Graph Decomposition in Routing and Compilers

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt beim Fachbereich Informatik und Mathematik
der Johann Wolfgang Goethe-Universität
in Frankfurt am Main

von
Philipp Klaus Krause
aus Mainz

Frankfurt 2015
(D 30)

vom Fachbereich Informatik und Mathematik der
Johann Wolfgang Goethe-Universität als Dissertation angenommen

Dekan:

Gutachter:

Datum der Disputation:

Kapitel 0

Zusammenfassung

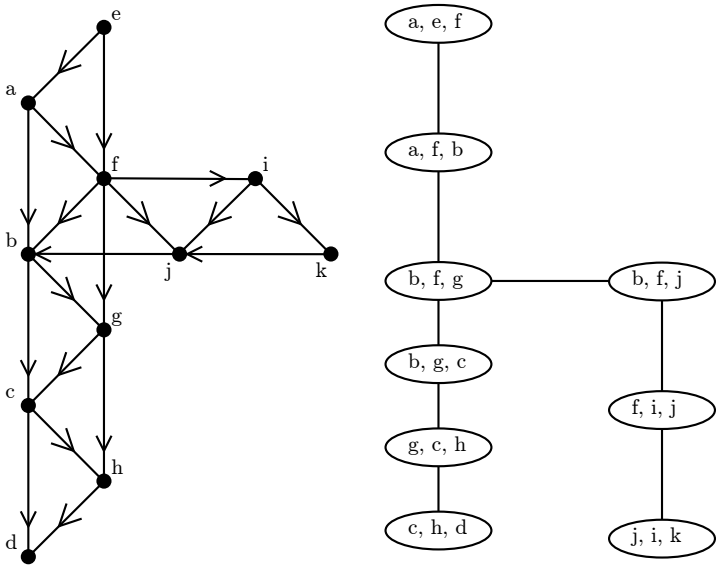
0.1 Schwere Probleme effizient lösen

Viele auf allgemeinen Graphen NP-schwere Probleme (z. B. Hamiltonkreis, k-Färbbarkeit) sind auf Bäumen einfach effizient zu lösen. *Baumzerlegungen*, Zerlegungen von Graphen in kleine Teilgraphen entlang von Bäumen, erlauben, dies zu effizienten Algorithmen auf baumähnlichen Graphen zu verallgemeinern. Die Baumähnlichkeit wird dabei durch die *Baumweite* abgebildet: Je kleiner die Baumweite, desto baumähnlicher der Graph.

Die Bedeutung der Baumzerlegungen [61, 113, 114] wurde seit ihrer Verwendung in einer Reihe von 23 Veröffentlichungen von Robertson und Seymour zur Graphminorentheorie allgemein erkannt. Das Hauptresultat der Reihe war der Beweis des Graphminorensatzes[1], der aussagt, dass die Minorenrelation auf den Graphen Wohlquasiordnung ist. Baumzerlegungen wurden in verschiedenen Bereichen angewandt. So bei probabilistischen Netzen[89, 69], in der Biologie [129, 143, 142, 110], bei kombinatorischen Problemen (wie dem in Teil II) und im Übersetzerbau [132, 7, 84, 83] (siehe Teil III). Außerdem gibt es algorithmische Metatheoreme [31, 9], die zeigen, dass sie für weite Problemklassen nützlich sind. Baumzerlegungen sind in dieser Arbeit von zentraler Bedeutung.

Die mittels Baumzerlegungen erzielten Erfolge auf baumähnlichen Graphen motivieren Versuche, diese auf größere Graphklassen zu verallgemeinern. Ein erfolgreicher Ansatz beruht auf *irrelevanten Knoten* und reduziert damit die Probleme auf der größeren Graphklasse auf Probleme auf einer Graphklasse kleiner Baumweite: Wenn der Eingabegraph zu einem Problem kleine Baumweite hat, wird das Problem mittels Baumzerlegungen gelöst. Andernfalls gibt es einen irrelevanten Knoten, so dass das Problem genau dann eine Lösung auf dem ursprünglichen Graphen hat, wenn es auch im Graphen ohne diesen irrelevanten Knoten eine Lösung hat. Es werden solange irrelevante Knoten gefunden und entfernt, bis ein Graph kleiner Baumweite verbleibt. Ein wichtiges Hilfsmittel zum Finden irrelevanter Knoten ist der Gitterminorensatz [115]: Nach diesem Satz enthalten Graphen großer Baumweite auch große Gitter als Minoren. So wird auch in Kapitel 5 vorgegangen. Die Gitter-Baumweite-Dualität ist auch in der Bidimensionalitätstheorie [37], einem weiteren erfolgreichen Ansatz, um auf

[1] Auch als „Wagners Vermutung“ bezeichnet, obschon nie von Wagner vermutet ([39], Seite 355); Wagner ging davon aus, dass ein es ein Gegenbeispiel gäbe ([136], Seite 61, Problem 9).



(a) Graph (b) Baumzerlegung

Abbildung 1: Graph mit Baumzerlegung minimaler Weite

größeren Graphklassen, als nur denen kleiner Baumweite, Probleme effizient zu lösen, von zentraler Bedeutung.

0.2 Ergebnisse

0.2.1 Baumzerlegungen

Die meisten unserer Ergebnisse erzielten wir mit Hilfe von Baumzerlegungen [61, 113, 114].

Definition (Baumzerlegung)**.** *Eine Baumzerlegung eines endlichen Graphen G ist ein Paar (T, χ), aus einem Baum T und einer Abbildung $\chi\colon V(T) \to 2^{V(G)}$ von der Knotenmenge des Baumes in die Potenzmenge der Knotenmenge des Graphen. Die $\chi(t)$ heißen Beutel. Es muss gelten: Für jeden Knoten $v \in V(G)$ gibt es ein $t \in V(T)$ mit $v \in \chi(t)$; für jede Kante $e \in E(G)$ gibt es einen Knoten $t \in V(T)$ dessen Beutel $\chi(t)$ beide Enden von e enthält; und für jeden Knoten $v \in V(G)$ ist $\{t \in V(T) \mid v \in \chi(t)\}$ zusammenhängend in T.*

Die Weite einer Baumzerlegung (T, χ) ist

$$\mathrm{w}(T, \chi) := \max\left\{ \left|\chi(t)\right| - 1 \;\middle|\; t \in V(T) \right\}.$$

Die Baumweite eines Graphen G ist

$$\mathrm{tw}(G) := \min\left\{ \mathrm{w}(T, \chi) \;\middle|\; (T, \chi) \text{ ist Baumzerlegung von } G \right\}.$$

Unsere Definition gilt gleichermaßen für gerichtete wie ungerichtete Graphen. Abbildung 1b zeigt eine Baumzerlegung minimaler Weite für den Graphen in Abbildung 1a.

Auf Graphklassen beschränkter Baumweite gilt: Baumzerlegungen lassen sich in Linearzeit finden [17], und mittels Baumzerlegungen lassen sich viele Probleme in Linearzeit lösen, so beispielsweise alle in monadischer Logik 2. Stufe formalisierbaren [31]. Allerdings sind die resultierenden Algorithmen zum Berechnen von Baumzerlegungen und Lösen von Problemen aufgrund hoher konstanter Faktoren in den Laufzeiten im Allgemeinen nicht praktisch anwendbar [120]. Daher werden andere Ansätze verwendet, um Baumzerlegungen zu erhalten, auch wenn diese im Allgemeinen nicht die kleinstmögliche Weite erreichen [132, 20, 19]. Für einzelne Probleme werden üblicherweise Algorithmen von Hand entworfen. Diese verwenden meist, wie auch die oben genannten unpraktischen, dynamische Programmierung [14] bottom-up entlang der Baumzerlegung. Auch die von uns gefundenen Algorithmen für Compileroptimierungen sind von dieser Art.

Die *parametrische Komplexitätstheorie* ermöglicht es, präziser darüber zu sprechen, wie Einschränkungen an Parameter der Eingabe von Problemen, beispielsweise an die Baumweite, in die Laufzeit eingehen. Dort gibt es die Klasse fpt der Probleme, die bei Parameter k und Eingabegröße n in Zeit $f(k)p(n)$ für eine berechenbare Funktion f und ein Polynom p gelöst werden können. Darüber hinaus gibt es weitere Komplexitätsklassen, die auch Probleme enthalten, bei denen solche Algorithmen nicht zu erwarten sind (vergleichbar dazu, wie die klassische Komplexitätstheorie keine Polynomialzeitalgorithmen für NP-schwere Probleme erwarten lässt), bis hin zur Klasse XP, die alle Probleme enthält, die bei Parameter k und Eingabegröße n in Zeit $n^{f(k)}$ für eine berechenbare Funktion f gelöst werden können.

$$\text{fpt} \subseteq W[1] \subseteq W[2] \subseteq \ldots \subseteq W[\text{SAT}] \subseteq W[P] \subseteq XP.$$

0.2.2 Disjunkte Wege

Beim Disjunkte-Wege-Problem, einem wohlbekannten Problem aus der Graphentheorie, ist bei Eingabe eines Graphen G und k Paaren von Knoten (s_1, t_1), $\ldots, (s_k, t_k)$ in G zu entscheiden, ob G k knotendisjunkte Wege $P_1, \ldots, P_k$, wobei jedes P_i Weg von s_i nach t_i ist, enthält. Das Problem (auch in seinen kantendisjunkten und gerichteten Varianten) ist selbst auf plättbaren Graphen NP-schwer [72, 135, 100, 81, 96], falls die Anzahl k der Wege Teil der Eingabe ist.

Robertson und Seymour zeigten, dass das Problem in Zeit $g(k)|V(G)|^3$ für eine berechenbare Funktion g gelöst werden kann, also insbesondere für beschränktes k in Polynomialzeit. Dies geschieht über irrelevante Knoten [116]: Falls der Graph G gewisse Anforderungen an seine Struktur erfüllt, wird das Problem mittels Baumzerlegungen gelöst. Falls der Graph die Anforderungen nicht erfüllt, lässt sich ein irrelevanter Knoten (d. h. das Problem ist genau dann in G lösbar, wenn es in G ohne diesen Knoten lösbar ist) finden, und aus dem Graphen entfernen. Das Entfernen irrelevanter Knoten wird solange wiederholt, bis der Graph die Anforderungen erfüllt.

Satz (Robertson und Seymour [118])**.** *Es gibt eine Funktion* $f\colon \mathbb{N} \to \mathbb{N}$*, so dass aus* $\text{tw}(G) \geq f(k)$ *folgt: Es gibt einen irrelevanten Knoten in* G*.*

Die Funktion $g(k)$, und damit die Laufzeit des Algorithmus zum Lösen des Disjunkte-Wege-Problems, ist exponentiell in $f(k)$. Bisher ist über die Funktion f allerdings nur bekannt, das sie berechenbar ist [75].

Obere Schranke[2]

Wir finden für plättbare Graphen eine einfach exponentielle Schranke an f:

Satz. *Für alle* $k \in \mathbb{N}$ *gilt:* $f(k) \leq (72k \cdot 2^k - 72 \cdot 2^k + 18)\lceil\sqrt{2k+1}\rceil$.

Unser Beweis ist abgesehen von der Verwendung des Gitterminorensatzes [115] elementar, und unabhängig von dem von Robertson und Seymour. Wir erhalten damit für plättbare Graphen nicht nur eine schärfere Schranke an f, sondern auch einen im Vergleich zu den über 500 Seiten bei Robertson und Seymour deutlich kürzeren Beweis für ein zentrales Resultat. Dies verwenden wir, um den asymptotisch schnellsten bisher bekannten Algorithmus für das Disjunkte-Wege-Problem auf plättbaren Graphen zu erhalten:

```
disjunkte_wege(Graph G)
{
  while (Wir finden einen irrelevanten Knoten in G)
    Entferne den irrelevanten Knoten aus G;
  Berechne eine Baumzerlegung kleiner Weite von G;
  Löse das Problem auf G mittels der Baumzerlegung;
}
```

Aus unserem Beweis des Satzes geht auch hervor, dass wir, so lange die Baumweite des Graphen größer als unsere Schranke an f ist, immer einen irrelevanten Knoten finden, und dass wir in Linearzeit überprüfen können, ob ein Knoten irrelevant ist. Die gesamte Schleife hat also Laufzeit in $O(|V(G)|^2)$. Für den nächsten Schritt kann der Separatorenalgorithmus [116] verwendet werden, der eine Baumzerlegung der Weite höchstens $4\,\mathrm{tw}(G)+1$ in Zeit $2^{O(\mathrm{tw}(G))}|V(G)|^2$ liefert. Der letzte Schritt lässt sich dann mittels dynamischer Programmierung in Zeit $O((k+1)^{4\,\mathrm{tw}(G)+1}|V(G)|)$ erledigen, womit die Laufzeit des gesamten Algorithmus in $2^{2^{O(k)}}|V(G)|^2$ liegt.

Aufbauend darauf wurde ein fpt-Algorithmus für gerichtete plättbare Graphen gefunden [34], während zuvor nur ein XP-Algorithmus bekannt war [126].

Untere Schranke[3]

Wir beweisen auch eine untere Schranke, die selbst für plättbare Graphen gilt:

Satz. *Für alle* $k \in \mathbb{N} \setminus \{0\}$ *gilt:* $f(k) \geq 2^k$.

Zum Beweis werden Graphen G_k, die keine irrelevanten Konten, aber $((2^k-1)\times(2^k-1))$-Gitter enthalten, und damit Baumweite $\geq 2^k-1$ haben, konstruiert.

Diese Schranke ist einfach exponentiell, wie auch die obige obere Schranke für plättbare Graphen. Sie zeigt somit, dass Algorithmen zur Lösung des Disjunkte-Wege-Problems, die irrelevante Knoten und Baumzerlegungen verwenden, keine weitere deutliche Verbesserung der Laufzeit gegenüber unserem Algorithmus aus dem vorigen Absatz erreichen können, auch wenn sie sich auf plättbare Graphen beschränken.

[2] Gemeinsame Arbeit mit Isolde Adler, Stavros G. Kolliopoulos, Daniel Lokshtanov, Saket Saurabh und Dimitrios M. Thilikos; zuerst auf der ICALP vorgestellt [2], ausführliche Fassung zur Veröffentlichung eingereicht [3].

[3] Gemeinsame Arbeit mit Isolde Adler; zuerst auf der ICALP vorgestellt [2], ausführliche Fassung zur Veröffentlichung eingereicht [5].

0.2.3 Übersetzer

Übersetzer (*Compiler*) übersetzen zwischen Computersprachen. Der klassische Fall ist die Übersetzung von einer höheren Programmiersprache, wie C, in Assemblercode. Übersetzer zerfallen üblicherweise in mehrere Phasen, wobei sukzessive die Eingabe über einen abstrakten Syntaxbaum und Zwischencode in Assemblercode übersetzt wird. Auf dem Zwischencode werden verschiedene Optimierungen ausgeführt; eine besonders wichtige ist die Registerallokation (Details siehe unten in den entsprechenden Abschnitten). Wenn wir jeden Befehl im Code als Knoten auffassen, und je zwei dieser Knoten mit einer Kante verbinden, falls sie direkt aufeinander folgend ausgeführt werden können, erhalten wir den *Kontrollflussgraphen*. In Übersetzern werden insbesondere die Kontrollflussgraphen des Zwischencodes in vielen Algorithmen verwendet.

Optimierende Compiler enthalten *Optimierungen*, die den erzeugten Code schneller, kürzer oder energieeffizienter machen. Obschon „Optimierungen“ genannt, können diese üblicherweise den Code zwar verbessern, aber nicht optimal machen, da für Optimalität NP-schwere Probleme zu lösen wären. Da Programme aller Art durch einfaches Übersetzen mit einem optimierenden Compiler von Optimierungen profitieren, und dadurch auf gegebener Hardware weniger Zeit, Energie und Speicher benötigen, beziehungsweise es ermöglichen, gegebene Aufgaben auf einfacherer Hardware zu erledigen, leisten diese einen bedeutenden Beitrag zur Reduzierung des Ressourcenverbrauchs.

Wir erreichen entscheidende Verbesserungen bei wichtigen Optimierungen: Der Platzierung von Bank-Selection-Instruktionen, der Redundanzelimination und der Registerallokation. Dazu verwenden wir Baumzerlegungen von Kontrollflussgraphen, und erhalten auch neue Erkenntnisse dazu, wie diese Baumzerlegungen berechnet werden können, und welche Baumweiten bei Kontrollflussgraphen auftreten. Wir implementierten unsere Ansätze in SDCC, einem C-Compiler für eingebettete Systeme erreichen auch empirisch deutliche Verbesserungen im erzeugten Code.

Definition (Strukturiertes Programm)**.** *Sei* $\mathfrak{k} \in \mathbb{N}$ *fest. Ein Programm heißt* $\mathfrak{k}$*-strukturiert, falls sein Kontrollflussgraph Baumweite höchstens* $\mathfrak{k}$ *hat.*

0.2.4 Baumweite von C[4]

Bei vielen Optimierungen werden Graphen verwendet, insbesondere Kontrollflussgraphen der Programme. Für einige klassische dabei auftretende Probleme gibt es Baumzerlegungen verwendende Algorithmen [132, 18, 7, 83, 84], von denen einige (die in den folgenden Abschnitten genannten) in SDCC [41], einem C-Compiler für eingebettete Systeme, implementiert wurden. Für Laufzeit und Ergebnisqualität dieser Algorithmen ist es wichtig, dass die verwendeten Baumzerlegungen des Kontrollflussgraphen *möglichst kleine Weite* haben.

SDCC verwendete zum Finden der Baumzerlegungen bisher Thorups Heuristik [132]. Thorup behauptet, dass die Heuristik Baumzerlegungen der Weite höchstens 6 fände, so lange der C-Code kein `goto` enthalte. Wir fanden C-Code, der kein `goto` enthält, und dessen Kontrollflussgraph Baumweite 7 hat. Wir können C-Code konstruieren, der kein `goto` enthält, und dessen Kontrollflussgraph Weite 2 hat, für den Thorups Heuristik aber Baumzerlegungen beliebig

[4] Gemeinsame Arbeit mit Lukas Larisch, zur Veröffentlichung eingereicht [87].

großer Weite liefert. Wir finden und zeigen die folgende Schranke:

Satz. *Für C-Funktionen können Baumzerlegungen der Kontrollflussgraphen der Weite* $\mathfrak{k} \leq 7 + \mathfrak{g}$ *in Linearzeit in der Anzahl der Knoten des Kontrollflussgraphen gefunden werden, wobei* $\mathfrak{g}$ *die Anzahl der Sprungmarken für* `goto` *ist.*

Insbesondere sind in C geschriebene Programme, soweit sie nicht exzessiv `goto` verwenden, strukturiert. Wir zeigen auch die Schärfe der Schranke:

Satz. *Für jedes* $\mathfrak{k} \geq 7$ *gibt es eine C-Funktion, deren Kontrollflussgraph Baumweite mindestens* $\mathfrak{k}$ *hat, und die höchstens* $\mathfrak{g} = \mathfrak{k} - 7$ *Sprungmarken für* `goto` *enthält.*

Dies korrigiert Thorup, und zeigt, wie `goto` sich auf die Baumweite auswirkt. Anders als bisherige Algorithmen zur Berechnung von Baumzerlegungen orientiert sich unserer eng an der verschachtelten Blockstruktur der C-Programme. Da er C-Code als Eingabe verwendet, is es allerdings aufwändig, ihn in Compilern als Ersatz für Algorithmen, die einen Kontrollflussgraphen als Eingabe erwarten, zu verwenden.

Empirisch untersuchen wir alternative Ansätze um Baumzerlegungen der Kontrollflussgraphen zu berechnen, sowohl im Hinblick auf ihre Laufzeit und die Weiten der erhaltenen Zerlegungen, als auch im Hinblick auf die Auswirkungen auf die Compilerlaufzeit und die Qualität des erzeugten Code. Es zeigt sich, dass Heuristiken für allgemeine Graphen [20] auch bei Kontrollflussgraphen Baumzerlegungen kleinerer Weiten finden als Thorups Heuristik. Die Kombination aus einem Präprozessor, der Vereinfachungsregeln [19] anwendet, gefolgt von einer allgemeinen Heuristik erreicht nochmals geringfügig kleinere Weiten bei mit Thorups Heuristik vergleichbaren Laufzeiten. Die führt in SDCC zu kürzeren Compilerlaufzeiten bei vergleichbarer Codequalität.

0.2.5 Optimale Platzierung von Bank-Selection-Instruktionen[5]

In vielen 8- und 16-Bit Mikrocontrollern wird für den Hauptspeicher ein logischer Adressraum verwendet, der kleiner als der physische Adressraum ist. Dann wird ein Teil des logischen Adressraums zur Einblendung verschiedener Teile des physischen Adressraums verwendet. Welcher Teil des physischen Adressraums dort eingeblendet wird, kann mittels *Bank-Selection-Instruktionen* geändert werden. Falls nötig, wird durch Einführen temporärer Variablen sichergestellt, dass jede Instruktion im Kontrollflussgraphen nur auf höchstens einen der einzublendenden Adressbereiche zugreift. Um das Problem graphtheoretisch zu modellieren, werden die verschiedenen einblendbaren Teiladressbereiche dann als Farben aufgefasst; damit sind die Knoten des Kontrollflussgraphen, deren Instruktionen auf die Teiladressbereiche zugreifen, entsprechend gefärbt; Instruktionen, die nicht auf einen einzublendenden Adressbereich zugreifen, bleiben ungefärbt. Die Kosten für das Einfügen von Bank-Selection-Instruktionen werden durch Kantengewichte am Kontrollflußgraphen modelliert. Um Bank-Selection-Instruktionen optimal zu platzieren, ist die Knotenfärbung so auf den ganzen Kontrollflußgraphen zu erweitern, dass die Kosten für Farbwechsel (also die Summe über die

[5] Vorgestellt auf der M-SCOPES 2013 [83].

Kantengewichte der Kanten, deren Enden verschieden gefärbt sind) minimiert werden.

Die optimale Platzierung von Bank-Selection-Instruktionen im Programm ist Aufgabe des Compilers und ein im Allgemeinen NP-schweres Problem [91], selbst wenn nur auf Codegröße optimiert wird (modelliert dadurch dass das Gewicht aller Kanten 1 ist). Bisherige Ansätze sind daher entweder nicht optimal [91, 93], oder haben keine polynomielle Schranke an die Laufzeit [122, 123]. In Übersetzern werden bisher meist einfache, nichtoptimale Ad-Hoc-Ansätze verwendet.

Wir zeigen:

Satz. *Für strukturierte Programme können Bank-Selection-Instruktionen in Polynomialzeit optimal platziert werden, für jede feste Anzahl an einblendbaren Teiladressbereichen sogar in Linearzeit.*

Der Beweis mittels Baumzerlegungen ist konstruktiv und wurde in SDCC implementiert. Er ersetzte den dort zuvor wie in anderen Compilern verwendeten Ad-Hoc-Ansatz. Wir können 2-strukturierte Programme der Länge n konstruieren, die zwei einblendbare Teiladressbereiche verwenden, für die alle bisherigen mit Polynomialzeit auskommenden Ansätze $O(n)$ Bank-Selection-Instruktionen einfügen, während unser Ansatz nur 3 einfügt.

0.2.6 Redundanzelimination[6]

Programme enthalten oft Redundanzen, was bedeutet, dass Berechnungen an mehr Stellen im Programm enthalten sind als notwendig, oder öfters als notwendig ausgeführt werden. *Redundanzelimination* ist eine Optimierung, die solche Programme verbessern kann, und bewirkt, dass diese Berechnungen an weniger Stellen im Programm vorkommen, oder seltener ausgeführt werden. Dazu ist es üblicherweise erforderlich, die Berechnungen an anderen Stellen als zuvor im Programm zu platzieren.

Mit Common Subexpression Elimination (CSE) verfügten bereits frühe Übersetzer über eine einfache Redundanzelimination für Codeabschnitte ohne Verzweigungen (also im Kontrollflussgraphen von Knoten mit Grad 2 induzierten zusammenhängenden Teilgraphen). Global CSE (GCSE) [29] erweiterte dies auf den gesamten Kontrollflussgraphen. Partielle Redundanzelimination (PRE) [102] verallgemeinerte GCSE auf Berechnungen, die nur auf manchen Pfaden durch den Kontrollflussgraphen redundant sind. PRE wurde einerseits zu Lazy Code Motion (LCM) [78], einer Lifetime-optimalen Variante von PRE, und andererseits zu spekulativer partieller Redundanzelimination (SPRE) [55, 125, 25], welche auf manchen Pfaden die Anzahl der Berechnungen erhöhen kann, um an anderer Stelle Berechnungen entfernen zu können, verbessert. Lifetime-optimale spekulative partielle Redundanzelimination (lospre) kombiniert die Vorteile beider Weiterentwicklungen von PRE. Allerdings haben alle bisherigen Algorithmen [139, 144] für lospre eine relativ hohe asymptotische Laufzeit (größer als quadratisch; Implementierungen haben bisher kubische Laufzeit). Wir zeigen:

Satz. *lospre ist für strukturierte Programme optimal in Linearzeit möglich.*

[6] Zur Veröffentlichung eingereicht [82].

Das Optimierungsziel (beispielsweise Codegröße oder -geschwindigkeit) kann dabei flexibel über eine Kostenfunktion vorgegeben werden, und sowohl die Elimination von Redundanz, als auch die sich auf neu eingeführte temporäre Variablen beziehende lifetime-Optimalität, berücksichtigen. Im Gegensatz zu den bisher bekannten Ansätzen kann auch die Verkleinerung der life-ranges bestehender Variablen berücksichtigt werden. Der Beweis mittels Baumzerlegungen ist konstruktiv und wurde in SDCC implementiert, womit SDCC erstmals über eine partielle Redundanzelimination verfügt

0.2.7 Registerallokation

Übersetzer legen Programmvariablen in physischem Speicher im Rechner ab, wobei es die Wahl zwischen Registern und Hauptspeicher gibt. Die Entscheidung, welche Variablen in welche Register abgelegt werden und welche im Hauptspeicher abgelegt werden, wird bei der *Registerallokation* getroffen. Üblicherweise ist der Hauptspeicher deutlich größer und langsamer als die Register und im Maschinencode sind Instruktionen, die auf Register zugreifen meist deutlich kürzer als solche, die auf den Hauptspeicher zugreifen. In Verbindung damit, dass nahezu jede Instruktion mehrere Variablen schreibt oder liest macht dies Registerallokation zur wichtigsten Optimierung im Übersetzer.

Bei der Registerallokation sind auch Registeraliasing (überlappende Register die nicht gleichzeitig genutzt werden können), Registerpräferenzen (z. B. durch Instruktionen, die abhängig davon, in welchen Registern die Operanden liegen, unterschiedlich schnell sind) und Coalescing (Wegoptimieren von Zuweisungen, indem die Operanden im gleichen Register abgelegt werden) zu berücksichtigen. Abhängig vom Optimierungsziel sind die Auswirkungen auf Codegröße, Geschwindigkeit und Energieverbrauch zu berücksichtigen.

Ohne diese Aspekte kann Registerallokation vereinfacht als Suche nach einer konfliktfreien Färbung eines möglichst großen induzierten Teilgraphen des *Konfliktgraphen* der Variablen aufgefasst werden. Wenn die Werte zweier Variablen zur gleichen Zeit im Speicher vorliegen müssen, und sie somit nicht im gleichen Register abgelegt werden können, wird dies im Konfliktgraphen durch eine Kante repräsentiert. Der Konfliktgraph ist Schnittgraph zusammenhängender Teilgraphen des Kontrollflussgraphen.

Aufgrund der Bedeutung der Registerallokation wurde sie seit geraumer Zeit ausführlich erforscht, beginnend mit dem klassischen heuristisch Graphen färbenden Ansatz von Chaitin [27, 26], der später auch auf komplexere Architekturen verallgemeinert wurde [128]. Neuere Ansätze übersetzen in andere NP-Schwere Probleme (ILP, PBQP), und gehen diese heuristisch mit Solvern an [53, 48, 124, 62]. Für Just-In-Time-Compiler wird ein heuristischer Linearzeitalgorithmus vorgezogen, der optimal ist, falls der Kontrollflussgraph ein Pfad ist [109].

Komplexität der Registerallokation[7]

Für Schwereresultate kann eine vereinfachte Variante der Registerallokation (ohne Aliasing, Präferenzen, Coalescing) auf $r \in \mathbb{N}$ Registern betrachtet werden. Dann ist bei Eingabe eines Kontrollflussgraphen und zugehörigen Konfliktgraphen mit Knotengewichten ein r-färbbarer Teilgraph des Konfliktgraphen von

[7] Veröffentlicht in DAM [85].

möglichst großem Gewicht zu finden. Die Knotengewichte modellieren dabei die Häufigkeit der Zugriffe auf die entsprechenden Variablen im zu übersetzenden Programm. Wenn wir eine zusätzliche Zahl g in der Eingabe haben, erhalten wir ein Entscheidungsproblem: Bei Eingabe eines Kontrollflussgraphen und zugehörigen Konfliktgraphen mit Knotengewichten ist zu entscheiden: Gibt es einen r-färbbaren Teilgraphen des Konfliktgraphen, so dass die Summe der Gewichte der nicht im Teilgraph enthaltenen Knoten höchstens g ist? Die folgenden Schwereresultate gelten für das Entscheidungsproblem.

Im Allgemeinen ist Registerallokation selbst bei festem $g = 0$ und $r = 3$ NP-schwer [27, 72]. Falls die Anzahl r der Register Teil der Eingabe ist, ist das Problem selbst für $g = 0$ und 2-strukturierte Programme NP-schwer [50]. Mit $g = 0$ und Parameter r ist das Problem für strukturierte Programme in fpt [18].

Wir zeigen mittels einer Konstruktion von Programmen von Baumweite 2 aus Schaltkreisen, das letzteres bei allgemeinem g nicht zu erwarten ist:

Satz. *Registerallokation mit Parameter Registerzahl r ist W[SAT]-schwer, selbst für strukturierte Programme.*

Somit ist zu erwarten, dass die Laufzeit jedes optimalen Algorithmus für Registerallokation mindestens ein Polynom ist, dessen Exponent mit r wächst. Dies lässt keine entscheidende Verbesserung der Laufzeit gegenüber unserem Ergebnis im nächsten Abschnitt zu.

Im Beweis reduzieren wir *Weighted Circuit Satisfiability* auf Registerallokation. Dazu werden logische Gatter aus dem Eingabeschaltkreis in Teilgraphen des Kontrollflussgraphen übersetzt, und dann entsprechend dem Schaltkreis zusammengesetzt. Auch g wird abhängig vom Schaltkreis gewählt.

Optimale Registerallokation in Polynomialzeit[8]

Wir zeigen, dass selbst unter Berücksichtigung von Registeraliasing, Registerpräferenzen, Coalescing und für Optimierung auf Codegröße, Geschwindigkeit, Energieverbrauch oder Kombinationen davon gilt:

Satz. *Bei fester Registerzahl gilt: Für strukturierte Programme können Register optimal in Polynomialzeit alloziert werden.*

Der Beweis mittels Baumzerlegungen ist konstruktiv und wurde in SDCC implementiert; im Vergleich mit dem zuvor in SDCC verwendeten Registerallokator wird damit deutlich besserer Code erzeugt. Bei verschiedenen Architekturen und Benchmarks konnten wir im Vergleich zu einem herkömmlichen Registerallokator Codegrößenreduzierungen um 20% erreichen. Für die meisten von SDCC unterstützten Architekturen wird inzwischen bei Standardeinstellungen unser Registerallokator verwendet. Registerallokation ist im Wesentlichen ein Problem auf dem Konfliktgraphen der Variablen, der keine beschränkte Baumweite hat, aber Schnittgraph des Kontrollflussgraphen ist. Im Gegensatz zu den meisten Algorithmen auf Baumzerlegungen, nutzt unser Ansatz somit Baumzerlegungen eines Graphen beschränkter Baumweite, um ein Problem auf einem anderen Graphen, der selbst keine beschränkte Baumweite hat, zu lösen.

[8] Vorgestellt auf der CC 2013 [84].

Byteweise Registerallokation[9]

Traditionell wurden Variablen in der Registerallokation als unteilbare Einheiten angesehen: Jede Variable wurde entweder in einem Register oder im Hauptspeicher abgelegt. Eine gewisse Flexibilität entstand aus der Definition der einzelnen Register: Beispielsweise gibt es in der Zilog Z80-Architektur Registerpaare, die entweder als einzelnes 16-Bit-Register oder als als zwei einzelne 8-Bit-Register angesehen werden, wobei der Registerallokator dann das Aliasing berücksichtigen muss.

Wir haben untersucht, welchen Vorteil eine flexiblere Handhabung der einzelnen Bytes in der Registerallokation bietet: Für jedes einzelne Byte jeder einzelnen Variable wird sowohl der Hauptspeicher als auch jedes einzelne Byte jedes Registers als möglicher Speicherort betrachtet. Ein verwandtes, aber orthogonales Thema ist *Bitbreiten berücksichtigende Registerallokation* [130, 12, 103, 11].

Aufbauend auf dem im vorhergehenden Abschnitt vorgestellten Registerallokator wurde byteweise Registerallokation im eigens dafür von uns geschriebenen STM8-backend von SDCC implementiert und ermöglicht die Erzeugung deutlich besseren Codes im Vergleich zu herkömmlicher Registerallokation. In Standardbenchmarks (Whetstone, Dhrystone, Coremark) wurde eine Codegrößenreduzierung von 12,1% bis 28,2% bei einer Erhöhung der Scores von 2,5% bis 31,4% erzielt. Dazu werden im Registerallokator einzelne Registerbytes als Register behandelt werden, was zwar bedeutende Änderungen in der Codegenerierung und Kostenfunktion erfordert, aber es ermöglicht, den im vorherigen Abschnitt beschriebenen Registerallokator weitgehend unverändert einzusetzen. Der Exponent in der theoretischen Schranke an die Laufzeit der optimalen Registerallokation erhöht sich dabei um einen Faktor, der der Breite des größten Registers in Bytes entspricht (beim STM8 ist dies das zwei Byte breite Register `x`).

[9] Vorgestellt auf der SCOPES 2015 [86].

Contents

0 Zusammenfassung **3**
- 0.1 Schwere Probleme effizient lösen . . . 3
- 0.2 Ergebnisse . . . 4
  - 0.2.1 Baumzerlegungen . . . 4
  - 0.2.2 Disjunkte Wege . . . 5
  - 0.2.3 Übersetzer . . . 7
  - 0.2.4 Baumweite von C . . . 7
  - 0.2.5 Optimale Platzierung von Bank-Selection-Instruktionen . 8
  - 0.2.6 Redundanzelimination . . . 9
  - 0.2.7 Registerallokation . . . 10

I Introduction **17**

1 Graph Decomposition in Routing and Compilers **19**
- 1.1 Solving hard problems efficiently . . . 19
- 1.2 Main results . . . 20

2 Graphs and Tree-decompositions **27**
- 2.1 Notation . . . 27
- 2.2 Graphs . . . 27
- 2.3 Tree-Decompositions . . . 32
- 2.4 Coloring using tree-decompositions . . . 35

3 Compilers **37**
- 3.1 Role and Structure . . . 37
- 3.2 Register Allocation . . . 39

II Disjoint Paths Problem **43**

4 The Disjoint Paths Problem **45**
- 4.1 Introduction . . . 45
- 4.2 Irrelevant vertices and the Disjoint Paths Problem . . . 46

5 Irrelevant vertices in Planar Graphs **49**
- 5.1 Concentric Cycles and Segments . . . 50
- 5.2 Bounding the number of segment types . . . 52
- 5.3 Bounding the size of segment types . . . 57

6 Graphs without Irrelevant Vertices **63**
6.1 Disc-with-edges embeddings 64
6.2 Linkages 69

III Compilers **75**

7 Structured Programs and the Tree-width of C **77**
7.1 Structured Programs 78
7.2 Thorup's algorithm 80
7.3 Upper bound and algorithm 81
7.4 Tightness of bound 86
7.5 Experimental results 88
7.6 Conclusion 91

8 Optimal Placement of Bank Selection Instructions in Polynomial Time **95**
8.1 Introduction 95
8.2 Problem Description 96
8.3 Optimal Placement of Bank Selection Instructions in Polynomial Time 100
8.4 Prototype Implementation 103
8.5 Examples 104
8.6 Conclusion 104

9 lospre in linear time **109**
9.1 Introduction 109
9.2 Problem Description 110
9.3 lospre in linear time 115
9.4 Prototype 118
9.5 Extending lifetime optimality 120
9.6 Conclusion 120

10 Optimal Register Allocation in Polynomial Time **121**
10.1 Introduction 121
10.2 Problem Description 123
10.3 Optimal Polynomial Time Register Allocation 125
10.4 Remarks 128
10.5 Complexity of Register allocation 129
10.6 Prototype implementation 130
10.7 Experimental results 131
10.8 Conclusion 137

11 Bytewise Register Allocation **139**
11.1 Introduction 139
11.2 Motivation 141
11.3 Implementation 143
11.4 Experiments 144
11.5 Results 145
11.6 Conclusion 147

12 The Complexity of Register Allocation **149**
12.1 Introduction 149
12.2 Preliminaries 150
12.3 Previous results 152
12.4 W[SAT]-hardness of register allocation 153
12.5 Conclusion 157

Conclusion **161**

Bibliography **163**

Index **175**

Part I

Introduction

Chapter 1

Graph Decomposition in Routing and Compilers

1.1 Solving hard problems efficiently

Many problems that are computationally hard on general graphs (e. g. Hamiltonian circuit, k-colorability) are trivial to handle efficiently on trees. *Tree-decompositions*, which decompose graphs into small subgraphs along trees, allow to generalize this to efficient algorithms on classes of tree-like graphs. *Tree-width* (tw) captures the property of being tree-like: The smaller the tree-width, the more tree-like the graph.

Tree-decompositions [61, 113, 114] became popular after being used in a series of 23 papers on graph minor theory by Robertson and Seymour. The main theoretical result of the series was the proof of the graph minor theorem[1], which states that graphs are well-quasi-ordered by the minor relation.

Tree-decompositions have been applied in various areas including probabilistic networks [89, 69], computational biology [129, 143, 142, 110], various combinatorial problems (such as the one in Part II) and compiler construction [132, 7, 84, 83] (see Part III), and there are algorithmic meta-theorems [31, 9] that show that they can be useful for large classes of problems. Tree-decompositions are a central concept in this thesis. Algorithms using tree-decompositions usually use dynamic-programming [14], bottom-up along the tree-decomposition (and so do the ones we found for compiler optimizations).

Due to the huge success of tree-decompositions in solving problems efficiently on tree-like graphs, there were many attempts to generalize such approaches beyond tree-like graphs. A particularly successful approach is the irrelevant vertex technique, which reduces problems on larger graph classes to problems on graph classes of small tree-width: If the input graph to a problem has small tree-width, the problem is solved using tree-decompositions. If it does not have small tree-width, then an *irrelevant vertex* can be found in the graph, a vertex such that the problem has a solution on the original graph exactly if it has a solution on the graph with the irrelevant vertex removed. One then keeps

[1]The graph minor theorem has also been called "Wagner's conjecture", although Wagner never conjectured it ([39], page 355) and in the reference given by Robertson and Seymour to justify the name, Wagner asked for a counterexample ([136], page 61, Problem 9).

finding and removing irrelevant vertices until the tree-width of the remaining graph is small. An important tool in finding irrelevant vertices is the *grid minor theorem* [115], which states that graphs of high tree-width have large grid minors. We use the irrelevant vertex technique and the grid minor theorem for our result in Chapter 5. The relationship between tree-width and grid minors is also of central importance in bidimensionality theory [37], another very successful approach to generalize beyond tree-width, discovered by Demaine, Hajiaghayi et alii over a series of 10 papers.

Parametrized complexity theory allows us to speak precisely about how restrictions on the input of a problem (such as bounding the tree-width on graphs in the input) affect algorithm runtime. The class fpt contains the problems that parametrized by k and with input size n can be solved in time $f(k)p(n)$ for a computable function f. There are further parametrized complexity classes that contain problems, for which such algorithms are considered unlikely (similar to how classic complexity theory considers polynomial-time algorithms unlikely for NP-hard problems), such as $W[1]$, up to the class XP, which contains all problems that parametrized by k and with input size n can be solved in time $n^{f(k)}$ for a computable function f.

$$\text{fpt} \subseteq W[1] \subseteq W[2] \subseteq \ldots \subseteq W[\text{SAT}] \subseteq W[P] \subseteq XP.$$

1.2 Main results

Disjoint Paths

The Disjoint-Paths Problem, a well known graph-theoretic problem, asks, given a graph G and a set of pairs of terminals $(s_1, t_1), \ldots, (s_k, t_k)$, whether there is a collection of k pairwise vertex-disjoint paths linking s_i and t_i, for $i = 1, \ldots, k$. The problem is NP-complete, along with its edge-disjoint or directed variants, even when the input graph is planar [72, 135, 100, 81, 96], when k is part of the input.

Robertson and Seymour show that it can be solved in time $g(k)|V(G)|^3$ for a computable function g, and thus in polynomial time for bounded k. They use irrelevant vertices [116]: In case the graph G satisfies certain structural conditions, the problem is solved using tree-decompositions. Otherwise, an irrelevant vertex (i. e. a vertex such that G has a solution to the problem exactly if G with the vertex removed has a solution) is found and removed. This is repeated until the structural conditions are satisfied.

Theorem 1.1 (Robertson und Seymour [118])**.** *There is a function* $f\colon \mathbb{N} \to \mathbb{N}$*, such that* $\text{tw}(G) \geq f(k)$ *implies: there is an irrelevant vertex in* G*.*

The function $g(k)$ and thus the runtime of the resulting algorithm for solving the Disjoint-Paths Problem using irrelevant vertices is exponential in $f(k)$. The function f is computable [75], but little else is known about it.

Upper bound[2]

For planar graphs we give a singly exponential bound on f:

Theorem 1.2. *$f(k) \leq (72k \cdot 2^k - 72 \cdot 2^k + 18)\lceil\sqrt{2k+1}\rceil$ for all $k \in \mathbb{N}$.*

Our proof uses the grid minor theorem [115], but is otherwise elementary. It is independent of the proof by Robertson and Seymour. We thus obtain for planar graphs both a better bound on f, and a much shorter (compared to more than 500 pages by Robertson and Seymour) proof for a central result. We use it to give the asymptotically fastest known algorithm for the Disjoint-Paths Problem on planar graphs:

```
disjoint_paths(Graph G)
{
  while (We find an irrelevant vertex v in G)
    Remove the vertex v from G;
  Compute a tree-decomposition of small width of G;
  Solve the problem using the tree-decomposition;
}
```

Our proof implies that, as long as the tree-width is bigger than our bound on f, we can always find an irrelevant vertex, and checking if a vertex is irrelevant can be done in linear time. This gives us time $O(|V(G)|^2)$ for the loop. The tree-decomposition can be obtained using e. g. the separator algorithm [116], which finds a tree-decomposition of width at most $4\,\mathrm{tw}(G)+1$ in time $2^{O(\mathrm{tw}(G))}|V(G)|^2$. The last step then can be done using dynamic programming in time $O((k+1)^{4\,\mathrm{tw}(G)+1}|V(G)|)$, resulting in total runtime $2^{2^{O(k)}}|V(G)|^2$ for the algorithm.

For directed planar graphs, an XP algorithm was previously known [126]. Building on our work, an fpt-algorithm has been found [34].

Lower bound[3]

We also prove a lower bound, that holds even for planar graphs:

Theorem 1.3. *$f(k) \geq 2^k$ for all $k \in \mathbb{N} \setminus \{0\}$.*

In the proof, we construct graphs G_k, that do not contain irrelevant vertices. They contain $((2^k-1) \times (2^k-1))$-grids, and thus have tree-width $\geq 2^k - 1$.

This bound is singly exponential, just like our upper bound for planar graphs. It thus implies that any faster algorithm for the Disjoint-Paths Problem on planar graphs would have to use methods different from the irrelevant vertex technique.

Compilers

Compilers translate between computer languages. In the classical setting they translate from a higher level programming language, such as C, to a lower level language such as assembler code. Optimizing compilers contain *optimizations*,

[2]Based on joint work with Isolde Adler, Stavros G. Kolliopoulos, Daniel Lokshtanov, Saket Saurabh and Dimitrios M. Thilikos, which has been presented at ICALP 2011 [2]. A journal version has been submitted [3] for publication.

[3]Based on joint work with Isolde Adler, which has been submitted [5] for publication.

which attempt to improve the generated code, to make the resulting program faster, smaller or less energy-consuming. Even though they are called optimizations, usually they only improve the code, but do not make it optimal; generating code that is, by some measure, optimal usually is a computationally hard problem. Since all it takes for a program to benefit from optimizations is being compiled with an optimizing compiler, compiler optimizations have a huge, immediate effect on resource usage in the real world.

Using tree-decomposition of control-flow graphs, we achieve decisive improvements in important compiler optimizations: The placement of bank selection instructions, redundancy elimination and register allocation. The use of tree-decompositions allows efficient, provably optimal algorithms. We also obtain new knowledge on how to obtain the tree-decompositions and which tree-widths occur in control-flow graphs. We implemented our approaches in SDCC, a C compiler for embedded systems and empirically achieved substantial improvements in the generated code.

The tree-width of C[4]

Many optimizations use graphs, in particular control-flow graphs. Starting with Thorup's work on an approximation problem in register allocation [132] in 1998, approaches based on tree-decompositions of control-flow graphs have been introduced for many classical optimization problems in compilers [18, 7, 83, 84, 85], some of which have proven their practical usefulness in implementations in SDCC [41], a mainstream C compiler for embedded systems. For the runtime and result quality of these algorithms it is important that the used tree-decompositions have small width.

In SDCC, so far Thorup's heuristic [132] has been used to obtain the tree-decompositions. Thorup claims that the heuristic yields tree-decompositions of width at most 6, when applied to control-flow graphs from C code that contains no `goto` statements. We found C code that contains no `goto`, but the control-flow graph has tree-width 7. We can also construct C code that contains no `goto`, its control-flow graph has tree-width 2, but Thorup's heuristic yields tree-decompositions of arbitrarily large width. We found and proved the following bound:

Theorem 1.4. *For C functions that contain $\mathfrak{g} \in \mathbb{N}$ labels targeted by `goto`, tree-decompositions of the control-flow graphs of width $\mathfrak{k} \leq 7 + \mathfrak{g}$ can be found in time linear in the number of nodes in the control-flow graph.*

We also show the tightness of the bound:

Theorem 1.5. *For every $\mathfrak{k} \geq 7$, there is a C function that has tree-width at least $\mathfrak{k}$ and at most $\mathfrak{g} = \mathfrak{k} - 7$ labels targeted by `goto`.*

This corrects Thorup and shows the effect that `goto` has on the tree-width. Unlike previous algorithms for computing tree-decompositions, our approach works closely along the nested block-structure of the C programs. However, since it expects C code as input using it to replace algorithms that expect a CFG as input in compilers is not straightforward.

[4] Joint work with Lukas Larisch, submitted for publication.

We empirically evaluate various approaches to finding tree-decompositions of control-flow graphs on many real-world C code examples, such as the standard library, standard benchmarks and operating systems. We also investigate the impact the different approaches have on compiler runtime and code quality in SDCC. We find that empirically, generic approaches to obtaining tree-decompositions of graphs [20, 19] are better than Thorup's heuristic. The combination of a preprocessor that uses reduction rules followed by a generic heuristic performs best. This results in shorter compiler runtime at comparable code quality.

Optimal Placement of Bank Selection Instructions in Polynomial Time[5]

Partitioned memory architectures are common in 8- and 16- bit microcontrollers. A part of the logical address space is used as a window into a larger physical address space. The mapable parts of the physical address space are called *memory banks*. Which part of the physical address space is mapped into the window is selected by *bank selection instructions*.

Graph theoretically, we consider the memory banks to be colors. Instructions in the control-flow graph are colored by the bank which they access. Instructions that do not access memory banks are uncolored. Edges in the control-flow graphs have weights representing the costs of inserting bank selection instructions there. Placing bank selection instructions optimally means extending the coloring to the whole control-flow graph in a way that minimizes costs for the inserted bank selection instructions. Placing bank selection instructions is NP-hard in general [91], even when just optimizing for code size (i. e. when all edge weights in the control-flow graph are 1). Thus, previous approaches are not optimal [91, 93], or do not have a polynomial bound on their runtime [122, 123]. Compilers tend to use simple non-optimal ad-hoc approaches.

Using tree-decompositions, we prove:

Theorem 1.6. *Optimal Placement of Bank Selection Instructions can be done in polynomial time for structured programs. For a fixed number of memory banks it can be done in linear time.*

We implemented the proof in SDCC, where it replaced the previously used ad-hoc approach. We can construct 2-structured programs of length n, using just two memory banks, for which our approach inserts 3 bank selection instructions, while all previous polynomial-time approaches insert $O(n)$ bank selection instructions.

Redundancy Elimination[6]

Programs tend to contain redundancies, which means that computations are done in more places or more often than necessary. *Redundancy Elimination* improves programs by reducing the number or frequency of computations, usually moving them elsewhere in the program.

Even early optimizing compilers had common subexpression elimination (CSE) for straight-line code. Global common subexpression elimination (GCSE) [29]

[5] Presented at M-SCOPES 2013 [83].
[6] Submitted for publication [82].

extended this to the whole control-flow graph. Partial redundancy elimination (PRE) [102] generalized GCSE (and some other techniques such as loop-invariant code-motion (LICM)) to computations that are redundant only on some paths in the CFG. Improvements led to lazy code-motion (LCM) [78], which is a lifetime-optimal variant of PRE, i. e. the lifetimes of introduced temporary variables are minimized, which is important to keep register pressure low. Another improvement to PRE is speculative PRE (SPRE) [55, 125, 25], which can increase the number of computations on some paths in order to reduce the total number of computations done (based on profiling information). The natural improvement is combining the advantages of LCM and SPRE, resulting in lifetime-optimal SPRE (lospre) [139, 144]. However, previously, the fastest known algorithms for lospre were randomized with expected superquadratic runtime (implementations tend to use deterministic algorithms with cubic runtime).

We show:

Theorem 1.7. *lospre can be done in linear time for structured programs.*

The optimization goal can be chosen through a cost function, which can take into account the elimination of redundancy, the lifetime-optimality and the changes in life-ranges of existing variables (the latter aspect has not been handled in previous approaches). Our proof uses tree-decompositions, is constructive and has been implemented in SDCC.

Register Allocation

Compilers place program variables in physical storage, deciding between registers and main memory. The problem of deciding which variables go into which registers, and which ones are *spilt* into main memory is called *register allocation*. Main memory is much bigger and slower than registers, and instructions using register operands tend to be shorter than instructions using operands in main memory. This makes register allocation the most important optimization. Further aspects to be considered are register aliasing (multiple registers mapping to overlapping physical hardware), register preferences (certain instructions being more or less efficient depending on which registers the operands reside in) and coalescing (optimizing out instructions by using suitable placement in registers) as well as different optimization goals, such as code size, speed or energy consumption.

Due to the importance of register allocation, it has been extensively researched, starting with Chaitin's classic graph-coloring heuristic approach [27, 26], which later has been generalized to more complex architectures [128]. Newer approaches translate to other NP-hard problems (ILP, PBQP), and attack these heuristically with solvers [53, 48, 124, 62]. For just-in time compilers a heuristic linear time approach that is optimal for straight-line code only is preferred [109].

Complexity of Register Allocation[7]

For hardness results, we consider a simplified version of register allocation (ignoring aliasing, preferences and coalescing) on $r \in \mathbb{N}$ registers. Then, register

[7] Published in in DAM [85].

allocation becomes the following problem: Given a control-flow graph, and a corresponding conflict graph with node weights, find an r-colorable induced subgraph of the conflict graph of maximum weight. By having an additional number g in the input we get a decision problem: Is there an r-colorable induced subgraph of the conflict graph, such that the sum of the weights of the nodes outside it are at most g? The following hardess results hold for the decision problem.

In general, register allocation is NP-hard even for fixed $g = 0$ and $r = 3$ [27, 72]. When r is part of the input, the program is NP-hard even for $g = 0$ and 2-structured programs [50]. For structured programs, $g = 0$ and parameter r, the problem is in fpt [18]. Constructing programs of tree-width 2 from circuits, we show that this is unlikely for general g:

Theorem 1.8. *Register allocation, when parametrized by the number of registers r, is W[SAT]-hard, even for structured programs.*

For optimal approaches to register allocation, this makes substantial improvements in runtime over our approach in the next section unlikely.

The proof reduces *Weighted Circuit Satisfiability* to register allocation by translating gates from the input circuit into subgraphs of the control-flow graph, and connecting them according to the circuit. g is chosen depending on the circuit.

Optimal Register Allocation in Polynomial Time[8]

We show that even when considering register aliasing, register preferences, coalescing and when optimizing for code size, speed, energy consumption or aggregates thereof the following holds:

Theorem 1.9. *For a fixed number of registers, register allocation can be done in polynomial time for structured programs.*

This is the first optimal approach that has polynomial runtime and works for such a huge class of programs. The proof uses tree decompositions, and has been implemented in SDCC. Compared to the previously used register allocator we obtain substantially better code, with code size reductions of about 20% at various architectures and benchmarks. For most architectures supported by SDCC, our register allocator has become the default. Graph-theoretically, register allocation is mostly a problem on the conflict graph, which does not have bounded tree-width, but is the intersection graph of connected subgraphs of a graph of bounded tree-width. Unlike most algorithms on tree-decompositions, our algorithm uses tree-decompositions of a graph of bounded tree-width to solve a problem on another graph that does not have bounded tree-width.

Bytewise Register Allocation[9]

Traditionally, variables have been considered as atoms by register allocation: Each variable was to be placed in one register, or spilt. Some flexibility arose from what would be considered a register: Register aliasing allowed to treat a

[8] Presented at CC 2013 [84].
[9] Presented at SCOPES 2015 [86].

register meant to hold a 16-bit variable as two registers that could hold an 8-bit variable each (with the register allocator handling the resulting aliasing). We allow for far more flexibility in register allocation: We decide on the storage of variables bytewise, i. e. we decide for each individual byte in a variable whether to store it in memory or a register, and consider any byte of any register as a possible storage location. A related, but orthogonal topic is bitwidth-aware register allocation [130, 12, 103, 11].

We wrote the STM8-backend for SDCC, and building on the register allocator from the previous section, implemented bytewise register allocation for it. In standard benchmarks (Whetstone, Dhrystone, Coremark) we achieved code size reductions of 12.1% to 28.2% and score increases of about 2.5% to 31.4%. For register allocation we treat individual bytes of variables as variables, and individual bytes of registers as registers. This allows us to easily use the register allocator from the previous section, even though implementing bytewise register allocation required substantial effort in code generation and the implementation of the cost function.

Chapter 2

Graphs and Tree-decompositions

This chapter gives a gentle introduction to graphs. Understanding the basic concepts related to graphs is necessary for understanding this thesis. We use mostly standard notation. Readers interested in graphs beyond what is presented here are referred to standard works on graph theory [88, 39].

2.1 Notation

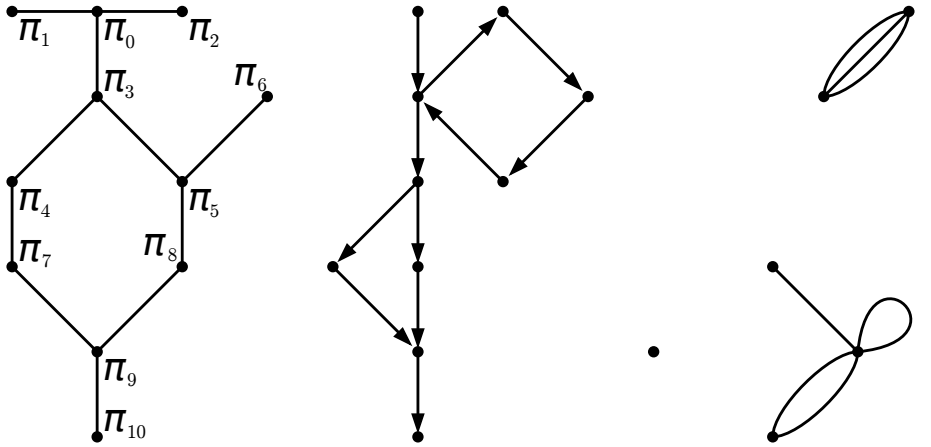
Let $\mathbb{N} := \{0, 1, 2, \ldots\}$ denote the set of *natural numbers*, $\mathbb{Z}$ the set of *integers*, and $\mathbb{R}$ the set of *real numbers*. For $k \in \mathbb{N}$, we let $[k] := \{1, \ldots, k\}$. For a set S we let $2^S := \{M \mid M \subseteq S\}$ denote the *power set* of S. For sets S and R, let $S \times R := \{(s, r) \mid s \in S, r \in R\}$ denote the *product* of S and R. For a set S and a natural number k let $S^k := S \times S \times \cdots \times S$ be the product of S k times with itself, and $S^* := \bigcup_{i \in \mathbb{N}} S^i$. We denote the *empty set* by $\varnothing := \{\}$. For a function $f\colon X \to Y$ and $Z \subseteq X$, we define the *restriction* $f|_Z\colon Z \to f(Z)$ for $z \in Z$ as $f|_Z(z) := f(z)$. A set S is said to be *partioned* into $n \in \mathbb{N}$ sets $S_1, \ldots, S_n$, if each element of S is contained in exactly one set S_i for an $i \in [n]$.

2.2 Graphs

Definition 2.1 (Graph)**.** *An (undirected) graph $G = (V, E)$ is a pair of a set of nodes or vertices V and a set of edges $E \subseteq \{e \mid e \in 2^V, |e| = 2\}$. For an edge $e = \{x, y\}$, the nodes x and y are called endpoints of the edge e, and e is called an edge from x to y.*

A directed graph is a pair $G = (V, E)$ of a set of nodes or vertices V and a set of edges $E \subseteq V \times V$. For an edge $e = (x, y)$, the nodes x and y are called endpoints of the edge e, and e is called an edge from x to y.

For a graph $G = (V, E)$ we define $V(G) := V$ and $E(G) := E$, and also use the notation $|G| := |V(G)|$. A graph G is called *finite*, if both $V(G)$ and $E(G)$ are finite. We believe it to be clear, which of our results hold for finite graphs only, and will usually not explicitly mention such restrictions. A *loop* is an edge, for which both endpoints are the same. There are minor variations in



(a) Undirected graph (b) Directed graph (c) Undirected Multigraph

Figure 2.1: Graphs

the way different authors define graphs, in particular with respect to allowing loops. Our above definitions allow loops for directed graphs, but disallow them for undirected graphs. In *multigraphs*, we have a multiset of edges instead of a set of edges, and we also always allow loops.

Graphs are usually visualized by drawing the nodes as dots and the edges as lines or arrows between nodes (we will later introduce drawings of graphs more formally), as in Figure 2.1: Figure 2.1a shows an undirected graph with node set $\{\pi_0, \pi_1, \ldots, \pi_{10}\}$. For the directed graph in Figure 2.1b and the undirected multigraph in Figure 2.1c, the visualization does not give us details on the node set.

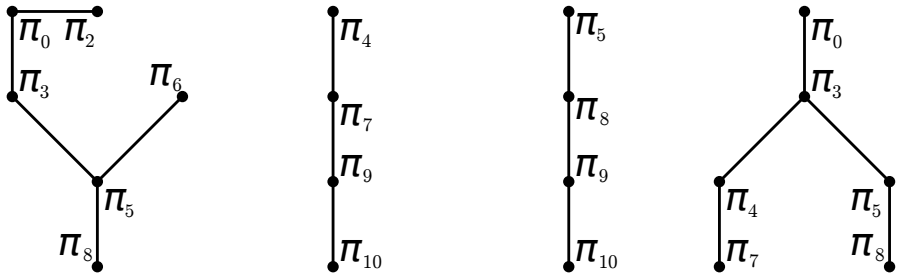
Definition 2.2 (Isomorphism)**.** *A (graph, multigraph) isomorphism from graph* (V, E) *to graph* (Π, K) *is a bijective function* $f\colon V \to \Pi$*, such that the number of edges from* v_0 *to* v_1 *in* (V, E) *is the same as the number of edges from* $f(v_0)$ *to* $f(v_1)$ *in* (Π, K)*.*

In general, we do not distinguish between isomorphic graphs.

For a node $v \in V(G)$, the number $d_G(v) := |\{e \in V(E) \mid v \in e\}|$ is called the *degree* of v in G. For a directed graph $G = (V, E)$ the number $|\{e \in E \mid e \text{ is an edge to } v\}|$ is called the *in-degree* of v, while $|\{e \in E \mid e \text{ is an edge from } v\}|$ is called the *out-degree* of v. In undirected multigraphs, loops are counted twice when calculating the degree of a node (once for each end).

Let H and G be graphs. H is a *subgraph* of G (denoted by $H \subseteq G$), if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. The graph G is then called a *supergraph* of H. For a set $X \subseteq V(G)$, the subgraph of G *induced by* X is the graph $G[X] := (X, \{e \in E(G) \mid \text{both endpoints of } e \text{ are in } X\})$. Often, when speaking about subgraphs, what really is meant are subgraphs up to isomorphism. $G - v := G[V(G) \setminus \{v\}]$.

A (possibly non-simple) *path* P in a (directed or undirected) graph $G = (V, E)$ is a sequence $n_0, \ldots, n_k \in V$ of nodes of G, such that for any $i \in \{0, \ldots, k-1\}$ there is an edge from n_i to n_{i+1} in the graph. The nodes



(a) T_0 (b) T_1 (c) T_2 (d) T_3

Figure 2.2: Trees that are subgraphs of the Graph in Figure 2.1a

$n_1, \ldots, n_{k-1}$ are called *inner* nodes of P, n_0 and n_k are called *endpoints* of P. The path P is called a path *from* n_0 *to* n_k. A path P is called *simple*, if its nodes are pairwise distinct. We will sometimes identify the path P in G with the subgraph $\big(\{n_0, \ldots, n_k\}, \{\{n_0, n_1\}, \ldots, \{n_{k-1}, n_k\}\}\big)$ of G. In this work, all paths are assumed to be simple, unless explicitly stated otherwise. A graph G is called *connected*, if it has at least one node and for any two nodes $x, y \in V(G)$, there is a path from x to y in G. The inclusion-maximal connected induced subgraphs of a graph a called *connected components* of the graph. An undirected connected graph, which either consists of a single node or in which all nodes have degree 2 is called a *cycle*. A *tree* T is a non-empty graph, such that for any two nodes $x, y \in V(T)$ there is exactly one simple path from x to y in T.

The graphs in Figures 2.1a and 2.1b are connected, while the one in Figure 2.1c is not. The trees in Figure 2.2 are subgraphs of the graph in Figure 2.1a.

Definition 2.3 (Clique)**.** *Let $n \in \mathbb{N}$. An n-clique or complete graph on n nodes is the graph $K_n := (V, E)$, such that $|V| = n$ and $E := \{e \mid e \in 2^V, |e| = 2\}$. 3-cliques are also called triangles.*

Usually the term clique is used when referring to subgraphs of another graph, while complete graph is used when talking about a graph by itself.

See Figure 2.3 for the K_5. An *independent set* is a graph that has no edges. Finding cliques of maximal size as subgraphs in graphs and finding independent sets of maximal size as induced subgraphs are NP-hard problems [72].

Definition 2.4 (Grid)**.** *Let $m, n \in \mathbb{N}$, with $m, n \geq 1$. The $(m \times n)$ grid is a graph $H = (V, E)$ with $V := [m] \times [n]$ and $E := \{\{(y, x), (w, z)\} \mid (y, x) \in V, (w, z) \in V, x = z$ and $|y - w| = 1$ or $y = w$ and $|x - z| = 1\}$. In case of a square grid where $m = n$, we say that n is the size of the grid. An edge $\{(y, x), (w, z)\}$ in the grid is called horizontal, if $y = w$, and vertical, if $x = z$.*

See Figure 2.4 for the (5×5) grid.

Besides being a subgraph there are other ways that a graph can "contain" another. An important one is the minor relation:

Definition 2.5 (Model, minor)**.** *Let H be a graph with nodes $\pi_0, \ldots, \pi_{|H|-1}$. A graph M is called model of H, if the node set of M can be partitioned into $|H|$ sets $\Pi_0, \ldots, \Pi_{|H|-1}$, such that:*

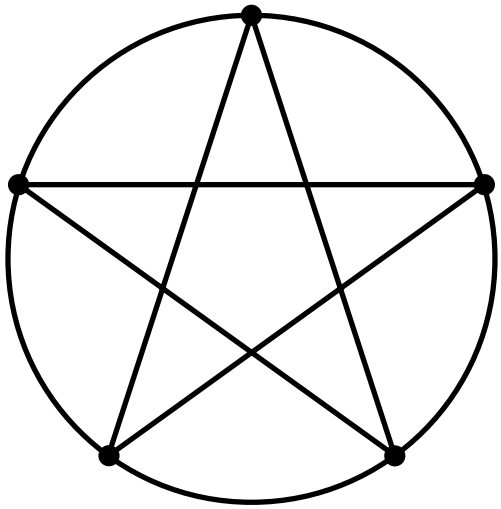


Figure 2.3: K_5

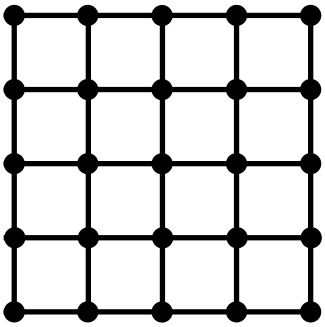


Figure 2.4: (5×5) grid

- *Each* $\Pi_i, i = 0, \ldots, |H| - 1$ *induces a connected subgraph in* M*.*
- *If there is an edge from* π_i *and* π_j *for some* $0 \leq i, j < |H|$ *in* H*, then there is an edge between* Π_i *and* Π_j *in* M *.*

The $\Pi_i, 0 \leq i < |H|$ *are called branch sets of the model.* H *is called a minor of a graph* G *(notation* $H \preccurlyeq G$*), if there is a model of* H *that is a subgraph of* G*.*

There are alternative ways to define minors, including the following one: For a graph G with $e = \{v, w\} \in E(G)$ let G/e denote the graph obtained from G by *contracting* e, i.e. $V(G/e) := (V(G) \setminus e) \cup \{x_e\}$, where x_e is a new node, and

$$E(G/e) := \Big(E(G) \setminus \big\{\{u, u'\} \mid \{u, u'\} \cap e \neq \varnothing\big\}\Big) \cup \big\{\{u, x_e\} \mid \{u, v\} \in E(G) \text{ or } \{u, w\} \in E(G)\big\}.$$

A graph H is a minor of a graph G, if H can be obtained from a subgraph of G by a sequence of edge contractions.

Figure 2.5 shows a graph that contains a K_4 as a minor.

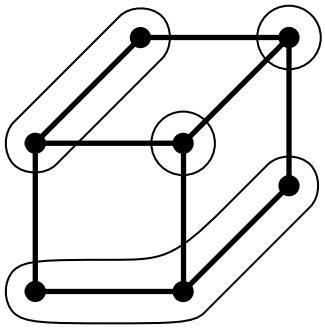
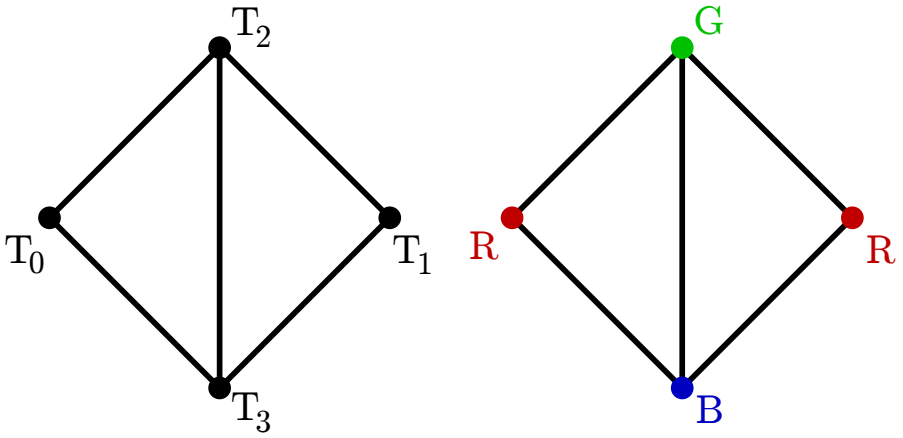


Figure 2.5: A graph containing a K_4 as a minor (branch sets are highlighted)



(a) Intersection Graph of the edge sets of the trees in Figure 2.2

(b) 3-Coloring

Figure 2.6: Coloring an intersection graph

Definition 2.6 (Intersection Graph)**.** *Let S_I be a family of sets for some index set I. The intersection graph of S_I is the undirected graph with node set I in which two nodes are connected by an edge, if the intersection of the corresponding sets is nonempty:*

$$\Big(I, \big\{\{i,j\} \mid S_i \cap S_j \neq \varnothing\big\}\Big).$$

Figure 2.6a shows the intersection graph of the edge sets of the trees in Figure 2.2 (with the nodes labeled by the trees).

Definition 2.7 (Vertex coloring)**.** *A vertex coloring of an undirected graph (V, E) for a set of colors C is a function $f\colon V \to C$, such that for every edge $\{v_0, v_1\} \in E$ the endpoints have different colors, i. e. $f(v_0) \neq f(v_1)$. The graph is called $|C|$-colorable. The chromatic number of a graph G is $\chi(G) :=$ $\min\{k \mid G$ is k-colorable$\}$.*

Figure 2.6b shows a 3-coloring of the Graph in Figure 2.6a. Since we cannot color the graph with fewer than 3 colors due to the triangle subgraphs, the chromatic number is 3. Deciding if a graph is 3-colorable is NP-hard [72].

A *drawing* of a graph G is a representation of G in the Euclidean plane $\mathbb{R}^2$, where nodes are represented by distinct points of $\mathbb{R}^2$ and edges by simple curves joining the points that correspond to their endpoints, such that the interior of every curve representing an edge may not contain points representing vertices. A *planar drawing* (or *embedding*) is a drawing, where the interiors of any two curves representing distinct edges of G are disjoint. A graph G is *planar*, if G has a planar drawing (See [101] for more details on planar graphs). A *plane graph* is a planar graph G together with a fixed embedding of G in $\mathbb{R}^2$. We will identify a plane graph with its image in $\mathbb{R}^2$. A region of the plane bounded by curves representing edges is called a *face* of the plane graph. Note that if $V(E)$ is finite, then G has exactly one infinite face. Once we have fixed the embedding, we will also identify a planar graph with its image in $\mathbb{R}^2$. A graph is *outerplanar*, if it has an embedding in the plane where all nodes are incident to the infinite face.

Figure 2.6a shows an outerplanar graph with three faces.

2.3 Tree-Decompositions

Tree-decompositions [61, 113, 114], have commonly been used to find polynomial algorithms on restricted graph classes for many problems that are computationally hard on general graphs. This includes well known problems such as graph coloring and maximum independent set. Algorithmic meta-theorems state that large classes of problems can be solved efficiently on such graph classes (see below after the definition of tree-decompositions).

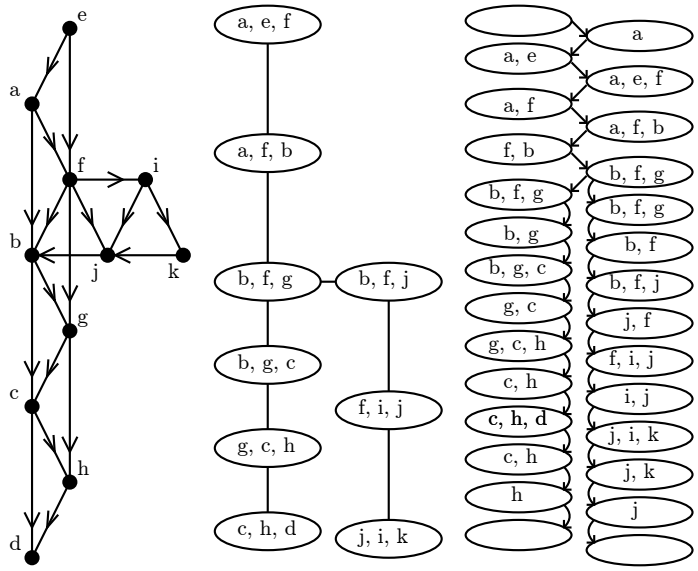
Definition 2.8 (Series-Parallel Graph)**.** *A two-terminal graph (TTG) is a graph with two distinguished nodes, source s and sink t, formed as follows. A graph consisting of a single edge, where one endpoint is the source, and the other endpoint is the sink is a TTG, and so are graphs that can be constructed by applying the following compositions: The parallel composition of two TTGs can be obtained by taking their disjoint union, and then merging the two sources into the new source and merging the two sinks into the new sink. The series composition of two TTGs X and Y can be obtained by taking their disjoint union, and then merging the sink of X with the source of Y. The source of X and the sink of Y become source and sink of the new graph. The graphs that can be obtained from the TTGs by forgetting about the distinction of source and sink are the series-parallel graphs.*

Definition 2.9 (Tree-Decomposition)**.** *A tree-decomposition of a (directed or undirected) graph G is a pair (T, χ), consisting of a tree T and a mapping $\chi\colon V(T) \to 2^{V(G)}$. Such that the following conditions hold:*

- *For each $v \in V(G)$ there exists $t \in V(T)$ with $v \in \chi(t)$.*
- *For each edge $e \in E(G)$ there exists a node $t \in V(T)$ so that both endpoints of e are in $\chi(t)$.*
- *For each $v \in V(G)$ the set $\{t \in V(T) \big| v \in \chi(t)\}$ is connected in T.*

The $\chi(t)$ are called the bags of the tree-decomposition.

If T is a path, (T, χ) is also called a path-decomposition.



(a) Graph (b) Tree-dec. (c) Nice tree-dec.

Figure 2.7: A graph and tree-decompositions of minimum width

The width of a tree-decomposition (T, χ) *is*

$$\mathrm{w}(T, \chi) := \max\left\{ \left|\chi(t)\right| - 1 \;\middle|\; t \in V(T) \right\}.$$

The tree-width of G *is*

$$\mathrm{tw}(G) := \min\left\{ \mathrm{w}(T, \chi) \;\middle|\; (T, \chi) \text{ is a tree-decomposition of } G \right\}.$$

The path-width of G *is*

$$\mathrm{pw}(G) := \min\left\{ \mathrm{w}(T, \chi) \;\middle|\; (T, \chi) \text{ is a path-decomposition of } G \right\}.$$

While the above definition is useful for algorithmic uses, it might be more intuitive to consider k-trees:

Definition 2.10 (k-tree)**.** *Let* $k \in \mathbb{N}$*. The* $(k+1)$*-clique is a* k*-tree. Given a* k*-tree, choosing a* k*-clique and adding a new node connecting it to all nodes of the* k*-clique results in a* k*-tree.*

For $k \in \mathbb{N}$, the graphs of tree-width at most k are the subgraphs of k-trees.

Path-decompositions and path-width were first introduced by Robertson and Seymour [112]. Intuitively, a tree-decomposition is a decomposition of a graph into small subgraphs along a tree and tree-width indicates how tree-like a graph is. Any tree has tree-width at most 1 and any path has path-width at most 1. Series-parallel graphs have tree-width at most 2. The complete graph K_n has tree-width $n-1$ for $n \geq 1$. It is well-known that for $n \geq 2$ the $(n \times n)$ grid has both tree-width and path-width n. Obviously, any graph G satisfies $\mathrm{pw}(G) \geq \mathrm{tw}(G)$. Moreover, if $H \preccurlyeq G$, then $\mathrm{tw}(H) \leq \mathrm{tw}(G)$ and

$\mathrm{pw}(H) \leq \mathrm{pw}(G)$ [114]. For some important applications, it has been found that practically occurring graphs tend to have low tree-width, e. g. in chemistry [140] and compiler construction [23, 58] (more information on bounded tree-width in compiler construction including our own results can be found in Chapter 7).

Figure 2.7b gives an example of a tree-decomposition of minimum width for the graph in Figure 2.7a.

Definition 2.11 (Bounded tree-width)**.** *A class C of graphs has bounded tree-width, if there is a $k \in \mathbb{N}$, such that each graph in C has tree-with at most k.*

For graphs from such classes, tree decompositions can be found in linear time:

Theorem 2.12 (Bodlaender [17])**.** *For all $k \in \mathbb{N}$, there exists a linear time algorithm, that tests whether a given graph G has tree-width at most k, and if so, outputs a tree-decomposition of G with width at most k.*

On graphs of bounded tree-width, any problem that can be formalized in monadic second-order logic can be decided in linear time [31]. Monadic second-order logic is decidable on graphs of bounded tree-width [9].

Unfortunately, the resulting algorithms are often impractical due to huge constants in run-time and prohibitively hard to implement [47, 120]. However, there also are more practical ways of obtaining tree-decompositions [132, 20, 19], which for many practical applications are sufficient, even though they do not result in tree-decompositions of minimum width on general graphs.

Often proofs and algorithms are easier to describe, understand and implement when using nice tree-decompositions. Nice tree-decompositions have been used at least since 1994 [77], with minor variation in the definitions, sometimes relaxing conditions where they are not needed. We use the following definition, in which the tree in the tree-decomposition is directed. A *root* in a directed tree is a unique node r, such that for every node n in the tree there is a directed path from r to n. A directed tree that has a root is called *oriented*. If there is an edge (i, j) in an oriented tree, then the node j is called a *child* of i.

Definition 2.13 (Nice Tree-Decomposition)**.** *A tree-decomposition (T, χ) of a graph G is called nice, if*

- *T is oriented, with root t, $\chi(t) = \varnothing$.*
- *Each node i of T is of one of the following types:*
  - *Leaf node, has no children, $\chi(i) = \varnothing$.*
  - *Introduce node, has one child $j, \chi(j) \subsetneq \chi(i), |\chi(i) \setminus \chi(j)| = 1$.*
  - *Forget node, has one child $j, \chi(j) \supsetneq \chi(i), |\chi(j) \setminus \chi(i)| = 1$.*
  - *Join node, has two children $j_1, j_2, \chi(i) = \chi(j_1) = \chi(j_2)$.*

The terminology is inspired by a bottom-up approach, which is how most algorithms on tree-decompositions work. Given a tree-decomposition, a nice tree-decomposition of the same width can be found easily and in linear time. Figure 2.7c shows a nice tree-decomposition for the graph in Figure 2.7a.

Besides tree-width and path-width there are other known graph width parameters. In particular, cliques have a quite simple structure, but high tree-width. This simplicity is captured by the *clique-width* $\mathrm{cw}(G)$ of a graph G, for which $\mathrm{cw}(G) \leq 2^{\mathrm{tw}(G)} + 1$ holds ($\mathrm{cw}(G) \leq 2^{2\,\mathrm{tw}(G)} + 2$ for directed graphs)[32].

2.4 Coloring using tree-decompositions

This section presents a simple example application of tree-decompositions. The notation and proof structure used are meant to prepare the reader for Chapters 8, 9 and 10.

An algorithmic application example for the use of tree-decompositions is deciding k-colorability: Given a graph G of bounded tree-width, decide if G is k-colorable. As noted in the previous section we can obtain a nice tree-decomposition (T, χ) of minimum width of G in linear time (note that the linear time also implies that $V(T)$ is linear in $V(G)$). Then, using the nice tree-decomposition of G of width $\operatorname{tw}(G)$, we can decide k-colorability in time $|V(T)|k^{\operatorname{tw}(G)+1}\operatorname{tw}(G)$: For $i \in V(T)$, let T_i be the set of nodes of $V(G)$ which are contained in bags in the subtree of T rooted at i. For each $i \in V(T)$ we calculate the set $S(i)$ of $[k]$-colorings of $\chi(i)$ that can be extended to a $[k]$-coloring of the subgraph of G induced by T_i. Then G is k-colorable exactly if $S(r) \neq \varnothing$ for the root r of T.

To find the sets $S(i)$, we define sets $s(i)$ and then show that $S(i) = s(i)$ and that $s(r)$ can be calculated in time $|V(T)|k^{\operatorname{tw}(G)+1}\operatorname{tw}(G)$. We define s inductively depending on the type of i:

- Leaf: $s(i) := \{f\colon \varnothing \to [k]\}$, the set containing the empty function.
- Introduce node with child j, $\chi(i) \setminus \chi(j) = \{v\}$:

$$s(i) := \Big\{f\colon \chi(i) \to [k] \;\Big|\; f|_{\chi(j)} \in s(j) \text{ and } f(v) \neq f(u) \text{ for all } \{u,v\} \in E(G), u \in \chi(j)\Big\}$$

- Forget node with child j: $s(i) := \Big\{f|_{\chi(i)} \;\Big|\; f \in s(j)\Big\}$.
- Join node with children j_1 and j_2: $s(i) := s(j_1) \cap s(j_2)$

We show $S = s$ by induction: Assume $S(j) = s(j)$ for all children j of a node i. Then $S(i) = s(i)$:

Case 1: i is a leaf. $s(i) = \{f\colon \varnothing \to [k]\} = S(i)$: The empty coloring is the only possibility to color an empty set $\chi(i)$ of nodes.

Case 2: i is an introduce node with child j.

$$s(i) =$$
$$\Big\{f \;\Big|\; f|_{\chi(j)} \in s(j) \text{ and } f(v) \neq f(u) \text{ for all } \{u,v\} \in E(G) \cap \big(\chi(j) \times V(G)\big)\Big\} \overset{\text{ind.}}{=}$$
$$\Big\{f \;\Big|\; f|_{\chi(j)} \in S(j) \text{ and } f(v) \neq f(u) \text{ for all } \{u,v\} \in E(G) \cap \big(\chi(j) \times V(G)\big)\Big\} =$$
$$S(i).$$

Since there are no edges between $(T_i \setminus \chi(i))$ and v, we only need to ensure that the endpoints of edges between v and other nodes in $\chi(i)$ are colored differently.

Case 3: i is a forget node with child j. $s(i) = \{f|_{\chi(i)} \mid f \in s(j)\} = \{f|_{\chi(i)} \mid f \in S(j)\} = S(i)$: Since $T_i = T_j$, each coloring of $\chi(i)$ that can be extended to one on T_i is a restriction of one on $\chi(j)$ that can be extended to one on T_j.

Case 4: i is a join node with children j_1 and j_2. $s(i) = s(j_1) \cap s(j_2) = S(j_1) \cap S(j_2) = S(i)$: Since there are no edges between $(T_{j_1} \setminus \chi(j_1))$ and $(T_{j_2} \setminus \chi(j_2))$ in G, any coloring of $\chi(i) = \chi(j_1) = \chi(j_2)$ that can be extended to T_{j_1} and to T_{j_2} can be extended to $T_i = T_{j_1} \cup T_{j_2}$.

For showing that s can be computed in time $O\left(|V(T)| k^{\mathrm{tw}(G)}\, \mathrm{tw}(G)\right)$, it suffices to show that we can calculate each $s(i)$ in time $O\left(k^{\mathrm{tw}(G)+1}\, \mathrm{tw}(G)\right)$:

Case 1: i is a leaf. $|s(i)| = 1$ And for each element of $s(i)$ we use constant time.

Case 2: i is an introduce node. $|s(i)| \leq k^{|\chi(i)|} \leq k^{\mathrm{tw}(G)+1}$. And for each element of $s(i)$ we need to check at most $|\chi(i)| - 1 \leq \mathrm{tw}(G)$ edges.

Case 3: i is a forget node with child j. $|s(i)| \leq k^{|\chi(i)|} \leq k^{\mathrm{tw}(G)}$. And for each element of $s(i)$ there are at most k different $f \in s(j)$ to be considered.

Case 4: i is a join node. $|s(i)| \leq k^{|\chi(i)|} \leq k^{\mathrm{tw}(G)+1}$. And for each element of $s(i)$ we use constant time.

Chapter 3

Compilers

This chapter gives a gentle high-level introduction to compilers, with a focus on optimization. Readers interested in compilers beyond what is presented here are referred to standard works on compilers [6, 59].

3.1 Role and Structure

Compilers translate between computer languages. E.g. from XeLaTeX to pdf. In the classical setting they translate from a higher level programming language, such as C to a lower level language such as assembler code. They are usually accompanied by other, simpler tools, such as preprocessor, assembler and linker (Figure 3.1), even though recently, there is a trend to integrate these into the compiler.

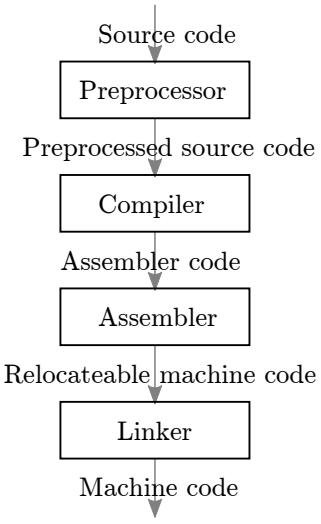


Figure 3.1: Toolchain

Preprocessors perform relatively simple tasks, such as pasting other files at specified places (e. g. the #include directive in C, as in Figure 3.2a to 3.2b)

```
#include "g.h"

int f(int x)
{
        return(g(3) + 1);
}
```

(a) C code

```
int g(int);

int f(int x)
{
        return(g(3) + 1);
}
```

(b) Preprocessed C code

```
_f:
        ld      hl,#0x0003      21 03 00
        push    hl              E5
        call    _g              CDr00r00
        pop     af              F1
        inc     hl              23
        ret                     C9
```

(c) Assembler code

(d) Relocatable machine code

Figure 3.2: Code in the toolchain (example using SDCC [41] targeting the Z80)

and expanding macros. Instructions in assembler code map one-to-one to instructions in machine code, but are more human-readable, as can be seen in Figure 3.2c to 3.2d. The linker links multiple files containing machine code together, and resolves external memory symbols (in the example in Figure 3.2, the linker would get the definition of function g() from another file and the r00r00 in Figure 3.2d would be replaced by the location of g() in memory).

Compilers typically have multiple phases (Figure 3.3). The input to the compiler is preprocessed source code, as a character stream, from which the *lexical analyzer* produces a token stream. The individual tokens are e. g. identifiers or keywords of the input language. The lexical analyzer is relatively simple, the theoretical foundations are covered by regular languages. From the token stream, the *syntax analyzer* produces an abstract syntax tree (AST). The syntax analyzer typically deals with context-free and sometimes context-sensitive languages. The *semantic analyzer* works on the syntax tree; typically the most important task it performs is type checking. From the abstract syntax tree, the intermediate code generator produces intermediate code (iCode). The iCode is typically mostly three-address-code, a sequence of operations that each have at most one destination operand and two source operands. *Optimizations* transform the iCode and annotate it with further information. Optimizations are a complex topic; two very important optimizations are redundancy elimination and the most important one, register allocation. Programs tend to contains redundancies, i. e. computations are done more often or at more places than necessary; *redundancy elimination* improves the program by doing those computations less often or in fewer places, which typically requires doing them in different places in the program than before. We cover register allocation in Section 3.2. From the iCode, the *code generator* generates assembler code. This assembler code is further transformed by a *peephole optimizer*, which improves the assembler code by locally substituting assembler code sequences.

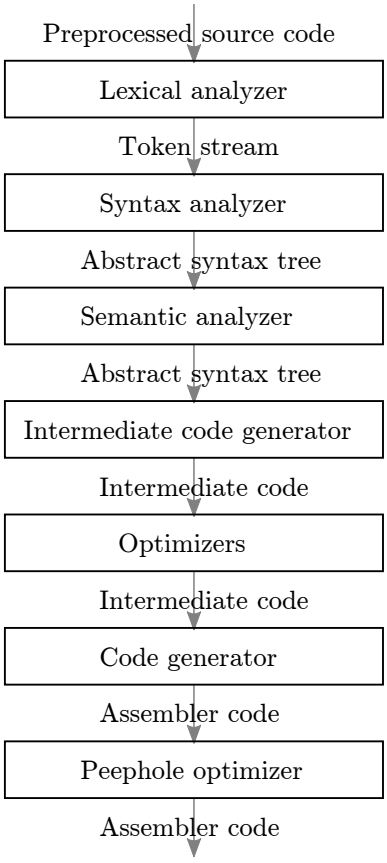


Figure 3.3: Compiler Phases

By creating a node for every instruction in code and adding an edge for every pair of instructions where one can be executed directly after the other, we get the *control-flow graph* (CFG). Often, CFGs are directed, but for some purposes (such as the following Section 3.2), undirected graphs are sufficient. Also, CFGs are often annotated with further information on the nodes or edges. We can create CFGs for all the types of code in Figure 3.3. Most algorithms in compilers dealing with CFGs do so with CFGs of the iCode. For simplicity, in this introduction we will use CFGs of source code in examples. Figure 3.4b shows a CFG for the source code in Figure 3.4a.

3.2 Register Allocation

Compilers map program variables to physical storage space in a computer. The compiler can use registers and memory for storage. The problem of deciding which variables to store into which registers or into memory is called *register allocation*. Variables allocated to memory instead of to registers are said to be *spilt*. Typically, there is much more storage available in memory than in registers, but accessing registers is faster than accessing main memory, and

the speed gap is widening. Instructions accessing registers are shorter than instructions accessing main memory. This makes register allocation typically the most important optimization in a compiler. There are further aspects to register allocation, such as *register aliasing* (i. e. multiple register names mapping to the same physical hardware and thus not being able to be used at the same time) and *register preferences* (e. g. due to certain instructions taking a different amount of time depending on which registers the operands reside in) that have to be handled to generate good code. *Coalescing* (eliminating moves by assigning variables to the same registers, if they do not interfere, but are related by a copy instruction) is another aspect, where register allocation can have a significant impact on code size and speed. Register allocation is a hard problem (See Chapter 12 and Section 10.5 for details).

Program variables have *live-ranges*. The live-ranges are connected subgraphs of the CFG where the variable is *alive*, i. e. its value needs to be in storage somewhere, since it might be read later on. We will usually, in particular when considering their intersections identify live-ranges with their edge sets. Figure 3.5a shows live-ranges for variables k and delta from Figure 3.4.

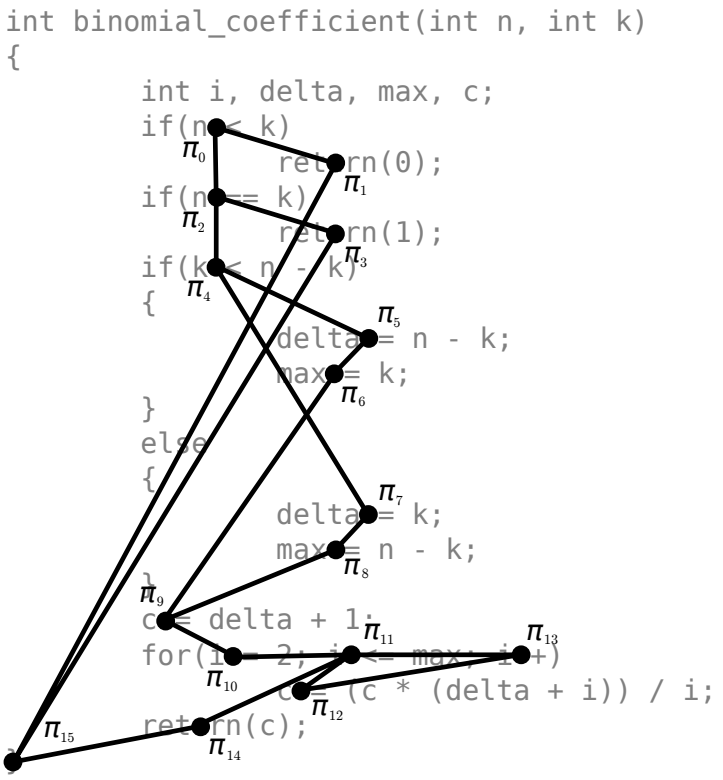
Definition 3.1 (Conflict Graph)**.** *Let V be the set of variables of a program. The conflict graph of the program is the intersection graph of their live-ranges.*

Figure 3.5b shows the conflict graph for the program from Figure 3.4, with the k and delta marked in the same color as their live-ranges in Figure 3.5a.

Assuming a simplified scenario, where any variable can go into any register and there is no register aliasing, the problem of register allocation to $r \in \mathbb{N}$ registers then becomes the problem of finding a suitable r-colorable induced subgraph of the conflict graph. Simplifying even further by ignoring coalescing and register preferences and assuming that allocating a variable to registers gives the same benefit, no matter which variable it is, the problem of register allocation to $r \in \mathbb{N}$ registers then becomes the problem of finding a maximal r-colorable induced subgraph of the conflict graph. Since in general, even this simplified problem is hard [27, 72], heuristics approaches are used. The classic one is Chaitin's algorithm, which is based on the degree of nodes in the conflict graph (the version presented in Figure 3.6 is somewhat simplified compared to Chaitin's original work) [27, 26].

```
int binomial_coefficient(int n, int k)
{
        int i, delta, max, c;
        if(n < k)
                return(0);
        if(n == k)
                return(1);
        if(k < n - k)
        {
                delta = n - k;
                max = k;
        }
        else
        {
                delta = k;
                max = n - k;
        }
        c = delta + 1;
        for(i = 2; i <= max; i++)
                c = (c * (delta + i)) / i;
        return(c);
}
```

(a) Source code

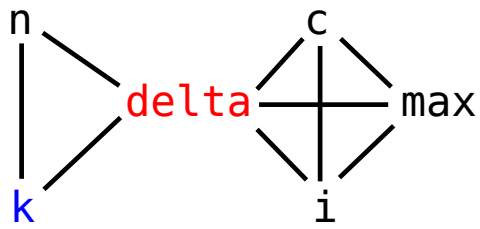


(b) Control-flow graph

Figure 3.4: Control-flow graph of source code

```
int binomial_coefficient(int n, int k)
{
        int i, delta, max, c;
        if(n < k)
                return(0);
        if(n == k)
                return(1);
        if(k < n - k)
        {
                delta = n - k;
                max = k;
        }
        else
        {
                delta = k;
                max = n - k;
        }
        c = delta + 1;
        for(i = 2; i <= max; i++)
                c = (c * (delta + i)) / i;
        return(c);
}
```

(a) Live-ranges



(b) Conflict graph

Figure 3.5: Some live-ranges and the conflict graph corresponding to Figure 3.4

```
Chaitin(graph G)
{
  if(G is the empty graph)
    return;
  Let v be a node of smallest degree in G;
  Color G[V(G)] \ v] by Chaitin(G[V(G)] \ v]);
  Extend the coloring to G by coloring v using the smallest color
    not used by it neighbors. If no such color is available, spill v;
}
```

Figure 3.6: Chaitin's algorithm

Part II

Disjoint Paths Problem

Chapter 4

The Disjoint Paths Problem[1]

4.1 Introduction

One of the famous classical problems in graph algorithms is the Disjoint-Paths Problem (DPP): *Given a graph G, and k pairs of terminals, (s_1, t_1), $\ldots, (s_k, t_k)$, decide whether G contains k vertex-disjoint paths $P_1, \ldots, P_k$ where P_i has endpoints s_i and t_i, $i = 1, \ldots, k$* . In addition to its numerous applications in areas such as network routing and VLSI layout, this problem has been the catalyst for extensive research in algorithms and combinatorics [127]. DPP is NP-complete, along with its edge-disjoint or directed variants, even when the input graph is planar [72, 135, 100, 81, 96]. It can be solved in linear time when the input graph has bounded tree-width [121]; on the other hand, it is still NP-hard even when the clique-width is bounded [56]. Also, the edge-disjoint variant is NP-complete for series-parallel graphs, a subclass of the graphs of tree-width at most 2 [106].

Robertson and Seymour showed that the DPP can be solved in time $g(k) \cdot |V(G)|^3$ for some computable function g, i. e. the problem is fixed-parameter tractable (and, in particular, solvable in polynomial time for fixed k). The runtime was later improved from $O(|V(G)|^3)$ to $O(|V(G)|^2)$ for fixed k [65]. The Robertson-Seymour algorithm is the central algorithmic result of the Graph Minors series of papers, one of the deepest and most influential bodies of work in graph theory.

The basis of the algorithm ((10.5) in [116]) is the *irrelevant-vertex technique* which can be summarized very roughly as follows. As long as the input graph G violates certain structural conditions, it is possible to find an *irrelevant vertex* v: every collection of paths certifying a solution to the problem can be rerouted to an *equivalent* one, that links the same pairs of terminals, but in which the new paths avoid v. One then iteratively removes such irrelevant vertices until the structural conditions are met. By that point the graph has been simplified

[1] This chapter and the following Chapters 5 and 6 are based on joint work with Isolde Adler, Stavros G. Kolliopoulos, Daniel Lokshtanov, Saket Saurabh and Dimitrios M. Thilikos; a first version of the results was presented at ICALP [2]. Recently, more refined versions with full proofs have been submitted [5, 3] for publication.

enough so that the problem can be attacked via dynamic programming. The algorithm by Roberson and Seymour [116] uses two structural conditions: (1) G excludes a clique, whose size depends on k, as a minor and (2) G has tree-width bounded by some function of k. If condition (1) is not satisfied, it is relatively easy to find the irrelevant vertex using the clique minor. For (2), the aim is to prove that in graphs without big clique-minors and with tree-width at least $f(k)$ there is always a solution-irrelevant vertex. This is the most complicated part of the proof and it was postponed until the later papers in the series [117, 118]. The complicated proofs also imply an *immense* dependence of the running time on the parameter k. This puts the algorithm outside the realm of feasibility even for elementary values of k.

The ideas above were powerful enough to be applicable also to problems outside the context of the Graph Minors series. During the last decade, they have been applied to many other combinatorial problems and now they constitute a basic paradigm in parametrized algorithm design (see, e.g., [35, 36, 52, 73, 74, 80]). However, in most applications, the need for overcoming the high parameter dependence emerging from the structural theorems of the Graph Minors series, especially those in [117, 118], remains imperative. Hence two natural directions of research are: (1) Simplify parts of the original proof for the general case to get upper bounds on f and also find good lower bounds for it or (2) focus on specific graph classes that may admit proofs with better parameter dependence.

An important step in the first direction was taken recently by Kawarabayashi and Wollan [75] who gave an easier and shorter proof of the results in [117, 118]. While the parameter dependence of the new proof is certainly much better than the previous, immense, function, it is still huge.

In Chapter 6 we present a lower bound: $f(k) \geq 2^k$. We prove that this bound holds even for planar graphs.

In Chapter 5 we obtain an upper bound for the class of planar graphs: $f(k) \leq (72k \cdot 2^k - 72 \cdot 2^k + 18)\lceil\sqrt{2k+1}\rceil$. This also results in the fastest known fpt algorithm for the DPP on planar graphs.

4.2 Irrelevant vertices and the Disjoint Paths Problem

Definition 4.1 (Disjoint Paths Problem (DPP))**.** *Given a graph G and k pairs of terminals $(s_1, t_1) \in V(G)^2, \ldots, (s_k, t_k) \in V(G)^2$, the* Disjoint Paths Problem *is the problem of deciding whether G contains k vertex-disjoint paths $P_1, \ldots, P_k$ such that P_i connects s_i to t_i, (for $i = 1, \ldots, k$). If such paths $P_1, \ldots P_k$ exist, we refer to them as a solution. We denote an instance of* DPP *by $G, (s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$.*

Let $G, (s_1, t_1), \ldots, (s_k, t_k)$ be an instance of DPP. A non-terminal vertex $v \in V(G)$ is *irrelevant*, if $G, (s_1, t_1), \ldots, (s_k, t_k)$ has a solution if and only if $G - v, (s_1, t_1), \ldots, (s_k, t_k)$ has a solution.

Theorem 4.2 (Robertson and Seymour [118])**.** *There is a function $f\colon \mathbb{N} \to \mathbb{N}$ such that if $\mathrm{tw}(G) \geq f(k)$, then $G, (s_1, t_1), \ldots, (s_k, t_k)$ has an irrelevant vertex (for any choice of terminals $(s_1, t_1), \ldots, (s_k, t_k)$ in G).*

Definition 4.3 (Linkage)**.** *A linkage in a graph G is a subgraph $L \subseteq G$, such that each connected component of L is a path. The endpoints of a linkage L are the endpoints of these paths, and the pattern of L is the matching on the endpoints induced by the paths, i. e. the pattern is the set*

$$\big\{\{s,t\} \;\big|\; L \text{ has a connected component that is a path from } s \text{ to } t\big\}.$$

A linkage L in a graph G is a vital linkage in G, if $V(L) = V(G)$ and there is no other linkage $L' \neq L$ in G with the same pattern as L.

Theorem 4.4 (Robertson and Seymour [118])**.** *There are functions $g, h\colon \mathbb{N} \to \mathbb{N}$ such that if a graph G has a vital linkage with k components then* $\mathrm{tw}(G) \leq g(k)$ *and* $\mathrm{pw}(G) \leq h(k)$.

```
I(graph G)
{
  while(we can find an irrelevant vertex v in G)
    remove v from G;
  Compute a tree-decomposition of small width of G;
  Decide the DPP on G using the tree-decomposition;
}
```

Figure 4.1: Solving the DPP with irrelevant vertices

Algorithm 4.1 is the irrelevant vertex technique for solving the DPP. The runtime of this algorithm depends crucially on the f from Theorem 4.2, since deciding the DPP using tree-decompositions takes exponential time in $\mathrm{tw}(G)$. For planar graphs, our Theorem 5.26 allows us to execute the `while`-loop in time $O(|V(G)|^2)$, and guarantees that after the loop, $tw(G) < f(G) \leq (72k \cdot 2^k - 72 \cdot 2^k + 18)\lceil\sqrt{2k+1}\rceil$. Our Theorem 6.1 shows that there are planar graphs G, such that after the loop $\mathrm{tw}(G)$ will always be at least $2^k - 1$. This means that that Algorithm 4.1 cannot be made substantially faster compared to what we achieved, and any faster algorithm for the DPP would require different techniques.

Chapter 5

Irrelevant vertices in Planar Graphs[1]

In this chapter we offer a solid advance in improving the application of the irrelevant vertex-technique on planar graphs. For directed planar graphs, there was an XP algorithm [126]. Partially building on our techniques and results from this chapter, an fpt algorithm has been found for directed planar graphs [34]. We prove that, for planar graphs, the function $f(k)$ from Theorem 4.2 is singly exponential. In particular we prove the following result.

Theorem 5.1. *There is a constant c such that every n-vertex planar graph G with tree-width at least c^k contains a vertex v such that every solution to* DPP *with input G and k pairs of terminals can be replaced by an equivalent one avoiding v.*

Given the above result, our Theorem 5.26 shows how to reduce, in $O(n^2)$ time, an instance of DPP to an an equivalent one whose graph G' has tree-width $2^{O(k)}$. Then, using dynamic programming, a solution, if one exists, can be found in $k^{O(\mathrm{tw}(G'))} \cdot n = 2^{2^{O(k)}} \cdot n$ steps. In particular, $f(k) \leq (72k \cdot 2^k - 72 \cdot 2^k + 18)\lceil\sqrt{2k+1}\rceil$ holds for the f from Theorem 4.2.

The proof of Theorem 5.1 deviates significantly from those in [117, 118, 75]. It is self-contained and exploits extensively the combinatorics of planar graphs.

The basic results were first presented at ICALP [2] (with $f(k) \leq (72k \cdot 2^k)\sqrt{2k}$). A journal version, which has been submitted [3] for publication, improves this to $f(k) \leq 26k \cdot 2^k\sqrt{2k}$. This chapter differs significantly from the joint work journal version. The proofs here are intuitive, self-contained and elementary, obtaining $f(k) \leq (72k \cdot 2^k - 72 \cdot 2^k + 18)\lceil\sqrt{2k+1}\rceil$.

The main result of this chapter is Theorem 5.26 stating that there is an $O(n^2)$-step algorithm that, given an instance G of DPP of tree-width $2^{\Omega(k)}$, can find a set of irrelevant vertices whose removal from G creates an equivalent instance of tree-width $2^{O(k)}$.

[1] Based on joint work with Isolde Adler, Stavros G. Kolliopoulos, Daniel Lokshtanov, Saket Saurabh and Dimitrios M. Thilikos, which has been presented at ICALP 2011 [2]. A journal version has been submitted [3] for publication.

5.1 Concentric Cycles and Segments

A *subdivided grid* is a graph obtained from a grid by replacing some edges of the grid by pairwise internally vertex disjoint paths of length at least one.

We use the fact that a subdivided grid has a unique embedding in the plane (up to homeomorphism). For (1×1) grids and subdivided (2×2) grids this is clear, and for $(n \times n)$ grids with $n \geq 2$ this follows from Tutte's Theorem [133] stating that 3-connected graphs have unique embeddings in the plane (up to homeomorphism). This implies that subdivisions of $(n \times n)$ grids have unique embeddings as well. The *perimeter* of a subdivided grid H is the cycle in H that is incident to the outer face (in every planar embedding of H). For an edge (u, v) in a directed graph we say that u is the *tail* of (u, v), $u = \text{tail}(u, v)$, and v is the *head* of (u, v), $v = \text{head}(u, v)$.

Definition 5.2 (Tight concentric cycles)**.** *Let G be a plane graph and let $C_0, \ldots, C_n$ be a sequence of cycles in G such that each cycle bounds a closed disc D_i in the plane. We call $C_0, \ldots, C_n$ concentric, if for all $i \in \{0, \ldots, n-1\}$, the cycle C_i is contained in the interior of D_{i+1}. The concentric cycles $C_0, \ldots, C_n$ are tight, if, in addition, C_0 is a single vertex and for every $i \in \{0, \ldots, n-1\}$, $D_{i+1} \setminus D_i$ does not contain a cycle C bounding a disc D in the plane with $D_{i+1} \supsetneq D \supseteq D_i$.*

Remark 5.3. *Let G be a plane graph and let $C_0, \ldots, C_n$ be tight concentric cycles in G bounding closed discs $D_1, \ldots, D_n$, respectively, in the plane. Let P be a path connecting vertices u and v with $u, v \notin D_n$. If a vertex of P is contained in the interior of D_i (i.e. in $D_i \setminus C_i$), then P has a vertex on C_{i-1}.*

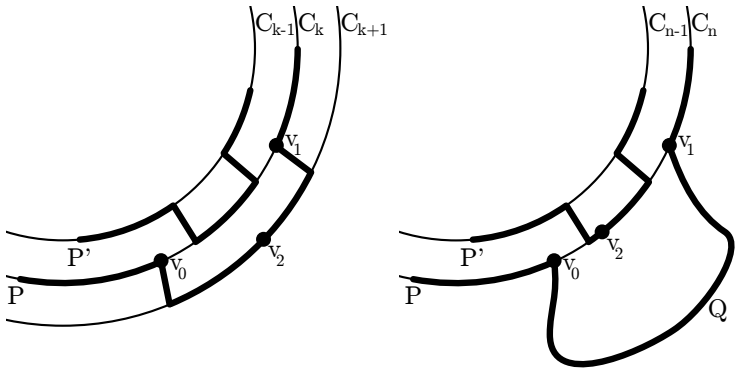
Lemma 5.4. *Let G be a planar graph containing a $(x \times x)$-grid Γ as a minor. Let $G, (s_t, t_1), \ldots, (s_k, t_k)$ be an instance of* DPP*. If $x \geq 2 \cdot (n+1) \cdot \lceil 2k+1 \rceil$, then G contains a sequence of $n+1$ tight concentric cycles $C = C_0, \ldots, C_n$ such that none of the vertices in the closed interior of C_n is a terminal.*

Proof. As such a grid can be partitioned into $2k+1$ vertex disjoint $(2(n+1) \times 2(n+1))$-grids, G contains a $(2(n+1) \times 2(n+1))$-grid Γ' that does not contain any terminals. Notice that $V(\Gamma')$ can be partitioned into $n+1$ sets $V_0, \ldots, V_n$, corresponding to $n+1$ concentric cycles of Γ' that are arranged from inside to outside, i. e., V_0 contains the centers of Γ'. In each $G[(V_i)]$, pick a cycle C_i meeting the models of all vertices of V_i. Because G is a plane graph, it is easy to verify that $C_0, \ldots, C_n$ is a sequence of concentric cycles in G. q. e. d.

Definition 5.5 (Segment)**.** *Let G be a plane graph and let C be a cycle in G bounding a closed disc D in the plane. For a path P in G we say that a subpath P_0 of P is a D-*segment *of P, if P_0 is a non-empty path obtained by intersecting P with D.*

For a linkage $\mathcal{P}$ in G we say that a path P_0 is a D-segment of $\mathcal{P}$, if P_0 is a D-segment of some component P of $\mathcal{P}$.

Let G be a plane graph containing a sequence $C_0, \ldots, C_n$ of concentric cycles bounding closed discs $D_0, \ldots, D_n$, respectively, in the plane. Let G contain a DPP with terminals outside the interior of D_n, and assume that the DPP has a solution $\mathcal{P}$. We will often refer to the D_n-segments of $\mathcal{P}$ simply as the *segments* of $\mathcal{P}$.



(a) Lemma 5.8 (b) Lemma 5.9

Figure 5.1: Constructions in proofs of Lemmata 5.8 and 5.9

Definition 5.6 (I)**.** *Let G be a plane graph containing a sequence $C_0, \ldots, C_n$ of concentric cycles bounding closed discs $D_0, \ldots, D_n$, respectively, in the plane. Let P be some path with endpoints in D_n that has vertices outside of D_n. Then $I(P)$ is the subgraph that has the boundary $C_n \cup P$ and contains neither the infinite face nor a vertex in the interior of D_n.*

Definition 5.7 (Cheap solution)**.** *Let G be a plane graph containing a sequence $C_0, \ldots, C_n$ of concentric cycles bounding closed discs $D_0, \ldots, D_n$, respectively, in the plane. Let $c : \{H \subseteq G \mid H \text{ Graph}\} \to \mathbb{N}$ be the function given by*

$$H \mapsto \left|\left\{xy \in E(H) \;\middle|\; \exists i \in \{0, \ldots, n\} \text{ with } x \in C_i \not\ni y\right\}\right|.$$

Let G contain a DPP*. A solution $\mathcal{P}$ of the* DPP *is called* cheap*, if there is no other solution $\mathcal{Q}$ of the* DPP*, such that $c(\mathcal{P}) > c(\mathcal{Q})$.*

Lemma 5.8. *Let G be a plane graph with a* DPP*. Let G contain a sequence $C_0, \ldots, C_n$ of tight concentric cycles bounding closed discs $D_0, \ldots, D_n$, respectively, in the plane, such that no terminal of the DPP lies inside D_n. Let $\mathcal{P}$ be a cheap solution of the* DPP*.*

Then there is no segment P of $\mathcal{P}$ with vertices $\ldots, v_0, \ldots, v_1, \ldots, v_2, \ldots$ where v_0 and v_2 are vertices of C_k, and v_1 is a vertex of C_j, for any $n \geq j > k \geq 0$.

Proof. Assume that such a segment exists. Then there is an innermost one, i.e. one for which k is minimal. Let P be such an innermost segment. P has vertices $\ldots, v_0, \ldots, v_2, \ldots, v_1, \ldots$ where v_3 is in C_{k+1}, since P has to go through C_{k+1} to reach C_j from C_k. Let v_0 and v_1 be the vertices nearest to v_2 in P, such that v_0 and v_1 are part of C_k. The part of P between v_0 and v_1 cannot be replaced by the part of C_k that connects them, since $\mathcal{P}$ is cheap: It contains vertices and these vertices are used in the solution. This means that vertices in C_k between v_0 and v_1 are part of some other segment P'. However to reach these vertices P' has to share a vertex with C_{k-1} according to Remark 5.3, since there is no terminal in D_n and P blocks all other ways to leave (Figure 5.1a). This contradicts the choice of P. q. e. d.

Lemma 5.9. *Let G be a plane graph with a* DPP*. Let G contain a sequence $C_0, \dots, C_n$ of tight concentric cycles bounding closed discs $D_0, \dots, D_n$, respectively, in the plane, such that no terminal of the* DPP *lies inside D_n. Let $\mathcal{P}$ be a cheap solution of the* DPP*. Let P be a path of the solution that contains two segments. Let Q be a subpath of P, in between two segments, such that Q shares exactly its ends with these two segments and $Q \cap C_{n-1} = \varnothing$. Let H be a subgraph of $G \setminus D_{n-1}$, such that the boundary of H is the union of Q and a part of C_n that connects the endpoints of Q. Then there is a terminal in $H \setminus (Q \cup C_n)$.*

In particular, there is a terminal in $I(Q)$, if $I(Q)$ is defined.

Proof. Assume that there is no such terminal. Let P be an innermost such path (i.e. there is no other such path that contains a vertex in H). Let v_0 and v_1 be the endpoints of Q. Q cannot be replaced by the part of C_n that connects v_0 to v_1, since $\mathcal{P}$ is cheap: The part contains a vertex v_2, and v_2 is used in the solution. This means that v_2 is used by some segment P'. Since no terminal can be reached from v_2 without using a vertex in C_{n-1} according to Remark 5.3 (Q together with v_0 and v_1 block any way out). P' contains vertices in C_{n-1} before and after v_2, and v_2 is part of C_n (Figure 5.1b). However, this is not possible according to the previous Lemma 5.8. q. e. d.

5.2 Bounding the number of segment types

In this section we define a notion of segment types and obtain an upper bound on the number of segment types.

Definition 5.10 (Segment Type)**.** *Let G be a plane graph containing a sequence $C_0, \dots, C_n$ of concentric cycles bounding closed discs $D_0, \dots, D_n$, respectively, in the plane. Let $G, (s_t, t_1), \dots, (s_k, t_k)$ be an instance of* DPP *such that no terminal is contained in D_n.*

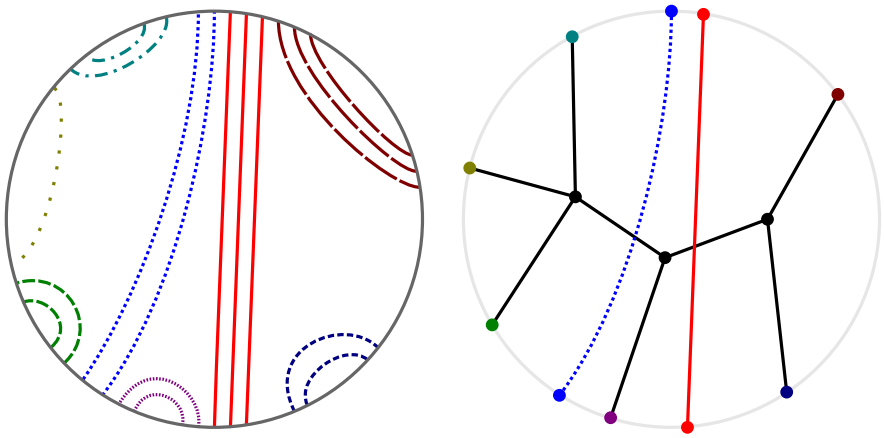
Let $\mathcal{P}$ be a solution to this instance. Let R and S be two D_n-segments. Let Q and Q' be the two paths on C_n connecting an endpoint of R with an endpoint of S and passing through no other endpoint of R or S. We say that R and S are equivalent, and we write $R \parallel S$, if no D_n-segment of $\mathcal{P}$ has both endpoints on Q and no D_n-segment has both endpoints on Q'. A type of D_n-segments is an equivalence class of D_n-segments under the relation $\parallel$.

The idea of segment types is illustrated by Figure 5.2a.

Definition 5.11 (Segment graph)**.** *Let G be a plane graph containing a sequence $C_0, \dots, C_n$ of concentric cycles bounding closed discs $D_0, \dots, D_n$, respectively, in the plane. Let $G, (s_t, t_1), \dots, (s_k, t_k)$ be an instance of* DPP *such that no terminal is contained in D_n.*

We start with the subgraph of G contained in D_n. Retain only the edges and vertices of $\bigcup \mathcal{P} \cup C_n$. Choose an edge. If it is part of C_n, contract it unless it connects endpoints of segments of different type. If it is not part of C_n, contract it unless it connects endpoints of segments. Repeat until there are no contractible edges left. Remove duplicate edges and loops, such that the graph becomes simple again. The resulting graph is the segment graph of D_n.

Segment graphs are outerplanar graphs. An example can be seen in Figure 5.2.



(a) Segments of 8 types, with 6 of the types being tongue tips

(b) Segment graph (colored) and segment dual graph (black)

Figure 5.2: Segments

Definition 5.12 (Tongue tip)**.** *Let G be a plane graph containing a sequence $C_0, \dots, C_n$ of concentric cycles bounding closed discs $D_0, \dots, D_n$, respectively, in the plane. Let $G, (s_t, t_1), \dots, (s_k, t_k)$ be an instance of* DPP *such that no terminal is contained in D_n.*

A D_n-segment type is called tongue tip, if it is a single isolated vertex in the segment graph of D_n.

Definition 5.13 (Segment dual graph)**.** *Let G be a plane graph containing a sequence $C_0, \dots, C_n$ of concentric cycles bounding closed discs $D_0, \dots, D_n$, respectively, in the plane. Let $G, (s_t, t_1), \dots, (s_k, t_k)$ be an instance of* DPP *such that no terminal is contained in D_n. We take the dual graph of the segment graph of D_n. Delete the vertex that represents the infinite face. Add the vertices representing the tongue tips of the segment graph and connect them to the vertices representing neighboring faces in the segment graph. The resulting graph is the segment dual graph of D_n.*

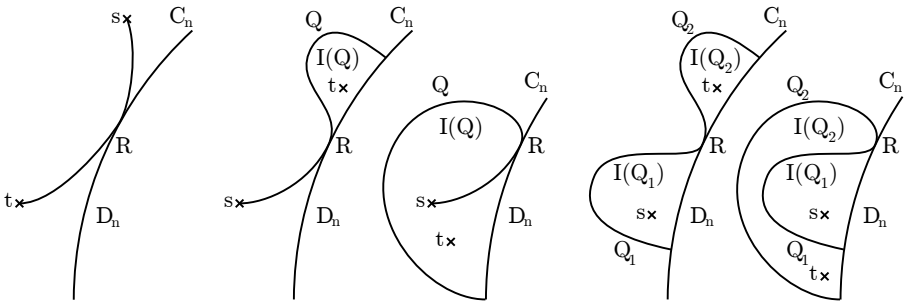
See Figure 5.2b for an example.

Remark 5.14. *Since the segment graph is outerplanar, the segment dual graph is a tree. All inner nodes of the segment dual graph have degree at least* 3*.*

Theorem 5.15 (Folklore, see e. g. [88, Kapitel II, Satz 3])**.** *In any finite multigraph, the number of nodes of odd degree is even.*

Lemma 5.16 (Tongue-taming)**.** *Let G be a plane graph containing a sequence $C_0, \dots, C_n$ of tight concentric cycles bounding closed discs $D_0, \dots, D_n$, respectively, in the plane. Let $(s_0, t_0), \dots, (s_k, t_k)$ be a sequence of terminals that are not contained in D_n. Let $\mathcal{P}$ be a cheap solution of the* DPP *instance $G, (s_1, t_1), \dots, (s_k, t_k)$.*

Then there are at most $2k - 1$ tongue tips.



(a) Type 1 (b) Type 2 (c) Type 3

Figure 5.3: Representatives and terminals in the proof of Lemma 5.16

Proof. We prove that there is an injective map from the set of tongue tips to the set of terminals that is not surjective, proving that the number of tongue tips is at most the number of terminals minus one. We will construct this map by subsequently marking all tongue tips and removing a terminal each time we mark a tongue tip, while another terminal remains.

We represent each tongue tip by the outermost segment R that it contains, i. e. R is the segment, that has a vertex in common with C_n, but not with C_{n-1}. Throughout this proof, we call these outermost segments of tongue tips *representatives*.

For each representative R we will find two terminals, $s(R)$ and $t(R)$. We will then remove $s(R)$, while $t(R)$ remains. s will define the injective map mentioned above.

We distinguish three types of representatives: First, those where the path $P(R)$ to which R belongs contains no nodes in C_n except for those in R. Second those, where $P(R)$ connects to a terminal $s(R)$ without any intermediate nodes in C_n on one side of R, but enters D_n on the other. Third, those where $P(R)$ enters D_n on both sides of R. See Figure 5.3.

Let M be a set of representatives. We call the representatives in M marked. Depending on the set M of marked representatives, we define, which representatives are M-available. Representatives of type 1 are *M-available* if they are unmarked. For an unmarked representative R of type 2 let $Q(R)$ be the subpath of $P(R)$ that has a node in common with R, and that otherwise lies outside of D_n, except for its two endpoints which lie in D_n. R is called *M-available*, if it is unmarked. For an unmarked representative R of type 3 let $Q_1(R) \subseteq P(R)$ and $Q_2(R) \subseteq P(R)$ be the paths outside of D_n which each have a node in common with R and lie outside of D_n except for their endpoints. R is called *M-available*, if for at least one of them, say Q_1, there is no other unmarked representative in $I(Q_1(R))$ and $Q_1(R)$ has no unmarked representative at its other end.

Assuming the algorithm from Figure 5.4 is correct, the lemma is proven: In the end all representatives are marked, and each time we marked a representative, one terminal was removed, and one terminal remained, thus the number of tongue tips is at most the number of terminals minus one. In the following three claims we argue that the algorithm is correct.

Claim 1: There is always an available representative when there is an unmarked

```
M := ∅;
while(there are unmarked representatives)
{
  Choose an M−available representative R;
  if(R is of type 1)
    Choose one of the terminals of P(R) as s(R),
    the other as t(R);
  else if (R is of type 2)
    Choose s(R) as in the definition of type 2,
    choose t(R) ≠ s(R) in I(Q);
  else // R is of type 3
    Choose terminals s(R) in I(Q1), t(R) ≠ s(R) in I(Q2);
  Remove s(R).
  M := M ∪ {R}; // Mark R
}
```

Figure 5.4: Tongue-taming algorithm

representative.

Proof of the Claim: Assume that there is no available representative. This means that all unmarked representatives are of type 3, since representatives of types 1 and 2 are always available when unmarked. We now define a relation $<$ on the set $\mathcal{Q} := \{Q \mid Q = Q_i(R), i = 1, 2, R \text{ is an unmarked representative}\}$ by $Q < Q'$ if $I(Q) \subsetneq I(Q')$. Depending on $\mathcal{Q}$, we will define a multigraph $\mathcal{E}$.

Case 1: All elements in $\mathcal{Q}$ are pairwise incomparable. We define $\mathcal{E}$: The nodes of $\mathcal{E}$ are the unmarked representatives. If $Q_1(R) = Q_1(R')$ for two nodes R, R' of $\mathcal{E}$ we join them by an edge. We do the same if $Q_1(R) = Q_2(R')$, if $Q_2(R) = Q_1(R')$ and if $Q_2(R) = Q_2(R')$. Since all elements of $\mathcal{Q}$ are $<$-minimal, each unavailable representative R has unavailable representatives at the ends of $Q_1(R)$ and $Q_2(R)$. Thus every node in $\mathcal{E}$ has degree 2.

Case 2: There are $Q, Q' \in \mathcal{Q}, Q' < Q$. We can then find such a non-minimal Q, such that there are no $Q'', Q' \in \mathcal{Q} : Q'' < Q' < Q$. I. e. there are elements Q' of $\mathcal{Q}$ that are smaller than Q, and all such Q' are minimal. We define $\mathcal{E}$: The nodes of $\mathcal{E}$ are the unmarked representatives in $I(Q)$. If $Q_1(R) = Q_1(R')$ for two nodes R, R' of $\mathcal{E}$ we join them by an edge. We do the same if $Q_1(R) = Q_2(R')$, if $Q_2(R) = Q_1(R')$ and if $Q_2(R) = Q_2(R')$. Since all $Q' \neq Q$ in $I(Q)$ are $<$-minimal, each unavailable representative R in $I(Q)$ has representatives at the ends of $Q_1(R)$ and $Q_2(R)$. Thus every node R in $\mathcal{E}$ for which $Q \neq Q_1(R)$ and $Q \neq Q_2(R)$ has degree 2. If there is exactly one representative R in $\mathcal{E}$ for which $Q = Q_1(R)$ or $Q = Q_2(R)$, then $\mathcal{E}$ is a contradiction to Theorem 5.15. There cannot be more than 2 such nodes since Q only has two ends. Thus there are zero or two such nodes in $\mathcal{E}$, which means that they have degree 2, too. Thus all nodes in $\mathcal{E}$ have degree 2.

Since each node of $\mathcal{E}$ is of even, non-zero degree, there is a cycle in $\mathcal{E}$ [42, 92, 63], which means there is a cycle in the solution of the DPP, a contradiction. $\dashv$

Claim 2: We can always choose terminals $s(R)$ and $t(R)$ as in the algorithm of Figure 5.4.

Proof of the Claim: We prove it for each type of R:

Type 1: The terminals $s(R)$ and $t(R)$ for the unmarked representative R of type 1 have not been removed before: These terminals are not part of any $P(R')$ for any representative $R' \neq R$, thus they could only have been removed while processing a representative R' of type 3. But then R is in $I(Q_1(R'))$. Which means that R' was not available, because R was still unmarked, a contradiction.

Type 2: For terminal $s(R)$ the reasoning from type 1 holds. According to Lemma 5.9 there is a terminal $t(R)$ in $I(Q(R))$. Assume that $t(R) = s(R)$ and that there is no other terminal in $I(Q)$ that we could have chosen as $t(R)$ instead. The path $C_n \cap I(Q(R))$ cannot replace $Q(R)$ in the solution, because the solution is cheap. Hence there is a vertex in $C_n \cap I(Q(R)) \setminus Q(R)$ that is used by some segment, contradicting Lemma 5.8 above. Thus we can choose $t(R) \neq s(R)$. We argue below (Claim 3) that we can choose $t(R)$ so it was not removed before.

Type 3: According to Lemma 5.9 there are terminals $s(R)$ in $I(Q_1)$ and $t(R)$ in $I(Q_2)$. We can choose $s(R) \neq t(R)$ since otherwise $Q_1(R) \cup R \cup Q_2(R)$ gives a counterexample to Lemma 5.9. We then mark R and remove the terminal s, while t remains. Again, we can choose s and t in such a way that they were not removed before. We now argue in Claim 3 why this is possible. ⊣

Claim 3: The choice of an not yet removed terminal $s(R)$ for type 2 and the choice of terminals $s(R)$ and $t(R)$ for type 3 are possible.

Proof of the Claim: Let the graph H be $H = I(Q(R))$ for type 2 and $H = I(Q_1(R))$ or $H = I(Q_2(R)) \setminus I(Q_1(R))$ for type 3. Previous loop iterations may have removed terminals in $I(Q(R)), I(Q_1(R))$ or $I(Q_2(R))$, but they only removed such terminals in the graph H. Assume that another representative R', has been marked before. This R' was of one of the three types. We claim that at least one terminal in H remained. To see this, let

$$T := \begin{cases} P(R'), & \text{for } R' \text{ of type 1,} \\ P'(R') \cup R' \cup Q(R'), & \text{for } R' \text{ of type 2,} \\ Q_1(R') \cup R' \cup Q_2(R'), & \text{for } R' \text{ of type 3.} \end{cases}$$

Intuitively, T is one of the three configurations in Figure 5.3. Then $T \subseteq H$: This follows from planarity, tightness and from the choice of R' as the outermost segment in the tongue tip, in particular, from the fact that $R' \cap C_{n-1} = \varnothing$. Hence for all three types, both the removed terminal $s(R')$ and the remaining terminal $t(R')$ are in H. ⊣

q. e. d.

Theorem 5.17. *Let G be a plane graph containing a sequence $C_0, \ldots, C_n$ of tight concentric cycles bounding closed discs $D_0, \ldots, D_n$, respectively, in the plane. Let $G, (s_t, t_1), \ldots, (s_k, t_k)$ be an instance of* DPP *such that no terminal is contained in D_n. Let $\mathcal{P}$ be a cheap solution to this instance. Then $\mathcal{P}$ has at most $4k - 4$ different types of D_n-segments.*

Proof. The segment types correspond to the edges in the segment dual graph. The tongue tips correspond to the leaves of the segment dual graph. According to Lemma 5.16 this tree has at most $2k - 1$ leaves, and according to Remark 5.14 all inner nodes have degree at least three. Thus the segment dual graph has at most $4k - 4$ edges. q. e. d.

5.3 Bounding the size of segment types

In this section we find a bound on the size of segment types in cheap solutions and we combine it with the bound on the number of segment types obtained in the previous section to find irrelevant vertices. Indeed, we find that cheap solutions only pass through a bounded number of concentric cycles.

We find the bound on the size of segment types by rerouting in the presence of a large segment type. In a first step, we allow ourselves to freely reroute in a disc (making sure that the graph remains planar), and we bound the number of segments of solution paths in the disc. In a second step, we realize our rerouting in a sufficiently large grid.

A subword $w_i \cdots w_j$ of a word $w = w_1 \cdots w_n$ for some $n \in \mathbb{N}$ is called *infix*. An infix with $i = 1$ is called *prefix*, an infix with $j = n$ is called *suffix*.

Lemma 5.18. *Let Σ be an alphabet of size $|\Sigma| = k$. Let $w \in \Sigma^*$ be a word over Σ. If $|w| > 2^k$, then w contains an infix y with $|y| \geq 2$, such that every letter occurring in y occurs an even number of times in y.*

Proof. Let $\Sigma = \{a_1, \ldots, a_k\}$, and let $w = w_1 \cdots w_n$ with $n > 2^k$. Define vectors $z_i \in \{0,1\}^k$ for $i \in \{1, \ldots, n\}$, and we let the jth entry of vector z_i be 0 if and only if letter a_j occurs an even number of times in the prefix $w_1 \cdots w_i$ of w and 1 otherwise. Since $n > 2^k$, there exist $i, i' \in \{1, \ldots, n\}$ with $i \neq i'$, such that $z_i = z_{i'}$. Then $y = w_{i+1} \cdots w_{i'}$ proves the lemma. q. e. d.

The following lemma is essentially the main combinatorial result from [4]. The proof is included here for the sake of completeness.

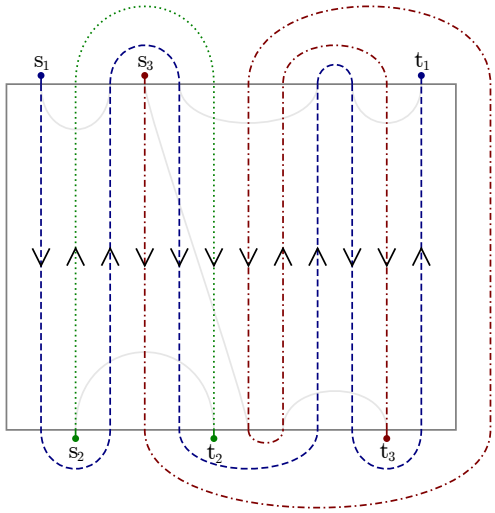
Lemma 5.19 (Rerouting in a disc)**.** *Let G be a plane graph, let $G, (s_1, t_1), \ldots, (s_k, t_k)$ be an instance of* DPP *that has a solution $\mathcal{P}$. Let G contain a cycle C bounding a closed disc D in the plane such that no terminal lies in D. Assume that every D-segment of $\mathcal{P}$ is simply an edge and, except for vertices and edges of D-segments, the interior of D contains no other vertices or edges of G.*

Then, if there is a segment type that contains more than 2^k segments, then we can replace the outerplanar graph O consisting of all D-segments of $\mathcal{P}$ by a new outerplanar graph O' such that in $(G \setminus O) \cup O'$ the DPP *(with the original terminals) has a solution and $|E(O')| < |E(O)|$.*

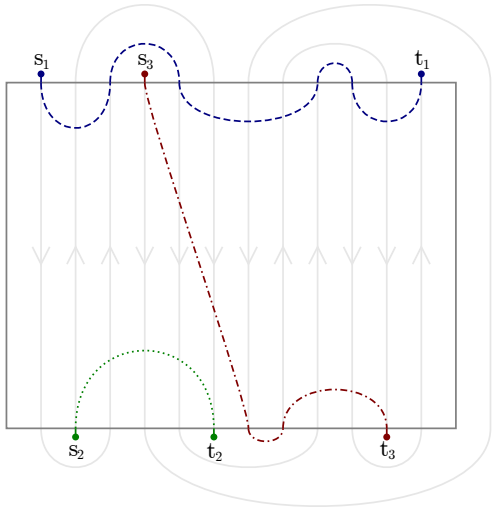
Proof. Assume there is a segment type in O that contains more than 2^k edges. Choose an ordering of the edges of this segment type that corresponds to their order in the plane. Label each of these edges by a word in $\{1, \ldots, k\}$ according to the number i of the path $P_i \in \mathcal{P}$ that it belongs to. Hence we obtain a word w over the alphabet $\{1, \ldots, k\}$ with $|w| > 2^k$, and by Lemma 5.18 we know that w contains an infix y with $|y| \geq 2$, such that every letter occurring in y occurs an even number of times in y. Let $Y := \{e_1, e_2, \ldots, e_{|y|}\}$ be the set of edges of O that correspond to the letters in y. For every path $P_i \in \mathcal{P}$ we orient all edges of P_i from s_i to t_i. For a path $P_i \in \mathcal{P}$ with $E(P_i) \cap Y \neq \varnothing$, let $e^i_1, \ldots, e^i_{2n_i}$ be the (even number of) edges of Y appearing on P_i in this order when moving from s_i to t_i. We introduce a shortcut for P_i as follows:

For every odd number $j \in \{1, \ldots, 2n_i\}$, we replace the subpath of P_i from $\text{tail}(e^i_j)$ to $\text{head}(e^i_{j+1})$ by a new edge f^i_j in the disc D. Having done this for all odd numbers $j \in \{1, \ldots, 2n_i\}$, we obtain a new path P'_i from s_i to t_i that uses strictly less edges in D than P_i. Let O' be the graph obtained from O by

modifying all paths $P \in \mathcal{P}$ with $E(P) \cap Y \neq \varnothing$ in this way (see Figure 5.5). Since $|y| \geq 2$ we have $\left|E(O')\right| < \left|E(O)\right|$. Obviously, the DPP has a solution in $(G \setminus O) \cup O'$.



(a) Segments before rerouting



(b) Segments after rerouting

Figure 5.5: Construction in the proof of Lemma 5.19

Finally, we give a geometric argument showing that O' is outerplanar: Given the planar embedding of G as above, we transform D homeomorphically into a large rectangle R. The upper side of R lies on a straight line U, and the lower side lies on a straight line L. Let S be the straight line parallel to U and L, that divides D into halves. We assume that every edge in Y is represented by a vertical straight line segment from U to L. We now take a smaller rectangle $R_0 \subseteq R$ bounded by U and L, such that the only D_n-segments it contains are the segments in Y. Let ∂R_0 be the boundary of R_0.

For every path P_i using an edge of Y, let F^i_j denote the subpath of P_i from $\text{head}(e^i_j)$ to $\text{tail}(e^i_{j+1})$ (for $j \in \{1, \dots, 2n_i\}$, j odd). We replace F^i_j by a single edge c^i_j. Then the graph H with vertex set $V(O)$ and edge set

$$\{c^i_j \mid i \in \{1, \dots, k\},\ E(P_i) \cap Y \neq \varnothing,\ j \in \{1, \dots, 2_{n_i}\},\ j \text{ odd}\}$$

is outerplanar. Take the embedding of H obtained from the embedding of G using the rectangle as above, where we embed the edges c^i_j along the subpaths F^i_j. We regard the embedded c^i_j as simple curves. Now we reflect the curves c^i_j inward at ∂R_0. We may assume that after reflection, none of the c^i_j crosses S or the drawing of an edge in $E(O) \setminus Y$. (If not, transform the drawing accordingly.) After reflection, the curves still have their endpoints on ∂R_0 and they are pairwise non-crossing. In a second step, we now reflect the curves at S. Let $(c^i_j)'$ denote the curve obtained from c^i_j by these two reflections. Now $(c^i_j)'$ connects $\text{tail}(e^i_j)$ to $\text{head}(e^i_{j+1})$ (while c^i_j connects $\text{head}(e^i_j)$ to $\text{tail}(e^i_{j+1})$). Due to symmetry, the $(c^i_j)'$ are pairwise non-crossing, and none of them crosses a drawing of an edge in $E(O) \setminus Y$. Hence the $(c^i_j)'$ together with the drawing of edges in $E(O) \setminus Y$ provide an outerplanar drawing of O' (where $(c^i_j)'$ is the drawing of f^i_j). Hence O' is outerplanar. This concludes the proof of the lemma. q. e. d.

Definition 5.20. *Let $n, m \in \mathbb{N}$. An untidy $(n \times m)$ grid is a graph obtained from a set $\mathcal{H}$ of n pairwise vertex-disjoint (horizontal) paths and a set $\mathcal{V} = \{V_1, \dots, V_m\}$ of m pairwise vertex-disjoint (vertical) paths as follows: Every path in $\mathcal{V}$ intersects every path in $\mathcal{H}$ in precisely one non-empty path, and each path $H \in \mathcal{H}$ consists of m vertex-disjoint segments such that V_i intersects H only in its ith segment (for every $i \in \{1, \dots, m\}$). A subdivided untidy $(n \times m)$ grid is obtained from an untidy $(n \times m)$ grid by subdividing edges.*

Let τ be a segment type in the plane graph. Recall that all the segments in a type are "parallel" to each other. We say that segments $S_1, \dots, S_n \in \tau$ are *consecutive*, if they appear in this order (or in the reverse order) in the plane. Segment types that go far into the concentric cycles yield subdivided untidy grids. More precisely, we show the following lemma, that is an easy consequence of Lemma 5.8.

Lemma 5.21. *Let $l, n, r \in \mathbb{N}$ with $n \geq l-1$. Let G be a plane graph containing a sequence $C_0, \dots, C_n$ of tight concentric cycles bounding closed discs $D_0, \dots, D_n$, respectively, in the plane. Let $G, (s_t, t_1), \dots, (s_k, t_k)$ be an instance of* DPP*, such that no terminal is contained in D_n. Let $\mathcal{P}$ be a cheap solution to this instance. If there is a type τ of D_n-segments of $\mathcal{P}$ with $|\tau| \geq r$ such that r consecutive segments of τ each contain a vertex of D_{n-l+1}, then G contains a subdivided untidy $(2l \times r)$ grid as a subgraph, with the r consecutive segments of τ as vertical paths, and suitable subpaths of $C_n, \dots, C_{n-l+1}, C_{n-l+1}, \dots, C_n$ (in this order) as horizontal paths.*

The following lemma, whose proof is based on Lemmata 5.19 and 5.21, shows that we can reroute a sufficiently large segment type in the case that many segments of the type go far into the concentric cycles.

Lemma 5.22 (Rerouting in an untidy grid)**.** *Let $n, k \in \mathbb{N}$ with $n \geq 2^{k-1} - 1$. Let G be a plane graph containing a sequence $C_0, \ldots, C_n$ of tight concentric cycles bounding closed discs $D_0, \ldots, D_n$, respectively, in the plane. Let $G, (s_t, t_1), \ldots, (s_k, t_k)$ be an instance of* DPP*, such that no terminal is contained in D_n. Let $\mathcal{P}$ be a cheap solution to this instance. Then $\mathcal{P}$ has no type τ of D_n-segments with $|\tau| \geq 2^k + 1$, such that each of $2^k + 1$ consecutive segments in τ contains a vertex in $D_{n-2^{k-1}+1}$.*

Proof. Towards a contradiction, assume that τ is a type of D_n-segments of $\mathcal{P}$ with $|\tau| \geq 2^k + 1$, such that each of $2^k + 1$ consecutive segments in τ contains a vertex in $D_{n-2^{k-1}+1}$. Let $r := 2^k + 1$ and let $l := 2^{k-1}$. Let $H \subseteq G$ be a subdivided untidy $(2l \times r)$ grid as in Lemma 5.21. The proof that follows is similar to the proof of Lemma 5.19, that is, we reroute some segments of τ. In addition, we make sure that we can realize the rerouted segments in H. If we let $D := D_n$, contract every D_n-segment of $\mathcal{P}$ to a single edge and remove all other vertices and edges of G in the interior of D, then we can apply Lemma 5.19 to the segment τ_0 obtained from τ by the contraction. We do this as follows: order the segments of τ (and hence of τ_0) according to their occurrence in the plane. Color the first $2^k + 1$ segments by the number of the path of $\mathcal{P}$ they belong to. Among these consecutive paths, find an infix d of colors as in Lemma 5.18. Then $2 \leq |d| \leq 2^k$. Let H' be the subdivided untidy $(2l \times |d|)$ subgrid of H with vertical paths corresponding only to the segments in d, and the horizontal paths shortened accordingly, as much as possible.

Reroute the segments in τ that correspond to the the letters of d as in Lemma 5.19. Then we obtain $\frac{|d|}{2}$ new segments (indeed, in the lemma each of them is a single edge, but now we simply subdivide them if necessary) and a new solution $\mathcal{P}'$ of the DPP. For routing the new segments in H', we use the paths of the old segments as follows: For routing a new segment, we use at most two old segments corresponding to a letter in d (two vertical paths in H'), and, for crossing horizontally, one horizontal path in H. Notice that all D_n-segments of $\mathcal{P}'$ use subpaths of $\mathcal{P}$ and horizontal paths of H' only. Since $2 \leq |d| \leq 2^k$ it follows that $1 \leq \frac{|d|}{2} \leq 2^{k-1}$. But H' has $2l = 2^k$ horizontal paths and horizontal crossings of $\mathcal{P}'$ use at most $\frac{|d|}{2} < 2^k$ of them. Hence one of them, h say, is not used by any horizontal crossing of $\mathcal{P}'$. But then h has a crossing with at least $1 \leq \frac{|d|}{2}$ vertical path in H' that is not used by any path in $\mathcal{P}'$. With this it is easy to see that $c(\mathcal{P}') < c(\mathcal{P})$, a contradiction to the cheapness of the solution. q. e. d.

The following remark says that if we have a segment type of sufficiently large cardinality, then many segments will go far into the concentric cycles.

Lemma 5.23. *Let $n, l, r \in \mathbb{N}$. Let G be a plane graph containing a sequence $C_0, \ldots, C_n$ of tight concentric cycles bounding closed discs $D_0, \ldots, D_n$, respectively, in the plane. Let $G, (s_t, t_1), \ldots, (s_k, t_k)$ be an instance of* DPP *such that no terminal is contained in D_n. Let $\mathcal{P}$ be a cheap solution to this instance. Let τ be a type of D_n-segments with $|\tau| \geq 2l + r$. Then $n \geq l - 1$ and τ contains r consecutive segments such that each of them has a vertex in D_{n-l+1}.*

Proof. Order the segments in τ according to their occurrence in the planar embedding. Remove the l first and the l last segments. From Lemma 5.8 it

follows that $n \geq l - 1$ and each of the remaining segments contains a vertex in D_{n-l+1}. q. e. d.

Theorem 5.24. *Let G be a plane graph containing a sequence $C_0, \ldots, C_n$ of tight concentric cycles bounding closed discs $D_0, \ldots, D_n$, respectively, in the plane. Let $G, (s_t, t_1), \ldots, (s_k, t_k)$ be an instance of* DPP *such that no terminal is contained in D_n. Let $\mathcal{P}$ be a cheap solution to the input instance. Then there are at most $(8k-8) \cdot 2^k$ D_n-segments of $\mathcal{P}$.*

Proof. We first prove that every type of D_n-segments of $\mathcal{P}$ contains less than $2 \cdot 2^k + 1$ segments. Let $l := 2^{k-1}$ and $r := 2^k + 1$. Towards a contradiction, assume that τ is a type of D_n-segments with $|\tau| \geq 2 \cdot 2^k + 1 = 2l + r$. Then, by Lemma 5.23, τ contains $r = 2^k + 1$ consecutive segments such that each of them has a vertex in D_{n-l+1}, but this is not possible by Lemma 5.22, a contradiction.

Now, by Theorem 5.17 the number of types of D_n-segments of $\mathcal{P}$ is at most $4k - 4$, so the total number of segments is at most $(8k-8) \cdot 2^k$ D_n, concluding the proof. q. e. d.

Theorem 5.25 (Irrelevant Vertex)**.** *Let G be a plane graph, let $G, (s_1, t_1), \ldots, (s_k, t_k)$ be an instance of* DPP *that has a solution, $n = (8k-8) \cdot 2^k + 1$, and let G contain a sequence $C_0, \ldots, C_n$ of tight concentric cycles bounding closed discs $D_0, \ldots, D_n$, respectively, in the plane, such that no terminal of the* DPP *lies in D_n. Let $C_0 = \{v\}$. Then the* DPP *has a solution that avoids v.*

Proof. Let $\mathcal{P}$ be a cheap solution to the DPP. We argue that $\mathcal{P}$ avoids v. By Theorem 5.24 the number of D_n-segments of $\mathcal{P}$ is at most $n - 1$.

Consider a D_n-segment P of $\mathcal{P}$. Let i be the lowest integer such that $P \cap C_i \neq \varnothing$. We say that P *peaks* at C_i. Note that P peaks at exactly one cycle C_i. Suppose $\mathcal{P}$ does not avoid v. Since the number of D_n-segments of $\mathcal{P}$ is at most $n - 1$, there is an i such that no D_n-segment of $\mathcal{P}$ peaks at C_i and some D_n-segment P of $\mathcal{P}$ peaks at C_{i-1}. Let P' be the subpath of P with endpoints in C_i and internal vertices in $D_i \setminus C_i$, in particular P' contains $P \cap C_{i-1}$. Let Q be the path between the endpoints of P' such that $Q \subseteq C_i$ and v is not contained in a cycle formed by $Q \cup P$.

Since no segment peaks at C_i, Lemma 5.8 implies that no D_n-segments of $\mathcal{P}$ contains an interior vertex of Q. Hence we can reroute P along Q rather than along P', contradicting that $\mathcal{P}$ is a cheap solution. q. e. d.

Given a plane graph G and a vertex v we show how to check whether a particular vertex v satisfies the conditions of Theorem 5.25. We set $C_0 = \{v\}$ and given C_i we construct C_{i+1} by performing a depth first search from a neighbor u of a vertex in C_i, always choosing the rightmost edge leaving the vertex we are visiting. This search will either output an innermost cyclic walk (which then can be pruned to a cycle) around C_i or determine that no such walk exists. In the case that a cycle C_{i+1} is output, we check whether the cyclic walk contained a terminal s_i or t_i. If it did, it means that this terminal lies on C_{i+1} or in its interior. At this point (or when the search outputs that no cycle around C_i exists), we have determined that there are i tight concentric cycles around v with no terminal in the interior of C_i. If $i > (8k-8) \cdot 2^k + 1$ this implies that v satisfies the conditions of Theorem 5.25. Clearly this procedure can be implemented to run in linear time. This yields the following theorem.

The *face-vertex incidence graph* of a plane graph G is the bipartite graph G^* on the vertices and faces of G whose edges are sets$\{v, f\}$, where f is a face of G and v is a vertex of G incident to f.

Theorem 5.26. *Let G be a plane graph, let $G, (s_1, t_1), \ldots, (s_k, t_k)$ be an instance of* DPP*.*

There is an $O(|V(G)|^2)$ time algorithm that outputs an induced subgraph G' of G such that $tw(G') < (72k \cdot 2^k - 72 \cdot 2^k + 18)\lceil\sqrt{2k+1}\rceil$ and G is a YES-instance for DPP *if and only if G' is.*

Proof. W.l.o.g. we assume that G is connected. For each vertex $v \in V(G)$ we check whether v satisfies the conditions of Theorem 5.25. If it does, we delete v. The resulting graph is G'. By Theorem 5.25 G has a DPP solution if and only if G' does. Observe that no vertex of G' satisfies the conditions of Theorem 5.25 because deleting a vertex u from a graph cannot increase the number of concentric cycles around a vertex v.

It remains to argue that $tw(G') < (72k \cdot 2^k - 72 \cdot 2^k + 18)\lceil\sqrt{2k+1}\rceil$. Suppose not, we will prove that G' contains a vertex v and a sequence $C_0, \ldots, C_n$ of tight concentric cycles bounding closed discs $D_0, \ldots, D_n$, respectively, in the plane, such that no terminal of the DPP lies in D_n. This will contradict the construction of G'. By [119],[54, Theorem 1], G contains a grid minor of size $\eta \times \eta$, where $\eta = 2((8k-8) \cdot 2^k + 1)\lceil\sqrt{2k+1}\rceil$. By Lemma 5.4 there are tight concentric cycles $C = C_0, \ldots, C_r$. The vertex v in the innermost of these cycles satisfies the conditions of Theorem 5.25, contradicting the construction of G'.

For the running time, note that each vertex will be tested only once: Any vertex not removed after the test cannot be removed at any later time either. We claim that for each vertex $v \in V(G)$ checking whether v satisfies the conditions of Theorem 5.25 can be done in linear time. Let G^* be the face-vertex incidence graph of G. A breadth-first-search in G^* starting at v defines *layers* (sets of vertices) around v in G^* in the obvious way. Since G^* is planar, the breadth-first-search takes linear time in the number of nodes of G^*.

The number of concentric cycles around v in G equals half the number of breadth-first-search-layers $L \subseteq V(G^*)$ that satisfy the following property: The component of $G^* \setminus L$ that contains the outer face vertex also contains all terminals and does not contain v. q. e. d.

Chapter 6

Graphs without Irrelevant Vertices[1]

In this chapter we give a lower bound on the function f from Theorem 4.2, showing that $f(k) \geq 2^k$, even for planar graphs. For this we construct a family of planar input graphs $(G_k)_{k\geq 2}$, each with k pairs of terminals, such that the tree-width of G_k is $2^k - 1$; each member of the family has a unique solution to the Disjoint Paths Problem, where the paths of the solution use all vertices of the graph. Hence no vertex of G_k is irrelevant. As a corollary, we obtain a lower bound of $2^k - 1$ on the tree-width of graphs having *vital linkages* (also called *unique linkages*) [117] with k components.[2]

In the previous Chapter 5 we provided an upper bound on f on planar graphs, showing that $f(k) \leq (72k \cdot 2^k - 72 \cdot 2^k + 36k - 27)\lceil\sqrt{2k+1}\rceil - \frac{1}{2}$ (we also obtained a slightly better bound of $f(k) \leq 26k \cdot 2^k\sqrt{2k}$ [3]). Our lower bound shows that this is asymptotically optimal. Recently, Mazoit [97] gave an upper bound on f on graphs of bounded genus that is single exponential in k and the genus. Geelen et alii [51] independently prove a result similar to [97], but their analysis yields a weaker upper bound. The exact order of growth of f on general graphs is still unknown.

Our main result in this chapter is the following.

Theorem 6.1. *Let $f, g, h\colon \mathbb{N} \to \mathbb{N}$ be as in Theorems 4.2 and 4.4. Then $f(k) \geq 2^k$, $g(k) \geq 2^k - 1$, and $h(k) \geq 2^k - 1$. Moreover, this holds even if we consider planar graphs only.*

The Disjoint-Paths Problem cannot be solved in $2^{o(w \log w)} \cdot n^{O(1)}$ for graphs of tree-width at most w, unless the Exponential Time Hypothesis fails [95]. This, along with Theorem 6.1, reveals the limitations of the irrelevant vertex technique: any algorithm for the Disjoint-Paths Problem whose parameter dependence that is better than doubly exponential, will probably require drastically different techniques.

[1] This chapter is based on joint work with Isolde Adler which has been submitted [5] for publication.

[2] This result appeared in the last section of the conference paper [2]. While the main focus of the paper [2] was a single exponential upper bound on f on planar graphs (see previous Chapter 5), it only sketches the lower bound. Here we provide the full proof of the lower bound.

6.1 Disc-with-edges embeddings

In this section, we consider plane graphs that mainly reside inside some closed disc in the plane, such that the terminals are on the boundary cycle. In addition, they have specific edges embedded outside the disc to allow bypassing terminals. The precise definition is the following:

Definition 6.2 (Disc-with-edges embedding)**.** *Let $G, (s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$ be an instance of the disjoint paths problem. A disc-with-edges embedding of $G, (s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$ is a planar drawing Π of G with the property that there exist a subgraph $H_G \subseteq G$ and a cycle $C \subseteq H_G$ such that in the Π-embedded graph G, the cycle C bounds a closed disc $D \subseteq \mathbb{R}^2$ satisfying the following conditions:*

1. *$V(H_G) = V(G) \subseteq D$.*
2. *Every edge $e \in E(G)$ satisfies $e \in E(H_G) \iff e \subseteq D$.*
   (We will refer to C as the boundary of H_G.)
3. *There is a path $P \subseteq C$ from t_1 to s_k containing all the terminals. Moreover, walking from t_1 to s_k on P, the terminals appear in the following (interleaved) ordering:*

$$t_1, t_2, s_1, t_3, s_2, t_4, \ldots, s_i, t_{i+2}, \ldots, s_{k-2}, t_k, s_{k-1}, s_k.$$

4. *Every edge $e \in E(G) \setminus E(H_G)$ satisfies the following conditions.*
   (a) *No endpoint of e is a terminal, and e is disjoint from all other edges in $E(G) \setminus E(H_G)$.*
   (b) *The endpoints of e are connected by a path on the boundary of H_G that contains exactly one terminal t.*
   (We will refer to e as an edge around the terminal t.)
5. *For every $i \in [k]$, there are exactly $2^{k-i} - 1$ edges around t_i and no edges around s_i. We let E_i denote the set of all edges around t_i.*

Note that for $i \in [k]$ we have $|E_i| = 2^{k-i} - 1$, and, in particular, $E_k = \varnothing$.

See Figure 6.1 for an instance of DPP embedded by a disc-width-edges embedding for $k = 4$.

Remark 6.3. *Let $G, (s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$ be an instance of* DPP *embedded by a disc-width-edges embedding, and let $1 < i < k$. Then s_i is connected to t_i by a path p_i on the boundary of H_G that contains $s_i, t_{i+1}, s_{i-1}, t_i$, but no other terminals. The terminal s_k is connected to t_k by a path on the boundary of H_G that contains s_k, s_{k-1}, t_k, but no other terminals. The terminal s_1 is connected to t_1 by a path on the boundary of H_G that contains s_i, t_{i+1}, t_i, but no other terminals.*

Proof. This follows immediately from Definition 6.2.4. q. e. d.

Remark 6.4. *Let $G, (s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$ be an instance of* DPP *embedded by a disc-width-edges embedding, and let $1 \leq i \leq k$.*

Then there is a path q_i on the boundary of H_G, such that

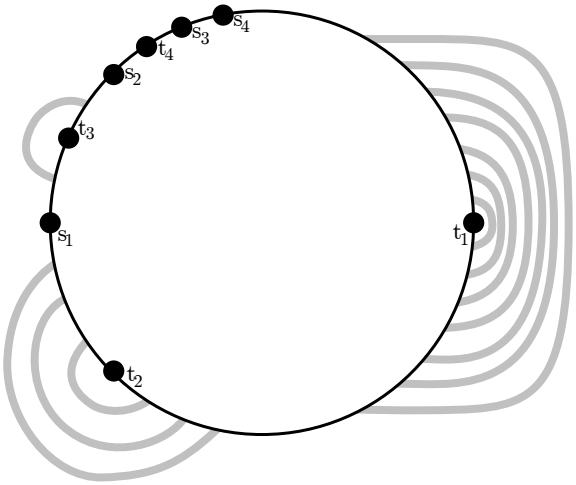


Figure 6.1: An instance of DPP embedded by a disc-width-edges embedding (the details of H_G are omitted).

- *q_i contains all endpoints of edges in E_i,*
- *q_i contains no endpoints of edges in $\bigcup_{\ell=1}^{k} E_\ell \setminus E_i$, and*
- *q_i contains terminal t_i and no other terminals.*

Proof. This follows immediately from Definition 6.2.4 and planarity. q. e. d.

Lemma 6.5. *Let $G, (s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$ be an instance of* DPP *embedded by a disc-with-edges embedding. Let $1 \leq i \leq k$,*

$$u \in \{s_{i+1}, s_{i+2}, s_{i+3}, \ldots, s_k, t_{i+2}, t_{i+3}, \ldots, t_k\} \cup \bigcup E_{i+2} \cup \bigcup E_{i+3} \cup \ldots \cup \bigcup E_k,$$
$$v \in \{t_{i+1}\} \cup \bigcup E_{i+1}.$$

Then there is a path p connecting u and v on the boundary of H_G that contains s_i, but no vertex in

$$\{t_i, t_{i-1}, \ldots t_1\} \cup \bigcup E_i \cup \bigcup E_{i-1} \cup \ldots \cup \bigcup E_1.$$

Proof. We prove the statements by inverse induction on i. For $i = k$, we cannot choose an u, thus the lemma holds.

Assume that the lemma holds for $i + 1$ with $k \geq i + 1 > 1$. We show that it holds for i. There are four choices for u. Case 1: $u = s_{i+1}$. Then we get p using Remark 6.3 and Remark 6.4. Case 2: $u = t_{i+2}$. Then by Remark 6.3 we get a path from u to s_{i+2} that contains s_{i+1}. This gives us a path from u to s_{i+1}, which we can combine with Case 1 to get p. Case 3: $u \in \bigcup E_{i+2}$. By Remark 6.4, we get a path from u to t_{i+2}, which we can combine with Case 2 to get p. Case 3: $u \in \{s_{i+2}, s_{i+3}, \ldots, s_k, t_{i+3}, \ldots, t_k\} \cup \bigcup E_{i+3} \cup \ldots \cup \bigcup E_k$. By induction, we get a path from u to t_{i+2}, which we can combine with Case 2 to get p. q. e. d.

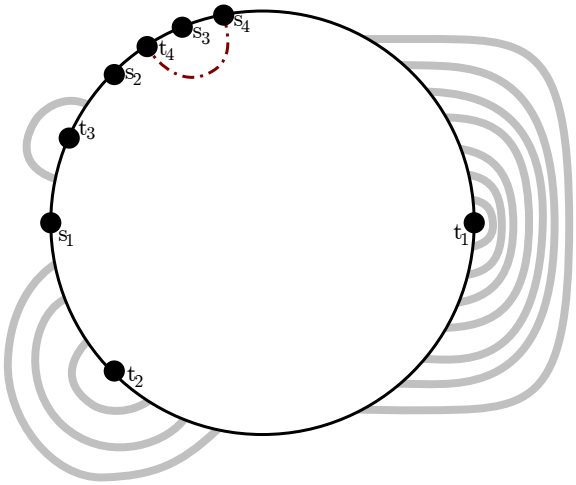


Figure 6.2: P_4.

Let $G, (s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$ be an instance of DPP embedded by a disc-with-edges embedding. Let $i, j \in [k]$ with $i \leq j$, and let P be a path in G from s_j to t_j. By $C_i(P)$ we denote the collection of paths obtained from P by removing all edges in $E_i \cup \ldots \cup E_k$. For an edge e around a terminal t, we let $y(e)$ denote the number of of edges around t, whose endpoints are contained in the path between the endpoints of e on the boundary of H_G, which contains t.

Lemma 6.6. *Let $G, (s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$ be an instance of* DPP *that has a solution $P_1, P_2, \ldots, P_k$. Let G be embedded by a disc-width-edges embedding. Let $i, j \in [k]$ with $1 < i \leq j$.*

1. *Let e be an edge around t_i, let $l \in \mathbb{N}$ be the smallest integer, such that $y(e)$ is a multiple of 2^{k-l}. Then the edge e is contained in the path P_l.*
2. *$\left|C_i(P_j)\right| = 2^{j-i}$.*
3. *Every path in $C_i(P_j)$ has exactly one endpoint in $\{t_i\} \cup \bigcup E_i$.*

Proof. We prove the statements by inverse induction on i.

Let $i = k$. Statements 1 trivially holds, as $E_i = \varnothing$. Observe that removing the empty set of edges from P_k, we obtain $C_k(P_k) = \{P_k\}$, and hence $\left|C_k(P_k)\right| = 1 = 2^{k-k}$, proving Statement 2. And P_k has endpoint t_k in $\{t_k\}$, proving Statement 3.

Assume that the lemma holds for $i + 1$ with $k \geq i + 1 > 1$. We show that it holds for i.

We consider the embedding of $G' := (V(G), E(G) \setminus E_i)$ obtained by deleting all curves representing edges in E_i in the disc-width-edges embedding of G. Let P'_i be the subpath of P_i starting at s_i and ending at the first vertex of P_i in $\{t_i\} \cup \bigcup E_i$. Then both endpoints of P'_i are on the outer face of the embedding of G'. Connecting the endpoints of P'_i by a closed curve L on the outer face, we obtain a closed curve $L \cup P'_i$ that divides the plane into two regions.

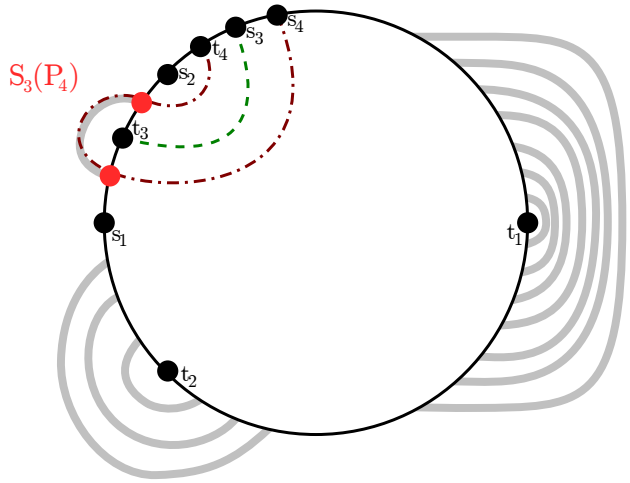


Figure 6.3: P_4, P_3.

Let A be the subgraph of G' induced by the vertices contained in the region that contains t_{i-1}, let B be the the subgraph of G' induced by the vertices contained in the region that does not contain t_{i-1}. Clearly, $A \cap B = P'_i$, and P'_i is an A-B-separator in G'. Then, if there are any edges between A and B in G, these are edges in E_i. By Remark 6.3, there is a path p_i, on the boundary of H_G, that contains s_{i-1} but no terminal t_j with $j < i$. Since by induction (Statement 1) P_i cannot contain edges around terminals t_j with $j \geq i$, the path P'_i crosses the path p_i exactly at s_{i-1}. Thus the endpoints of p_i are in different regions, i. e. s_{i-1} is in B. Thus P_{i+1} has one endpoint in A and the other endpoint in B. Let $p \in \bigcup_{j>i} C_{i+1}(P_j)$. By induction (Statement 3), p has exactly one endpoint in $\{t_{i+1}\} \cup \bigcup E_{i+1}$. By definition of $C_{i+1}(P_j)$, the other endpoint of p is in

$$\{s_{i+1}, s_{i+2}, s_{i+3}, \ldots, s_k, t_{i+2}, t_{i+3}, \ldots, t_k\} \cup \bigcup E_{i+2} \cup \bigcup E_{i+3} \cup \ldots \cup \bigcup E_k.$$

By Lemma 6.5, the endpoints of p are connected by a path p' on the boundary of H_G, that does not contain terminal t_i and does not contain any vertex in $\bigcup E_i \cup \bigcup E_{i-1} \cup \ldots \cup \bigcup E_1$. By the same argument as above for p_i, we get that P'_i crosses the path p' exactly at s_{i-1}, and thus p has one endpoint in A and the other endpoint in B. Therefore every path in $\bigcup_{j \geq i} C_{i+1}(P_j)$ contains at least one edge in E_i. By induction (Statement 2), and the definition of the disc-with-edges instance, we have

$$\left| \bigcup_{k \geq j \geq i+1} C_{i+1}(P_j) \right| \overset{\text{ind.}}{=} \sum_{j=i+1}^{k} 2^{j-(i+1)} = 2^{k-i} - 1 \overset{\text{def.}}{=} |E_i|.$$

We thus obtain the following.

Claim 1: Each path $p \in \bigcup_{j=i+1}^{k} C_{i+1}(P_j)$ contains *exactly* one edge in E_i, and P_i contains no edges in E_i.

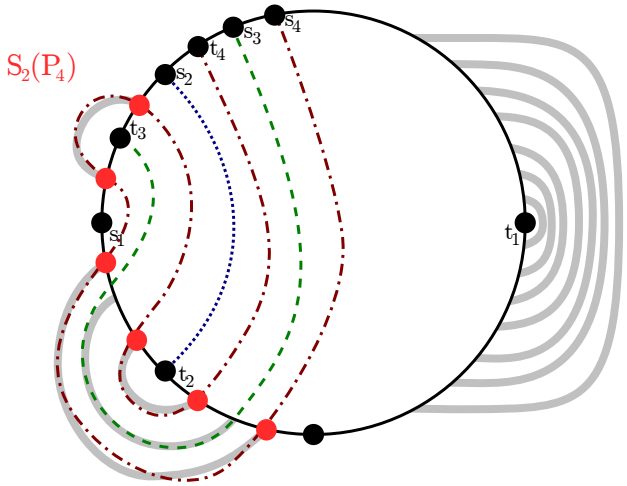


Figure 6.4: P_4, P_3, P_2.

From this Claim we immediately obtain Statement 3.

Towards a proof of Statement 1: Since A is plane and by Claim 1, the paths in $\bigcup_{j=i+1}^{k} C_{i+1}(P_j)$ are a planar linkage between $\bigcup E_{i+1} \cup \{t_i\}$ and the endpoints of edges in E_i, that lie on the path between t_i and s_i from Remark 6.3. Each edge in E_i falls into one of three cases (Figure 6.6): Case 1: $y(e) < (|E_{i+1}| + 1)$. The edge is contained in the same path as edge $e' \in E_{i+1}$ with $y(e) = (|E_{i+1}| + 1) - y(e')$. Case 2: $y(e) = |E_{i+1}| + 1$. The edge is in P_{i+1}. Case 3: $y(e) > (|E_{i+1}| + 1)$. The edge is contained in the same path as edge $e' \in E_{i+1}$ with $y(e) = (|E_{i+1}| + 1) + y(e')$. By induction (Statement 1) this proves Statement 1.

Towards a proof of Statement 2 we first consider the case $i < j$. By induction (Statement 2), $|C_{i+1}(P_j)| = 2^{j-(i+1)}$, and by Claim 1, each of the paths $p \in C_{i+1}(P_j)$ contains exactly one edge around t_i. Thus $|C_i(P_j)| = 2 \cdot 2^{j-(i+1)} = 2^{j-i}$, proving Statement 2 for $i < j$. We now consider $i = j$. By Claim 1, the path P_i does not contain an edge around t_i, thus $|C_i(P_i)| = 1 = 2^{i-i}$, which proves Statement 2 for $i = j$. q. e. d.

The ideas of the inductive proof of Lemma 6.6 are illustrated by Figures 6.2, 6.3, 6.4, and 6.5 (for $k = 4$). Figure 6.2 shows the basis of the induction at $i = 4$. Figure 6.3 shows the inductive step from $i + 1 = 4$ to $i = 3$, Figure 6.4 shows the step from 3 to 2, and Figure 6.5 the final step from 2 to 1.

Theorem 6.7. *Let $G, (s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$ be an instance of* DPP *embedded by a disc-width-edges embedding that has a solution $P_1, \ldots, P_k$. Let e be an edge around a terminal t. Let $l \in \mathbb{N}$ be the smallest integer, such that $y(e)$ is a multiple of 2^{k-l}. Then e belongs to the path P_l.*

Proof. This follows immediately from Lemma 6.6, Statement 1, by $t = t_i$. q. e. d.

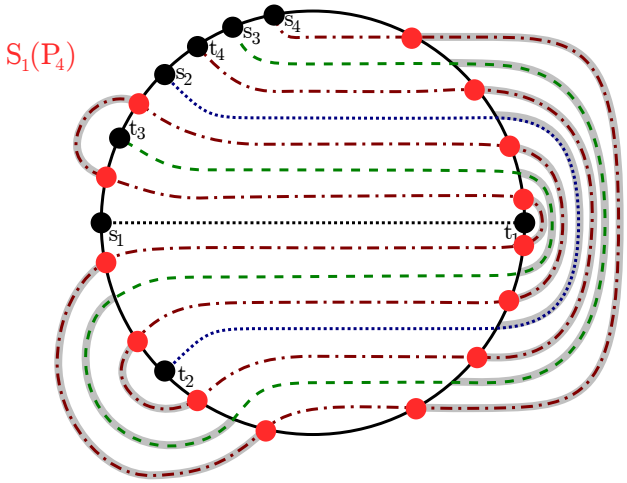


Figure 6.5: P_4, P_3, P_2, P_1.

Lemma 6.8. *Let $G, (s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$ be an instance of* DPP *embedded by a disc-width-edges embedding that has a solution $P_1, \ldots, P_k$. Let $p \in C_1(P_j)$ for some j. Then the endpoints of p are not endpoints of edges around the same terminal.*

Proof. This follows from Lemma 6.6, Statement 3. q. e. d.

In the following Section 6.2, we apply Theorem 6.7 and Lemma 6.8 to get the lower bound.

Remark 6.9. *In a* DPP *instance embedded by a disc-width-edges embedding, the number of edges around the terminals is crucial. Even just relaxing the conditions on disc-width-edges embeddings by having 2 edges instead of 1 around terminal t_{k-1} allows a quite different solution to the* DPP *(for a suitable H_G). This solution uses no edge around t_1, one edge around each of $t_2, t_3, \ldots, t_{k-2}$, and the two edges around t_{k-1} (Figure 6.7 shows this for $k = 4$).*

We would like to thank Frédéric Mazoit for helpful discussions and valuable input, especially for inspiring the above Remark 6.9.

6.2 Linkages

We now define a sequence $(G_k)_{k \geq 2}$ of graphs (with k pairs of terminals each), and show that they have a unique solution to the DPP and $\mathrm{tw}(G_k) \geq 2^k - 1$.

Definition 6.10. *For $k \in \mathbb{N} \setminus \{0, 1\}$ we define an instance $G_k, (s_1, t_1), \ldots, (s_k, t_k)$ of* DPP *as follows:*

- *Let H_k be the $((2^k - 1) \times (2^k - 1))$ grid.*
- *For $i \in [k]$, let $s_i := (2^{k-i}, 1) \in V(H_k)$.*

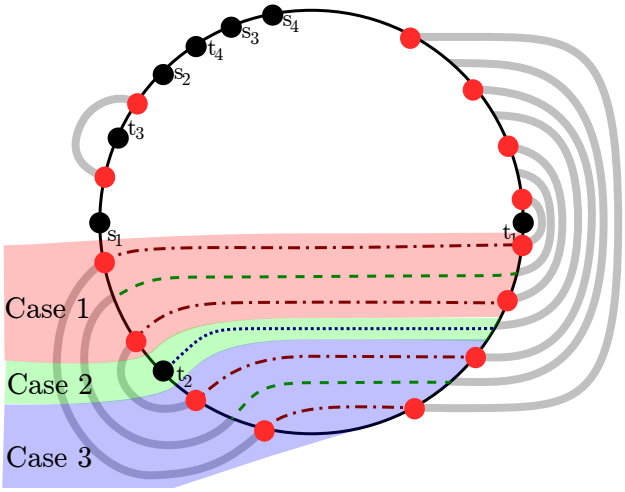


Figure 6.6: Cases in the proof Lemma 6.6, Statement 1 for $i = 1, k = 4$.

- *Let* $t_1 := (2^{k-1}, 2^k - 1)$, *and for* $i \in [k] \setminus \{1\}$ *let* $t_i := (3 \cdot 2^{k-i}, 1)$.
- *For* $i \in [k]$, *let* $E_i := \Big\{\{t_i - (y, 0), t_i + (y, 0)\} \mid y \in [2^{k-i} - 1]\Big\}$.
- $G_k := (V(H_k), E(H_k) \cup \bigcup_{i=1}^{k} E_i)$.

Note that the terminals also depend on k, which we omit for notational convenience. The graph G_2 is a supergraph of the 3×3 grid H, and a solution for $G_2, (s_1, t_1), (s_2, t_2)$ is shown in Figure 6.9. A larger example can be found in Figure 6.8, which shows $G_4, (s_1, t_1), (s_2, t_2), (s_3, t_3), (s_4, t_4)$.

Remark 6.11. *The instance* $G_k, (s_1, t_1), \ldots, (s_k, t_k)$ *of* DPP *has a disc-with-edges embedding with* $H_{G_k} = H_k$.

From now on, we assume G_k is embedded by this disc-width-edges embedding.

Lemma 6.12. *Let* $k \in \mathbb{N} \setminus \{0, 1\}$. *The instance* $G_k, (s_1, t_1), \ldots, (s_k, t_k)$ *of* DPP *has a solution that uses all horizontal edges in* H_{G_k}, *but none of the vertical edges in* H_{G_k}.

Proof. For each $i \in [k]$, we inductively define a subgraph $P'_i \subseteq G_k$ that uses horizontal edges, but no vertical edges. We then show that P'_i contains a solution path P_i from s_i to t_i.

- Let P'_1 be the graph induced by the vertex set $\{(2^{k-1}, x) \mid x \in [2^k - 1]\}$.
- Let P'_{i+1} be the graph induced by the vertex set

$$\{(y, x) \mid (y + 2^{k-(i+1)}, x) \in V(P'_i) \text{ or } (y - 2^{k-(i+1)}, x) \in V(P'_i)\}.$$

Claim 1: Let $i \in [k]$.

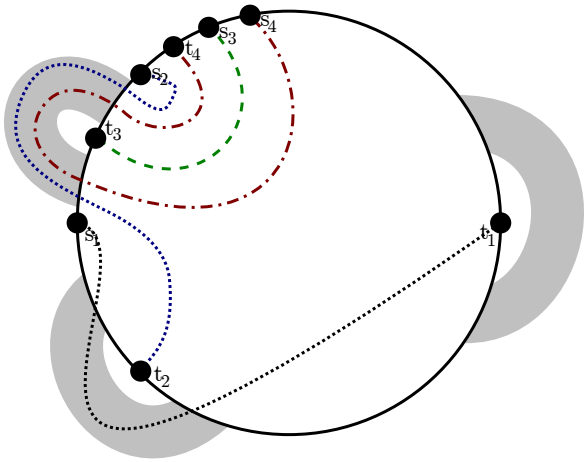


Figure 6.7: The number of edges around the terminals is crucial (cf. Remark 6.9).

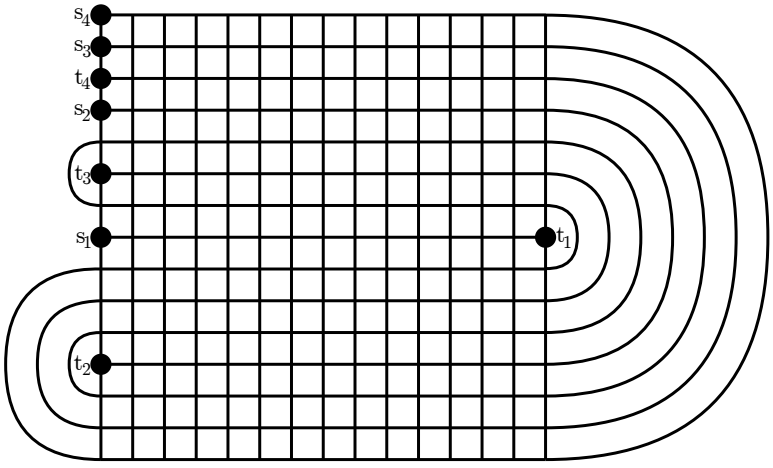


Figure 6.8: $G_4, (s_1, t_1), (s_2, t_2), (s_3, t_3), (s_4, t_4)$.

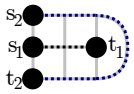


Figure 6.9: The instance $G_2, (s_1, t_1), (s_2, t_2)$ with a solution.

(i) $s_i \in P'_i, t_i \in P'_i$.

(ii) Let $j \in [k]$, $j \neq i$. Then P'_i is disjoint from P'_j.

(iii) Let $y \in [2^k-1], x \in [2^k-1] \setminus \{1, 2^k-1\}$. Then $(y, x) \in V(P'_i) \Leftrightarrow (y, x-1) \in V(P'_i) \Leftrightarrow (y, x+1) \in V(P'_i)$.

(iv) Let $(y, x), (y', x) \in V(P'_i)$. Then $|y - y'|$ is a multiple of 2^{k-i+1}.

Proof of the Claim: (i) is true by definition for $i = 1$. Let (i) be true for $i < k$. Then it is true for $i + 1$: Let $(y, x) := s_i, (y', x) := s_{i+1}, (y'', x) := t_{i+1}$. Then $|y' - y| = |y'' - y| = 2^{k-(i+1)}$. By the inductive hypothesis $s_i \in V(P'_i)$, thus $s_{i+1}, t_{i+1} \in V(P'_{i+1})$.

(ii) Let $i < j$ and $(y, x) \in V(P'_i)$. Then y is a multiple of 2^{k-i} and thus a multiple of $2^{k-(j+1)}$. But for any $(y', x) \in V(P'_j)$, y' is a multiple of $2^{k-(j+1)}$ plus or minus 2^{k-j}, and thus not a multiple of $2^{k-(j+1)}$.

(iii) holds by definition for $i = 1$. Let (iii) be true for i. Then it holds for $i + 1$: Let $y \in [2^k - 1], x \in [2^k - 1] \setminus \{1, 2^k - 1\}$. Then $(y, x) \in V(P'_{i+1})$ if and only if there is an $(y', x) \in V(P'_i)$ such that $|y - y'| = 2^{k-(i+1)}$. By the inductive hypothesis

$$(y', x) \in V(P'_i) \Leftrightarrow (y', x - 1) \in V(P'_i) \Leftrightarrow (y', x + 1) \in V(P'_i),$$

and thus

$$(y, x) \in V(P'_{i+1}) \Leftrightarrow (y, x - 1) \in V(P'_{i+1}) \Leftrightarrow (y, x + 1) \in V(P'_{i+1}).$$

⊣

(iv) is true by definition for $i = 1$. Let (iv) be true for i. Then it is true for $i + 1$, since for every $(y, x) \in V(P'_{i+1})$ there is a $(y', x) \in V(P'_i)$ such that $|y - y'| = 2^{k-(i+1)}$.

Claim 2: Let $i \in [k]$. Let $v = (y, x)$ be a vertex in P'_i.

(i) The vertex v is either a terminal and $d_{P'_i}(v) = 1$ or it has degree $d_{P'_i}(v) = 2$.

(ii) If $v \in e, e \in E_j$ for some $j \in [k]$, $t_j = (y', x)$ then $|y - y'| \leq \sum_{l=1}^{i-j} 2^{k-j-l}$.

Proof of the Claim: We use induction on i. For $i = 1$ the claim is true by the definition of P'_1.

Assume that the claim holds for $i < k$. We show that it holds for $i + 1$.

Let $(y, x) \in V(P'_{i+1})$. If $1 < x < 2^k - 1$, then $d_{P'_i}((y, x)) = 2$ by Claim 1 (iii, iv).

If $x \in \{1, 2^k - 1\}$, then by definition of P'_{i+1} there is a vertex $(y', x) \in P'_i$ such that $|y - y'| = 2^{k-(i+1)}$:

Case 1: $(y', x) = t_i$. Then $2^{k-(i+1)} \leq 2^{k-i} - 1$ and (y, x) is an endpoint of an edge $\{(y, x), (y'', x)\} \in E_i$ and $|y'' - y'| = |y - y'| = 2^{k-(i+1)}$, thus $(y'', x) \in V(P'_{i+1})$ and combined with Claim 1 (iii) $d_{P'_i}((y, x)) = 2$, proving (i). Moreover, $|y - y'| = 2^{k-(i+1)} = \sum_{l=1}^{1} 2^{k-i-l} = \sum_{l=1}^{(i+1)-i} 2^{k-i-l}$, proving (ii).

Case 2: $(y', x) = s_i$. Then $(y, x) \in \{s_{i+1}, t_{i+1}\}$ and (i) holds using Claim 1 (iii). (ii) trivially holds.

Case 3: $(y', x) \notin \{t_i, s_i\}$. By inductive hypothesis (i), $d_{P'_i}((y', x)) = 2$ and thus $(y', x) \in e'$ for some $e' \in E_j$ for some $j \in [k]$. Let $(y_j, x) := t_j$. By inductive hypothesis (ii), $|y' - y_j| \leq \sum_{l=1}^{i-j} 2^{k-j-l}$. Then

$$|y - y_j| \leq \sum_{l=1}^{i-j} 2^{k-j-l} + 2^{k-(i+1)} \leq \sum_{l=1}^{(i+1)-j} 2^{k-j-l} \leq 2^{k-j} - 1.$$

Thus $(y, x) \in e$ for some $e \in E_j$ and (ii) holds. Let $e = \{(y, x), (y'', x)\}$ and $e' = \{(y', x), (y''', x)\}$. Since $|y - y_j| = |y'' - y_j|$ and $|y' - y_j| = |y''' - y_j|$ we get $|y'' - y'''| = |y - y'| = 2^{k-(i+1)}$, thus $(y'', x) \in P'_{i+1}$ and combined with Claim 1 (iii) $d_{P'_{i+1}}((y, x)) \geq 2$. From Claim 1 (iv) it follows that $d_{P'_{i+1}}((y, x)) \leq 2$, thus $d_{P'_{i+1}}((y, x)) = 2$. ⊣

By Claim 1 (i), $s_i, t_i \in P'_i$ and by Claim 2 (i), all vertices in P'_i have degree 2, except for the terminals, which have degree 1. Thus each P'_i contains a path $P_i \subseteq P'_i$ that connects s_i to t_i. By Claim 1 (ii), the graphs $P'_i, i \in [k]$, are pairwise disjoint. Thus $P_1, \ldots, P_k$ is a solution of the DPP instance $G_k, (s_1, t_1), \ldots, (s_k, t_k)$. By Theorem 6.7.1, the solution uses all edges in $\bigcup_{i=1}^{k} E_i$ and hence $P_i = P'_i$. With Claim 1 (iii, iv) it follows that the solution uses all horizontal edges in H_{G_k}. By construction it uses none of the vertical edges in H_{G_k}. q. e. d.

Figures 6.9 and 6.10 show the solution to the instances $G_2, (s_1, t_1), (s_2, t_2)$ and $G_4, (s_1, t_1), \ldots, (s_4, t_4)$, respectively.

Theorem 6.13. *Let $k \in \mathbb{N} \setminus \{0, 1\}$. There is exactly one solution to the instance $G_k, (s_1, t_1), \ldots, (s_k, t_k)$ of* DPP*, and the solution paths use all vertices of G_k.*

Proof. By Lemma 6.12, there is a solution $P_1, \ldots, P_k$ that uses exactly the horizontal edges in G. For $y \in [2^k - 1]$, let Q_y denote the path that consists of exactly the edges $\{(y, x), (y, x + 1)\}, x \in [2^k - 2]$. This is a subpath of some path of the solution. Towards a contradiction, suppose there is a different solution $P'_1, \ldots, P'_k$. From Theorem 6.7 and Lemma 6.8 it follows that for each $y \in [2^k - 1]$ there is a subpath Q'_y of some path of the solution $P'_1, \ldots, P'_k$, such that the endpoints of Q'_y are $(y', 1)$ and $(y, 2^k - 1)$ for some y'. Since $P'_1, \ldots, P'_k$ is different from $P_1, \ldots, P_k$, at least one path Q'_y contains a vertical edge $e = \{(y, x), (y+1, x)\}$ for some $x \in [2^k - 1]$. Let G'_k be the subgraph of G_k induced by $V(G_k) \setminus \{(y, x), (y+1, x)\}$. In G'_k, the set $S := \{(y', x) \mid y' \in [2^k - 1] \setminus \{y, y+1\}\}$ separates $X := \{(y', 1) \mid y' \in [2^k - 1]\}$ from $Y := \{(y', 2^k - 1) \mid y' \in [2^k - 1]\}$. The paths $Q'_{y'}, y' \in [2^k - 1] \setminus \{y\}$ are pairwise vertex-disjoint paths between X and Y, and there are $2^k - 2$ such paths. But $|S| = 2^k - 3$, a contradiction to Menger's Theorem [98]. q. e. d.

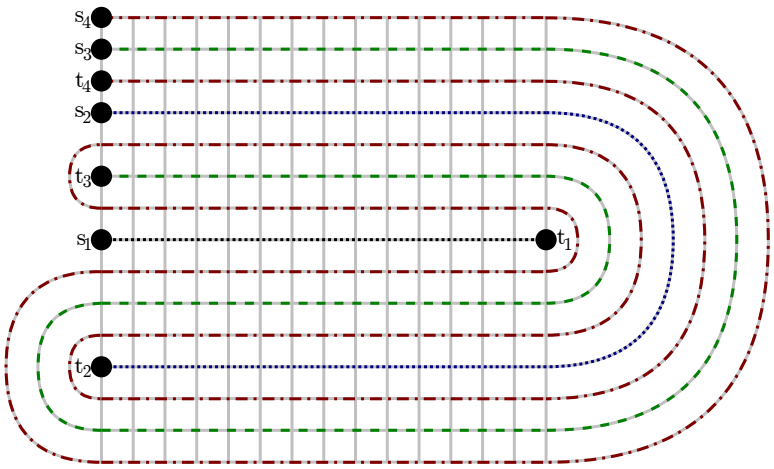


Figure 6.10: $G_4, (s_1, t_1), (s_2, t_2), (s_3, t_3), (s_4, t_4)$ with solution.

Proof of Theorem 6.1 From Theorem 6.13 and the fact that for every $k \in \mathbb{N} \setminus \{0, 1\}$, the planar graph G_k has a $((2^k - 1) \times (2^k - 1))$ grid H_k as a subgraph, it follows that $\mathrm{pw}(G_k) \geq \mathrm{tw}(G_k) \geq \mathrm{tw}(H_k) = 2^k - 1$. q. e. d.

Part III

Compilers

Chapter 7

Structured Programs and the Tree-width of C[1]

Recently, approaches based on tree-decompositions of control-flow graphs have been introduced for many classical problems in compiler construction [132, 18, 7, 83, 84], some of which have proven their practical usefulness in implementations in SDCC [41], a mainstream C compiler for embedded systems. In SDCC, Thorup's heuristic [132] is used to obtain the tree-decompositions of the control-flow graph. Thorup claims that his heuristic will give tree-decompositions of width at most 6 for C code that contains no `goto` statements.

We construct C code that contains no `goto` statements, for which the control-flow graph has tree-width 7. We can construct C code that contains no `goto` statements, for which the control-flow graph has tree-width 2, but Thorup's heuristic will yield tree-decompositions of arbitrarily large width. The issues with Thorup's heuristic also result in tree-decompositions of much higher than necessary width in real-world code. Since the width of the tree-decomposition is crucial for runtime or result quality in algorithms, better ways to obtain tree-decompositions of the control-flow graph are needed.

We present a constructive proof showing the tree-width of control-flow graphs of C programs is at most $7 + \mathtt{g}$, if the number of labels targeted by `goto` per function does not exceed $\mathtt{g}$. We also prove the tightness of our bound. This corrects Thorup and shows the effect that `goto` has on the tree-width. We empirically evaluate various approaches to finding tree-decompositions of control-flow graphs on many real-world C code examples, such as the standard library, standard benchmarks and operating systems. We also investigate the impact the different approaches have on compiler runtime and code quality in SDCC. We find that empirically, generic approaches to obtaining tree-decompositions of graphs [20, 19] are better than Thorup's heuristic. The combination of a preprocessor that uses reduction rules followed by a generic heuristic performs best. This results in shorter compiler runtime at comparable code quality.

[1] This chapter is based on joint work with Lukas Larisch, which has been submitted for publication [87].

7.1 Structured Programs

Definition 7.1 (Program)**.** *A program consists of a directed graph G, called the control-flow graph (CFG) of the program.*

This representation can be easily generated from other representations, such as pseudo-code. The nodes of the CFG are the program's instructions; there is an edge from i to j, if and only if there is some execution of the program where instruction j is executed directly after instruction i. Figure 7.8 (a) and (b) give an idea about how the CFG can correspond to code. In the following chapters we will often use variants of the above definition of a program, which contain additional information (e. g. weight functions for the nodes and edges of the CFG).

Definition 7.2 (Structured program)**.** *Let $\mathfrak{k} \in \mathbb{N}$ be fixed. A program is called $\mathfrak{k}$-structured if its control-flow graph has tree-width at most $\mathfrak{k}$.*

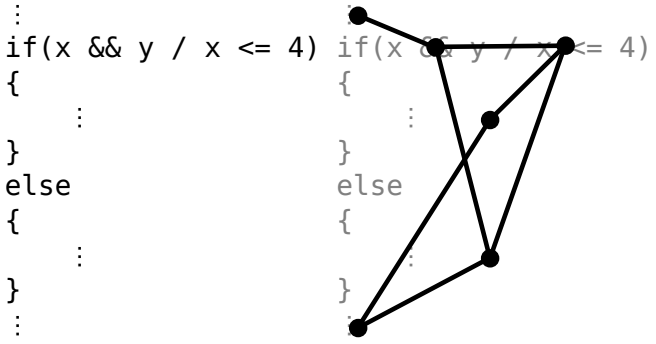
Different notions of "structured program" exist. Many of them, including famous ones, are rather informal or not directly related to the tree-width of control-flow graphs [79]. We now take a closer look at some formal definitions of "structured program". Under an early one, it meant programs using only if/else and while as control structures [24]. These are a subclass of the programs that have a series-parallel control-flow graph, and later "structured program" has been used to denote all programs that have a series-parallel control flow graph [70]. Some define "structured program" as a program that does not contain `goto` statements [18]. More recently, it denotes programs that are $\mathfrak{k}$-structured for a $\mathfrak{k}$ fixed in advance [84, 83]. Programs that have a series-parallel control flow graph are 2-structured, and programs that use only if/else and while as control structures have series-parallel control-flow graphs.

Programs written in Algol or Pascal are $2 + \mathfrak{g}$-structured, if the number of labels targeted by `goto` statements per function does not exceed $\mathfrak{g}$, Modula-2 programs are 5-structured [132]. Java programs are $(6 + \mathfrak{g})$-structured if the number of labels targeted by labeled breaks and labeled continues per function does not exceed $\mathfrak{g}$ [58]. Ada programs are $(6 + \mathfrak{g})$-structured if the number of labels targeted by `goto` and labeled loops per function does not exceed $\mathfrak{g}$ [23]. Programs written in C are $(7 + \mathfrak{g})$-structured if the number of labels targeted by `goto` per function does not exceed $\mathfrak{g}$ (see Section 7.3). Coding standards tend to place further restrictions, resulting e. g. in C programs being 5-structured when adhering to the widely adopted MISRA-C:2004 [1] standard (see Section 7.3).

A survey looking at 12522 Java methods from applications and the standard library using Thorup's heuristic found width above 3 to be very rare. With one exception (of width 5) for all methods Thorup found decompositions of width 4 or lower, the average width was about 2.7 [58]. In Section 7.5, we find similar results for C, but with higher maximum widths when using Thorup, and substantially lower average width when using methods other than Thorup's.

C is the most common programming language and has been so for a long time. Over time it has been updated, with the 1990, 1999 [66] and 2011 [67] ISO standards being the most important. C code consists of statements, which, mostly, are executed in sequence. However, there are a few statements, that allow execution to occur in a different order, and thus allow control-flow graphs that are not paths. These are the *selection statements* `if`, `else` and `switch`,

the *iteration statements* `while`, `do` and `for` and the *jump statements* `goto`, `continue`, `break` and `return`. There is one more aspect of C that can give rise to complex control-flow graphs in some composite logical expressions: Expressions of the form $e_l \circ e_r$ where $\circ$ is either the logical and `&&` or the logic or `||`. C has short-circuit evaluation: For `&&`, the subexpression e_r is never evaluated, if e_l evaluates to zero. For `||`, the subexpression e_r is never evaluated, if e_l does not evaluate to zero. Figure 7.1 illustrates the control-flow resulting from the use of short-circuit evaluation in the controlling expression of an `if/else` statement to avoid divisions by 0.



(a) C Code (b) CFG

Figure 7.1: Short-circuit evaluation

The earliest result on tree-with bounds depending of programming languages was by Thorup [132]. It also was the first application of graph-structure theory in compiler construction. For C, it states that programs written in C are $(6 + \mathtt{g})$-structured if the number of labels targeted by `goto` per function does not exceed `g`; an algorithm that was supposed to obtain the corresponding tree-decompositions was given as well. However, both the bound and algorithm are flawed. In the proof of the bound there was a mistake regarding the C `switch` statement, which can be exploited to create C programs of higher tree-width, as we do in Section 7.4. The algorithm gives tree-decompositions of arbitrarily large width when applied to some C programs containing nested `if/else` statements, even though these programs do not contain `goto` statements and have small constant tree-width, as shown in Section 7.2.

We give the correct bound and a working algorithm in Section 7.3, and prove the tightness of our bound in Section 7.4.

There are various approaches to obtaining tree-decompositions of control-flow graphs. An important theoretical result states that for every $k \in \mathbb{N}$, there is a linear time algorithm that tests if a given graph G has tree-width at most k, and if so, outputs a tree-decomposition of G with width at most k [17]. Unfortunately, it is impractical [120]. A commonly used algorithm is Thorup's [132], which unfortunately, is flawed, as mentioned above and shown in Section 7.2. Section 7.3 presents an algorithm for C code that could be easily generalized to other programming languages. For small tree-width, there are efficient exact algorithms [19]. There also are good heuristic algorithms [20]. We empirically evaluate them on control-flow graphs in Section 7.5.

7.2 Thorup's algorithm

The classic approach to obtaining tree-decompositions of control-flow graphs is Thorup's algorithm [132].

```
ThorupE(I)
{
  M = ∅;
  s = 0;
  (i_0, j_0) = (0, n+1);
  for (i = 1; i ≤ n; i++)
    if (There is j > i : (i, j) ∈ I)
    {
      j = max{j | (i, j) ∈ I};
      while (j_s ≤ i)
      {
        M = M ∪ {(i_s, j_s)};
        s--;
      }
      while (j ≥ j_s > i)
      {
        i = i_s;
        s--;
      }
      s++;
      (i_s, j_s) = (i, j);
    }
  return (M);
}
```

Figure 7.2: Thorup's algorithm E for finding maximal I-chains

Let $n \in \mathbb{N}$ and $I \subseteq [n]^2$. An *I-chain* from i to j is a sequence of pairs $(i_1, j_1), \ldots, (i_l, j_l) \in I$ such that for all $k < l$ holds: $i_k < i_{k+1} < j_k < j_{k+1}$. An I-chain from i to j is *maximal*, if there is neither an I-chain from some $i' < i$ to j nor an I-chain from i to some $j' > j$.

Thorup presents an algorithm for finding maximal I chains given I (Figure 7.2). It can be easily fixed by in the end adding all pairs left on the stack (except for $(0, n+1)$) to M.

Given a sequence $s_1, \ldots, s_n$ of statements in iCode let J be the set of pairs (i, j) such that s_i contains a jump to s_j. Let S be the symmetric closure of J. Thorup presents an algorithm (Figure 7.3) that uses maximal J-chains and maximal S-chains to find an elimination ordering for the control-flow graph (which then can be used directly or after transforming it into a tree-decomposition).

Thorup introduces a Modula-2-inspired toy programming language STRUCTURED and claims that "Although it is very technical it can be shown that the above heuristic will give" tree-decompositions of width at most 5 if applied to iCode generated "from a program with structural statements from STRUCTURED". He states "Intuitively maximal J-chains are used to bring us from

```
ThorupD()
{
  i = 0;
  for(j = n; j ≤ 0; j−−)
    if(There is j > i : (i, j) ∈ I)
    {
      if(s_j is not marked)
        Mark s_j by i++;
      if(s_j there is a maximal S−chain from
        k to j and s_k is not marked)
        Mark s_k by i++;
      if(s_j there is a maximal J−chain from
        k to j and s_k is not marked)
        Mark s_k by i++;
    }
}
```

Figure 7.3: Thorup's algorithm D for finding elimination orderings

the beginning to the end of a conditional structure." "Intuitively maximal S-chains are used to bring us from from loop or conditional structures to their exit points." This seems to work well as long as there are no (C) `return`, `break`, `continue` or `goto` statements in the program. STRUCTURED has equivalents to `return` and `break`.

However, by adding a few lines within maximal J-chains, every program can be changed so it has exactly one maximal J-chain and exactly one maximal S-chain and those two are the same. We can do this e. g. by inserting a conditional `return` (then for every previously existing maximal chain from i to j, there will now be a chain from i to the target of the `return` statement, thus the one from i to j is no longer maximal). Thus for the modified program, there is no useful information contained in the maximal J-chains and maximal S-chains, resulting in Thorup's algorithm giving tree-decompositions of arbitrarily large width even for classes of programs where the CFG has small constant tree-width. Figure 7.4 shows a program with a CFG of tree-width 2. By further nesting of the conditional statements we get a class of programs of tree-width 2 for which, after standard jump optimizations, Thorup's algorithm gives tree-decompositions of width linear in the nesting depth. Figure 7.5 shows a program with a CFG of tree-width 3. By further nesting of the conditional statements we get a class of programs of tree-width 3 for which Thorup's algorithm gives tree-decompositions of width linear in the nesting depth.

7.3 Upper bound and algorithm

When finding tree-decompositions for control-flow graphs of programs written in C, it is mostly sufficient to work at the statement level. The only exception are some composite logical expressions: Those where short-circuit evaluation matters. Short-circuit evaluation makes control-flow graphs more complex (without

```
int x0, x1, x2, x3, x4;
int a0, a1, a2, a3, a4;
int b0, b1, b2, b3, b4;

void c(void)
{
        if(a0)
        {
                if(a1)
                {
                        if(a2)
                        {
                                if(a3)
                                {
                                        x0++;
                                }
                                else
                                        b3++;
                        }
                        else
                                b2++;
                }
                else
                        b1++;
        }
        else
                b0++;
}
```

Figure 7.4: C function from a class of functions of tree-width 2 for which, after standard jump optimizations, Thorup gives arbitrarily large width

short-circuit evaluation, the bound on the tree-width of C would be lower by 1).

Lemma 7.3. *For every logical expression in C, we can find a tree-decomposition of width at most* 2 *in linear time. The logical expression is separated by the rest of the control-flow graph by a separator of size at most* 3*.*

Proof. Let e be a logical expression. The logical expression can be used in other expressions or as a controlling expression for `if`, `switch` or a loop. In the former case, execution will continue in a manner defined by the standard or implementation, no matter to which value the logical expression evaluates. In the latter case, execution will continue in a place depending on the value of the logical expression. Let π_t be the place where execution continues if the expression evaluates to `true`, let π_f be the place where executing continues if the expression evaluates to `false`. Within the logical expression there is some non-composite expression that is evaluated first. Let π be this non-composite expression.

```
int x0, x1, x2, x3, x4;
int a0, a1, a2, a3, a4;
int b0, b1, b2, b3, b4;

void s(void)
{
        if(a0)
        {
                if(a1)
                {
                        if(a2)
                        {
                                if(a3)
                                {
                                        if(x4)
                                                return;
                                }
                                else
                                        b3++;
                                if(x3)
                                        return;
                        }
                        else
                                b2++;
                        if(x2)
                                return;
                }
                else
                        b1++;
                if(x1)
                        return;
        }
        else
                b0++;
        if(x0)
                return;
}
```

Figure 7.5: C function from a class of functions that can be written in STRUCTURED of tree-width 3 for which Thorup gives arbitrarily large width

Then $\{\pi, \pi_t, \pi_f\}$ separates all the subexpressions of e from the rest of the control-flow graph. We thus create a bag $\mathfrak{B}_e = \{\pi, \pi_t, \pi_f\}$. If e is not a composite logical expression, we are done since $\pi = e$.

If e is a composite logical expression, let it w. l. o. g. be of the form e_l&&e_r (for $e_l \,||\, e_r$ we would get $\mathfrak{B}_{e_l} = \{\pi, \pi_t, \pi'\}$ below instead). We then recurse on e_l and e_r. We create a bag $\mathfrak{B} = \mathfrak{B}_e \cup \mathfrak{B}_{e_l} \cup \mathfrak{B}_{e_r}$, and connect bags $\mathfrak{B}_e, \mathfrak{B}_{e_l} = \{\pi, \pi', \pi_f\}$ and $\mathfrak{B}_{e_r} = \{\pi', \pi_t, \pi_f\}$ to it. $|\mathfrak{B}| \leq 4$. q. e. d.

```
b(π_t, π_f, b, e)
{
  Create a node b_e in the tree-decomposition;
  Add an edge between b_e and b;

  Let π be the non-composite subexpression of e that is evaluated first;

  Put π, π_t, π_f into the bag for b_e.

  if(e is not of the form e_l ∘ e_r, where ∘ is either && or ||)
    return π;

  Create a node b' in the tree-decomposition;
  Add an edge between b' and b_e;

  if(∘ is &&)
  {
    Let π' := b(π_t, π_f, b', e_r);
    b(π', π_f, b', e_l);
  }
  else // i.e. ∘ is ||
  {
    Let π' := b(π_t, π_f, b', e_r);
    return b(π_t, π', b', e_l);
  }

  Put π, π'π_t, π_f into the bag for b';
  return π;
}
```

Figure 7.6: Algorithm for finding a tree-decomposition for a logical expression

Figure 7.6 shows the algorithm just described in pseudocode.

Theorem 7.4. *For every C function, that contains at most* $\mathfrak{g}$ *labels targeted by* `goto` *we can find a tree-decomposition of width at most* $\mathfrak{k} = 7 + \mathfrak{g}$ *in linear time. When adhering to the required rules of MISRA-C:2004, the width is at most* 5.

Proof. For programs that do not contain any branches (i. e. the control-flow graph is a path), we can trivially create a tree-decomposition in linear time: Let $\pi_0, \ldots \pi_k$ be the nodes in the control-flow graph, then we can use a path as the tree for the tree-decomposition and bags $\{\pi_0, \pi_1\}, \{\pi_1, \pi_2\}, \ldots$; for the most part we can look at the C source at the statement level (the only exception is short-circuit evaluation). Individual expressions are evaluated in some order decided by the standard or implementation and thus only result in a subdivision of the path when looking at individual expressions instead of statements.

When we arrive at an iteration statement (`while`, `do while`, `for`) or a selection statement (`if`, `if else`, `switch`), special handling is required.

In each case, we generate the bag X containing the iteration / selection statement π_0 and the following statement, π_1 normally. π_1 is the one after the end of the loop, not one of the statements inside the loop. In case of an `if` or `if else` statement we set the current if-node to π_1. In case of a switch statement we set the current switch node to π_0 and the current break node to π_1. In case of a loop statement we set the current continue node to π_0 and the current break node to π_1. The bags for the loop body, `if` / `else` part or body of the `switch`

statement are added branching off from the bag X and include the current if-node, break-node, switch-node and continue-node if they exist. If the controlling expression of an `if`, `if else`, `switch` or loop statement is a logical expression we proceed slightly differently: Branching off from X we create another bag X', which is mostly identical to X, but does not include the nodes that will be replaced (thus there is at least one node in X that is not in X'. Instead it contains the separator that separates the logical expression from the rest of the CFG (it is the return value from $b()$ on the logical expression). By Lemma 7.3 this separator has size 3, one of the elements is π_0. Thus $|X'| \leq |X| + 1$ and the handling of short-circuit evaluation increases tree-width by at most 1.

In the end, we add all labels to all bags and if there is a return statement in the program, we add the return-node (the node targeted by return statements) to all bags.

Figure 7.7 shows the algorithm just described in pseudocode (for clarity the handling of short-circuit evaluation has been omitted).

This obviously results in bag sizes of at most $\mathfrak{k} + 1 = 8 + \mathfrak{g}$ (two nodes due to the sequence of statements, one for the handling of short-circuit evaluation, the if-node, switch-node, break-node and continue-node, the node for handling return statements and one for each label) and can be done in linear time. We will now argue that this process results in a tree-decomposition of the program: The condition that every node of G has to be in a bag is true by construction, as is the condition that each node of G induces a connected subgraph in the tree T. There only exist a few types of edges in the control-flow graph. The edges that result from one statement being subsequent to another are trivially covered by the bags. The edges resulting from short-circuit evaluation are covered by the construction in the special handling of logical expressions. Edges from `return` and `goto` statements are covered, since their targets are included in all bags. The edges from `break` and `continue` statements are covered, since the construction includes their targets in all bags from where they can be reached. The symmetric situation holds for edges due to `switch` statements: We include the originating node from the switch statement in every place that could potentially be reached by a case label from that switch statement. A similar argument holds for edges due to if and else statements.

The required rules of MISRA-C:2004 [1] result in a smaller upper bound: Rule 14.5 disallows the use of `goto`, so we do not need add the labels to the bags. Rule 14.4 disallows the use of `continue`, making it impossible to terminate a loop from within an inner compound statement; we can thus identify the continue-node and the if-node. Rules 15.1 and 15.2 restricts `switch` in a way that allows us to handle it like `if else`, allowing us to identify the if-node and the switch-node. Thus when adhering to the required rules of MISRA-C:2004, we can obtain tree-decompositions of width at most 5. q. e. d.

Figure 7.8 shows two examples of tree-decompositions obtained by our algorithm. In the left example (real-world code calculating the binomial coefficient) we have return-node π_{15}, if-nodes π_2, π_4, π_9, break-node π_{14} and continue-node π_{11}. In the right example (constructed to show how our algorithm handles nested control constructs) we have if-node π_3, break-nodes π_{11}, π_{10}, switch-node π_3 and continue-node π_0.

```
next(π)
{
  if(π is followed by a switch−, if−, if/else− or loop−body)
    return the statement after that body;
  else
    return the statement after π;
}

k(π0, π1, πif, πswitch, πbreak, πcontinue, b, πend)
{
  Create a node b′ in the tree−decomposition;
  Add an edge between b′ and b, if b ≠ ⊥;
  Put each of π0, π1, πif, πswitch, πbreak, πcontinue, that is not ⊥
  into the bag for b′

  If there is a switch−body between π0 and π1
  {
    Let π′1 be the first statement of that body, π′end the last;
    k(π0, π′1, πif, π0, π1, πcontinue, b′, π′end);
  }
  If there is an if−body between π0 and π1
  {
    Let π′1 be the first statement of that body, πend the last;
    k(π0, π′1, π1, πswitch, πbreak, πcontinue, b′, π′end);
  }
  If there is an else−body between π0 and π1
  {
    Let π′1 be the first statement of that body, π′end the last;
    k(π0, π′1, π1, πswitch, πbreak, πcontinue, b′, π′end);
  }
  If there is an loop−body between π0 and π1
  {
    Let π′1 be the first statement of that body, π′end the last;
    k(π0, π′1, π1, πswitch, π1, π0, b′, π′end);
  }

  if(π1 == πend)
    return;

  k(π1, next(π1), πif, πswitch, πbreak, πcontinue, b′, πend);
}

K()
{
  Let π1 be the first statement of the program;
  Let πreturn be the return node;

  k(π1, next(π1), ⊥, ⊥, ⊥, ⊥, ⊥, πreturn);

  Add πreturn into all bags if there is a return statement;
  Add all nodes targeted by goto statements into all bags;
}
```

Figure 7.7: Algorithm for finding tree-decompositions

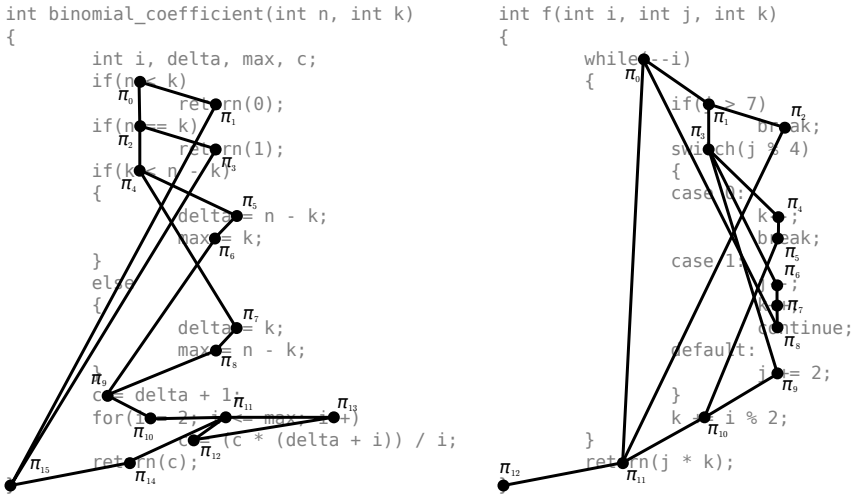
7.4 Tightness of bound

Theorem 7.5. *For every $\mathfrak{k} \geq 7$, there is a C function that has tree-width at least $\mathfrak{k}$ and at most $\mathfrak{g} = \mathfrak{k} - 7$ labels targeted by* `goto`.

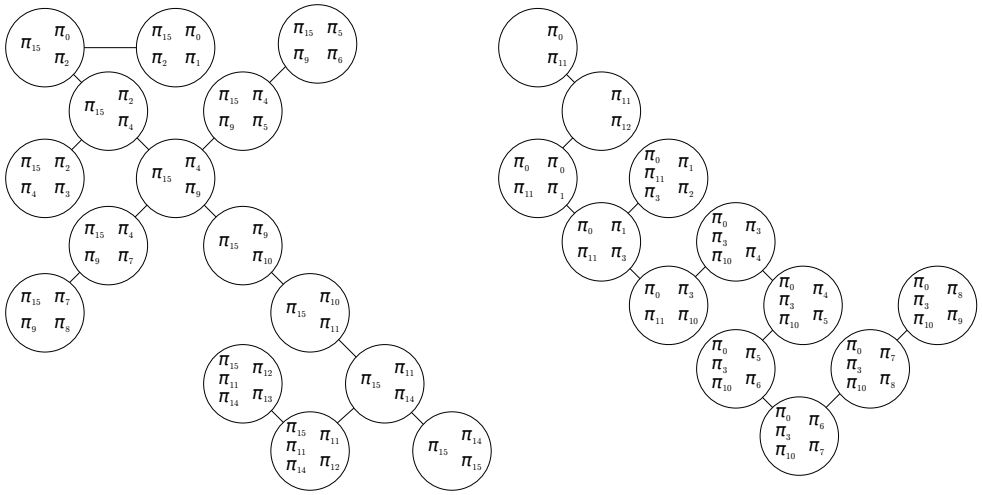
```
int binomial_coefficient(int n, int k)
{
        int i, delta, max, c;
        if(n < k)
                return(0);
        if(n == k)
                return(1);
        if(k < n - k)
        {
                delta = n - k;
                max = k;
        }
        else
        {
                delta = k;
                max = n - k;
        }
        c = delta + 1;
        for(i = 2; i <= max; i++)
                c = (c * (delta + i)) / i;
        return(c);
}
```

```
int f(int i, int j, int k)
{
        while(--i)
        {
                if(j > 7)
                        break;
                switch(j % 4)
                {
                case 0:
                        k--;
                        break;
                case 1:
                        j--;
                        k++;
                        continue;
                default:
                        j += 2;
                }
                k += i % 2;
        }
        return(j * k);
}
```

(a) Source code



(b) Control-flow graph



(c) Tree-decomposition found by the algorithm in Figure 7.7

Figure 7.8: Examples

Proof. We can construct a C function that has a K_8-minor. If a graph H is a minor of a graph G, $\mathrm{tw}(G) \geq \mathrm{tw}(H)$ [114]. The K_8 has tree-width 7, and thus the function has tree-width at least 7 (Figure 7.9). We can use this as the start of an induction:

We inductively assume that there is a C-function that has a $K_{\mathfrak{k}+1}$-minor and at most $\mathfrak{g}$ labels targeted by `goto`. We then add variables $x_{\mathfrak{g}+1,i}, i = 0, \ldots \mathfrak{k}$ and the following line at the start of the function: $l_{\mathfrak{g}+1}$`: if(`$x_{\mathfrak{g}+1,\mathfrak{k}}$`) return;` This line corresponds to the additional node we need to add to the $K_{\mathfrak{k}+1}$ to get a $K_{\mathfrak{k}+2}$. It already has edges to two nodes of the old $K_{\mathfrak{k}+1}$: The following one and the one targeted by return. We now need to add the necessary additional edges: For the nodes $n_i, i = 0, \ldots, \mathfrak{k}$ of the $K_{\mathfrak{k}+1}$ that do not have an edge to the new node, we insert the following line somewhere among the lines of their model: `if(`$x_{\mathfrak{g}+1,i}$`) goto` $l_{\mathfrak{g}+1}$`;` This obviously results in a C function that has a $K_{\mathfrak{k}+2}$-minor (and thus tree-width at least $\mathfrak{k}+1$) and at most $\mathfrak{g}+1$ labels targeted by `goto`. q. e. d.

7.5 Experimental results

We implemented various algorithms and evaluated them on control-flow graphs of various C functions. The control-flow graphs were obtained from the iCode in the SDCC compiler [41] by compiling the SDCC standard C library from the 3.5.0 SDCC release, a C version of the floating-point benchmark Whetstone [33], an ISO C version of version 2 of the integer benchmark Dhrystone [137, 138] and the integer benchmark Coremark [49], version 2.5 of the operating system Contiki [40] and the operating system FUZIX (from the git repository as of 2015-01-08).

The algorithms considered were Thorup's [132], heuristics [20], postprocessing rules [20, 15], and a set of exact reduction rules [19, 10, 134] for preprocessing which can reduce all graphs of tree-width up to 3 to the empty graph. We also looked into further approaches, such as the separator algorithm [116] and exact algorithms. We here present results for Thorup's algorithm, due to its current use in compiler construction, and for the three most promising alternatives: Preprocessing followed by a greedy degree-based heuristic, a greedy fill-in heuristic followed by triangulation minimization postprocessing, preprocessing followed by a greedy fill-in heuristic followed by triangulation minimization postprocessing. While we looked into some other approaches as well, those other approaches performed worse in terms of width of the decomposition compared to the approaches presented here, or had prohibitively high runtime. Besides looking into the runtime and the widths of the resulting tree-decompositions, we also looked into the overall effect on the compiler: The three benchmarks were compiled with SDCC, using the different approaches to obtaining tree-decompositions. In SDCC, tree-decompositions are used in redundancy elimination[82], placement of bank selection instructions[83] and register allocation[84]. Since a smaller width affects the tree-decomposition based algorithms in the compiler, we looked into the impact on total compiler runtime and code quality. We used SDCC version 3.5.0 with default options, targeting a variety of 8-bit architectures. For the STM8 target, we also executed the resulting binaries on an STM8/128-EVAL evaluation board to measure benchmark scores. The host system was running Debian GNU/Linux and has an Intel Core i7-4500U processor.

```
int x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15, x16, x17;

void g(void);

void f(void)
{
  switch(x0)
  {
  case 0:
    if(x1)
      return;
    while(x2)
    {
  case 1:
      if(x3)
        break;
      else if(x4)
        return;
      if(x5)
      {
        if(x6 || x7)
        {
  case 2:
          if(x8)
            break;
          else if(x9)
            return;
          else if(x10)
            continue;
        }
        else
        {
  case 3:
          if(x11)
            break;
          else if(x12)
            return;
          else if(x13)
            continue;
        }
      }
  case 4:
      if(x14)
        break;
      else if(x15)
        return;
      else if(x16)
        continue;
  case 5:
      if(x17)
        break;
    }
    g();
  default:
    ;
  }
}
```

Figure 7.9: C function with model of K_8 (branch sets are highlighted)

Figure 7.10 shows that all approaches except for the fill-in-heuristic without preprocessing perform similar in terms of algorithm runtime. The fill-in-heuristic approach without preprocessing and the min-degree-heuristic with preprocessing perform only slightly worse in terms of width of the found tree-decompositions compared to the fill-in-heuristic with preprocessing and triangulation minimization. Among the 1817 graphs there was only one for which the min-degree-heuristic with preprocessing and the fill-in-heuristic with preprocessing and triangulation minimization did not find a decomposition of minimal width (the found decomposition had width 2 higher than the tree-width for the min-degree-heuristic with preprocessing and width 1 higher than the tree-width for the fill-in-heuristic with preprocessing and triangulation minimization). All three heuristic approaches result in substantially lower width compared to Thorup's. In particular, in all benchmarks except for Whetstone, there was at least one control-flow graph for which Thorup's approach results in much higher width than the other approaches.

Figure 7.11 shows how this affects a compiler that uses tree-decompositions when replacing Thorup's approach by preprocessing followed by the fill-in heuristic followed by triangulation minimization. Neither one of the small benchmarks Whetstone and Dhrystone is affected much. However the situation is different for the bigger Coremark benchmark. We see a substantial reduction in compiler runtime for all target architectures. Since register allocation has a bigger impact on compiler runtime and code quality compared to other optimizations, we will discuss the results with respect to register allocation. The register allocator in SDCC does computations on a number of intermediate results at each node of the tree in the tree-decomposition. The number of intermediate results considered at each node is bounded by the argument to `--max-allocs-per-node`, with a default value of 3000. The number of intermediate results that need to be considered for provably optimal register allocation depends exponentially on the product of the number of registers and the width of the bag corresponding to the node.

This explains the results: Dhrystone and Whetstone are small and consist of functions of small tree-width. For most functions, the heuristic limit is never reached, since the number of intermediate results needed for provably optimal allocation is lower than the limit. This holds for both approaches to finding tree-decompositions, and thus these benchmarks are not affected much. In Coremark, for the default options, the different functions in the benchmark contribute relatively equally to the runtime, since for many of them when using Thorup's approach the number of intermediate results considered is bounded by the limit of 3000. By using a better approach to finding tree-decompositions, we have more nodes where we do not reach the limit of 3000, which results in a reduction of compiler runtime.

The impact on code size (Figure 7.11b) and benchmark scores (Figure 7.11c) is rather small, below 3% in all situations.

While doing the experiments, we found that due to the use of a heuristic in the register allocator, there are other aspects of the tree-decomposition besides the width that can have a substantial influence on code quality and compiler runtime. In particular, very branched tree-decompositions can make the register allocator perform worse. We found that the generic approaches to obtaining tree-decompositions tend to lead to more branched decompositions than Thorup's, which partially compensates the improvements from the lower width. We

conjecture that paying attention to aspects other than the width of the tree-decompositions can result in even bigger improvements than what we presented in this section, but further research is needed.

7.6 Conclusion

We have shown that the control-flow graphs of C programs have bounded tree-width, and that our bound is tight. We found that Thorup's heuristic is flawed, and also empirically approaches other than Thorup's heuristic result in tree-decompositions of lower width. After the 3.5.0 release of SDCC, Thorup's heuristic will be replaced by preprocessing followed by the fill-in heuristic and triangulation minimization.

package	#CFGs	avg width	max width	avg time[ms]
stdlib	142	1.852	4	0.063
Whetstone	7	1.429	2	0.214
Dhrystone	15	1.667	2	0.136
Coremark	42	1.643	3	0.117
Contiki	1082	1.714	7	0.068
FUZIX	529	1.966	6	0.066

(a) preprocessing + min-degree-heuristic

package	#CFGs	avg width	max width	avg time[ms]
stdlib	142	1.852	4	0.237
Whetstone	7	1.429	2	1.039
Dhrystone	15	1.667	2	0.995
Coremark	42	1.643	3	0.656
Contiki	1082	1.713	7	0.280
FUZIX	529	1.966	6	0.266

(b) fill-in-heuristic + triangulation minimization

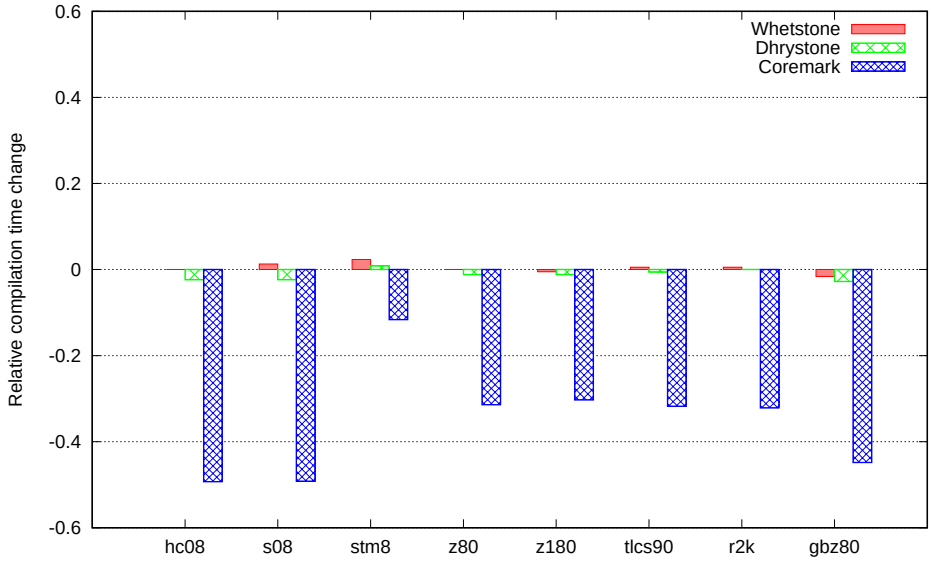
package	#CFGs	avg width	max width	avg time[ms]
stdlib	142	1.852	4	0.059
Whetstone	7	1.429	2	0.217
Dhrystone	15	1.667	2	0.137
Coremark	42	1.643	3	0.110
Contiki	1082	1.712	7	0.069
FUZIX	529	1.966	6	0.067

(c) preprocessing + fill-in-heuristic + triangulation minimization

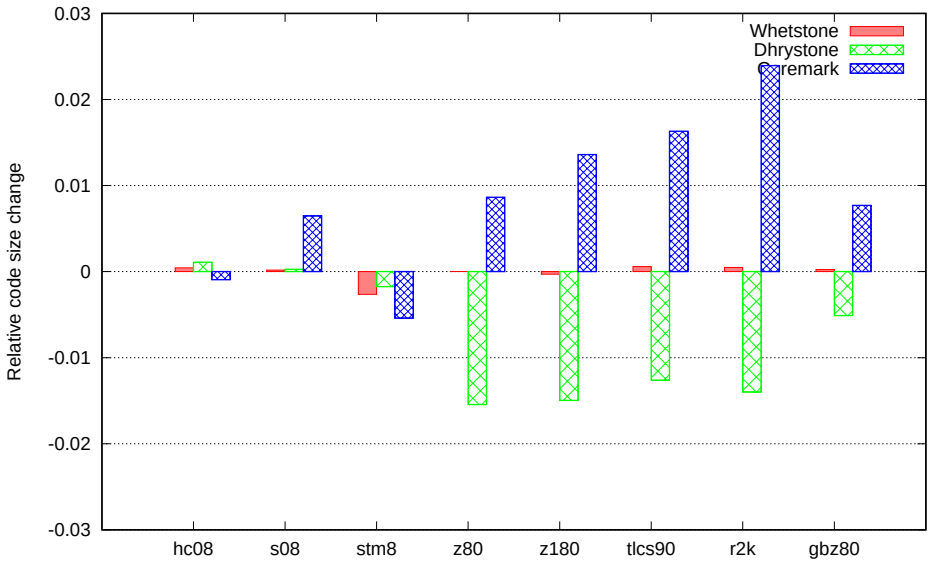
package	#CFGs	avg width	max width	avg time[ms]
stdlib	142	2.430	11	0.073
Whetstone	7	1.571	3	0.186
Dhrystone	15	2.200	7	0.157
Coremark	42	2.310	8	0.106
Contiki	1082	2.296	22	0.092
FUZIX	529	2.767	29	0.092

(d) Thorup's heuristic

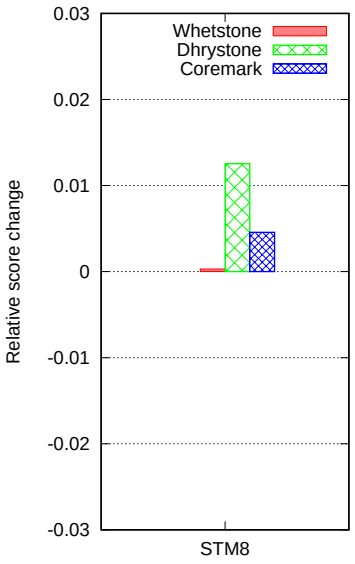
Figure 7.10: Experimental results: Graph processing



(a) Impact on compilation time



(b) Impact on code size



(c) Impact on benchmark scores

Figure 7.11: Experimental results

Chapter 8

Optimal Placement of Bank Selection Instructions in Polynomial Time[1]

We present the first approach to Optimal Placement of Bank Selection Instructions in Polynomial Time; previous approaches were not optimal or did not provably run in polynomial time. Our approach requires the input program to be structured, which is automatically true for many programming languages and for others, such as C, is equivalent to a bound on the number of `goto` labels per function. When not restricted to structured programs, the problem is NP-hard. A prototype implementation in a mainstream compiler for embedded systems shows the practical feasibility of our approach. Our approach and implementation are easy to re-target for different optimization goals and architectures.

8.1 Introduction

Placement of bank selection instructions is not a new problem; it is commonly encountered in both older architectures and current embedded systems. However, approaches implemented in compilers have mostly been ad-hoc. The simplest and least-efficient approach inserts a bank switching instruction before every access to a memory location in a memory bank. An obvious improvement is leaving out the switching instruction if a switching instruction to the same memory bank has been inserted before and no other bank switching instruction, function call or jump target can be found in between. The latter approach runs in linear time, and is optimal for straight-line code.

Even when just optimizing for code size, optimal placement of bank selection instructions is an NP-hard problem when not restricted to structured programs [91].

One approach by Scholz et alii, that can take into account different optimization goals such as code size or energy consumption, formulates a discrete

[1] Previously presented at M-SCOPES 2013 [83].

optimization problem and solves it using Partitioned Boolean Quadratic Programming (PBQP) [122, 123]. PBQP is NP-hard, and there is no known polynomial bound on the runtime.

There is an approach by Liu et alii, that handles bank selection instruction placement and variable partitioning at the same time [93]. It runs in polynomial time, but offers no guarantee on the quality of the results.

A more recent approach for acyclic control-flow graphs by Li et alii runs in polynomial time, and gives a guarantee on the result quality when there are no transparent basic blocks (basic blocks that do not contain accesses to banked memory) and some further restrictions apply (it then is a 2-approximation when optimizing for code size) [91]. The requirement that there are no transparent basic blocks is a significant restriction, considering that many architectures support both some banked and some non-banked memory at the same time, and placing some variables in the small memory that requires no bank selection can significantly improve code [99].

Our approach is based on the bounded tree-width of the control-flow graphs of structured programs (see Chapter 7).

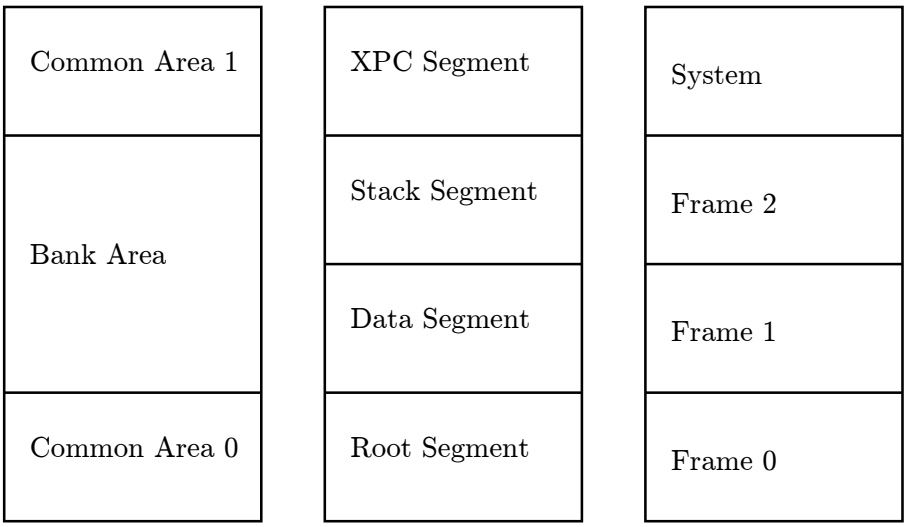
We present an algorithm that places bank selection instructions optimally. Optimality is defined with respect to a cost function, such as code size, speed, energy consumption or an aggregate thereof. Our algorithm has polynomial runtime (when the number of memory banks is bounded it even becomes linear) and in practice the time spent in it is negligible compared to the rest of the compiler, as witnessed by a prototype implementation in a C compiler for embedded systems. Approach and implementation can be easily re-targeted for different architectures and optimization goals, such as code size, performance, energy consumption or aggregates thereof.

The placement of connection instructions in Register Connection [76] is a closely related problem, where currently simple ad-hoc approaches are used; with minor changes our approach could be applied there, too.

8.2 Problem Description

Partitioned memory architectures are common in 8- and 16- bit microcontrollers. A part of the logical address space is used as a window into a larger physical address space. The parts of the physical address space that can be mapped into the window are called *memory banks*. There is some mechanism to select which part of the physical address space is visible in the window. It is used by means of bank selection instructions. We will briefly describe three typical examples. They all use a 16 bit logical address space to access a 20-bit physical address space.

The Z180 is a classic processor once used in general-purpose computers, now mostly used in embedded systems. It partitions the logical address space into three areas (Figure 8.1a): The Common Area 0, which is always directly mapped to the same physical addresses in the larger address space, and two areas that can be mapped to different parts of the physical address space. The size of the areas is configurable, both the size and upper address bits are set by writing to I/O registers. Typically the Common Area 0 is used for startup code, and data that should be accessed quickly, such as the stack. The Bank Area can then be used to store more variables (accessing those will require the use of bank



(a) Z180 (b) Rabbit 3000 (c) SMS

Figure 8.1: Examples of partitioned memory architectures

selection instructions in the form of writing the I/O register) and the Common Area 1 can be used for code.

The Rabbit 3000 is a microcontroller designed for embedded systems. It has one more area (Figure 8.1b). Again they can be configured by writing I/O registers. However the XPC segment is special: Its size is fixed and there are special instructions to set the upper address bits during jumps and calls, which improves efficiency when using this area for code.

The video game console Sega Master System (SMS) uses an ordinary Z80 processor. Neither the processor nor the system include hardware support for the paged memory architecture. This hardware is included in the video game cartridges, and can thus vary. Figure 8.1c shows a typical example. There are three configurable areas of fixed size. The upper address bits are set by writing to a special memory location. The fourth area is used to access system RAM at a fixed location.

A less typical architecture example are the low- and mid-range devices among the Microchip PIC microcontrollers. Depending on the device type e. g. an 7-bit logical address space is used to access a 9-bit physical address space. The upper address bits are set by bit-manipulation instructions. This means that the cost for switching between different memory banks depends on the Hamming distance between their binary representations. Since these devices are very common, it is important that approaches to the placement of bank selection instructions are able to take this quirk into account.

In all these architectures, typically one area is used for global variables that require bank selection instructions, while the other areas are used for different purposes, such as the stack or code. In particular, there practically always is a separate area for code, which means we can consider the bank selection problem independent of the problem of where in memory code is placed.

The placement of variables in memory banks is typically done by the pro-

grammer (e. g. using named address spaces in Embedded C) or the compiler (e. g. using a bin-packing heuristic to minimize RAM usage). There are some approaches combining the placement of variables in memory banks and the placement of bank selection instructions [93]. But embedded systems tend to have more space for code than for data, and variables placed in banked memory are often big, which makes it attractive to optimize for good packing of variables into the banks first and for other aspects, such as code size and speed later. So typically the placement of variables in memory happens in an earlier stage of the compiler than the insertion of bank-switching instructions.

The following formal definition of a program will help us tackle the placement of bank selection instructions. We use the individual instructions as nodes of our control-flow graph; using basic blocks instead would slightly lengthen the presentation of our algorithm and the proofs, but would not make any substantial difference.

Definition 8.1 (Program)**.** *Let $\mathfrak{B}$ be the set of memory banks, including a special symbol $\bot \in \mathfrak{B}$ that represents that the currently selected bank is unknown. A program consists of a control-flow graph $G = (\Pi, E), E \subseteq \Pi^2$, which is partially colored: Each node n of G can be colored by a $\mathfrak{b} \in \mathfrak{B}$.*

Typically $\mathfrak{B}$ consists of all memory banks accessed in the program and $\bot$. Figure 8.2a shows a small example program, with three nodes colored by $\mathfrak{b} \in \mathfrak{B} \setminus \{\bot\}$, four nodes colored by $\bot$, and the remaining 8 nodes uncolored.

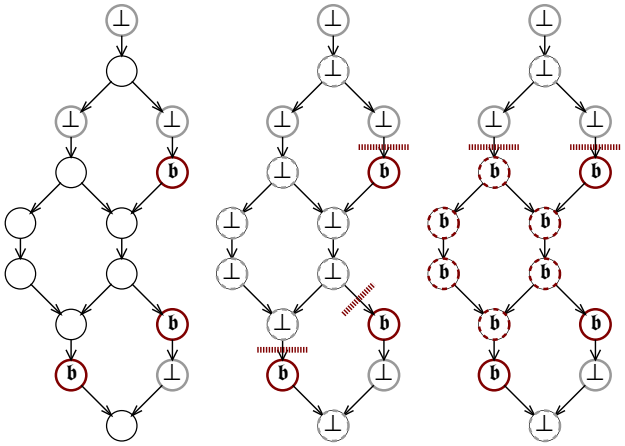
For an edge e in a directed graph we denote the source of e by e_0 and the target of e by e_1.

We color node n by $\mathfrak{b} \in \mathfrak{B} \setminus \{\bot\}$ to indicate that bank $\mathfrak{b}$ needs to be active at instruction n. We color node n by $\bot$ to indicate that the selected bank at n is unknown (e. g. at the start of a function or when calling another function that might set the active bank). The other nodes are uncolored (called *transparent nodes*), signifying that the instruction does not access banked memory.

Definition 8.2 (Cost Function)**.** *A cost function for a program $G = (\Pi, E)$ is a function $c\colon E \times \mathfrak{B} \times \mathfrak{B} \to \mathbb{R}$.*

$c(e, \mathfrak{b}_0, \mathfrak{b}_1)$ is the cost of having bank $\mathfrak{b}_0$ active at e_0 and having bank $\mathfrak{b}_1$ active at e_1. This potentially involves splitting the edge e in the control-flow graph and inserting a bank selection instruction switching between two given banks. Depending on the optimization goal the cost function can be code size for the inserted instructions, time to execute the instructions multiplied by relative execution frequency obtained from a profiler, etc or an aggregate thereof. A typical cost function when optimizing for code size would be, $c(e, \mathfrak{b}, \mathfrak{b}) = c(e, \mathfrak{b}, \bot) = 0$ since no instructions need to be inserted; $c(e, \mathfrak{b}_0, \mathfrak{b}_1) = c_1 > 0, \mathfrak{b}_0 \neq \mathfrak{b}_1 \neq \bot$ when e is an edge from a taken conditional branch, since splitting such an edge results in an additional unconditional jump instruction being generated; $c(e, \mathfrak{b}_0, \mathfrak{b}_1) = c_0 > 0, \mathfrak{b}_0 \neq \mathfrak{b}_1 \neq \bot, c_0 < c_1$ for all other cases.

For the PIC architecture mentioned above, the situation would be a bit different: To be able to represent knowledge about the state of individual bank-selection bits, we would choose $\mathfrak{B} = \{\bot = \bot\bot, \bot 0, \bot 1, 0\bot, 00, 01, 1\bot, 10, 11\}$. When optimizing for code size the cost $c(e, \mathfrak{b}_0, \mathfrak{b}_1)$ would be the number of bits in $\mathfrak{b}_1$, which are neither $\bot$, nor the same as in $\mathfrak{b}_0$. E. g. $c(e, \mathfrak{b}, \bot\bot) = 0 = c(e, 00, 0\bot) = c(e, 01, \bot 1)$, $c(e, \bot 0, 0\bot) = 1, c(e, \bot 0, 11) = 2$.



(a) Program (b) Ad-hoc approach (c) Optimal solution

Figure 8.2: Example instance

Two (partial) colorings of a graph are *compatible*, if they are identical on the nodes colored by both of them. In the following we call a (partial) coloring *compatible*, if it is compatible with the partial coloring that came with the control-flow graph.

Definition 8.3 (Optimal Placement of Bank Selection Instructions)**.** *For a fixed set of memory banks $\mathfrak{B}$, the problem of Optimal Placement of Bank Selection Instructions with input program consisting of a partially colored control-flow graph $G = (\Pi, E)$ and cost function c is the following: Find a compatible coloring $f\colon \Pi \to \mathfrak{B}$ of G, such that*

$$\sum_{e \in E} c(e, f(e_0), f(e_1))$$

is minimal over all compatible colorings.

Solving this problem is NP-hard when not restricted to structured programs [91], even for a simple cost function such as $c(e, \mathfrak{b}, \mathfrak{b}) = c(e, \mathfrak{b}, \bot) = 0$, and $c(e, \mathfrak{b}_0, \mathfrak{b}_1) = 1$ otherwise.

The definition of optimality via the minimization of the sum $\sum_{e \in E} c(e, f(e_0), f(e_1))$ allows to easily optimize for minimum code size, minimum runtime overhead, minimum energy consumption, etc, or aggregates thereof.

Figure 8.2a shows a small example program, which contains three accesses to variables in bank $\mathfrak{b}$, and three calls to other functions. The accesses are precolored by $\mathfrak{b}$, the function calls and the entry node are precolored by $\bot$. With a typical cost function, such as the one mentioned above, the optimal solution would look like Figure 8.2c: We insert bank selection instructions selecting bank $\mathfrak{b}$ at the two marked edges. The ad-hoc approach typically used in compilers today would need three bank selection instructions (Figure 8.2b).

Compared to Definition 2.13, we can relax the requirements on $|\chi(j)|$ and $|\chi(i)|$ in the introduce and forget nodes:

Definition 8.4 (Nice Tree-Decomposition)**.** *A tree-decomposition (T, χ) of a graph G is called nice, if*

- *T is oriented, with root t, $\chi(t) = \varnothing$.*
- *Each node i of T is of one of the following types:*
  - *Leaf node, has no children, $\chi(i) = \varnothing$.*
  - *Introduce node, has one child $j, \chi(j) \subsetneq \chi(i)$.*
  - *Forget node, has one child $j, \chi(j) \supsetneq \chi(i)$.*
  - *Join node, has two children $j_1, j_2, \chi(i) = \chi(j_1) = \chi(j_2)$.*

We call an edge e of G *covered* in the subtree rooted at node i of T, if each endpoint of e is contained in some $\chi(j)$ for a j in the subtree of t rooted at i.

8.3 Optimal Placement of Bank Selection Instructions in Polynomial Time

Our approach uses dynamic programming, bottom-up along a nice tree-decomposition.

Let $\mathfrak{B}$ be the set of memory banks in the architecture. Let $G = (\Pi, E)$ be the control-flow graph of the program, let $c\colon E \times \mathfrak{B} \times \mathfrak{B} \to \mathbb{R}$ be the cost function. Let (T, χ) be a nice tree-decomposition of minimum width of G with root t.

Let S be the function that gives the minimum possible costs for bank switching instructions on edges covered in the subtree rooted at node i of T, excluding edges between nodes in $\chi(i)$. S depends on the memory banks active at instructions in $\chi(i)$. The functions $f\colon \chi(i) \to \mathfrak{B}$ give these active memory banks.

$$S\colon \left\{(i, f) \;\middle|\; i \text{ node of } T, f\colon \chi(i) \to \mathfrak{B}\right\} \to \mathbb{R}.$$

$$S(i, f) := \min_{\substack{g \text{ compatible} \\ g|_{\chi(i)} = f|_{\chi(i)}}} \left\{ \sum_{e \in E_i} c\left(e, g(e_0), g(e_1)\right) \right\}.$$

Where E_i is the set of edges between nodes in the subtree rooted at node i of T, excluding edges between nodes in $\chi(i)$. This function at the root t of T, and the corresponding coloring that results in the minimum is what we want:

$$S(t, f) = \min_{\substack{g \text{ compatible} \\ g|_{\chi(t)} = f|_{\chi(t)}}} \left\{ \sum_{e \in E_t} c\left(e, g(e_0), g(e_1)\right) \right\} =$$

$$\min_{\substack{g \text{ compatible} \\ g|_{\varnothing} = f|_{\varnothing}}} \left\{ \sum_{e \in (E \setminus \varnothing^2)} c\left(e, g(e_0), g(e_1)\right) \right\} =$$

$$\min_{g \text{ compatible}} \left\{ \sum_{e \in E} c\left(e, g(e_0), g(e_1)\right) \right\}.$$

To get S, we first define a function s, and then show that $S = s$ and that s can be calculated in polynomial time. We define s inductively depending on the type of i:

- Leaf: $s(i, f) := 0$.
- Introduce node with child j: $s(i, f) :=$

$$\begin{cases} \infty \text{ if there is } \pi \in \chi(i) \setminus \chi(j) \text{ colored, } f(\pi) \text{ is not the color of } \pi \\ s\left(j, f|_{\chi(j)}\right) \text{ otherwise.} \end{cases}$$

- Forget node with child j: $s(i, f) :=$

$$\min \left\{ s(j, g) + \sum_{e \in \left(E \cap (\chi(j)^2 \setminus \chi(i)^2)\right)} c\left(e, g(e_0), g(e_1)\right) \;\middle|\; g|_{\chi(i)} = f \right\}.$$

- Join node with children j_1 and j_2: $s(i, f) := s(j_1, f) + s(j_2, f)$.

The following lemma states $s = S$:

Lemma 8.5. *Let i be a node of $T, f \colon \chi(i) \to \mathfrak{B}$. If there is no compatible coloring of the instructions covered in the subtree rooted at i that, when restricted to $\chi(i)$ is f, then $s(i, f) = \infty$. Otherwise $s(i, f)$ is the minimum possible cost for bank selection instructions on edges covered in the subtree rooted at i, excluding edges between nodes in $\chi(i)$ for such a coloring.*

Proof. By induction we can assume that the lemma is true for all children of the node i of T. Let T_i be the set of instructions in the subtree rooted at node i of T, excluding instructions in $\chi(i)$.

Case 1: i is a leaf. There are no edges in the subgraph of G induced by $T_i = (\chi(i) \setminus \chi(i)) = \varnothing$, thus the cost is 0: $s(i, f) = 0 = S(i, f)$.

Case 2.1: i is an introduce node with child j, f is compatible. $T_i = T_j$, since $\chi(i) \supseteq \chi(j)$, thus the cost remains the same: $s(i, f) = s(j, f) = S(j, f) = S(i, f)$.

Case 2.2: i is an introduce node with child j, f is not compatible: $s(i, f) = \infty = \min \varnothing = S(i, f)$.

Case 3: i is a forget node with child j. $T_i = T_j \cup (\chi(j) \setminus \chi(i))$, the union is disjoint. Thus we get the correct result by adding the costs for the edges

between $\chi(i)$ and $\pi, \pi \in \chi(j) \setminus \chi(i)$:

$$s(i,f) =$$

$$\min\left\{ s(j,g) + \sum_{e \in (E \cap (\chi(j)^2 \setminus \chi(i)^2))} c\left(e, g(e_0), g(e_1)\right) \;\middle|\; g|_{\chi(i)} = f \right\} =$$

$$\min\left\{ S(j,g) + \sum_{e \in (E \cap (\chi(j)^2 \setminus \chi(i)^2))} c\left(e, g(e_0), g(e_1)\right) \;\middle|\; g|_{\chi(i)} = f \right\} =$$

$$\min\left\{ \min_{\substack{h \text{ compatible} \\ h|_{\chi(j)} = g|_{\chi(j)}}} \left\{ \sum_{e \in E_j} c\left(e, h(e_0), h(e_1)\right) \right\} + \sum_{e \in (E \cap (\chi(j)^2 \setminus \chi(i)^2))} c\left(e, g(e_0), g(e_1)\right) \;\middle|\; g|_{\chi(i)} = f \right\} =$$

$$\min\left\{ \sum_{e \in E_j} c\left(e, g(e_0), g(e_1)\right) + \sum_{e \in (E \cap (\chi(j)^2 \setminus \chi(i)^2))} c\left(e, g(e_0), g(e_1)\right) \;\middle|\; \begin{matrix} g|_{\chi(i)} = f \\ g \text{ compatible} \end{matrix} \right\} =$$

$$\min_{\substack{g \text{ compatible} \\ g|_{\chi(i)} = f|_{\chi(i)}}} \left\{ \sum_{e \in E_i} c\left(e, g(e_0), g(e_1)\right) \right\} =$$

$$S(i,f).$$

Case 4: i is a join node with children j_1 and j_2. $T_i = T_{j_1} \cup T_{j_2}$, since $\chi(i) = \chi(j_1) = \chi(j_2)$. The union is disjoint and there are no edges between T_{j_1} and T_{j_2} in G. Thus we get the correct result by adding the costs from both subtrees: $s(i,f) = s(j_1,f) + s(j_2,f) = S(j_1,f) + S(j_2,f) = S(i,f)$. q. e. d.

Lemma 8.6. *Given the nice tree-decomposition of minimum width, s can be calculated in time $O(|T| \operatorname{tw}(G)^2 |\mathfrak{B}|^{\operatorname{tw}(G)+1})$.*

Proof. At each node i of T time $O(\operatorname{tw}(G)^2 |\mathfrak{B}|^{\operatorname{tw}(G)+1})$ is sufficient:

Case 1: i is a leaf. There is only one function $f\colon \varnothing \to \mathfrak{B}$.

Case 2: i is an introduce node. There are at most $|\mathfrak{B}|^{|\chi(i)|} \leq |\mathfrak{B}|^{\operatorname{tw}(G)+1}$ different f and for each one we need constant time.

Case 3: i is a forget node with child j. There are at most $|\mathfrak{B}|^{|\chi(i)|}$ different f and for each one we need to consider at most $|\mathfrak{B}||\chi(j) \setminus \chi(i)|$ different g and for each g we need to consider less than $\operatorname{tw}(G)^2$ different edges. Thus the total time is in

$$O\left(|\mathfrak{B}|^{|\chi(i)|} |\mathfrak{B}|^{|\chi(j) \setminus \chi(i)|} \operatorname{tw}(G)^2\right) =$$

$$O\left(\operatorname{tw}(G)^2 |\mathfrak{B}|^{|\chi(j)|}\right) \subseteq O\left(\operatorname{tw}(G)^2 |\mathfrak{B}|^{\operatorname{tw}(G)+1}\right).$$

Case 4: i is a join node. The reasoning from case 2 holds. q. e. d.

Theorem 8.7. *Optimal Placement of Bank Selection Instructions can be done in polynomial time for structured programs.*

Proof. Given an input program of bounded tree-width we can calculate a tree-decomposition of minimum width in linear time [17]. We can then transform this tree-decomposition into a nice one of the same width and calculate the s as above in polynomial time. Using standard book-keeping techniques we can keep track of which placement of bank selection instructions corresponds to each s. The one remaining s at the root t of T then gives us the minimum total cost. The corresponding placement of bank selection instructions is a solution to the Optimal Placement of Bank Selection Instructions problem. q. e. d.

8.4 Prototype Implementation

We implemented a prototype in SDCC [41], a C compiler for embedded systems, targeting the MCS-51, DS390, DS400, HC08, S08, Z80, Z180, Rabbit 2000/3000, Rabbit 3000A, LR35902, PIC14 and PIC16 architectures. Our approach is used for the placement of bank selection instructions since SDCC 3.2.0, released in mid-2012. The code can be found in the SDCC project's public source code repository.

The placement of variables in memory banks is done by the programmer using Embedded C's named address spaces. SDCC generates intermediate code, where each instruction uses operands from at most one named address space. We color the control-flow graph, giving instructions that access variables in a named address space the respective color and giving the function entry and function call nodes the color $\perp$.

In a pre-processing step (not mentioned in Section 8.3 above, since it does not change the complexity bound) we calculate for each connected component of non-colored nodes the set of adjacent colors. We can restrict color choice to these sets. For this pre-processing step we assume that our cost function c is non-negative, and not inserting instructions is free (i. e. $c(e, \mathfrak{b}, \mathfrak{b}) = c(e, \mathfrak{b}, \perp) = 0$).

Bank selection is done by calls to programmer-supplied helper functions, resulting in a flexible approach to bank selection, which is necessary to support systems with custom bank selection hardware. We implemented code size as the cost function due to its simplicity: $c(e, \mathfrak{b}, \mathfrak{b}) = c(e, \mathfrak{b}, \perp) = 0$ since no instructions need to be inserted; $c(e, \mathfrak{b}_0, \mathfrak{b}_1) = 6, \mathfrak{b}_0 \neq \mathfrak{b}_1 \neq \perp$ when e is an edge from a taken conditional branch, (3 bytes for the call and 3 bytes for an unconditional jump in the architectures we targeted); $c(e, \mathfrak{b}_0, \mathfrak{b}_1) = 3, \mathfrak{b}_0 \neq \mathfrak{b}_1 \neq \perp, c_0 < c_1$ for all other cases. Our implementation is easy to re-target to other optimization goals, architectures and ways of bank-switching by choice of the cost function.

Since Bodlaender's algorithm [17] is not a practical choice due to large constant factors in the runtime, we used Thorup's algorithm [132] to obtain the tree-decomposition.

The pre-processing step mentioned above and the low typical tree-width practically always results in less than ten colorings being considered at each node of the tree-decomposition. This means that the run-time typically is much lower than the theoretical polynomial bound. The time spent by the compiler in our algorithm for Optimal Placement of Bank Selection Instructions is thus negligible compared to the time spent on other compiler tasks.

8.5 Examples

Figure 8.3 shows an artificial example of a C function that uses named address spaces from the Embedded C standard: There are address spaces a and b corresponding to memory banks. The global variables a0, a1 and a2 are placed in bank `a`, while b0, b1 and b2 are placed in bank `b`. Global variables x0 and x1 are placed in a memory that is always visible in the address space (i. e. requiring no bank selection instructions), as is the local variable i and any additional local variables generated by the compiler. While this example is artificial, we think that it captures the essential aspects of a program that has moderately complex control-flow and uses variables both in banked and non-banked memory.

Figure 8.4 shows the control-flow graph SDCC generates, after the preprocessing step: Each node is annotated with the set of banks to choose from. We see that the entry nodes, the call to g() and accesses to variables in memory banks only have one choice. When optimizing for code size, our prototype implementation inserts four bank selection instructions at the marked edges. The ad-hoc approach would insert six. Li et alii's approach [91] offers two algorithms (called "Heuristic Algorithm (A)" and "Heuristic Algorithm (B)" in their paper) that can deal with transparent nodes, but neither is optimal. "Heuristic Algorithm (A)" would insert at least six bank selection instructions, "Heuristic Algorithm (B)" would insert at least five.

Graphs similar to Figure 8.4 can be obtained from SDCC when compiling using the –dump-graphs option.

Figure 8.5 shows pseudocode where our optimal approach gives a decisive advantage over common non-optimal approaches: A switch statement with $n+1$ cases, slightly less than half of which access a bank `b`, while the other cases do not access any bank. The switch statement is followed by an access to a bank `a`, and preceded by an if-else construct that accesses bank `a` in one branch, and no bank in the other. When optimizing for code size, our optimal approach will insert 3 bank selection instructions. The ad-hoc approach will insert $O(n)$ bank selection instructions, since one will be placed for each case of the switch statement that accesses bank `b`. Both "Heuristic Algorithm (A)" and "Heuristic Algorithm (B)" from Li et alii's approach [91] would insert $O(n)$ bank selection instructions (essentially, both algorithms fail to handle the bank selection for the switch statement in an efficient way, for "A" this is triggered by the code before the switch, and for "B" by the code after the switch).

8.6 Conclusion

We present the first approach to Optimal Placement of Bank Selection Instruction in Polynomial Time. There are previous approaches to the placement of bank selection instructions, but they are restricted to a narrow class of problems, and otherwise not applicable or not optimal [91], or do not run in polynomial time [122, 123]. Our approach uses tree-decompositions of the control-flow graph. It is easy to re-target for different optimization goals and is implemented in a mainstream compiler for embedded systems.

```
void set_a(void);
void set_b(void);

__addressmod set_a a;
__addressmod set_b b;

a int a0, a1, a2;
b int b0, b1, b2;
int x0, x1;

void g(void);

int f(void)
{
        int i;
        switch(x0)
        {
        case 0:
                if(a0 && (b0 + x1) > 2)
                {
                        if(x1 % 2)
                                a1++;
                        if(x1 > 4)
                                break;
                }
                else
                {
                        if(x1 < 7)
                                break;
                        else if(x1 > 0)
                                return(a1);
                        x0++;
                }
                return(b2);
        case 1:
                x0++;
        default:
                x0++;
                g();
                return(x1);
        }
        return(a2);
}
```

Figure 8.3: Embedded C program: Source

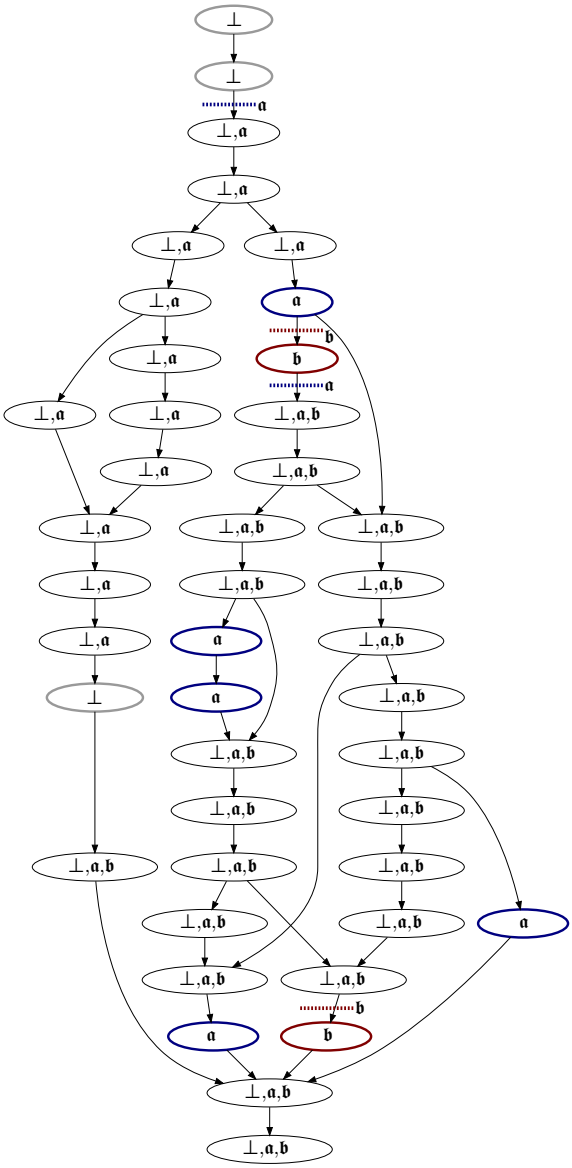


Figure 8.4: Embedded C program: Graph

```
if(...)
        access_bank_a;
else
        do_not_access_banked_memory;

switch(...)
{
case 0:
        do_not_access_banked_memory;
        break;
case 1:
        access_bank_b;
        break;
case 2:
        do_not_access_banked_memory;
        break;
.
.
.
case n-1:
        access_bank_b;
        break;
default:
        do_not_access_banked_memory;
        break;
}

access_bank_a;
```

Figure 8.5: Pseudocode

Chapter 9

lospre in linear time[1]

Lifetime-optimal speculative partial redundancy elimination (lospre) is the most advanced currently known redundancy elimination technique. It subsumes many previously known approaches, such as common subexpression elimination, global common subexpression elimination, and loop-invariant code motion. However, previously known lospre algorithms have high time complexity; faster but less powerful approaches have been used and developed further instead. We present a simple linear-time algorithm for lospre for structured programs that can also handle some more general scenarios compared to previous approaches. We prove that our approach is optimal and that the runtime is linear in the number of nodes in the control-flow graph. The condition on programs of being structured is automatically true for many programming languages and for others, such as C, is equivalent to a bound on the number of `goto` labels per function. A prototype implementation in a mainstream C compiler demonstrates the practical feasibility of our approach. Our approach is based on graph-structure theory and uses tree-decompositions.

9.1 Introduction

Redundancy elimination is a technique commonly used in current optimizing compilers. Even early optimizing compilers had common subexpression elimination (CSE) for straight-line code. Global common subexpression elimination (GCSE) [29] extended this to the whole control-flow graph (CFG). Partial redundancy elimination (PRE) [102] generalized GCSE (and some other techniques such as loop-invariant code-motion (LICM)) to computations that are redundant only on some paths in the CFG. Improvements led to lazy code-motion (LCM) [78], which is a lifetime-optimal variant of PRE, i. e. the lifetimes of introduced temporary variables are minimized, which is important to keep register pressure low. Another improvement to PRE is speculative PRE (SPRE) [55, 125, 25], which can increase the number of computations on some paths in order to reduce the total number of computations done (based on profiling information). The natural improvement is combining the advantages of LCM and SPRE, resulting in lifetime-optimal SPRE (lospre), which was first achieved by the min-cut-PRE (MC-PRE) algorithm [139].

[1] Submitted for publication [82].

However, MC-PRE relies on solving a weighted minimum cut problem on directed graphs. This problem seems to be harder than its equivalent on undirected graphs. The fastest known algorithm [71] is randomized, and will likely give a result in $O(n^2 \log^3(n))$. Typical implementations use deterministic algorithms resulting in cubic runtime. Some researchers consider this too much for some applications, in particular for just-in-time compilation and thus developed faster approaches that are not optimal, but perform close to lospre for many real-world scenarios [64, 107]. MC-SSAPRE [144] is a newer optimal approach that works for programs in static single assignment form. In practice it is often faster than MC-PRE, but it has the same worst case complexity, since it also needs to solve a weighted minimum cut problem on directed graphs. MC-SSAPRE is also considered hard to implement [68].

An alternative to lospre is complete PRE (ComPRE) [16], which completely eliminates redundancies and is lifetime-optimal. It is not speculative, and eliminates more dynamic computations than speculative approaches, which can offer benefits when optimizing for code speed in some scenarios. However, it is not suitable for optimization for code size, and it restructures the CFG, resulting in additional conditional jumps being introduced. For most common architectures, conditional jumps are far more expensive in terms of speed than typical computations [111, 44], which makes the code resulting from ComPRE slower than the code from lospre.

We present a simple approach to lospre, which does not sacrifice optimality, and runs in linear time. It is based on graph-structure theory, in particular the bounded tree-width of control-flow graphs of structured programs. A prototype has been implemented in a mainstream C compiler and has low compilation time overhead. It also generalizes lospre further than previous approaches, by allowing trade-offs between costs from computations and costs from variable lifetimes. Unlike previous approaches, our approach can also consider benefits from reduction in the life-times of operands of redundant expressions, while previous approaches only considered costs from the newly introduced temporary variable.

9.2 Problem Description

Programs tend to contain redundancies, and eliminating them is an important goal in optimizing compilers. Figure 9.1 shows a function written in the C programming language. It will be transformed by compilers into a form of intermediate code. Figure 9.2 shows the control-flow graph and intermediate code for this function as it is used in the SDCC [41] compiler before redundancy elimination. The array-index style access got transformed into a sequence of operations, first multiplying the index `i` by the size of `long` (the multiplication already was further transformed into a left-shift), followed by an addition of the result to the array base address and then a read from memory. Redundancy elimination would move these operations before the branch, as can be seen in Figure 9.3.

We will now formally define the control-flow graph as we use it in our approach. In particular, it is a weighted graph to allow the representation of costs from calculations and variable lifetimes. Often, such costs will be e. g. represented by natural numbers or something similar, but we want to keep things a

```
#include <stdbool.h>

extern long *a;
extern long c;

void f(bool b, int i)
{
    if(b)
        c = a[i] + 8;
    else
        c = a[i] - 13;
}
```

Figure 9.1: Function written in C

bit more general for now:

Definition 9.1 (Control-flow graph (CFG))**.** *A control-flow graph is a weighted directed graph* (V, E, c, l) *with node set* V *and edge set* $E \subseteq V^2$ *and weight functions* $c\colon E \to \mathcal{K}$ *and* $l\colon V \to \mathcal{K}$ *for some ordered* $\mathcal{K}$ *that has addition. There is a unique source (node without predecessors).*

For simplicity we will sometimes ignore the weights and treat G as the directed graph (V, E). The nodes of the CFG correspond to instructions in a program. We chose this notion over the more common one of using basic blocks as nodes in the CFG, since it simplifies the discussion of our approach a bit and makes it subsume CSE as well. For applications where compilation speed is essential, it is easy to reformulate our approach to use basic blocks, and do CSE as a pre-processing step. The weight function c gives the cost of subdividing an edge and inserting a computation there. It depends on the optimization goal. E. g. when optimizing for speed or energy consumption, execution frequencies (estimated or obtained from a profiler) would be used, and $\mathcal{K}$ could be the set of possible values of a floating-point data type. When optimizing for code size one could use a constant $c \in \mathbb{N}$ and $\mathcal{K} = \mathbb{N}$ instead. The weight function l gives the cost of having a new temporary variable alive at a node. When just minimizing the life-time, a constant can be used. More sophisticated approaches to l could take other aspects, such as register pressure at the nodes, into account. This latter aspect, and the possibility of handling costs from calculations and lifetimes in a unified way is something previous approaches, such as MC-PRE and MC-SSAPRE could not do.

For a given expression the set of nodes in the CFG where it is calculated is called the *use set* $\mathcal{U} \subseteq V$. In redundancy elimination techniques, such calculations from $\mathcal{U}$ are replaced by assignments from a new temporary variable, which is initialized by new calculations. For each node in the CFG, we decide whether the new temporary variable should be alive there. We call the set of such nodes the *life set* $\mathcal{L} \subseteq V$. There also can be invalidating nodes, which we have to be careful about. They invalidate the result of the calculation in the expression: E. g. when the expression we want to replace is $a + b$, the node where the instruction $a = 7$ is done would be invalidating. We call the set of such nodes the

invalidation set $\mathcal{I} \subseteq V$. We always consider the source and sinks (nodes without successors) of the CFG to be invalidating, since we have no knowledge about how variables might change before or after our program. Depending on these three sets some edges need to be subdivided and new calculations inserted. We call the set of such edges the *calculation set*

$$\mathcal{C}(\mathcal{U}, \mathcal{L}, \mathcal{I}) := \left\{(x, y) \in E \;\middle|\; x \notin \mathcal{L} \setminus \mathcal{I}, y \in \mathcal{U} \cup \mathcal{L}\right\}.$$

Definition 9.2 (lospre)**.** *Given a CFG* (V, E, c, l)*, use set* $\mathcal{U} \subseteq V$ *and invalidation set* $\mathcal{I} \subseteq V$*, the problem of lospre is to find a life set* $\mathcal{L} \subseteq V$*, such that the cost*

$$\sum_{e \in \mathcal{C}(\mathcal{U}, \mathcal{L}, \mathcal{I})} c(e) + \sum_{v \in \mathcal{L}} l(v)$$

is minimized.

For the typical lospre application, one could use $\mathcal{K} = \mathbb{Z}^2$ with lexicographical ordering. Optimizing for code size using $c(e) = (1, 0)$ or optimizing for speed using $c(e) = (p(e), 0)$, where p gives an execution probability estimated using a profiler. $l = (0, 1)$ would then guarantee the lifetime-optimality.

Sometimes *safety* is required. Safety means that no calculations may be done on values on which the original program would not perform them. In general, safety is not desirable, since it restricts choices and thus results in less optimization. However, on some architectures division by zero results in undesirable behaviour; in this case we need safety when the expression is division and we cannot predict the value of the divisor. Other examples would be memory reads from a calculated address on architectures with memory management (where reads from invalid addresses could result in a SIGSEGV terminating the program), accesses to variables declared using C `volatile`, or I/O accesses. The safety requirement can be handled by using a different invalidation set $\mathcal{I}'$ in place of $\mathcal{I}$ in lospre.

Definition 9.3 (safety)**.** *Given a CFG* (V, E)*, use set* $\mathcal{U} \subseteq V$ *and invalidation set* $\mathcal{I} \subseteq V$*, the problem of safety is finding a set* $\mathcal{I}' \supseteq \mathcal{I}$ *of minimal size, such that no node outside of* $\mathcal{I}'$ *lies on a path between two nodes in* $\mathcal{I}$ *that does not contain a node in* $\mathcal{U}$*.*

Once we have $\mathcal{I}'$, we can use it in place of $\mathcal{I}$ in lospre.

Figure 9.4 shows a CFG with redundancies. $a * b$ is calculated in two places. Even a simple redundancy elimination technique, such as GCSE, would split e_0 and place the calculation there. For lospre we would have $\mathcal{U}$ consisting of the two nodes where $a * b$ is used, no invalidating nodes except for sink and source, thus $\mathcal{I}$ would consist of the black nodes only. The life set found by lospre would consist of the nodes with dashed contours, and thus $\mathcal{C}(\mathcal{U}, \mathcal{L}, \mathcal{I}) = \{e_0\}$.

c/d is a slightly more complicated case: When optimizing for code size, and when safety is not required for division, one might want to split e_1 and do the calculation there. But when division requires safety this is not possible ($\mathcal{I}'$ would consist of the black and grey nodes). When optimizing for speed we will not want to pay the cost of calculating c/d when it is not needed.

Compared to Definition 2.13, we can relax the requirement on $|\chi(j)|$ and $|\chi(i)|$ in the introduce nodes:

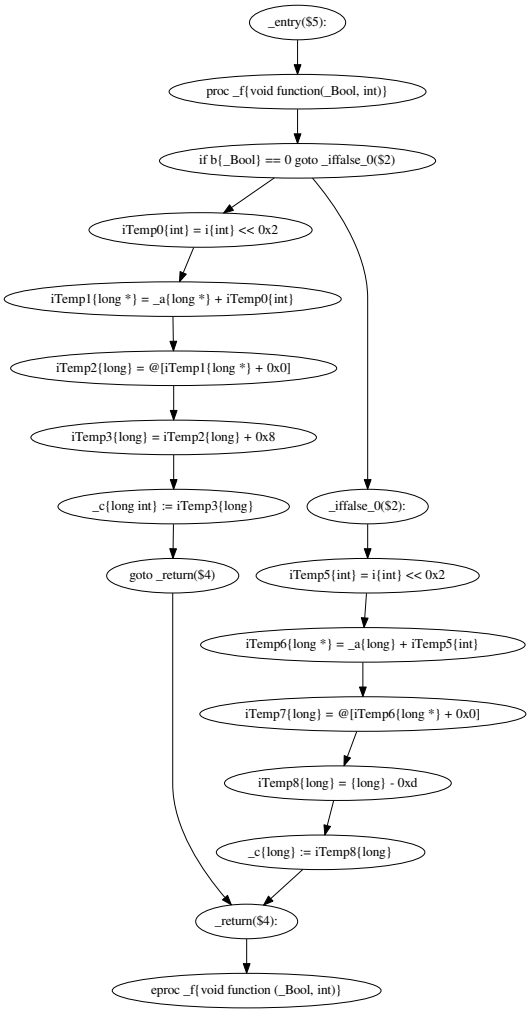


Figure 9.2: CFG before redundancy elimination

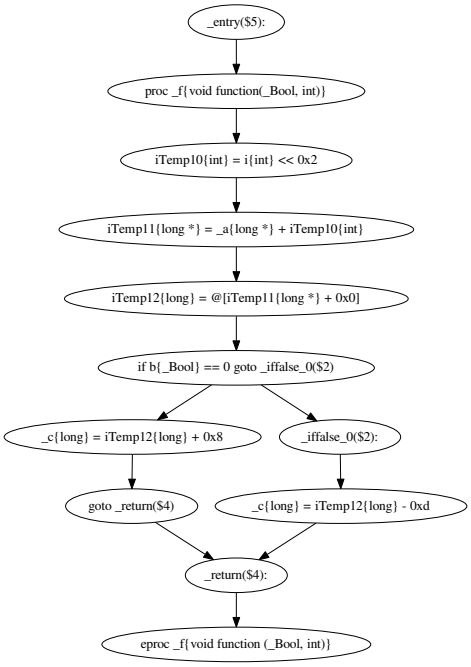


Figure 9.3: CFG after redundancy elimination

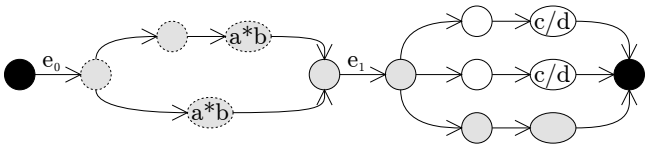


Figure 9.4: Simplified CFG with redundancies

Definition 9.4 (Nice Tree-Decomposition)**.** *A tree-decomposition (T, χ) of a directed graph G is called nice, if*

- *T has a root t, $\chi(t) = \varnothing$.*
- *Each node i of T is of one of the following types:*
  - *Leaf, no children, $\chi(i) = \varnothing$.*
  - *Introduce node, has one child $j, \chi(j) \subsetneq \chi(i)$.*
  - *Forget node, has one child $j, \chi(j) \supsetneq \chi(i), |\chi(j) \setminus \chi(i)| = 1$.*
  - *Join node, has two children $j_1, j_2, \chi(i) = \chi(j_1) = \chi(j_2)$.*

One approach to improving the runtime of MC-PRE would be to replace the min-cut step by one based on tree-decompositions. However this would not affect other parts of MC-PRE, and thus not yield a linear time variant of MC-PRE. Also it would not simplify the MC-PRE algorithm, and not allow us to extend lospre to better handle lifetime-optimality. We therefore instead replace all of MC-PRE by an approach based on tree-decompositions.

9.3 lospre in linear time

Our approach uses dynamic programming [14], bottom-up along a nice tree-decomposition of minimum width of the CFG; as noted above, there are established methods for obtaining this tree-decomposition. Otherwise, our approach only uses elementary operations.

Let $G = (V, E, c, l)$ be the control-flow graph as in Section 9.2. Let $(T, \mathcal{X})$ be a nice tree-decomposition of minimum width of G with root t. Let T_i be the set of nodes in bags in the subtree rooted at node i of T, excluding the nodes in $\chi(i)$. Let S be the function that gives the minimum possible costs on edges and nodes covered in the subtree rooted at node i of T, excluding edges and nodes in $\chi(i)$. S depends on where in $\chi(i)$ the new temporary variable is alive, which is captured by the functions $f\colon \chi(i) \to \{0, 1\}$. For these f (and any function from a $X \subseteq V$ to $\{0, 1\}$), we define a local version of the calculation set:

$$\mathcal{C}(\mathcal{U}, f, \mathcal{I}) := \left\{ (x, y) \in E \;\middle|\; x \notin f^{-1}(1) \setminus \mathcal{I}, y \in \mathcal{U} \cup f^{-1}(1) \right\}.$$

$$T_i := \left\{ v \in (\chi(j) \setminus \chi(i)) \;\middle|\; j \text{ in the subtree of } T \text{ rooted at } i \right\},$$

$$S(i, f) := \min_{\substack{g\colon V \to \{0,1\} \\ g|_{\chi(i)} = f|_{\chi(i)}}} \left\{ \sum_{v \in (T_i \cap g^{-1}(1))} l(v) + \sum_{e \in \mathcal{C}(\mathcal{U}, g, \mathcal{I}) \cap T_i^2} c(e) \right\}.$$

At the root t of T, this function S, and the corresponding g are what we want,

since this g gives the life-set that is the solution to lospre:

$$S(t,f) = \min_{\substack{g\colon V\to\{0,1\}\\ g|_{\chi(t)}=f|_{\chi(t)}}} \left\{ \sum_{v\in(T_t\cap g^{-1}(1))} l(v) + \sum_{e\in\mathcal{C}(\mathcal{U},g,\mathcal{I})\cap T_t^2} c(e) \right\} =$$

$$\min_{\substack{g\colon V\to\{0,1\}\\ g|_{\varnothing}=f|_{\varnothing}}} \left\{ \sum_{v\in(g^{-1}(1))} l(v) + \sum_{e\in\mathcal{C}(\mathcal{U},g,\mathcal{I})} c(e) \right\} =$$

$$\min_{g\colon V\to\{0,1\}} \left\{ \sum_{v\in(g^{-1}(1))} l(v) + \sum_{e\in\mathcal{C}(\mathcal{U},g,\mathcal{I})} c(e) \right\} =$$

$$\min_{\mathcal{L}\subseteq V} \left\{ \sum_{e\in\mathcal{C}(\mathcal{U},\mathcal{L},\mathcal{I})} c(e) + \sum_{v\in\mathcal{L}} l(v) \right\}.$$

To calculate S, we define a function s, and then proceed to show that $s = S$ and that s can be computed in linear time. We define s inductively depending on the type of i:

- Leaf: $s(i,f) := 0$.
- Introduce node with child j: $s(i,f) := s(j, f|_{\chi(j)})$.
- Forget node with child $j, \chi(j) \setminus \chi(i) = \{v\}$:

$$s(i,f) := \min_{\substack{g\colon \chi(j)\to\{0,1\}\\ g|_{\chi(i)}=f}} \left\{ s(j,g) + f(v)l(v) + \sum_{\substack{e\in\mathcal{C}(\mathcal{U},f,\mathcal{I})\cap\\ (\{v\}\times E)\cup(E\times\{v\}))}} c(e) \right\}.$$

- Join node with children j_1 and j_2: $s(i,f) := s(j_1,f) + s(j_2,f)$.

Lemma 9.5. *$s(i,f)$ gives the minimum possible costs on edges and nodes covered in the subtree rooted at node i of T, excluding edges and nodes in $\chi(i)$, i. e. $s = S$.*

Proof. By induction we can assume that the lemma is true for all children of node i of T.

Case 1: i is a leaf. There are no edges or nodes in the subgraph of G induced by $T_i = \chi(i) \setminus \chi(i) = \varnothing$, thus the cost is zero:

$$s(i,f) = 0 = S(i,f).$$

Case 2: i is an introduce node with child j. $T_i = T_j$, since $\chi(i) \subseteq \chi(j)$, thus the cost remains the same:

$$s(i,f) = s(j,f) = S(j,f) = S(i,f).$$

Case 3: i is a forget node with child j. $T_i = T_j \cup (\chi(j) \setminus \chi(i)) = T_j \cup \{v\}$, the union is disjoint. Thus we get the correct result by adding the costs for the edges between $\chi(i)$ and $v, \{v\} = (\chi(j) \setminus \chi(i))$ and the lifetime cost for v itself:

$$s(i,f) = \min_{\substack{g\colon \chi(j)\to\{0,1\}\\ g|_{\chi(i)}=f}} \left\{ s(j,g) + f(v)l(v) + \sum_{\substack{e\in\mathcal{C}(\mathcal{U},f,\mathcal{I})\cap\\ (((\{v\}\times E)\cup(E\times\{v\}))}} c(e) \right\} =$$

$$\min_{\substack{g\colon \chi(j)\to\{0,1\}\\ g|_{\chi(i)}=f}} \left\{ S(j,g) + f(v)l(v) + \sum_{\substack{e\in\mathcal{C}(\mathcal{U},f,\mathcal{I})\cap\\ (((\{v\}\times E)\cup(E\times\{v\}))}} c(e) \right\} =$$

$$\min_{\substack{g\colon \chi(j)\to\{0,1\}\\ g|_{\chi(i)}=f}} \left\{ \min_{\substack{h\colon V\to\{0,1\}\\ h|_{\chi(j)}=g|_{\chi(j)}}} \left\{ \sum_{u\in(T_j\cap h^{-1}(1))} l(u) + \sum_{e\in\mathcal{C}(\mathcal{U},h,\mathcal{I})\cap T_j^2} c(e) \right\} + \right.$$

$$\left. f(v)l(v) + \sum_{\substack{e\in\mathcal{C}(\mathcal{U},f,\mathcal{I})\cap\\ (((\{v\}\times E)\cup(E\times\{v\}))}} c(e) \right\} =$$

$$\min_{\substack{g\colon V\to\{0,1\}\\ g|_{\chi(i)}=f}} \left\{ \sum_{u\in(T_j\cap g^{-1}(1))} l(u) + \sum_{e\in\mathcal{C}(\mathcal{U},g,\mathcal{I})\cap T_j^2} c(e) + \right.$$

$$\left. f(v)l(v) + \sum_{\substack{e\in\mathcal{C}(\mathcal{U},f,\mathcal{I})\cap\\ (((\{v\}\times E)\cup(E\times\{v\}))}} c(e) \right\} =$$

$$\min_{\substack{g\colon V\to\{0,1\}\\ g|_{\chi(i)}=f|_{\chi(i)}}} \left\{ \sum_{v\in(T_i\cap g^{-1}(1))} l(v) + \sum_{e\in\mathcal{C}(\mathcal{U},g,\mathcal{I})\cap T_i^2} c(e) \right\} = S(i,f).$$

Case 4: i is a join node with children j_1 and j_2. $T_i = T_{j_1} \cup T_{j_2}$, since $\chi(i) = \chi(j_1) = \chi(j_2)$. The union is disjoint and there are no edges between T_{j_1} and T_{j_2} in G. Thus we get the correct result by adding the costs from both subtrees:

$$s(i,f) = s(j_1,f) + s(j_2,f) = S(j_1,f) + S(j_2,f) = S(i,f).$$

q. e. d.

Lemma 9.6. *Given a nice tree-decomposition $(T,\mathcal{X})$ of minimum width of G, s can be calculated in time $O(\mathrm{tw}(G)2^{\mathrm{tw}(G)}|T|)$.*

Proof. At each node i of T time $O(\mathrm{tw}(G)2^{\mathrm{tw}(G)})$ is sufficient:

Case 1: i is a leaf. There is only one function $f\colon \varnothing \to \{0,1\}$.

Case 2: i is an introduce node. There are at most $2^{|\chi(i)|} \leq 2^{\mathrm{tw}(G)+1}$ different f and for each one we use constant time.

Case 3: i is a forget node with child j. There are at most $2^{|\chi(i)|}$ different f and for each one we need to consider at most $2^{|\chi(j) \setminus \chi(i)|}$ different g and for each g we need to consider at most $2\,\mathrm{tw}(G)$ different edges. Thus the total time is in $O(2^{|\chi(i)|}2^{|\chi(j)\setminus\chi(i)|}2\,\mathrm{tw}(G)) = O(\mathrm{tw}(G)2^{|\chi(j)|}) \subseteq O(\mathrm{tw}(G)2^{\mathrm{tw}(G)+1}) = O(\mathrm{tw}(G)2^{\mathrm{tw}(G)})$.

Case 4: i is a join node. The reasoning from case 2 holds. q. e. d.

Theorem 9.7. *lospre can be done in linear time for structured programs.*

Proof. Given an input program of bounded tree-width we can calculate a tree-decomposition of minimum width in linear time [17]. We can then transform this tree-decomposition into a nice one of the same width. The linear time for these steps implies that $|T|$ is linear in $|V|$. We then calculate the s as in Lemma 9.6 above in linear time. Using standard book-keeping techniques we can keep track of which g corresponds to each f. The one remaining $s(t, f)$ at the root t of T then gives us the minimum total cost according to Lemma 9.5. The corresponding $g^{-1}(1) = \mathcal{L}$ is the solution. q. e. d.

The safety problem can be solved by a similar approach. This time f denotes which nodes of G are to be added to $\mathcal{I}$ to get $\mathcal{I}'$. We use cost values in $\mathcal{K} = \mathbb{Z} \cup \{\infty\}$. The function s is defined the same as above except for forget nodes. For a forget node i, with child $j, \{v\} = \chi(j) \setminus \chi(i)$, it is defined as follows:

$$s(i, f) := \min\left\{ s(j, g) \;\middle|\; g|_{\chi(i)} = f \right\} + \begin{cases} 0 \text{ if } f(v) = 0 \\ \infty \text{ if } f(v) = 1, v \in \mathcal{U} \\ \infty \text{ if } f(v) = 1, \text{ no successor of } v \text{ is in } f^{-1}(1) \cup \mathcal{I} \\ \infty \text{ if } f(v) = 1, \text{ no predecessor of } v \text{ is in } f^{-1}(1) \cup \mathcal{I} \\ -1 \text{ otherwise.} \end{cases}$$

With a proof very similar to the one for the previous theorem, we get:

Theorem 9.8. *The safety problem can be solved in linear time for structured programs.*

Together with the previous theorem, this allows us to do lospre in linear time, even when safety is required.

9.4 Prototype

We implemented a prototype in SDCC [41], since SDCC has infrastructure for handling tree-decompositions due to its tree-decomposition based register allocator (Chapter 10) and bank selection (Chapter 8). SDCC is a C compiler for embedded systems, targeting the MCS-51, DS390, DS400, HC08, S08, Z80, Z180, Rabbit 2000/3000, Rabbit 3000A, LR35902, STM8, PIC14 and PIC16 architectures. Our prototype is included in SDCC 3.3.0 released in May 2013.

The code can be found in the SDCC project's public source code repository. The prototype minimizes the total number of computations in the three-address code (corresponding to optimizing for code size), using $\mathcal{K} = \mathbb{Z}^2$ with lexicographical ordering, $w(e) = (1, 0)$ and $l = (0, 1)$. It does not yet take information from pointer analysis into account, and thus requires safety for all pointer reads. Thorup's heuristic [132] is used to obtain the tree-decomposition. As a benchmark, we compiled the Contiki operating system [40], version 2.5, consisting of 1083 C functions.

To measure the impact of lospre on compiled programs, we first counted the number of eliminated computations when using no other global redundancy elimination technique. To measure the advantage over the current GCSE implementation in SDCC, we also counted the number of computations eliminated by our lospre implementation when GCSE was run on the programs first. When using lospre as an additional compiler stage after GCSE, lospre was able to reduce the number of computations by 543. With GCSE disabled, lospre was able to reduce the number of computations by 1311; when the safety requirement on reads from calculated addresses was dropped, these numbers increased to 561 and 1329. This shows that even in its current state, our lospre prototype provides a significant advantage over the GCSE implementation used in SDCC. For further data on how lospre can improve a program, we refer the reader to the extensive experimental evaluation using the MC-PRE [139] (e. g. eliminating 90.13% more non-full redundancies in SPECint2000 compared to LCM, speedup of over 7% compared to LCM in the sixtrack SPECint2000 benchmark) and MC-SSAPRE algorithms [144]; the flexible handling of lifetime costs using the function l in our approach offers the potential for improvements over what MC-PRE and MC-SSAPRE can do.

We used the Callgrind tool of Valgrind [104] to measure the part of compilation time spent in our lospre prototype implementation. Again the numbers are from compiling the Contiki operating system. Only 1.75% of the total compilation time was spent in the prototype. This is a very low overhead, especially considering that our prototype is not optimized for compilation speed. Speed could easily improved further, e. g. by parallelization or by working on a CFG that uses basic blocks as nodes. Of the time spent in the prototype, about 66% were spent in our lospre algorithm, 29% in our safety algorithm, and the rest on other tasks, such as obtaining a tree-decomposition and transforming it into a nice tree-decomposition.

The prototype speed could be improved further by using a better way to obtain the tree-decomposition, working on the block graph and parallelizing it. It could be improved by using information from pointer analysis. Also, this prototype in SDCC only provides a complexity advantage over the previously known MC-PRE and MC-SSAPRE algorithms; it will be interesting to see the impact from using our approach in its full generality, i. e. the possibility to have more complex cost functions, which take register pressure into account and allow a trade-off between computation costs and lifetime costs; the next section will allow a first glimpse on this.

9.5 Extending lifetime optimality

Traditionally, the property of being lifetime optimal only referred to minimization of the life-times of newly introduced temporary variables. However, considering further aspects, such as register pressure and potential reductions in life-times of other variables, can result in better code. In this section we show how to extend our approach to handle these aspects, at the cost of increasing the runtime by a small constant factor; this also serves as an example for other similar extensions.

For calculations that have local variables as operands, we introduce $\mathcal{L}_l$ as the life-time of the left operand and $\mathcal{L}_r$ as the life-time of the right operand, and adjust the cost function accordingly:

$$\sum_{e \in \mathcal{C}(\mathcal{U},\mathcal{L},\mathcal{I})} c(e) + \sum_{v \in V} l\left(v, \mathcal{L} \cap \{v\}, \mathcal{L}_l \cap \{v\}, \mathcal{L}_r \cap \{v\}\right)$$

To account for the two additional sets we have to handle we also adjust the functions f and g to give values in $\{0,1\}^3$ instead of $\{0,1\}$.

Theorem 9.9. *lospre can be done in linear time for structured programs, even under the extended meaning of lifetime-optimality.*

Proof. The proof is similar to the one for Theorem 9.7 above. The most notable difference is that compared to the proof of Lemma 9.6 we have to consider $(2^3)^{|\chi(i)|}$ different f instead of just $2^{|\chi(i)|}$, resulting in total time $O((\mathrm{tw}(G)2^{\mathrm{tw}(G)} + 8^{\mathrm{tw}(G)})|T|)$ instead of $O(\mathrm{tw}(G)2^{\mathrm{tw}(G)}|T|)$ for the calculation of s. Since $\mathrm{tw}(G)$ is bounded, this is still linear time. q. e. d.

We have started extending our prototype accordingly, using register pressure (i. e. the sum over the sizes of all variables alive at an instruction) for l. For some test cases we see improvements in the size and runtime of the generated code of up to 2.5%. Results also hint at potential for further improvement when taking the availability of registers in the target architecture into account, so this is what we want to investigate next.

9.6 Conclusion

We have presented an algorithm for lospre, which is based on graph-structure theory. Like the earlier MC-PRE and MC-SSAPRE algorithms it is optimal, but it is more general, and much simpler and has much lower time complexity. We have proven that it is optimal and has linear runtime. A prototype implementation in a mainstream C compiler demonstrates the practical feasibility of our approach and the low compilation time overhead.

Chapter 10

Optimal Register Allocation in Polynomial Time[1]

A graph-coloring register allocator that optimally allocates registers for structured programs in polynomial time is presented. It can handle register aliasing. The assignment of registers is optimal with respect to spill and rematerialization costs, register preferences and coalescing. The register allocator is not restricted to programs in SSA form or chordal interference graphs. It assumes the number of registers is to be fixed and requires the input program to be structured, which is automatically true for many programming languages and for others, such as C, is equivalent to a bound on the number of `goto` labels per function. Non-structured programs can be handled at the cost of either a loss of optimality or an increase in runtime. This is the first optimal approach that has polynomial runtime and works for such a huge class of programs.

An implementation is already the default register allocator in most backends of a mainstream cross-compiler for embedded systems.

10.1 Introduction

Compilers map variables to physical storage space in a computer. The problem of deciding which variables to store into which registers or into memory is called register allocation. Register allocation is one of the most important stages in a compiler. Due to the ever-widening gap in speed between registers and memory the minimization of spill costs is of utmost importance. For CISC architectures, such as ubiquitous x86, register aliasing (i. e. multiple register names mapping to the same physical hardware and thus not being able to be used at the same time) and register preferences (e. g. due to certain instructions taking a different amount of time depending on which registers the operands reside in) have to be handled to generate good code. Coalescing (eliminating moves by assigning variables to the same registers, if they do not interfere, but are related by a copy instruction) is another aspect, where register allocation can have a significant impact on code size and speed.

Our approach is based on graph coloring and assumes the number of registers

[1] Previously presented at CC 2013 [84].

to be fixed. It can handle arbitrarily complex register layouts, including all kinds of register aliasing. Register preferences, coalescing and spilling are handled using a cost function. Different optimization goals, such as code size, speed, energy consumption, or some aggregate of them can be handled by choice of the cost function. The approach is particularly well-suited for embedded systems, which often have a small number of registers, and where optimization is of utmost importance due to constraints on energy consumption, monetary cost, etc. We have implemented a prototype of our approach, and it has become the default register allocator in most backends of SDCC [41], a mainstream C cross-compiler for architectures commonly found in embedded systems. Virtually all programs are structured, and for these the register allocator has polynomial runtime. This is the first optimal approach that has polynomial runtime and works for such a huge class of programs.

Chaitin's classic approach to register allocation [26] uses graph coloring. The approach assumes r identical registers, identical spill cost for all variables, and does not handle register preferences or coalescing. Solving this problem optimally is equivalent to finding a maximal r-colorable induced subgraph in the interference graph of the variables and coloring it. In general this is NP-hard [27]. Even when it is known that a graph is r-colorable it is NP-hard to find a r-coloring compatible with a fraction of $1 - \frac{1}{33r}$ of the edges [57]. Thus Chaitin's approach uses heuristics instead of optimally solving the problem. It has been generalized to more complex architectures [128]. The maximum r-colorable induced subgraph problem for fixed r can be solved optimally in polynomial time for chordal interference graphs [108, 141], which can be obtained when the input programs are in static single assignment (SSA) form [60]. Recent approaches have modeled register allocation as an integer linear programming (ILP) problem, resulting in optimal register allocation for all programs [53, 48]. However ILP is NP-hard, and the ILP-based approaches tend to have far worse runtime compared to graph coloring. There are also approaches modeling register allocation as a partitioned boolean quadratic programming (PBQP) problem [124, 62]. They can handle some irregularities in the architecture in a more natural way than older graph-coloring approaches, but do not handle coalescing and other interactions that can arise out of irregularities in the instruction set. PBQP is NP-hard, but heuristic solvers seem to perform well for many, but not all practical cases. Linear scan register allocation [109] has become popular for just in time compilation [43]; it is typically faster than approaches based on graph coloring, but the assignment is further away from optimality. Kannan and Proebsting [70] were able to approximate a simplified version of the register allocation problem within a factor of 2 for programs that have series-parallel control-flow graphs (a subclass of 2-structured programs). Thorup [132] uses the bounded tree-width of structured programs to approximate an optimal coloring of the intersection graph by a constant factor. Bodlaender et alii [18] present an algorithm that decides in linear time if it is possible to allocate registers for a structured program without spilling.

Section 10.2 introduces the basic concepts, including structured programs. Section 10.3 presents the register allocator in its generality and shows its polynomial runtime. Section 10.4 discusses further aspects of the allocator, including ways to reduce the practical runtime and how to handle non-structured programs. Section 10.5 discusses the complexity of register allocation and why certain NP-hardness results do not apply in our setting. Section 10.6 presents the

```
#include <stdint.h>
#include <stdbool.h>

bool get_pixel(uint_fast8_t x, uint_fast8_t y);
void set_pixel(uint_fast8_t x, uint_fast8_t y);

void fill_line_left(uint_fast8_t x, const uint_fast8_t y)
{
    for(;; x--)
    {
        if(get_pixel(x, y))
            return;
        set_pixel(x, y);
    }
}
```

Figure 10.1: C code example

prototype implementation, followed by the experimental results in Section 10.7. Section 10.8 concludes and proposes possible directions for future work.

10.2 Problem Description

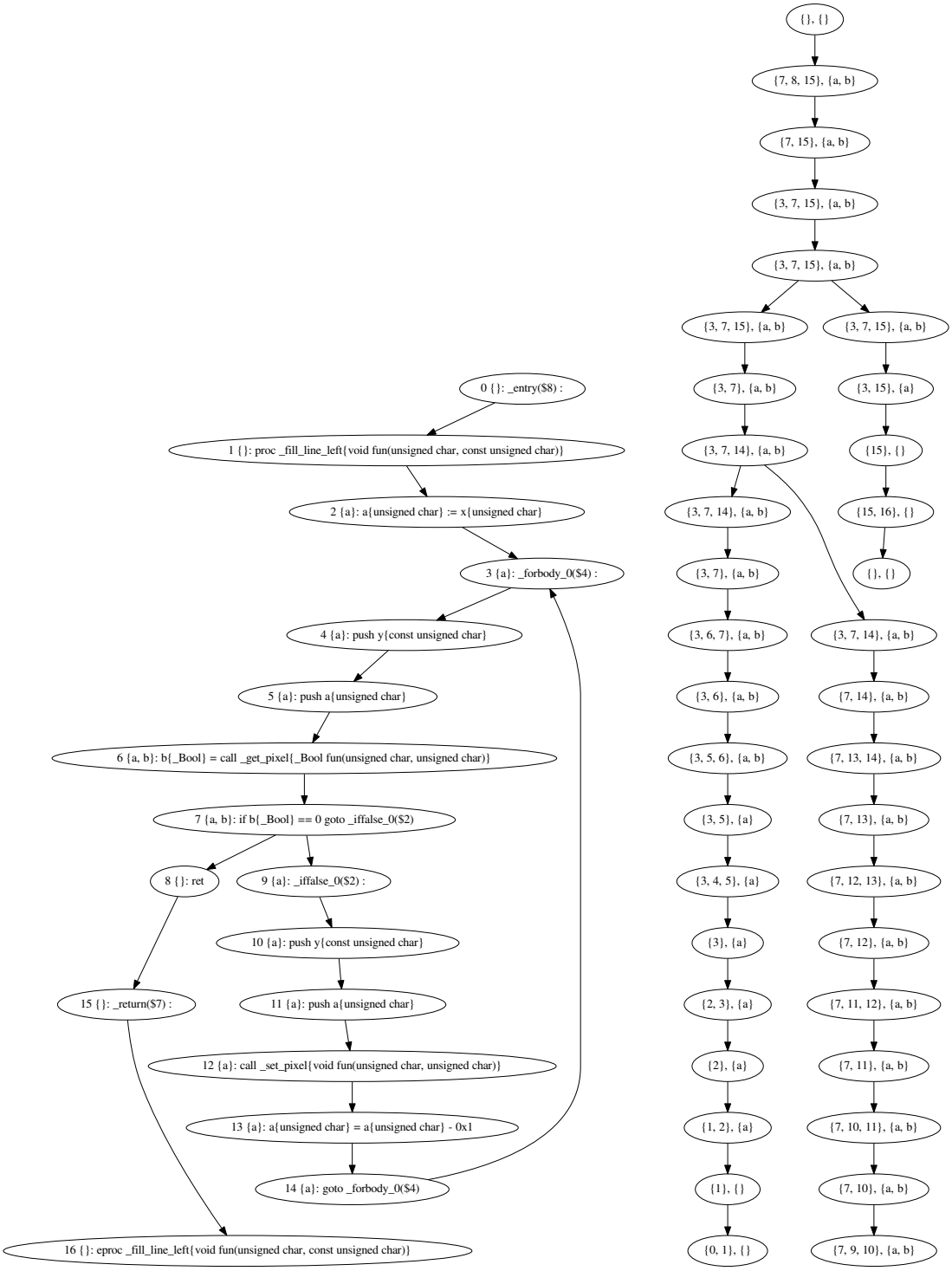
Compilers transform their input into an intermediate representation, on which they do many optimizations. At the time when register allocation is done, we deal with such an intermediate representation. We also have the *control-flow graph* (CFG) (Π, K), with node set Π and edge set $K \subseteq \Pi^2$ which represents the control flow between the instructions in the intermediate representation. For the code written in the C programming language from Figure 10.1, the C compiler SDCC [41] generates the CFG in Figure 10.2a. In this figure, the nodes of the the CFG are numbered, and annotated with the set of variables alive there, and the intermediate representation. As can be seen, SDCC introduced two temporary variables, which make up the whole set of variables $V = \{a, b\}$ to be handled by the register allocator for this code.

Let r be the number of registers. Let $[\![r]\!] := \{0, \ldots, r-1\}$ be the the set of registers.

Definition 10.1. *Let V be a set of variables. An assignment of variables V to registers $[\![r]\!]$ is a function $f\colon U \to [\![r]\!], U \subseteq V$. The assignment is valid if it is possible to generate correct code for it, which implies that no conflicting variables are assigned to the same register.*

Variables in $V \setminus U$ are to be placed in memory (spilt) or removed and their value recalculated as needed (rematerialized).

Definition 10.2 (Register allocation)**.** *Let the number of available registers be fixed. Given an input program containing variables and their live-ranges and a cost function, that gives costs for register assignments, the problem of register allocation is to find an assignment of variables to the registers that minimizes the total cost.*



(a) control-flow graph (b) nice tree-decomposition

Figure 10.2: CFG and decomposition example

Compared to Definition 2.13, we can relax the requirements on $|\chi(j)|$ and $|\chi(i)|$ in the introduce and forget nodes:

Definition 10.3 (Nice Tree-Decomposition)**.** *A tree-decomposition $(T, \mathcal{X})$ of a graph G is called nice, if*

- *T is oriented, with root t, $\chi(it) = \varnothing$.*
- *Each node i of T is of one of the following types:*
  - *Leaf node, no children*
  - *Introduce node, has one child $j, \chi(j) \subsetneq \chi(i)$*
  - *Forget node, has one child $j, \chi(j) \supsetneq \chi(i)$*
  - *Join node, has two children $j_1, j_2, \chi(i) = \chi(j_1) = \chi(j_2)$*

From now on let $G = (\Pi, K)$ be the control-flow graph of the program, let $I = (V, E)$ be the corresponding conflict graph of the variables of the program (i. e. the intersection graph of the variables' live-ranges). The live-ranges are connected subgraphs of G. Let $(T, \mathcal{X})$ be a nice tree-decomposition of minimum width of G with root t. For $\pi \in \Pi$ let V_π be the set of all variables $v \in V$, that are alive at π (in the example CFG in figure 10.2a this is the set directly after the node number). Let $\mathcal{V} := \max_{\pi \in \Pi}\{|V_\pi|\}$ be the maximum number of variables alive at the same node. For each node i of T let $V_i := \bigcup_{\pi \in \chi(i)} V_\pi$ be the set of variables alive at any of the corresponding nodes from the CFG (in the nice tree-decomposition in figure 10.2b this is the right one of the sets at each node).

10.3 Optimal Polynomial Time Register Allocation

The goal in register allocation is to minimize costs, including spill and rematerialization costs, costs from not respecting register preferences, costs from not coalescing, etc. These costs are modeled by a cost function that gives costs for an instruction π under register assignment f:

$$c\colon \left\{(\pi, f) \;\middle|\; f\colon U \to [\![r]\!], U \subseteq V_\pi, \pi \in \Pi\right\} \to [0, \infty]$$

Different optimization goals, such as speed or code size can be implemented by choosing c. E. g. when optimizing for code size c could give the code size for π under assignment f, or when optimizing for speed c could give the number of cycles π needs to execute multiplied by an execution probability obtained from a profiler. We assume that c can be evaluated in constant time. The goal is thus finding an f for which $\sum_{\pi \in \Pi} c(\pi, f|_{V_\pi})$ is minimal.

Let S be the function that gives the minimum possible costs for instructions in the subtree rooted at node i of T, excluding instructions in $\chi(i)$ when assigning variables alive in the subtree rooted at node i of T when choosing $f\colon U \to [\![r]\!], U \subseteq V$ as the assignment of variables alive at instructions $i \subseteq \Pi$ to registers, i. e.

$$S\colon \left\{(i,f) \;\middle|\; i \text{ node of } T, f\colon U \to [\![r]\!], U \subseteq V_i\right\} \to [0,\infty].$$

$$S(i,f) := \min_{g|_{V_i}=f|_{V_i}} \left\{ \sum_{\pi\in T_i} c(\pi, g|_{V_\pi}) \right\}.$$

Where T_i is the set of instructions in the subtree of T rooted at node i of T, excluding instructions in $\chi(i)$. This function at the root t of T, and the corresponding assignment that results in the minimum is what we want:

$$S(t,f) = \min_{g|_{V_t}=f|_{V_t}} \left\{ \sum_{\pi\in T_t} c(\pi, g|_{V_\pi}) \right\} =$$

$$= \min_{g|_{\varnothing}=f|_{\varnothing}} \left\{ \sum_{\pi\in \Pi} c(\pi, g|_{V_\pi}) \right\} = \min_{g} \left\{ \sum_{\pi\in \Pi} c(\pi, g|_{V_\pi}) \right\}.$$

To get S, we first define a function s, and then show that $S = s$ and that s can be calculated in polynomial time. We define s inductively, and depending on the type of i:

- Leaf: $s(i,f) := 0$
- Introduce with child j: $s(i,f) := s(j, f|_{V_j})$
- Forget with child j: $s(i,f) := \min\{\sum_{\pi\in\chi(j)\setminus\chi(i)} c(\pi, g|_{V_\pi}) + s(j,g) \mid g|_{V_i} = f\}$
- Join with children j_1 and j_2: $s(i,f) := s(j_1,f) + s(j_2,f)$

By calculating all the $s(i,f)$ and recording which g gave the minimum we can obtain an optimal assignment. We will show that s correctly gives the minimum possible cost and that it can be calculated in polynomial time.

Lemma 10.4. *For each node i of $T, f\colon U \to [\![r]\!], U \subseteq V_i$ the value $s(i,f)$ is the minimum possible cost for instructions in the subtree rooted at node i of T, excluding instructions in $\chi(i)$ when assigning variables alive in the subtree rooted at node i of T when choosing f as the assignment of variables alive at instructions $i \subseteq \Pi$ to registers, i. e. $s = S$. Using standard bookkeeping techniques we obtain the corresponding assignments for the subtree.*

Proof. By induction we can assume that the lemma is true for all children of i. Let T_i be the set of instructions in the subtree rooted at node i of T, excluding instructions in $\chi(i)$.

Case 1: i is a leaf. There are no instructions in $T_i = \chi(i) \setminus \chi(i) = \varnothing$, thus the cost is zero: $s(i,f) = 0 = S(i,f)$.

Case 2: i is an introduce node with child j. $T_i = T_j$, since $\chi(i) \supseteq \chi(j)$, thus the cost remains the same: $s(i,f) = s(j,f) = S(j,f) = S(i,f)$

Case 3: i is a forget node with child j. $T_i = T_j \cup (\chi(j) \setminus \chi(i))$, the union is disjoint. Thus we get the correct result by adding the costs for the instructions in $\chi(j) \setminus \chi(i)$:

$$s(i,f) = \min_{g|_{V_i}=f} \left\{ \sum_{\pi \in \chi(j) \setminus \chi(i)} c(\pi, f|_{V_\pi}) + s(j,g) \right\} =$$
$$\min_{g|_{V_i}=f} \left\{ \sum_{\pi \in \chi(j) \setminus \chi(i)} c(\pi, f|_{V_\pi}) + S(j,g) \right\} =$$
$$\min_{g|_{V_i}=f|_{V_i}} \left\{ \sum_{\pi \in \chi(j) \setminus \chi(i)} c(\pi, f|_{V_\pi}) + \sum_{\pi \in T_j} c(\pi, g|_{V_\pi}) \right\} =$$
$$\min_{g|_{V_i}=f|_{V_i}} \left\{ \sum_{\pi \in T_i} c(\pi, g|_{V_\pi}) \right\} = S(i,f).$$

Case 4: i is a join node with children j_1 and j_2. $T_i = T_{j_1} \cup T_{j_2}$, since $\chi(i) = \chi(j_1) = \chi(j_2)$. The union is disjoint. Thus we get the correct result by adding the costs from both subtrees: $s(i,f) = s(j_1,f) + s(j_2,f) = S(j_1,f) + S(j_2,f) = S(i,f)$. q. e. d.

Lemma 10.5. *Given the tree-decomposition of minimum width, s can be calculated in polynomial time.*

Proof. Each $V_\pi, \pi \in \Pi$ is the union of two cliques, each of size at most $\mathcal{V}$: The variables alive at the start of the instruction form the clique, and so do the variables alive at the end of the instruction. Thus V_i, i node of T is the union of at most $2(\mathrm{tw}(G)+1)$ cliques. From each clique at most r variables can be placed in registers.

At each node i of the tree-decomposition time $O(\mathcal{V}^{2(\mathrm{tw}(G)+1)r})$ is sufficient:

Case 1: i is a leaf. There are at most $O(\mathcal{V}^{2|\chi(i)|r}) \subseteq (\mathcal{V}^{2(\mathrm{tw}(G)+1)r})$ possible f, and for each one we do a constant number of calculations.

Case 2: i is an introduce node with child j. The reasoning from case 1 holds.

Case 3: i is a forget node. There are at most $O(\mathcal{V}^{2|\chi(i)|r})$ possible f. For each one we need to consider at most $O(\mathcal{V}^{2|\chi(j) \setminus \chi(i)|r})$ different g. Thus time $O(\mathcal{V}^{2|\chi(i)|r}) \cdot (\mathcal{V}^{2|\chi(j) \setminus \chi(i)|r}) \subseteq O(\mathcal{V}^{2|\chi(i)|r+2|\chi(j) \setminus \chi(i)|r}) = O(\mathcal{V}^{2|\chi(j)|r}) \subseteq O(\mathcal{V}^{2(\mathrm{tw}(G)+1)r})$ is sufficient.

Case 4: i is a join node with children j_1 and j_2. The reasoning from case 1 holds.

The tree-decomposition has at most $|T|$ nodes, thus the total time is in $O(|T|\mathcal{V}^{2(\mathrm{tw}(G)+1)r}) = O(|T|\mathcal{V}^{\mathfrak{c}})$ for a constant $\mathfrak{c}$ and thus polynomial. q. e. d.

Theorem 10.6. *The register allocation problem can be solved in polynomial time for structured programs.*

Proof. Given an input program of bounded tree-width we can calculate a tree-decomposition of minimum width in linear time [17]. We can then transform this tree-decomposition into a nice one of the same width. The linear time for these steps implies that $|T|$ is linear in $|G|$. Using this nice tree-decomposition s is calculated in polynomial time as above. The total runtime is thus in $O(|G|\mathcal{V}^{2(\mathrm{tw}(G)+1)r}) = O(|G|\mathcal{V}^{\mathfrak{c}})$ for a constant $\mathfrak{c}$. q. e. d.

10.4 Remarks

Remark 10.7. *The runtime bound is reduced by a factor of $\mathcal{V}^{\mathrm{tw}(G)r}$, if the intermediate representation is three-address code.*

Proof. In that case there is at most one variable alive at the end of an instruction that was not alive at the start of the instruction, so in the proof of Lemma 10.5, we can replace $O(\mathcal{V}^{2(\mathrm{tw}(G)+1)r})$ by $O(\mathcal{V}^{(\mathrm{tw}(G)+2)r})$. q. e. d.

Remark 10.8. *Bodlaender's algorithm [17] used in the proof above is not a practical option. However there are other, more practical alternatives, including a linear-time algorithm that is not guaranteed to give decompositions of minimal width, but will do so for many programming languages [132, 38].*

Remark 10.9. *Implementations of the algorithm can be massively parallel, resulting in linear runtime.*

Proof. At each node i of T the individual $s(i, f)$ do not depend on each other. They can be calculated in parallel. By requiring that $|\chi(j)| = |\chi(i)| + 1$ at forget nodes, we can assume that the number of different g to consider is at most $\mathcal{V}^{2r}$, resulting in time $O(r)$ for calculating the minimum over the $s(j, g)$. Thus given enough processing elements the runtime of the algorithm can be reduced to $O(|G|r)$. q. e. d.

Remark 10.10. *Doing live-range splitting as a preprocessing step is cheap.*

The runtime bound proved above only depends on $\mathcal{V}$, not $|V|$. Thus splitting of non-connected live-ranges before doing register allocation doesn't affect the bound. When the splitting is done to allow more fine-grained control over spilling, then the additional cost is small (even the extreme case of inserting permutation instructions between any two original nodes in the CFG, and splitting all live-ranges there would only double Π and $\mathcal{V}$).

Remark 10.11. *Non-structured programs can be handled at the cost of either a loss of optimality or an increase in runtime.*

Programs of high tree-width are extremely uncommon (none have been found so far, with the exception of artificially constructed examples). Nevertheless they should be handled correctly by compilers. One approach would be to handle these programs like the others. Since $\mathrm{tw}(G)$ is no longer constant, the algorithm is no longer guaranteed to have polynomial runtime. Where polynomial runtime is essential, a preprocessing step can be used. This preprocessing stage would spill some variables (or allocate them using one of the existing heuristic approaches). Edges of G, at which no variables are alive, can be removed. Once enough edges have been removed, $\mathrm{tw}(G) \leq \mathfrak{k}$ and our approach can be applied to allocate the remaining variables. Another option is the heuristic limit used in our prototype as mentioned in Section 10.6.

Remark 10.12. *The runtime of the polynomial time algorithm can be reduced by a factor of more than $(2(\mathrm{tw}(G) + 1)r)!$, if there is no register aliasing and registers are interchangeable within each class. Furthermore r can then be chosen as the maximum number of registers that can be used at the same time instead of the total number of registers, which gives a further runtime reduction in case of register aliasing.*

Publication	Difference to our setting
Register allocation via coloring [27]	$\mathrm{tw}(G)$ unbounded
On the Complexity of Register Coalescing [21]	$\mathrm{tw}(G)$ unbounded
The complexity of coloring circular arcs and chords [50]	r is part of input
Aliased register allocation for straight line programs is NP-complete [90]	r is part of input
On Local Register Allocation [45]	r is part of input

Figure 10.3: Complexity Results

Proof. Instead of using $f\colon U \to [\![r]\!]$ we can directly use U.

- Leaf: $s(i, U) := 0$
- Introduce with child j: $s(i, U) := s(j, U \cup V_j)$
- Forget with child j: $s(i, U) := \sum_{\pi \in \chi(j) \setminus \chi(i)} c(\pi, U) + \min\{s(j, W) \mid W \cap V_i = U\}$
- Join with children j_1 and j_2: $s(i, U) := s(j_1, U) + s(j_2, U)$

Most of the proofs of the lemmata are still valid. However instead of the number of possible f we now look at the number of possible U, which is at most

$$\binom{\mathcal{V}}{2(\mathrm{tw}(G)+1)r}.$$

q. e. d.

Remark 10.13. *Using a suitable cost function and $r = 1$ we get a polynomial time algorithm for maximum independent set on intersection graphs of connected subgraphs of graphs of bounded tree-width.*

Remark 10.14. *The allocator is easy to re-target, since the cost function is the only architecture-specific part.*

10.5 Complexity of Register allocation

The complexity of register allocation in different variations has been studied for a long time and there are many NP-hardness results (Figure 10.3):

Given a graph I a program can be written, such that the program has conflict graph I [27]. Since 3-colorability is NP-hard [72], this proves the NP-hardness of register allocation, as a decision problem for $r = 3$. However the result does not hold for structured programs. Coalescing is NP-hard even for programs in SSA-form [21]. Again this result does not hold for structured programs. Register allocation, as a decision problem, is NP-hard, even for series-parallel control-flow graphs, i. e. for $\mathrm{tw}(G) \leq 2$ and thus for structured programs, when the number of registers is part of the input [50]. Register allocation, as a decision problem, is NP-hard when register aliasing is possible, even for straight-line programs, i. e. $\mathrm{tw}(G) = 1$ and thus for structured programs, when the number

of registers is part of the input [90]. Minimizing spill costs is NP-hard, even for straight-line programs, i. e. $\mathrm{tw}(G) = 1$ and thus for structured programs, when the number of registers is part of the input [45].

It is thus fundamental to our polynomial time optimal approach, which handles register aliasing, register preferences, coalescing and spilling, that the input program is structured and the number of registers is fixed.

The runtime bound of our approach proven above is exponential in the number of registers r. However, even a substantially simplified version of the register allocation problem is W[SAT]- and co-W[SAT]-hard when parametrized by the number of registers even for $\mathrm{tw}(G) = 2$ (Chapter 12). Thus doing optimal register allocation in time faster than $\mathcal{V}^{O(r)}$ would imply a collapse of the parametrized complexity hierarchy. Such a collapse is considered highly unlikely in parametrized complexity theory. This means that not only we cannot get rid of the r in the exponent, but we can't even separate it from the $\mathcal{V}$ either.

10.6 Prototype implementation

We have implemented a prototype of the allocator in C++ for the HC08, S08, Z80, Z180, Rabbit 2000/3000, Rabbit 3000A and LR35902 ports of SDCC [41], a C compiler for embedded systems. It is the default register allocator for these architectures as of the SDCC 3.2.0 release in mid-2012 and can be found in the public source code repository of the SDCC project.

S08 is the architecture of the current main line of Freescale microcontrollers, a role previously filled by the HC08 architecture. Both architectures have three 8-bit registers, which are assigned by the allocator. The Z80 architecture is a classic architecture designed by Zilog, which was once common in general-purpose computers. It currently is mostly used in embedded systems. The Z180, Rabbit 2000/3000 and Rabbit 3000A are newer architectures derived from the Z80, which are also mostly used in embedded systems. The differences are in the instruction set, not in the register set. The Z80 architecture is simple enough to be easily understood, yet has many of the typical features of complex CISC architectures. Nine 8-bit registers are assigned by the allocator (A, B, C, D, E, H, L, IYL, IYH). IYL and IYH can only be used together as 16-bit register IY; there are instructions that treat BC, DE or HL as 16-bit registers; many 8-bit instructions can only use A as the left operand, while many 16-bit instructions can only use HL as the left operand. There are some complex instructions, like djnz, a decrement-and-jump-if-not-zero instruction that always uses B as its operand, or ldir, which essentially implements memcpy() with the pointer to the destination in DE, source pointer in HL and number of bytes to copy in BC. All these architectural quirks are captured by the cost function. The LR35902 is the CPU used in the Game Boy video game system. It is inspired by the Z80 architecture, but has a more restricted instruction set and fewer registers. Five 8-bit registers are assigned by the allocator.

The prototype still has some limitations, e. g. current code generation does not allow the A or IY registers to hold parts of a bigger variable in the Z80 port. Code size was used as the cost function, due to its importance in embedded systems and relative ease of implementation (optimal speed or energy optimization would require profiler-guided optimization). We obtain the tree-decomposition using Thorup's method [132], and then transform it into a nice

tree-decomposition. The implementation of the allocator essentially follows Section 10.3, and is neither very optimized for speed nor parallelized. However a configurable limit on the number of assignments considered at each node of the tree-decomposition has been introduced. When this limit is reached, some assignments are discarded heuristically. The heuristic mostly relies on the $s(i, f)$ to discard those assignments that have the highest cost so far first, but takes other aspects into account to increase the chance that compatible assignments will exist at join nodes. When the limit is reached, and the heuristic applied, the assignment is no longer provably optimal. This limit essentially provides a trade-off between runtime and quality of the assignment.

The prototype was compared to the current version of the old SDCC register allocator, which has been improved over years of use in SDCC. The old allocator is basically an improved linear scan [109, 43] algorithm extended to take the architecture into account, e.g. preferring to use registers HL and A, since accesses to them typically are faster than those to other registers and taking coalescing, register aliasing and some other preferences into account. This comparison was done using the Z80 architecture, which has been around for a long time, so there is a large number of programs available for it.

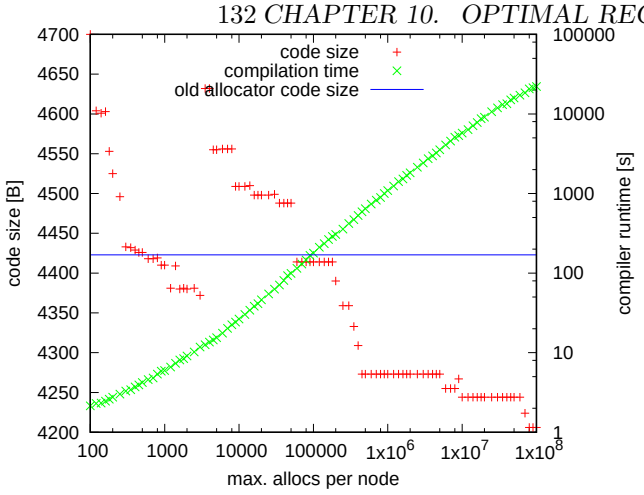
Furthermore we did a comparison between the different architectures, which shows the impact of the number of registers on the performance of the allocator.

10.7 Experimental results

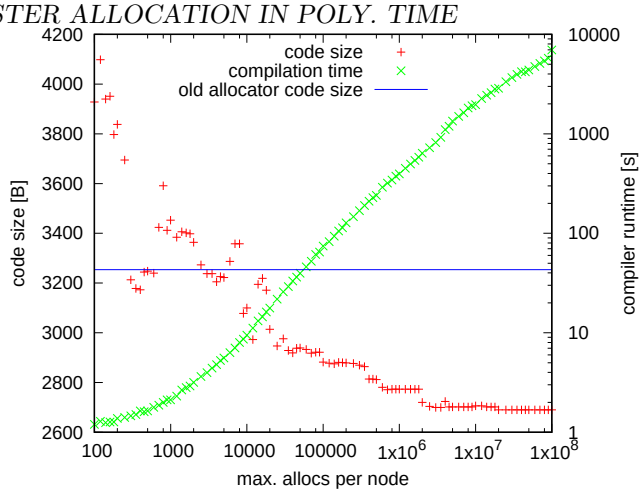
Six benchmarks considered representative of typical applications for embedded systems have been used to evaluate the register allocator, by compiling them with SDCC 3.2.1 #8085:

- The Dhrystone benchmark [137], version 2 [138]. An ANSI-C version was used, since SDCC does not yet support K&R C.
- A set of source files taken from real-world applications and used by the SDCC project to track code size changes over SDCC revisions and to compare SDCC to other compilers.
- The Coremark benchmark [49], version 1.0.
- The FatFS implementation of the FAT filesystem [28], version R0.09.
- Source code from two games for the ColecoVision video game console. All C source code has been included, while assembler source files and C source files that only contain data have been omitted.
- The Contiki operating system [40], version 2.5.

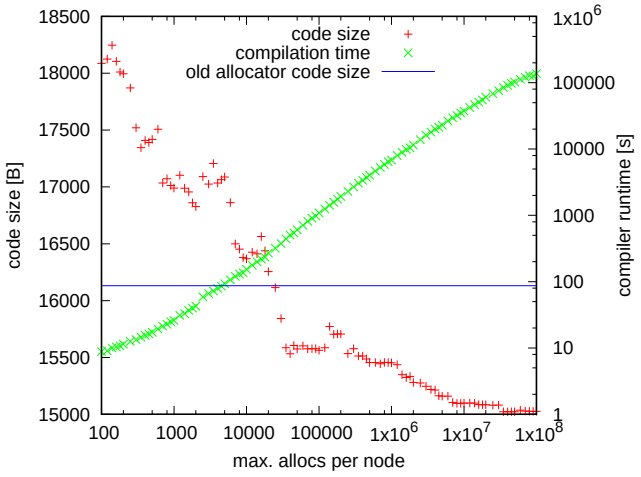
We first discuss the results of compiling the benchmarks for the Z80 architecture. Figure 10.4 shows the code size with the peephole optimizer (a post code-generation optimization stage not taken into account by the cost function) enabled, Figure 10.5 with the peephole optimizer disabled. Furthermore, Figure 10.4 shows the compilation time, and Figure 10.5 shows the fraction of provably optimally allocated functions (i. e. those functions for which the heuristic never was applied); the former is little affected by enabling the peephole optimizer and the latter not at all.



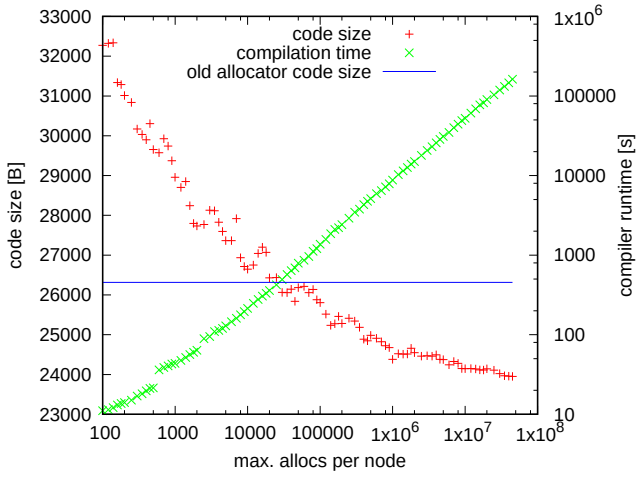
(a) Dhrystone



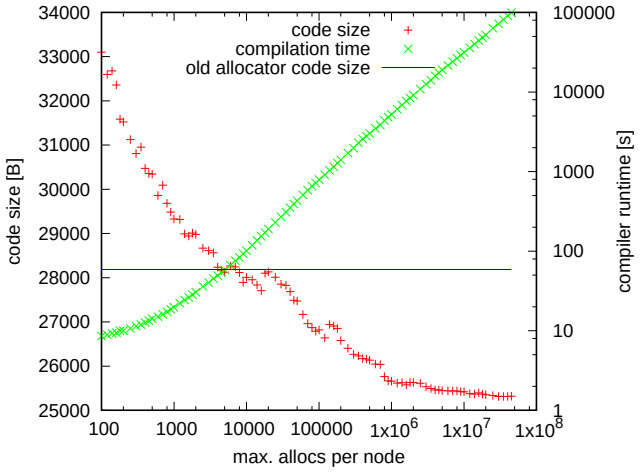
(b) SDCC benchmark



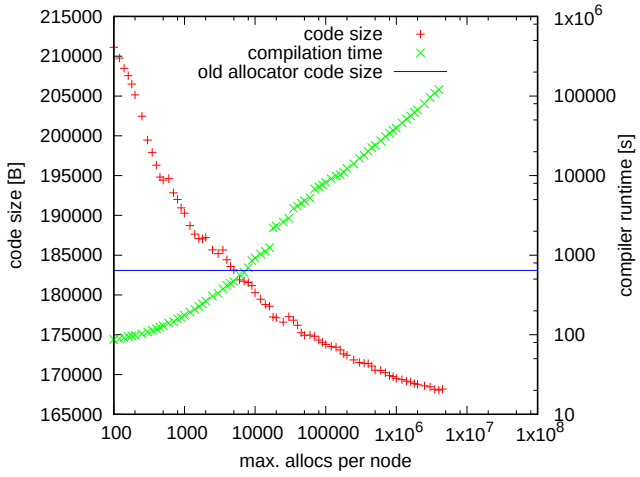
(c) Coremark



(d) FatFS

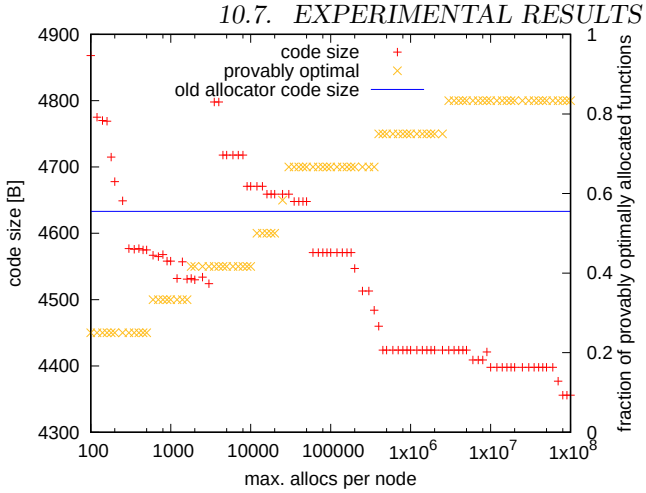


(e) games

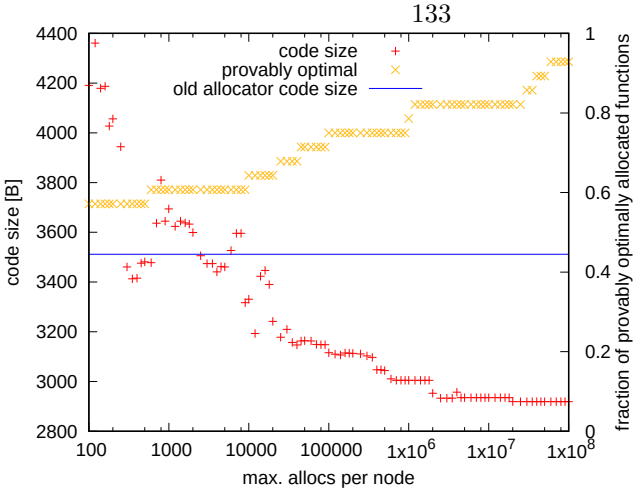


(f) Contiki

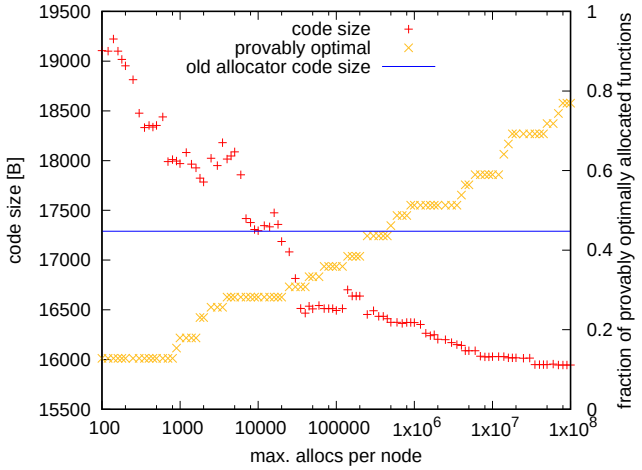
Figure 10.4: Experimental Results (Z80, with peephole optimizer)



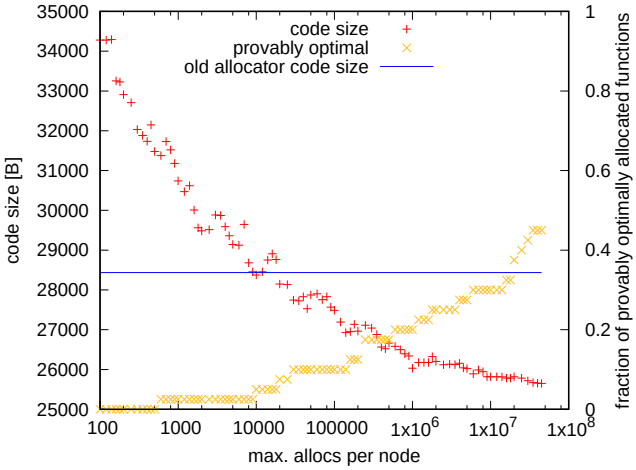
(a) Dhrystone



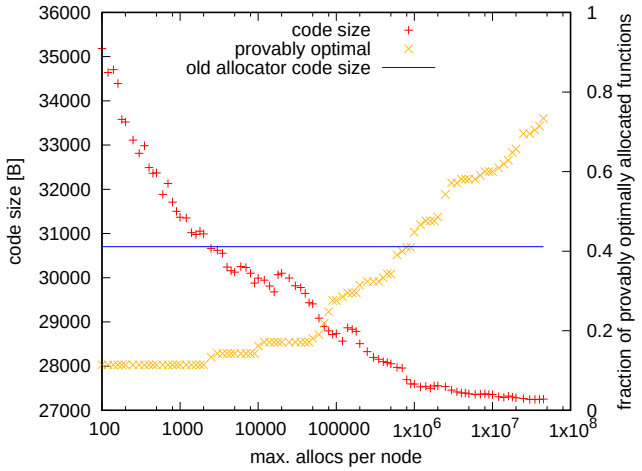
(b) SDCC benchmark



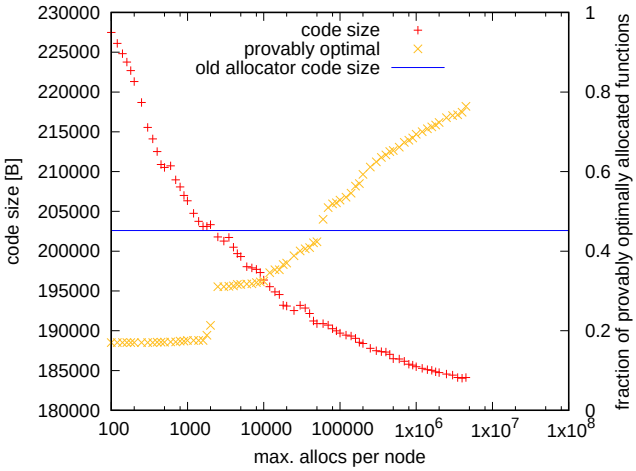
(c) Coremark



(d) FatFS



(e) games



(f) Contiki

Figure 10.5: Experimental Results (Z80, without peephole optimizer)

The Dhrystone benchmark is rather small. At 10^8 assignments per node we find a provably optimal assignment for 83.3% of the functions. This also results in a moderate reduction in code size of 6.0% before and 4.9% after the peephole optimizer when compared to the old allocator. The SDCC benchmark, even though small, contains more complex functions; at 10^8 assignments per node we find a provably optimal assignment for 93.9% of the functions. However code size seems to be stable from 6×10^7 onwards. We get a code size reduction of 16.9% before and 17.3% after the peephole optimizer. For Coremark, we find an optimal assignment for 77% of the functions at 10^8 assignments per node. We get a code size reduction of 7.8% before and 6.9% after the peephole optimizer.

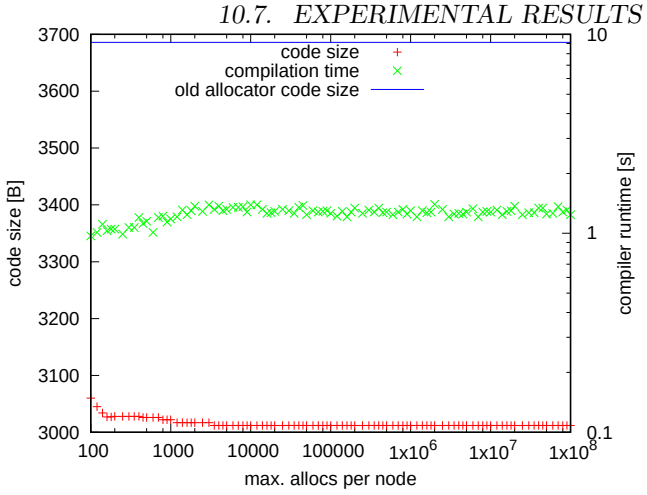
FatFS is the benchmark which is the most problematic for our allocator; it contains large functions with complex control flow, some containing nearly a kilobyte of local variables. Even at 4.5×10^7 assignments per node (we did not run compilations at higher values due to lack of time) only 45% of the functions are provably optimally allocated. We get a reduction in code size of 9.8% before and 11.4% after the peephole optimizer. Due to the low fraction of provably optimally allocated functions the code size reduction and compilation time are likely to be much higher for a higher number of assignments per node.

In the games benchmark, about 73% of the functions are provably optimally allocated at 4.5×10^7 assignments per node; at that value the code size is reduced by 11.2% before and 12.3% after the peephole optimizer. This result is consistent with the previous two: The source code contains both complex and simple functions (and some data, since only source files containing data only were excluded, while those that contain both code and data were included).

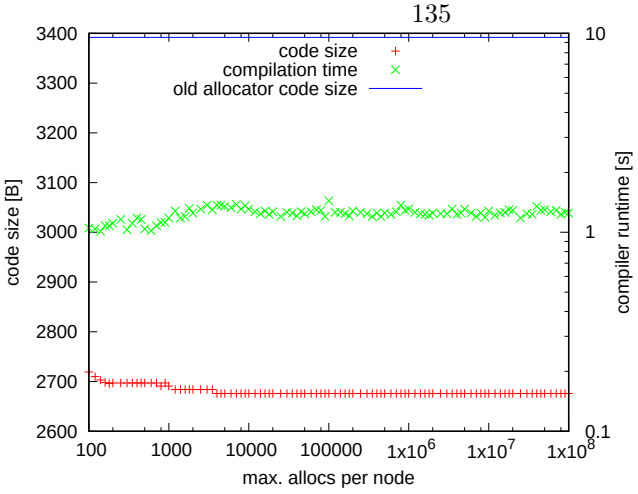
For Contiki, about 76% of the functions are provably optimally allocated at 4.5×10^6 assignments per node (we did not run compilations at higher values due to lack of time); at that value the code size is reduced by 9.1% before and 8.2% after the peephole optimizer. Contiki contains some complex control flow, but it tends to use global instead of local variables; where there are local variables they are often 32-bit variables, of which neither the optimal nor the old allocator can place more than one in registers at a given time (due to the restriction in code generation that allows the use of IY for 16-bit variables only).

We also did a comparison of the different architectures (except for the Rabbit 3000A, since it is very similar to the Rabbit2000/3000). Figure 10.6 shows the code size with the peephole optimizer enabled, Figure 10.7 with the peephole optimizer disabled. Furthermore, Figure 10.6 shows the compilation time, and Figure 10.7 shows the fraction of provably optimally allocated functions.

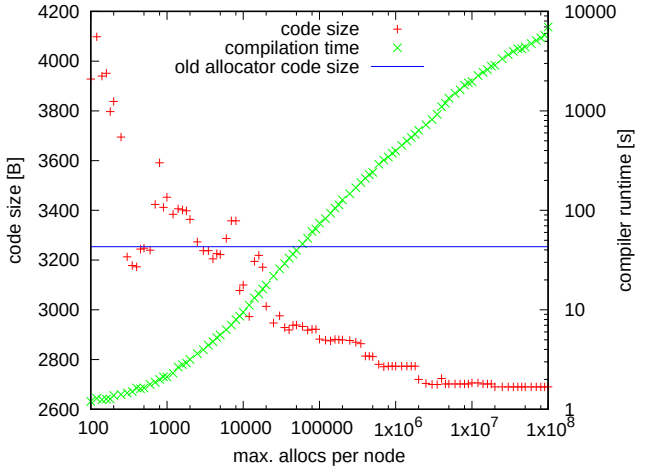
The results clearly show that for the runtime of the register allocator and the fraction of provably optimally allocated function the number of registers is much more important than the other aspects of the architecture: For the architectures with 3 registers, code size is stable from 3.8×10^3 (for HC08) and 4.0×10^3 (for S08) assignments, and all functions are provably optimally allocated from 2.5×10^4 assignments onwards. The effect of the register allocator on compiler runtime is mostly lost in noise. For the architecture with 5 registers (LR35902), code size is stable from 9.0×10^3 assignments onwards, and all functions are provably optimally allocated from 1.4×10^5 assignments onwards. For the architectures with 9 registers, there are still functions for which a provably optimal assignment is not found at 1.0×10^8 assignments. Architectural differences other than the number of registers have a substantial impact on code size, but only a negligible one on the performance of the register allocator.



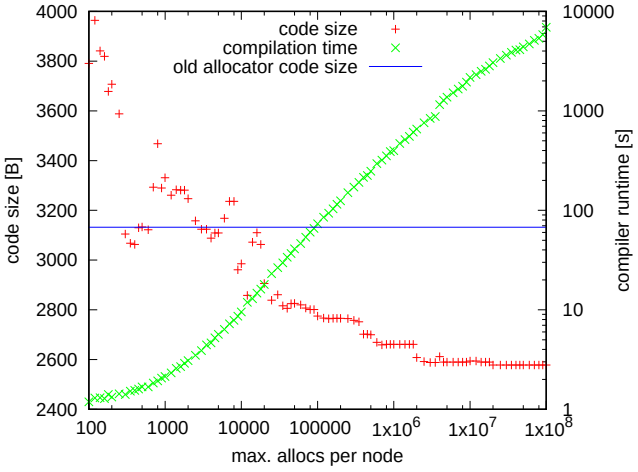
(a) HC08



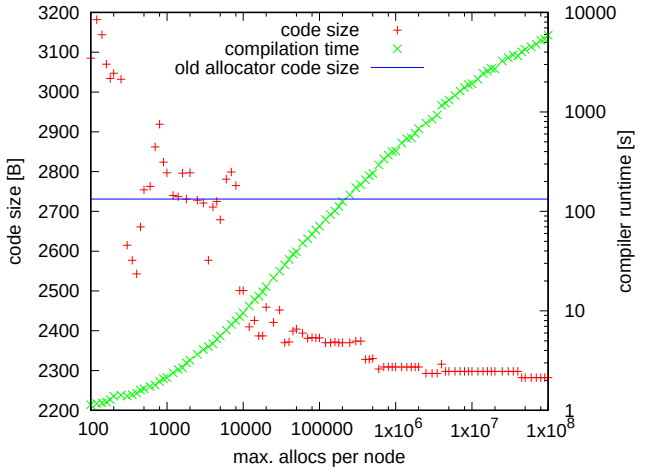
(b) S08



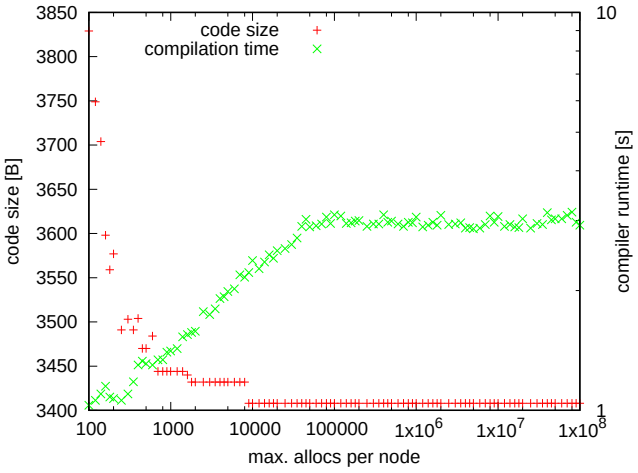
(c) Z80



(d) Z180

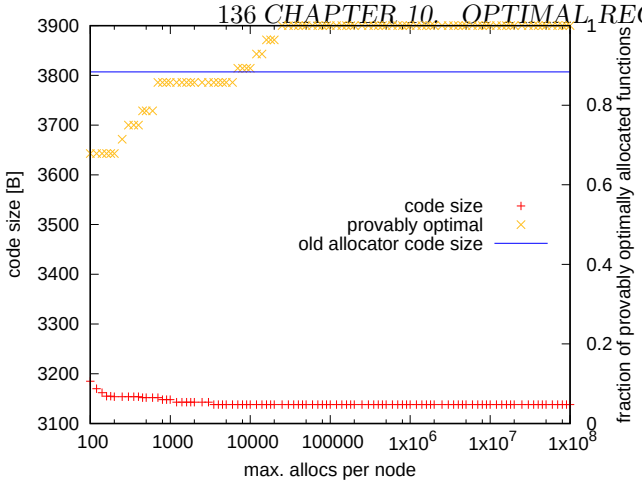


(e) Rabbit 2000/3000

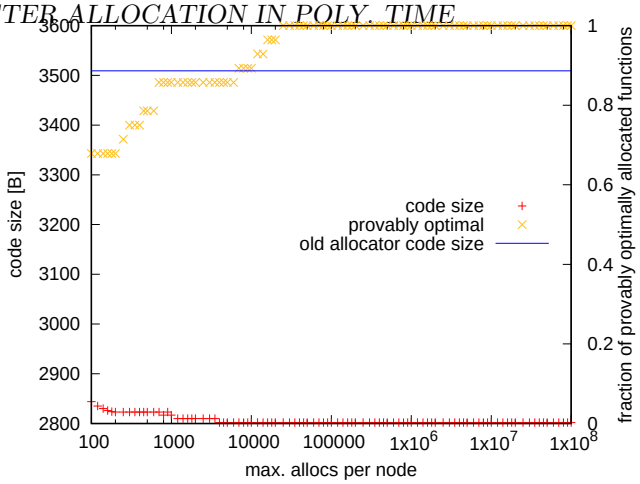


(f) LR35902

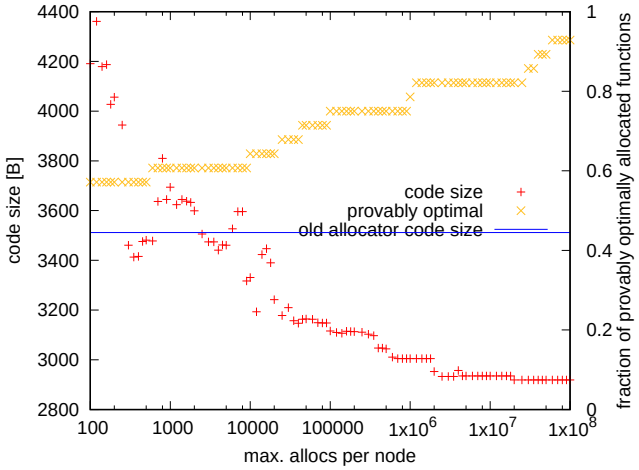
Figure 10.6: Experimental Results (SDCC benchmark, with peephole optimizer)



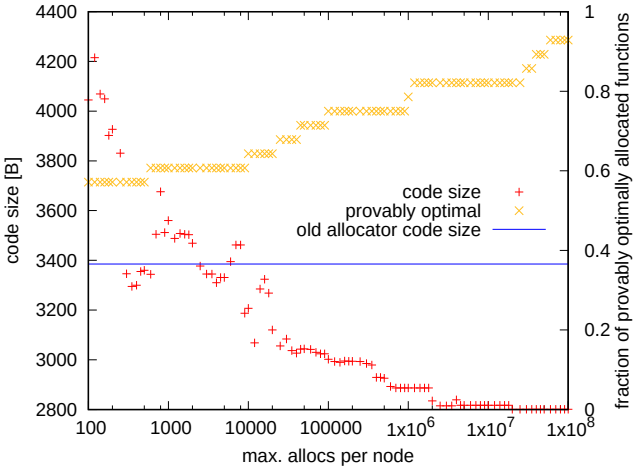
(a) HC08



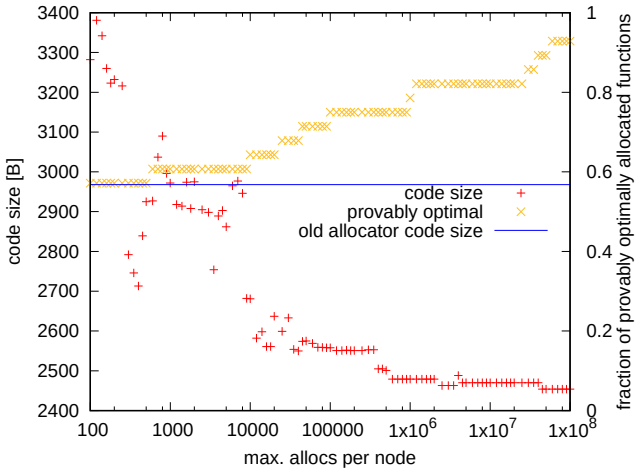
(b) S08



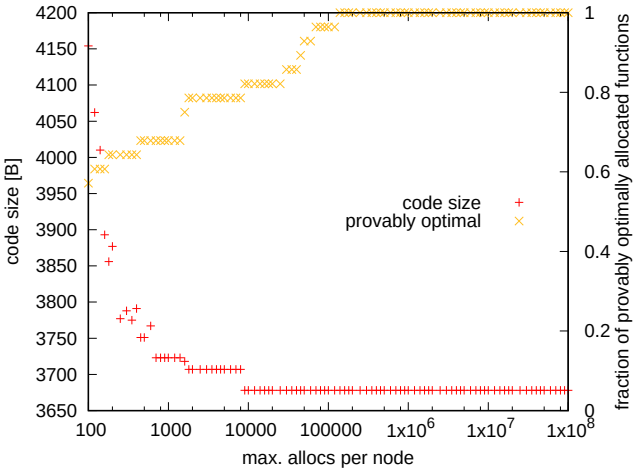
(c) Z80



(d) Z180



(e) Rabbit 2000/3000



(f) LR35902

Figure 10.7: Experimental Results (SDCC benchmark, without peephole optimizer)

We also see that the improvement in code size compared to the old allocator was the most substantial for architectures that have just three registers: For the HC08 17.6% before and 18% after the peephole optimizer, for the S08 20.1% before and 21.1% after the peephole optimizer. This has substantially reduced, but not completely eliminated the gap in generated code size between SDCC and the competing Code Warrior and Cosmic C compilers. The Z180 and Rabbit 2000/3000 behave similar to the Z80, which was already discussed above. Our register allocator makes SDCC substantially better in generated code size than the competing z88dk, HITECH-C and CROSS-C compilers for these architectures. The LR35902 backend had been unmaintained in SDCC for some time, and was brought back to life after the 3.1.0 release, at which time it was not considered worth the effort to make the old register allocator work with it. There is no other current compiler for the LR35902.

10.8 Conclusion

We presented an optimal register allocator, that has polynomial runtime. Register allocation is one of the most important stages of a compiler. Thus the allocator is a major step towards improving compilers. The allocator can handle a variety of spill and rematerialization costs, register preferences and coalescing.

A prototype implementation shows the feasibility of the approach, and is already in use in a major cross-compiler targeting architectures found in embedded systems. Experiments show that it performs excellently for architectures with a small number of registers, as common in embedded systems.

Future research could go towards improving the runtime further, completing the prototype and creating a massive parallel implementation. This should make the approach feasible for a broader range of architectures.

Chapter 11

Bytewise Register Allocation[1]

Traditionally, variables have been considered as atoms by register allocation: Each variable was to be placed in one register, or spilt (placed in main memory) or rematerialized (recalculated as needed). Some flexibility arose from what would be considered a register: Register aliasing allowed to treat a register meant to hold a 16-bit variable as two registers that could hold an 8-bit variable each. We allow for far more flexibility in register allocation: We decide on the storage of variables bytewise, i. e. we decide for each individual byte in a variable whether to store it in memory or a register, and consider any byte of any register as a possible storage location.

We implemented a backend for the STM8 architecture (STMicroelectronics' current 8-bit architecture) in the C compiler SDCC, and experimentally evaluate the benefits of bytewise register allocation. The results show that bytewise register allocation can result in substantial improvements in the generated code. Optimizing for code size we obtained 27.2%, 13.2% and 9.2% reductions in code size in the Whetstone, Dhrystone and Coremark benchmarks, respectively, when using bytewise allocation and spilling compared to conventional allocation.

11.1 Introduction

Traditionally, variables have been considered as atoms by register allocation: Each variable was to be placed in one register, or spilt (placed in main memory) or rematerialized (recalculated as needed). Some flexibility arose from what would be considered as a register. E. g., in the Zilog Z80 architecture, there are register pairs that are meant to be used either as 16 bit registers (`hl`, `de`, `bc`) or as two individual 8-bit (`h` and `l`, `d` and `e`, `b` and `c`) registers each. In the case of the Z80, this flexibility was intended by the designers of the architecture. The 16 bit registers are said to alias with their 8-bit parts. Often, compilers will stick to the vision of the architects, but sometimes also come up with their own idea of what a register could be; e. g. some compilers consider `dehl` as a 32-bit register, which aliases with the 16-bit registers `hl` and `de`, and the 8-bit

[1] Previously presented at SCOPES 2015 [86].

registers `h`, `l`, `d` and `e`. However, in the end, all these registers originate in the vision of the architects and the imagination of the compiler writers.

We investigate whether more flexibility in register allocation is worth the effort: In a first step, we consider any combination of bytes of registers as a register: Any two bytes in registers form a 16-bit register, any 4 bytes in registers form a 32-bit register, resulting in extreme register aliasing. An equivalent perspective on the same technique would be to only consider 8-bit registers, and break each variable down into its individual bytes, and allocate each of the bytes of the variable into an 8-bit register. This perspective leads us to the second step, of also making the spill decision for each individual byte of each variable. We now have the full flexibility of putting any byte of any variable into any register or spilling the byte.

Bytewise register allocation is challenging for both code generation and the register allocator: Code generation needs to be able to deal with wherever the operands have been allocated to. Register allocation needs to be able to deal with a highly irregular register architecture. E. g., the Z80 mentioned above has instructions `add hl, rr` for `rr` being one of `bc`, `de` or `hl`. Thus 16-bit additions are cheap when one operand is in `hl` and the other in `bc` or `de`; the same applies for one operand in `hc` and the other in `bl` or one in `he` and the other in `dl`. But 16-bit additions would be expensive with e. g. one operand in `db` and the other in `ld`.

Traditional graph-coloring register allocators, such as Chaitin's original [26] approach, as well as later graph-coloring approaches [60, 94] are elegant when used for architectures with interchangeable, non-aliasing registers. While they can be generalized to handle some architectural irregularities [128, 131], they cannot handle the extreme irregularities and aliasing resulting from bytewise register allocation well. Many of the recent graph-coloring approaches are decoupled, i. e. they handle register assignment and spilling in separate stages of the register allocator [131, 13, 30]. There are some approaches based on integer linear programming (ILP) [53, 48] and partitioned boolean quadratic programming (PBQP) [124, 62]. For them, handling the irregularities well is somewhat easier, but still challenging; ILP and PBQP are NP-hard problems, which can be problematic for the runtime of the allocators. There also are decoupled approaches based on ILP [8] and PBQP [22]. Our approach in the previous Chapter 10 based on graph-structure theory combines the elegance of graph-coloring approaches with the ability to easily handle any irregularities and aliasing.

In register allocation, a topic related to bytewise register allocation is bitwidth-aware register allocation [130, 12, 103, 11]. In bitwidth-aware register allocation, variables of a certain width are allocated to continuous parts of registers. In the case of two 8-bit variables and a 16-bit register both bitwidth-aware register allocation and bytewise register allocation may allocate each variable to a part of the register. However, in the case of two 16-bit variables, in which all bits are important, but the lower bits are read more often, bytewise register allocation would consider to allocate the lower byte of each variable into part of the register, and spill the upper byte. Bitwidth-aware register allocation could not, since it does not split variables into individual bytes. On the other hand, in the case of a 16-bit variable, of which only 12 bits are really used, and an 8-bit variable, of which only 4 bits are really used, bytewise register allocation will need three bytes of storage (any of which might be in memory or part of

any register), while bitwidth-aware register allocation might be able to fit them both into the same 16-bit register. So bitwidth-aware register allocation and bytewise register allocation are mostly orthogonal.

11.2 Motivation

```
ld    x, (d1, sp)
addw  x, #c01
ld    (d1, sp), x
ld    a, (d2, sp)
adc   a, #c2
ld    (d2, sp), a
ld    a, (d3, sp)
adc   a, #c3
ld    (d3, sp), a
```

(a) Fully spilt 16 Bytes

```
ld    a, (d0, sp)
add   a, #c0
ld    (d0, sp), a
ld    a, (d1, sp)
adc   a, #c1
ld    (d1, sp), a
ld    a, (d2, sp)
adc   a, #c2
ld    (d2, sp), a
ld    a, (d3, sp)
adc   a, #c3
ld    (d3, sp), a
```

(b) Fully spilt 20 Bytes

```
addw  x, #c01
ld    a, yl
adc   a, #c2
ld    yl, a
ld    a, yh
adc   a, #c3
ld    yh, a
```

(c) Not spilt 13 Bytes

```
addw  x, #c01
ld    a, (d2, sp)
adc   a, #c2
ld    (d2, sp), a
ld    a, (d3, sp)
adc   a, #c3
ld    (d3, sp), a
```

(d) Partially spilt 13 Bytes

Figure 11.1: STM8 code for 32-bit addition of a constant to a temporary variable (`ld r, g` loads the 8-bit value from `g` into `r`, `add r, g` adds the 8-bit value in `g` to `r`, `adc r, g` adds the 8-bit value in `g` and the carry bit to `r`, `addw r, g` adds the 16-bit value in `g` to `r`)

Our research into bytewise register allocation was motivated both by the availability of a register allocator that can deal with irregularities and aliasing easily, and certain features of the the index registers in the STMicroelectronics STM8 architecture.

A register allocator based on tree-decompositions [61, 114, 113], a concept from graph-structure theory, was presented recently [84]. While there was earlier theoretical work on applications of tree-decompositions in register allocation [132, 18], this is the first such register allocator, that can handle many practical complications, such as register aliasing, spilling and rematerialization costs, register preferences and coalescing; and it also is the first allocator based on tree-decompositions which has been implemented. The allocator is optimal with respect to a cost function that, depending on an instruction and a register allocation for the variables alive at the instruction, gives a cost. One possible

choice for the cost function is the size of the generated code. The allocator has a theoretical runtime bound exponential in the number of registers (which probably cannot be avoided in any optimal allocator [85]), and exponential in the tree-width of the control-flow graph. The tree-width is a small constant (unless the program excessively uses `goto` statements). The runtime is otherwise polynomial in the size of the input. This register allocator can handle bytewise allocation and spilling easily: Treat the individual bytes of the variables from the previous compiler stages as variables for register allocation. This complicates the implementation of the cost function and the code generation, but does not really require many changes in the machine-independent parts of the allocator. The runtime is still bounded by a polynomial in the input size, but the exponents of this polynomial are bigger by a factor of the size of the largest data type in bytes, compared to the situation without bytewise allocation and spilling.

The STM8 is an 8-bit microcontroller, with an 8-bit accumulator `a`, and two 16-bit index registers, `x` and `y`. For the 8-bit accumulator standard operations, such as addition, subtraction, addition with carry bit and subtraction with carry bit are available. For the index registers, there are addition and subtraction instructions, that do set the carry bit, but no instructions that take into account the current state of the carry bit. STM8 arithmetic instructions mostly have one register operand and one memory operand. `ld a, (d, sp)` loads the value in memory at offset d from the stack pointer into register a. `ld (d, sp), a` stores the value in register a into the memory at offset d from the stack pointer. `add a, #c` adds the constant c to the value in register `a` and stores the result in `a`. `adc a, #c` adds the constant c plus the carry bit to the value in register `a` and stores the result in `a`. When considering this architecture, one notices, that a good way to handle program code that does not use pointers, and contains two 32-bit variables, to which other values are added or subtracted, is to put the lower half of each variable into an index register and spill the upper half: The traditional approach would be to spill both variables, since there are no 32-bit registers. Assuming register `x` is not used for anything else, an addition of a constant to a variable will take 16 bytes of code (Figure 11.1a). If `x` is not free, it will take 20 bytes (Figure 11.1b). Thus, assuming we have two subsequent additions, one for each variable, we get a total of at least 32 bytes of code. Assume we can allocate a variable into registers `x` and `y`, and fully spill the other, we get 13 bytes for the variable that was not spilt (Figure 11.1c) and 20 for the other (Figure 11.1b), for a total of 33. But if we spill the upper half of each variable, we get 13 bytes for the variable that was partially allocated into `x` (Figure 11.1d), and 14 bytes for the other (`addw` on `y` as operand is one byte longer than `addw` on `x`) for a total of 27 bytes. Thus we have an artificial example, for which bytewise register allocation and spilling reduces code size by 18.2%.

With bytewise register allocation being possible with current registers allocators, and there being artificial examples where it can provide an advantage, we investigate if bytewise register allocation provides an advantage substantial enough in practice to make it worth implementing.

11.3 Implementation

We implemented a backend for the STMicroelectronics STM8 architecture in SDCC [41], a C compiler for embedded systems, since SDCC already has an implementation of an optimal, polynomial-time register allocator [84], that was easy to extend for our scenario. SDCC is also the only free modern compiler targeting highly irregular 8-bit architectures. For the STM8 backend, we consider the 8-bit registers `a`, `xl`, `xh`, `yl` and `yh`. Unlike the other backends in SDCC, we allow arbitrary combinations of the 8-bit registers, and allow putting some of the bytes of a variable into registers while others are spilt. It was relatively easy to modify the register allocator for bytewise register allocation. But extra effort to be able to handle variables that have only some of their bytes spilt was required in implementing code generation. Even though the register allocator has a polynomial theoretical runtime bound, the implementation will sometimes use a heuristic to sacrifice optimality in exchange for faster compilation. Thus, even though bytewise allocation and spilling increases the theoretical runtime bound, the practical compilation time doesn't increase much; however the allocator might make more frequent use of the heuristic, resulting in sub-optimal code (one might wonder if this results in code quality worsening so much that the benefits from bytewise register allocation and spilling are dwarfed, but the results section shows that it is not a practical issue). Our new backend is included in the SDCC 3.4.0 release.

Describing the full details of the register allocator is beyond the scope of this chapter (see the previous Chapter 10 for that), so we here give a short overview with particular focus on the the interface and how we implemented bytewise allocation and spilling. In SDCC, register allocation is done on three-address-code, followed by code generation. This is a sensible approach for the highly irregular architectures with few registers targeted by SDCC (for RISC architectures with many registers, on the other hand, doing register allocation after code generation is now a standard approach). The register allocator first computes a *tree-decomposition* [61, 114, 113] of the control-flow graph, which is a data structure capturing some structural properties of the graph. The register allocator considers the local effects on small groups of instructions that particular assignments of variables to registers have. These partial solutions are then assembled into a globally optimal solution. The tree-decomposition allows doing this in polynomial time.

The input to the register allocator is three-address-code, which has already been scheduled. The register allocator uses a cost function c. For an instruction π in the three-address code and a register assignment f of the variables alive at π, we get a cost $c(\pi, f) \in [0, \infty]$. The choice of c depends on the optimization goal. When optimizing for code size, $c(\pi, f)$ would be the number of bytes of code code generation would generate for the instruction π if the variables alive at π were allocated according to f. When optimizing for code speed in a simple architecture, one could use the number of cycles the generated code would take to execute multiplied by a relative execution frequency obtained from a profiler. If code generation is unable to generate code for an instruction π when local variables are allocated according to f, we get $c(\pi, f) = \infty$. To allow bytewise allocation and spilling, each variable is broken into individual bytes for register allocation. A conventional backend could then use $c(\pi, f) = \infty$ for combinations considered too far-fetched, including any combination that partially puts part

of a variable into registers while spilling another part. For a backend that supports bytewise allocation and spilling, we get $c(\pi, f) \neq \infty$ even for these. In particular, register allocation will consider all possible ways to distribute the bytes of variables over registers and memory, and then choose the best one, depending on the cost function.

For the STM8, we implemented code generation that can generate code for three-address-code instructions π, even when the operands of π are distributed across multiple registers and partially in memory. Our implementation of a cost function c for optimization for code size is integrated into code generation: To evaluate the cost function we do a "dry run" of code generation that does not generate code, but instead gives the number of bytes of code that would be generated for an instruction π in three-address code under an assignment f of variables to registers.

11.4 Experiments

We created three variants of the STM8 backend in SDCC: A strongly restricted version ("Conventional") that allocates each variable either into 8-bit register `a`, 16-bit register `x`, 16-bit register `y`, or spills it. A less restricted version ("Bytewise allocation") that allocates each byte of each variable individually into any of the 8-bit registers `a`, `xl`, `xh`, `yl`, `yh`, but for each variable all bytes are either allocated into registers or spilt. Last, we used the full flexibility of our approach ("Bytewise allocation and spilling"), which can allocate any byte of any variable into any of the 8-bit register `a`, `xl`, `xh`, `yl`, `yh` or spill it.

We also extended some of the existing SDCC backends to allow much more flexibility in register allocation than conventional compilers. Since we had to extend existing backends we did not implement full bytewise register allocation, and did not implement bytewise spilling at all. Some of our modification already made it into the 3.3.0 release, others are part of the 3.4.0 release. For these backends we compared a restricted version ("Conventional") to the version allowing more flexibility ("Partially implemented bytewise allocation"). We did this for the Freescale HC08 and S08, the Zilog Z80 and Z180, the Toshiba TLCS-90, the Rabbit 2000 and the Sharp LR35902 backends. The Freescale HC08 and S08 architectures have an 8-bit register `a` and a 16-bit register `x`; register `x` is more restricted than the 16-bit registers in the STM8 in terms of available instructions. The other architectures on the other hand, have a higher number of registers, and are much richer in the instructions available on them. In particular, unlike the STM8, HC08 and S08 architectures they have more operations that have multiple register operands.

It is hard to find suitable benchmarks for STM8-based microcontrollers (many of which only have 2KB of program memory and 1KB of RAM), since most common benchmarks have far too high requirements on the hardware. This restricted our choice, even though we used the relatively powerful STM8S208MB microcontroller. We compiled three classic benchmarks for embedded systems: A C version of Whetstone [33]. Dhrystone [137], version 2 [138]; An ANSI-C version was used, since SDCC does not fully support K&R C. And Coremark [49], version 1.0. Whetstone is a floating-point benchmark. Dhrystone mostly uses 8- and 16-bit integer arithmetic, while Coremark also uses some 32-bit integer arithmetic. Both the benchmarks and the standard library were compiled

with the SDCC variants using options for strong optimization for code size ("--opt-code-size --max-allocs-per-node 10000000"). The benchmarks were run on an STM8S208MB, which features an STM8 core running at 24 MHz, 128 KB of flash program memory and 6 KB of RAM. It does not have caches.

11.5 Results

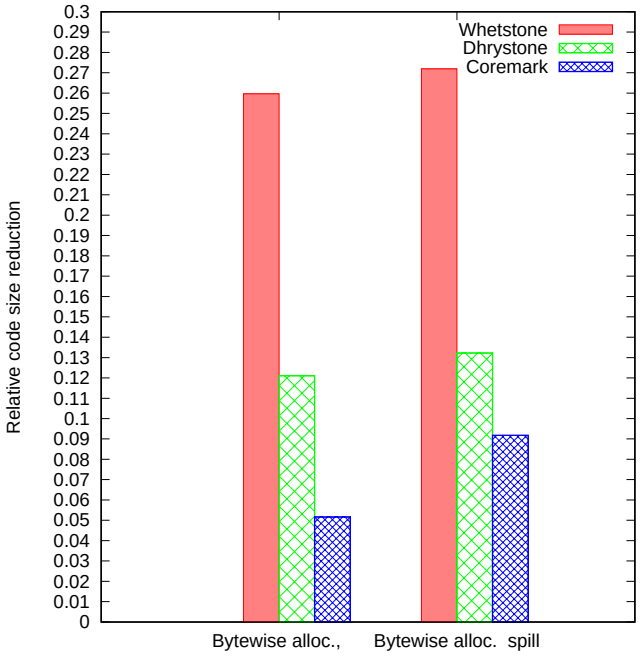


Figure 11.2: Benefits of bytewise allocation and spilling for the STM8 architecture

The impact of bytewise register allocation on code size is shown in Figure 11.2: In Whetstone, bytewise allocation results in a code size reduction of 26.0% compared to conventional allocation. With bytewise spilling this improves to 27.2%. For Dhrystone, bytewise allocation gives us a code size reduction of 12.1% compared to conventional allocation. The further benefits from bytewise spilling are small, resulting in bytewise spilling giving a code size reduction of 13.2% compared to conventional allocation. For Coremark, on the other hand, bytewise allocation without bytewise spilling gives us a code size reduction of 5.2% over conventional allocation, while bytewise allocation and spilling gives us a code size reduction of 9.2%.

We were surprised by the huge impact bytewise allocation had on the Whetstone benchmark. Further investigation showed that a large part of the code size reduction happened in the floating-point routines of the standard library. Since the STM8 does not have hardware support for floating-point numbers, those routines use bitwise instruction on 32-bit numbers. On the other hand, the results for Dhrystone and Coremark met our expectations: Unsurprisingly,

we see that the benefits of bytewise spilling are bigger when the code contains variables that would have to be spilt completely in conventional allocation.

The STM8S208MB does not have an instruction cache, so reduced code size does not immediately translate into a performance improvement. Since we were optimizing for code size, and bytewise allocation and spilling give additional freedom to the register allocator, one might wonder if the more aggressive optimization for code size comes at a performance cost. We found that with bytewise allocation the Whetstone performance improved by 29.6% compared to conventional allocation. With bytewise allocation and spilling it improved by 31.4%. Dhrystone performance increased from 4536 Dhrystones/second for conventional allocation to 4650 Dhrystones/second, a 2.5% performance improvement with bytewise allocation. With bytewise allocation and spilling it improved to 4658 Dhrystones/second, by 2.7%. Coremark performance increased by 12.0% with bytewise allocation, and by 14.1% with bytewise allocation and spilling.

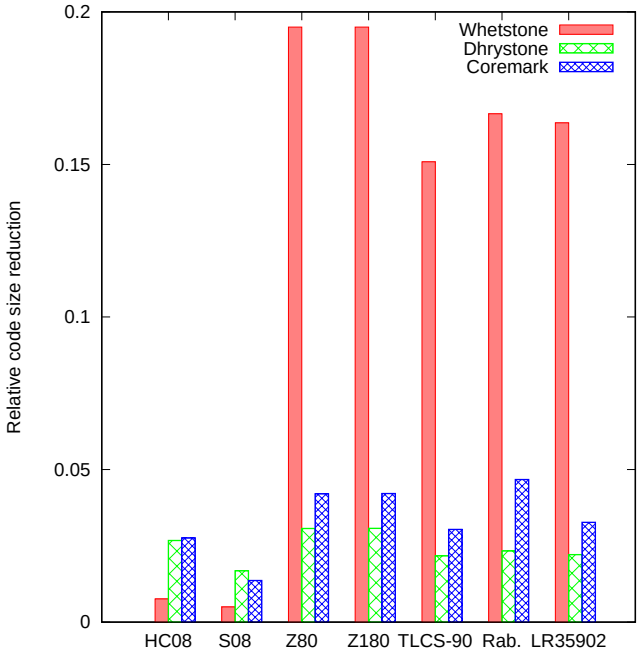


Figure 11.3: Benefits of partially implemented bytewise allocation over conventional allocation

The impact of partially implemented bytewise register allocation on code size is for various architectures shown in Figure 11.3. We see that the architectures basically fall into two groups: In the first group (HC08 and S08), the benefits for the Whetstone benchmark are relatively small and the benefits in the Dhrystone benchmark are similar to the benefits in Coremark. In the second group (other architectures) we see a huge improvement in Whetstone and the benefits in the Coremark benchmark are bigger than in Dhrystone. This is easily explained by the architecture: HC08 and S08 cannot allocate 32-bit variables in into registers, so the impact on Whetstone and Coremark is rela-

tively small (we suppose that they would benefit substantially from bytewise spilling though). The other architectures, however, have enough registers to allocate 32-bit variables into registers and gain a lot from the flexibility of partial bytewise allocation (we suppose that they would benefit further from full bytewise allocation, but not that much from bytewise spilling). The reduction in code size is between 0.5% and 27.6% for all these architecture / benchmark combinations. This is smaller than what we see for full bytewise allocation and spilling in the STM8 architecture, but still a good improvement.

11.6 Conclusion

We conclude that bytewise register allocation and spilling can result in substantial improvements of the generated code, for highly irregular architectures with a small number of registers. Bytewise spilling is particularly important for architectures with a tiny number of registers. We consider the advantages provided by bytewise allocation and spilling substantial enough to make support for bytewise allocation and spilling an important aspect when choosing a register allocator for a compiler for such targets.

Future research should investigate further how bytewise register allocation is affected by the architecture. In particular, the benefits of full bytewise register allocation and bytewise register allocation and spilling should be evaluated for more architectures, in particular those for which we already looked into the effect of partial bytewise allocation. Looking into the effects of combining bytewise register allocation with bitwidth-aware register allocation seems promising as well.

Chapter 12

The Complexity of Register Allocation[1]

In compilers, register allocation is one of the most important stages with respect to optimization for typical goals, such as code size, code speed, or energy efficiency. Graph theoretically, optimal register allocation is the problem of finding a maximum weight r-colorable induced subgraph in the conflict graph of a given program. The parameter r is the number of registers.

Large classes of programs are structured, i. e. their control-flow graphs have bounded tree-width [132, 58, 23] (see also Chapter 7). The decision problem of deciding if a conflict graph of a structured program is r-colorable is known to be fixed-parameter tractable [18]. Optimal register allocation for structured programs is known to be in XP (Chapter 10).

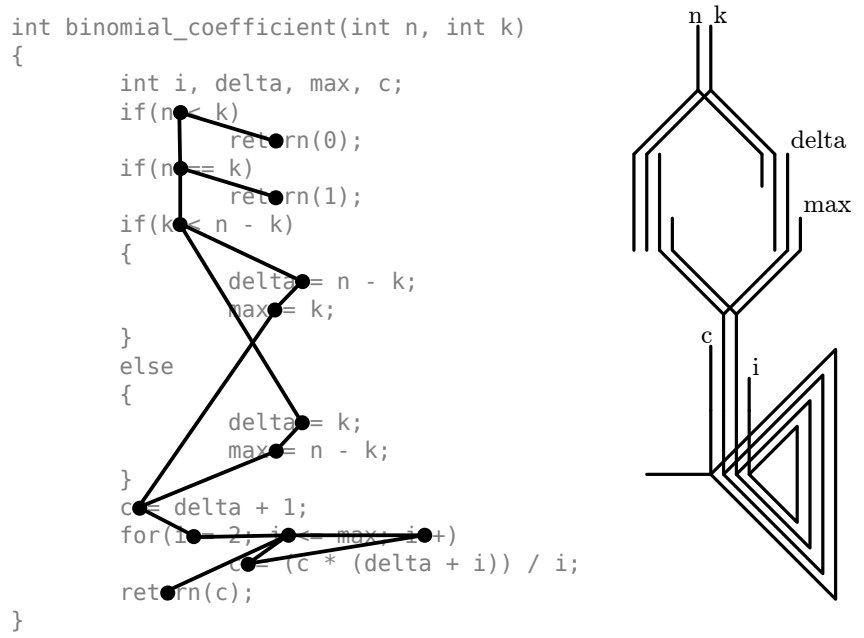
We complement these results by showing that optimal register allocation parametrized by r is W[SAT]-hard. This even holds for programs using only if/else and while as control structures; these programs form are subclass of the structured programs.

12.1 Introduction

Register allocation is a compiler stage that tries to assign variables in a computer program to hardware registers in a processor. Variables that are alive at the same time (conflicting variables) cannot be assigned to the same register, since this would result in values that are still needed being overwritten. Variables that are not assigned to registers are stored in main memory instead, which typically is slower by several orders of magnitude, and takes more or longer instructions to access. Register allocation is one of the most important stages in a compiler with respect to optimization for typical goals, such as code size, code speed or energy efficiency.

Register allocation can be seen as coloring the conflict graph of the variables of the program, with colors being the available registers. For r registers, finding an r-colorable induced subgraph of maximum weight in the conflict graph is a simplification of the the register allocation problem.

[1] Previously published in Discrete Applied Mathematics [85].



(a) Code and CFG (b) Variable live-ranges

Figure 12.1: Some program

We prove that finding an r-colorable induced subgraph of maximum weight in the conflict graph is $W[\text{SAT}]$-hard, even for a subclass of structured programs. This is a negative result complementing the earlier positive results.

The following Section 12.2 introduces the basic concepts necessary for the discussion of the related work in Section 12.3 and our results in Section 12.4. Section 12.5 concludes and states some questions that are still open.

12.2 Preliminaries

Definition 12.1 (Program)**.** *A program consists of a directed graph G, called the control-flow graph (CFG) of the program, a set of variables V and a weight function $c\colon V \to]0, \infty[$. Each node of G is marked by a subset of V. The set of nodes of G that are marked by a variable $v \in V$ are the live-range of v; v is said to be alive there. A live-range induces a connected subgraph of G.*

This representation can be easily generated from other representations, such as pseudo-code.The nodes of the CFG are the program's instructions; there is an edge from i to j, if there is some execution of the program where instruction j is executed directly after instruction i. Typically there is a cost (code size, runtime, energy consumption) associated with not placing variables in registers. The cost depends on how often the variable is accessed in the program. This is represented in the weight function.

Figure 12.1 shows code and CFG of a program and corresponding live-ranges.

Definition 12.2 (Conflict Graph)**.** *Let V be the set of variables of a program.*

The conflict graph of the program is the intersection graph of their live-ranges.

We use G for the control-flow graph and V for the variables throughout.

In $\mathfrak{k}$-structured programs the conflict graph is the intersection graph of connected subgraphs of a graph of tree-width at most $\mathfrak{k}$.

Definition 12.3 (ORA)**.** *Given an input program and parameter r, the number of registers, the optimization problem of register allocation (ORA) is to find an r-colorable induced subgraph S in the conflict graph, such that the sum $\sum_{v \in V \setminus V(S)} c(v)$ of the costs of the variables outside this subgraph is minimized.*

The subgraph S is induced by the variables, that will be placed in registers by optimal allocation.

This Definition 12.3 is a simplification of the problem encountered in real-world register allocation. It does not capture aspects such as different register classes, register preferences, register aliasing, coalescing, rematerialization or rescheduling. However, since we present a hardness result it is sufficient.

The decision problem corresponding to ORA is the following:

Definition 12.4 (DRA)**.** *Given an input program and parameter r, the number of registers and a number g, the decision problem of register allocation (DRA) is to decide if there is an r-colorable induced subgraph S in the conflict graph, such that the sum $\sum_{v \in V \setminus V(S)} c(v)$ of the costs of the variables outside this subgraph is at most g.*

Note that computationally, the optimization problem ORA is at least as hard as the decision problem DRA.

We use standard terminology from parametrized complexity theory [46, 105]:

Definition 12.5 (Parametrized Problem)**.** *A parametrized problem over a finite alphabet Σ is a set of pairs (x, k) with $x \in \Sigma^*$ and $k \in \mathbb{N}$.*

DRA above is an example of a parametrized problem under this definition.

Definition 12.6 (Parametrized Reduction)**.** *Let $L_1, L_1 \subseteq \Sigma^* \times \mathbb{N}$ be two parametrized problems. We say that L_1 reduces to L_2 by a parametrized reduction, if there are functions $k \mapsto k'$ and $k \mapsto k''$ from $\mathbb{N}$ to $\mathbb{N}$ and a function $(x, k) \mapsto x'$ from $\Sigma^* \times \mathbb{N}$ to Σ^* such that*

- *$(x, k) \mapsto x'$ is computable in time $k''|(x, k)|^c$ for some constant c,*
- *$(x, k) \in L_1 \Leftrightarrow (x', k') \in L_2$.*

Definition 12.7 (Circuit)**.** *A circuit is a directed, acyclic graph, with nodes representing logical gates (or, and, not). Not-gates have in-degree one, or- and and-gates have in-degree two or more. There is exactly one node of out-degree zero, called the output, and there are nodes of in-degree zero, called inputs.*

The circuit computes a Boolean function in the natural way.

The weft of a circuit is the maximum number of gates of in-degree more than two on a path from input to output. The depth of a circuit is the maximum number of gates on a path from input to output.

A circuit is called a SAT-circuit, if the undirected subgraph induced by all nodes that are not input nodes is a tree.

Except for the definition of $W[t]$ below, weft and depth are irrelevant in this work, so we can assume that all or- and and-gates have in-degree two by replacing the gates of in-degree more than two by trees of gates of in-degree two.

Definition 12.8 (WCS)**.** *Given a circuit C (input) and an integer k (parameter) the weighted circuit satisfiability problem asks if there is a satisfying assignment for C of weight exactly k (i. e. exactly k of the inputs are set to "true" in the assignment).*

Definition 12.9 (Parametrized Complexity Classes)**.**

- fpt *is the class of all parametrized problems parametrized by k that can be solved in time $f(k)p(n)$ with input size n, with a computable function f and a polynomial p.*
- *Let t be a positive integer. $W[t]$ is the class of all parametrized problems that can be reduced to the WCS problem on weft t, depth d SAT-circuits by a parametrized reduction, where $d \geq 1$ is a constant.*
- $W[\mathrm{SAT}]$ *is the class of all parametrized problems that can be reduced to the WCS problem on SAT-circuits by a parametrized reduction.*
- $W[P]$ *is the class of all parametrized problems that can be reduced to the WCS problem by a parametrized reduction.*
- XP *is the class of all problems parametrized by k that can be solved in time $f(k, n)$, with input size n and f polynomial in n for fixed k.*

$$\mathrm{fpt} \subseteq W[1] \subseteq W[2] \subseteq \ldots \subseteq W[\mathrm{SAT}] \subseteq W[P] \subseteq XP.$$

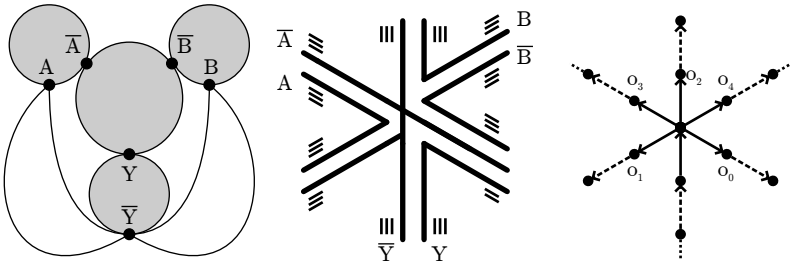
12.3 Previous results

Any graph can occur as the conflict graph of some program [27]. Together with the NP-hardness of graph-coloring [72] this proves the NP-hardness of register allocation when the input is not restricted to $\mathfrak{k}$-structured programs for some $\mathfrak{k}$. Garey et alii [50] have proven that when the number r of registers is part of the input (i.,e. not a parameter) the register allocation problem DRA is NP-hard even for structured programs. Kannan and Proebsting [70] were able to approximate a simplified version of the register allocation problem within a factor of 2 for programs that have series-parallel control-flow graphs (a subclass of 2-structured programs).

Large classes of programs are $\mathfrak{k}$-structured for some $\mathfrak{k}$. See Chapter 7 for details.

Based on earlier work by Thorup [132], Bodlaender et alii [18] obtained the following result:

Theorem 12.10. *Let $\mathfrak{k}$ be fixed. For $\mathfrak{k}$-structured programs, deciding if the register allocation problem has a solution such that $\sum_{v \in V \setminus V(S)} c(v) = 0$, (i. e. all variables can be placed in registers) is in* fpt *when parametrized by the number r of registers. I. e. DRA is in* fpt *for $g = 0$.*



(a) Cliques (b) Variable live-ranges (c) CFG

Figure 12.2: OR

We obtained the following result for the optimization problem (Chapter 10):

Theorem 12.11. *Let $\mathfrak{k}$ be fixed. For $\mathfrak{k}$-structured programs, the register allocation problem ORA can be solved in time $O(|G||V|^{2(\mathrm{tw}(G)+1)r})$, and thus is in XP.*

This result even holds when taking into account further aspects of register allocation such as different register classes, register preferences, register aliasing, coalescing and rematerialization.

12.4 W[SAT]-hardness of register allocation

We prove the hardness by constructing a program from the circuit in a rather natural way. The following Lemmata 12.12 and 12.13 and their proofs allow us to construct program parts from the individual gates in the circuit. Lemma 12.14 and its proof allow us to combine them into a program for the whole circuit.

Lemma 12.12. *For every $r \geq 3$, there is a structured program, such that there are variables $A, \overline{A}, B, \overline{B}, Y, \overline{Y}$, and when doing optimal register allocation for the program, exactly one of $X, \overline{X}$ is placed in registers for $X = A, B, Y$. Furthermore, Y is placed in a register exactly if A placed in a register or B is placed in a register.*

Proof. To abbreviate, and guide intuition we will say that a variable is "true", if it is placed in a register. Note that this notion of "true" is completely unrelated to any value the variable might hold in the program at run-time.

We construct a conflict graph with cliques as in Figure 12.2a. Each clique (represented by a grey shape in the figure) is filled by additional, unnamed nodes, so that it has size exactly $r + 1$. Thus e. g. the middle clique containing A, B and $\overline{Y}$ has $r - 2$ other nodes. These other nodes, each of which belongs to exactly one such added clique, are given a huge weight. Thus in each optimal assignment of variables to registers all unnamed variables will be placed in registers. Since each clique has size $r + 1$ this means that at least one of the named nodes in each clique will not be placed in a register. For the clique containing A and $\overline{A}$ it means that at most one of them goes into a register. The same hold for the clique containing B and $\overline{B}$ and the clique containing Y and

$\overline{Y}$. The clique containing A and $\overline{Y}$ and the one containing B and $\overline{Y}$ work the same way. The middle clique ensures that the value "true" can be assigned to at most two out of $\overline{A}, \overline{B}$ and Y. Since positive weights are assigned to all named variables, an optimal assignment will place as many of the variables in registers, as possible. It is possible to place half of them in registers (since $r \geq 3$), thus, for an optimal assignment, the following hold:

- At most one of A and $\overline{A}$ is "true".
- At most one of B and $\overline{B}$ is "true".
- At most one of Y and $\overline{Y}$ is "true".
- At most one of A and $\overline{Y}$ is "true".
- At most one of B and $\overline{Y}$ is "true".
- At most two of $\overline{A}, \overline{B}$ and Y are "true".
- At least three of $A, \overline{A}, B, \overline{B}, Y, \overline{Y}$ are "true".

As one can easily verify, this corresponds to Y being "true" exactly if A is "true" or B is "true".

Now that we have a conflict graph that gives us the desired properties, we need to show that this conflict graph can occur in a structured program. Figure 12.2b shows live-ranges, that result in the cliques just discussed in the conflict graph; the narrow, short lines represent the unnamed variables. As can be seen in Figure 12.2c, we get a star-shaped control-flow graph. The dashed lines represent paths where the additional, unnamed variables of the cliques are alive.

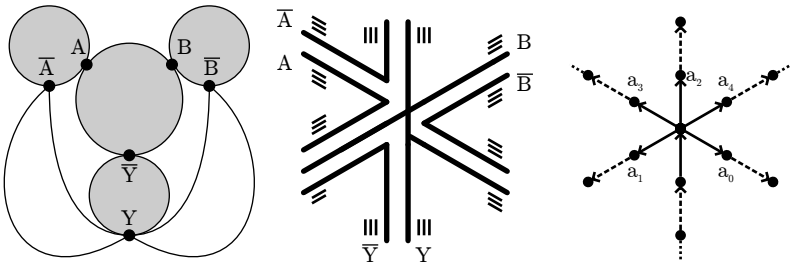
C-like pseudo-code for this or-gate could look like Figure 12.3. The cases of the switch statement correspond to the indices in Figure 12.2c. The macro CLIQUE1 expands to code that has $r - 1$ additional variables that, together with the other variables alive, form a clique of size $r+1$ in the conflict graph, and accesses them as often as necessary to give them the desired weight. CLIQUE2 does the same for $r - 2$ additional variables to form a clique of size $r + 1$. Since the only control construct used is a switch statement (which could be easily replaced by if/else), the program is structured. The assignments to the variables $A, \overline{A}, B, \overline{B}$ in the cases 0 to 2 of the switch statement only serve to overwrite these variables to ensure that they are not alive after the CLIQUE1/CLIQUE2 macro. Cases 3 and 4 are where other code is inserted later in the proof of Lemma 12.14. q. e. d.

Lemma 12.13. *For every $r \geq 3$, there is a structured program, such that there are variables $A, \overline{A}, B, \overline{B}, Y, \overline{Y}$, and when doing optimal register allocation for the program, exactly one of $X, \overline{X}$ is placed in registers for $X = A, B, Y$. Furthermore, Y is placed in a register exactly if A and B are placed in a register.*

Proof. The proof is similar to the one of Lemma 12.12; see also Figure 12.4. q. e. d.

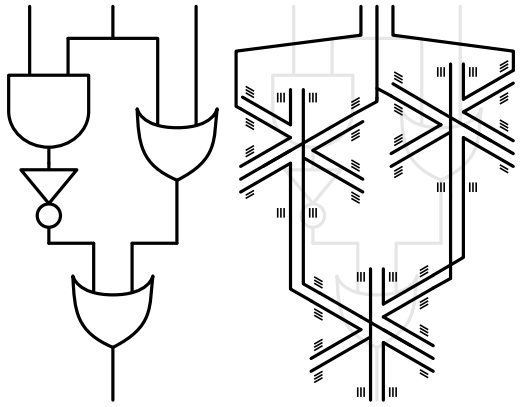
```
signal_y:
CLIQUE1;
A = f(), Ā = f(), B = f(), B̄ = f();
switch(f() % 5)
{
case 0:
        CLIQUE2;
        g(Y, Ā, B̄);
        A = f(), Ā = f(), B = f(), B̄ = f();
        break;
case 1:
        CLIQUE1;
        g(Ȳ, A);
        A = f(), Ā = f(), B = f(), B̄ = f();
        break;
case 2:
        CLIQUE1;
        g(Ȳ, B);
        A = f(), Ā = f(), B = f(), B̄ = f();
        break;
case 3:
        CLIQUE1;
        break; // Place goto signal_a; or nest here.
case 4:
        CLIQUE1;
        break; // Place goto signal_b; or nest here.
}
```

Figure 12.3: Pseudocode for OR from Figure 12.2



(a) Cliques (b) Variable live-ranges (c) CFG

Figure 12.4: AND



(a) Circuit (b) Variable live-ranges

Figure 12.5: Example transformation

Lemma 12.14. *For every circuit, we can construct a program, such that the solution to the register allocation problem* DRA *corresponding to the program gives a solution to the WCS for the circuit. The tree-width of the control-flow graph of the constructed program is at most one bigger than the tree-width of the underlying graph of the circuit without input nodes. For SAT-circuits, a series-parallel control-flow graph can be constructed. The number of registers in the register allocation problem is the maximum of* 3 *and the parameter* k *from the WCS instance of size* n*. The construction can be done in time* $O(nk)$*.*

Proof. We assume $k \geq 3$ and set $r = k$.

We represent each gate output from the circuit by a variable (and thus by a node in the conflict graph). We do the same for the complement of the output. Placing a variable in a register represents assigning the value "true" to the output. By adding cliques to the conflict graph and choosing suitable weights on the nodes of the conflict graph we can ensure that exactly one of these two variables will be placed in a register. The gates in the circuit are represented in a similar way. Lemmata 12.12 and 12.13 show that this can be done, and their proofs give the details.

We then combine the individual gates as in the circuit and add one node to the control-flow graph, at which all input variables of the circuit are alive. See Figure 12.5 for an example. The control-flow graph now is basically the graph from the circuit with input nodes replaced by a single node n, a few nodes of degree one added and some paths subdivided. The weight of the one variable representing the output from the circuit is increased slightly. The weight of the inputs is slightly increased as well, but much less than that of the output.

There are basically two ways to do the combining: A general one that works for any circuit and gives us a $\mathfrak{k}$-structured program for a circuit of tree-width $\mathfrak{k}-1$, and a specialized one that works for SAT-circuits and only uses switch/break (or, alternatively if/else) as control structure. Both essentially go bottom-up through the circuit while constructing the program.

In the general one, for each program fragment generated for a gate, we replace the `break` statements in the cases 0 to 3 by `return` statements. We replace the `break` statements in case 3 by `goto` statements: If the input A

of the gate is connected to the output of another gate, the `goto` targets the corresponding label. If the input of the gate is an input of the circuit, the `goto` targets the new node n. We handle case 4 in a similar way depending on input B. The control-flow graph is basically the graph from the circuit, subdivided, and with the addition of one node n, and paths that connect n to various parts of the graph. Thus the tree-width of the control-flow graph is at most one greater than the tree-width $\mathfrak{k} - 1$ of the circuit, and thus the program is $\mathfrak{k}$-structured. Figure 12.6 shows code generated for the circuit in Figure 12.5 this way (excluding some code we would add to increase the weights of the output and inputs).

In the specialized way of combining, we generate the program fragment for the gate that has the circuit's output as its output. If input A of our gate is the output of another gate, we replace the `break` statement in case 3 by the program fragment generated for that gate. We do this recursively, and also for case 4 with input B. This results in a program that uses only `switch` statements (`if/else` could be used instead) as control structure. This makes the program structured even under early, most restrictive notions of "structured program". Figure 12.7 shows code generated for the circuit in Figure 12.5 this way (excluding some code we would add to increase the weights of the output and inputs).

The construction can be done in linear time for fixed k: For each node in the input graph a fixed amount of nodes in the CFG (or a fixed amount of pseudocode) plus whatever is needed for the fixed amount of cliques, is generated. The size of the cliques is linear in the parameter k. We thus get a total runtime and output size of $O(kn) \subseteq O(n^2)$.

The one node in the control-flow graph where all the live-ranges of inputs meet ensures that at most $r = k$ of the inputs are assigned the value "true". Thus an optimal assignment will find a valid configuration of the circuit, with exactly k of the inputs set to "true". If it is possible to set the output to "true" under these conditions it will happen, since it has slightly higher weight than the other variables.

The sum of the weights of all the variables that would be placed in registers by such an assignment with the output set to "true" can easily be calculated. We then use this sum as g for the DRA. q. e. d.

Theorem 12.15. *The register allocation problem DRA, when parametrized by the number of registers r, is W[SAT]-hard, even for structured programs.*

Proof. Lemma 12.14 gives a reduction from WCS to DRA. The functions in the reduction are $k \mapsto k' = \max\{3, k\}$ and $k \mapsto k'' = k$, and $(x, k) \mapsto x'$ with constructed program x'. For SAT-circuits, our construction only uses one very basic control structure (switch/break or, alternatively, if/else) which is available in virtually all programming languages. Thus our hardness result is not restricted to a particular programming language and holds even for the early, most restrictive notion of "structured program". q. e. d.

12.5 Conclusion

We have proven that register allocation for structured programs is W[SAT]-hard. This complements a previous result that register allocation for structured

```
void h(void)
{
        Y = f();
        Ȳ = f();
        CLIQUE1;
        A = f(), Ā = f(), B = f(), B̄ = f();
        switch(f() % 5)
        {
        case 0:
                CLIQUE2;
                g(Y, Ā, B̄);
                return;
        case 1:
                CLIQUE1;
                g(Ȳ, A);
                return;
        case 2:
                CLIQUE1;
                g(Ȳ, B);
                A = f(), Ā = f(), B = f(), B̄ = f();
                return;
        case 3:
                CLIQUE1;
                goto signal_a;
                break;
        case 4:
                CLIQUE1;
                goto signal_b:
        }

signal_a:
        CLIQUE1;
        C = f(), C̄ = f(), D = f(), D̄ = f();
        switch(f() % 5)
        {
        case 0:
                CLIQUE2;
                g(A, C, D);
                return;
        case 1:
                CLIQUE1;
                g(Ā, C̄);
                return;
        case 2:
                CLIQUE1;
                g(Ā, D̄);
                return;
        case 3:
                CLIQUE1;
                D = f(); E = f();
                goto n;
        case 4:
                CLIQUE1;
                C = f(); E = f();
                goto n;
        }

signal_b:
        CLIQUE1;
        D = f(), D̄ = f(), E = f(), Ē = f();
        switch(f() % 5)
        {
        case 0:
                CLIQUE2;
                g(B, D̄, Ē);
                return;
        case 1:
                CLIQUE1;
                g(B̄, D);
                return;
        case 2:
                CLIQUE1;
                g(B̄, E);
                return;
        case 3:
                CLIQUE1;
                C = f(); E = f();
                goto n;
        case 4:
                CLIQUE1;
                C = f(); D = f();
                goto n;
        }
n:
        g(C, D, E);
}
```

Figure 12.6: 2-structured pseudocode for the example from Figure 12.5

```
void h(void)
{
        Y = f();
        Ȳ = f();
        CLIQUE1;
        A = f(), Ā = f(), B = f(), B̄ = f();
        switch(f() % 5)
        {
        case 0:
                CLIQUE2;
                g(Y, Ā, B̄);
                break;
        case 1:
                CLIQUE1;
                g(Ȳ, A);
                break;
        case 2:
                CLIQUE1;
                g(Ȳ, B);
                A = f(), Ā = f(), B = f(), B̄ = f();
                break;
        case 3:
                CLIQUE1;
                C = f(), C̄ = f(), D = f(), D̄ = f();
                switch(f() % 5)
                {
                case 0:
                        CLIQUE2;
                        g(A, C, D);
                        C = f(), D = f();
                        break;
                case 1:
                        CLIQUE1;
                        g(Ā, C̄);
                        C = f(), D = f();
                        break;
                case 2:
                        CLIQUE1;
                        g(Ā, D̄);
                        C = f(), D = f();
                        break;
                case 3:
                        CLIQUE1;
                        D = f(); E = f();
                        break;
                case 4:
                        CLIQUE1;
                        C = f(); E = f();
                }
                break;
        case 4:
                CLIQUE1;
                D = f(), D̄ = f(), E = f(), Ē = f();
                switch(f() % 5)
                {
                case 0:
                        CLIQUE2;
                        g(B, D̄, Ē);
                        D = f(), E = f();
                        break;
                case 1:
                        CLIQUE1;
                        g(B̄, D);
                        D = f(), E = f();
                        break;
                case 2:
                        CLIQUE1;
                        g(B̄, E);
                        D = f(), E = f();
                        break;
                case 3:
                        CLIQUE1;
                        C = f(); E = f();
                        break;
                case 4:
                        CLIQUE1;
                        C = f(); D = f();
                }
        }
        g(C, D, E);
}
```

Figure 12.7: Pseudocode for the example from Figure 12.5 using only switch/break as control construct

programs is in XP, even when taking into account further aspects, such as different register classes, register preferences, register aliasing, coalescing and rematerialization, which were not considered here [84]. Since $W[\mathrm{SAT}] \subseteq W[P] \subseteq XP$, open questions remain about which class exactly register allocation for structured programs falls into. The answer could depend on the further aspects of register allocation mentioned above.

Conclusion

Our contributions include new results for the Disjoint-Paths Problem and in compiler optimizations. A central concept in our work are tree-decompositions, which we applied to both fields.

We obtained a singly-exponential upper bound on the tree-width of planar graphs with vital linkages with k components, resulting in the fastest known fpt algorithm for the planar Disjoint-Paths Problem. We also found planar graphs of tree-width 2^k that have vital linkages with k components.

We presented our theoretical and empirical results on the tree-widths of the control-flow graphs of programs written in the programming language C, and how to obtain the corresponding decompositions. We used the tree-decompositions of the control-flow graphs to obtain the fastest known alorithms for the placement of bank selection instructions and for the redundancy elimination technique lospre. We found the first optimal polynomial time algorithm for register allocation, and investigated the effect of applying our approach at the level of individual bytes. We obtained a W[SAT]-hardness result that makes substantial further improvements over our register allocation algorithm unlikely. We implemented our approaches in SDCC, a C compiler for embedded systems and empirically achieved substantial improvements in the generated code.

Bibliography

[1] Guidelines for the Use of the C Language in Critical Systems (MISRA-C:2004, 2nd Edition). Technical report, MISRA, 2008.

[2] Isolde Adler, Stavros G. Kolliopoulos, Philipp K. Krause, Daniel Lokshtanov, Saket Saurabh, and Dimitrios M. Thilikos. Tight Bounds for Linkages in Planar Graphs. In *Proceedings of the 38th International Colloquium on Automata, Languages and Programming (ICALP 2011) (1)*, pages 110–121, 2011.

[3] Isolde Adler, Stavros G. Kolliopoulos, Philipp Klaus Krause, Daniel Lokshtanov, Saket Saurabh, and Dimitrios M. Thilikos. Irrelevant vertices for the planar disjoint paths problem. *CoRR*, abs/1310.2378, 2013.

[4] Isolde Adler, Stavros G. Kolliopoulos, and Dimitrios M. Thilikos. Planar Disjoint-Paths Completion. In Dániel Marx and Peter Rossmanith, editors, *Parameterized and Exact Computation*, volume 7112 of *Lecture Notes in Computer Science*, pages 80–93. Springer, 2012.

[5] Isolde Adler and Philipp K. Krause. A lower bound on the tree width of graphs with irrelevant vertices. Unpublished manuscript.

[6] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Dorling Kindersley, 2009.

[7] Stephen Alstrup, Peter W. Lauridsen, and Mikkel Thorup. Generalized Dominators for Structured Programs. *Algorithmica*, 27:244–253, 2000.

[8] Andrew W. Appel and Lal George. Optimal Spilling for CISC Machines with Few Registers. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 243–253, New York, NY, USA, 2001. Association for Computing Machinery.

[9] Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy Problems for Tree-Decomposable Graphs. *Journal of Algorithms*, 12(2):308–340, 1991.

[10] Stefan Arnborg and Andrzej Proskurowski. Characterization and recognition of partial 3-trees. *SIAM Journal on Algebraic Discrete Methods*, 7(2):305–314, April 1986.

[11] Rajkishore Barik, Christian Grothoff, Rahul Gupta, Vinayaka Pandit, and Raghavendra Udupa. Optimal Bitwise Register Allocation Using Integer

Linear Programming. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, LCPC'06, pages 267–282, Berlin, Heidelberg, 2007. Springer.

[12] Rajkishore Barik and Vivek Sarkar. Enhanced Bitwidth-Aware Register Allocation. In Alan Mycroft and Andreas Zeller, editors, *Compiler Construction*, volume 3923 of *Lecture Notes in Computer Science*, pages 263–276. Springer, 2006.

[13] Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. A Decoupled non-SSA Global Register Allocation Using Bipartite Liveness Graphs. *ACM Transactions on Architecture Code Optimization (TACO)*, 10(4):63:1–63:24, December 2013.

[14] Richard E. Bellman. On the theory of dynamic programming. In *Proceedings of the National Academy of Sciences*, volume 38, pages 716–719, 1952.

[15] Jean R.S. Blair, Pinar Heggernes, and Jan A. Telle. A practical algorithm for making filled graphs minimal. *Theoretical Computer Science*, 250(1–2):125 – 141, 2001.

[16] Rastislav Bodík, Rajiv Gupta, and Mary L. Soffa. Complete Removal of Redundant Expressions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 1–14. Association for Computing Machinery, 1998.

[17] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal of Computation*, 25(6):1305–1317, 1996.

[18] Hans L. Bodlaender, Jens Gustedt, and Jan A. Telle. Linear-Time Register Allocation for a Fixed Number of Registers. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '98, pages 574–583. Society for Industrial and Applied Mathematics, 1998.

[19] Hans L. Bodlaender and Arie M.C.A. Koster. Treewidth computations III. Exact algorithms and preprocessing. Unpublished manuscript.

[20] Hans L. Bodlaender and Arie M.C.A. Koster. Treewidth computations I. Upper bounds. *Information and Computation*, 208(3):259 – 275, 2010.

[21] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the Complexity of Register Coalescing. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 102–114. IEEE Computer Society, 2007.

[22] Sebastian Buchwald, Andreas Zwinkau, and Thomas Bersch. SSA-based Register Allocation with PBQP. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 42–61, Berlin, Heidelberg, 2011. Springer.

[23] Bernd Burgstaller, Johann Blieberger, and Bernhard Scholz. On the Tree Width of Ada Programs. In *Reliable Software Technologies - Ada-Europe 2004*, volume 3063 of *Lecture Notes in Computer Science*, pages 78–90. Springer, 2004.

[24] Corrado Böhm and Giuseppe Jacopini. Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules. *Communications of the ACM*, 9(5):366–371, May 1966.

[25] Qiong Cai and Jingling Xue. Optimal and Efficient Speculation-Based Partial Redundancy Elimination. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 91–102. IEEE Computer Society, 2003.

[26] Gregory J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–105. Association for Computing Machinery, 1982.

[27] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[28] ChaN. FatFS. `http://elm-chan.org/fsw/ff/00index_e.html`.

[29] John Cocke. Global Common Subexpression Elimination. In *Proceedings of a symposium on Compiler optimization*, pages 20–24. Association for Computing Machinery, 1970.

[30] Quentin Colombet, Benoit Boissinot, Philip Brisk, Sebastian Hack, and Fabrice Rastello. Graph-Coloring and Treescan Register Allocation Using Repairing. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '11, pages 45–54, New York, NY, USA, 2011. Association for Computing Machinery.

[31] Bruno Courcelle. The Monadic Second-Order Logic of Graphs I. Recognizable Sets of Finite Graphs. *Information and Computation*, 85(1):12–75, 1990.

[32] Bruno Courcelle and Stephan Olariu. Upper bounds to the clique width of graphs. *Discrete Applied Mathematics*, 101(1–3):77–114, 2000.

[33] H. J. Curnow and Brian A. Wichmann. A Synthetic Benchmark. *Computer Journal*, 19:43–49, 1976.

[34] Marek Cygan, Dániel Marx, Marcin Pilipczuk, and Michal Pilipczuk. The planar directed k-vertex-disjoint paths problem is fixed-parameter tractable. In *FOCS*, pages 197–206. IEEE Computer Society, 2013.

[35] Anuj Dawar, Martin Grohe, and Stephan Kreutzer. Locally excluding a minor. In *LICS'07*, pages 270–279. IEEE Computer Society, 2007.

[36] Anuj Dawar and Stephan Kreutzer. Domination problems in nowhere-dense classes. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009)*, pages 157–168, 2009.

[37] Erik D. Demaine and MohammadTaghi Hajiaghayi. The Bidimensionality Theory and Its Algorithmic Applications. *Computer Journal*, 51(3):292–302, 2008.

[38] Nick D. Dendris, Lefteris M. Kirousis, and Dimitrios M. Thilikos. Fugitive-search games on graphs and related parameters. *Theoretical Computer Science*, 172(1-2):233 – 254, 1997.

[39] Reinhard Diestel. *Graph Theory*. Springer, 2005.

[40] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, November 2004.

[41] Sandeep Dutta. Anatomy of a Compiler. *Circuit Cellar*, 121:30–35, 2000.

[42] Leonhard Eulero. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, 8:128–140, 1741.

[43] Alkis Evlogimenos. Improvements to Linear Scan register allocation, 2004. Technical report, University of Illinois, Urbana-Champaign.

[44] Stijn Eyerman, James E. Smith, and Lieven Eeckhout. Characterizing the branch misprediction penalty. In *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, pages 48–58, March 2006.

[45] Martin Farach and Vincenzo Liberatore. On local register allocation. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, SODA '98, pages 564–573. Society for Industrial and Applied Mathematics, 1998.

[46] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Springer, 1998.

[47] Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. *Annals of Pure and Applied Logic*, 130(1–3):3–31, 2004.

[48] Changqing Fu and Kent Wilken. A Faster Optimal Register Allocator. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 245–256. IEEE Computer Society Press, 2002.

[49] Shay Gal-On. Coremark, 2009. `http://www.coremark.org`.

[50] M. R. Garey, D. S. Johnson, Gary L. Miller, and C. H. Papadimitriou. The Complexity of Coloring Circular Arcs and Chords. *SIAM Journal on Algebraic Discrete Methods*, 1(2):216–227, June 1980.

[51] Jim Geelen, Tony Huynh, and Ronny B. Richter. Explicit bounds for graph minors. *CoRR*, abs/1305.1451, 2013.

[52] Petr A. Golovach, M. Kaminski, D. Paulusma, and Dimitrios M. Thilikos. Induced packing of odd cycles in a planar graph. In *Proceedings of the 20th International Symposium on Algorithms and Computation (ISAAC 2009)*, volume 5878 of *Lecture Notes in Computer Science*, pages 514–523. Springer, Berlin, 2009.

[53] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0–1 integer programming. *Software — Practice & Experience*, 26(8):929–965, 1996.

[54] Qian-Ping Gu and Hisao Tamaki. Improved bounds on the planar branchwidth with respect to the largest grid minor size. In *ISAAC (2)*, volume 6507 of *Lecture Notes in Computer Science*, pages 85–96, 2010.

[55] Rajiv Gupta, David A. Berson, and Jesse Z. Fang. Path Profile Guided Partial Redundancy Elimination Using Speculation. In *Computer Languages, 1998. Proceedings. 1998 International Conference on*, pages 230–239, May 1998.

[56] Frank Gurski and Egon Wanke. Vertex disjoint paths on clique-width bounded graphs. *Theor. Comput. Sci.*, 359(1):188–199, August 2006.

[57] Venkatesan Guruswami and Ali Sinop. Improved Inapproximability Results for Maximum k-Colorable Subgraph. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, volume 5687 of *Lecture Notes in Computer Science*, pages 163–176. Springer, 2009.

[58] Jens Gustedt, Ole Mæhle, and Jan A. Telle. The Treewidth of Java Programs. In *Algorithm Engineering and Experiments*, volume 2409 of *Lecture Notes in Computer Science*, pages 57–59. Springer, 2002.

[59] Ralf H. Güting and Martin Erwig. *Übersetzerbau: Techniken, Werkzeuge, Anwendungen*. Springer, 1998.

[60] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register Allocation for Programs in SSA-Form. In *Compiler Construction 2006*, volume 3923 of *Lecture Notes In Computer Science*, pages 247–262. Springer, March 2006.

[61] Rudolf Halin. Zur Klassifikation der endlichen Graphen nach H. Hadwiger und K. Wagner. *Mathematische Annalen*, 172(1):46–78, 1967.

[62] Lang Hames and Bernhard Scholz. Nearly Optimal Register Allocation with PBQP. In *Proceedings of the 7th joint conference on Modular Programming Languages*, JMLC'06, pages 346–361. Springer, 2006.

[63] Carl Hierholzer. Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6:30–32, 1873.

[64] R. Nigel Horspool, David J. Pereira, and Bernhard Scholz. Fast Profile-Based Partial Redundancy Elimination. In *Proceedings of the 7th joint conference on Modular Programming Languages*, JMLC'06, pages 362–376. Springer, 2006.

[65] Ken ichi Kawarabayashi, Yusuke Kobayashi, and Bruce A. Reed. The disjoint paths problem in quadratic time. *Journal of Combinatorial Theory, Series B*, 102(2):424–435, 2012.

[66] ISO JTC1/SC22/WG14. ISO/IEC 9899:TC3, Programming languages – C, 2007. N1256.

[67] ISO JTC1/SC22/WG14. ISO/IEC 9899:2011, Programming languages – C, 2011. N1570.

[68] Ben Jaiyen and Jamie Liu. Implementing Profile-Guided Speculative Code Motion in LLVM. Technical report, 2012.

[69] F. V. Jensen, Steffen L. Lauritzen, and Kristian G. Olesen. Bayesian updating in causal probabilistic networks by local computations. *Computational Statistics Quaterly*, 4:269–282, 1990.

[70] Sampath Kannan and Todd Proebsting. Register Allocation in Structured Programs. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, SODA '95, pages 360–368. Society for Industrial and Applied Mathematics, 1995.

[71] David R. Karger and Clifford Stein. An $\tilde{O}(n^2)$ Algorithm for Minimum Cuts. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 757–765. Association for Computing Machinery, 1993.

[72] Richard M. Karp. On the Computational Complexity of Combinatorial Problems. *Networks*, 5:45–68, 1975.

[73] Ken-ichi Kawarabayashi and Yusuke Kobayashi. The induced disjoint path problem. In *Proceedings of the 13th Conference on Integer Programming and Combinatorial Optimization (IPCO 2008)*, volume 5035 of *Lecture Notes in Computer Science*, pages 47–61. Springer, Berlin, 2008.

[74] Ken-ichi Kawarabayashi and Bruce Reed. Odd cycle packing. In *Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC 2010)*, pages 695–704, New York, NY, USA, 2010. Association for Computing Machinery.

[75] Ken-ichi Kawarabayashi and Paul Wollan. A shorter proof of the graph minor algorithm: the unique linkage theorem. In *Proc. of the 42nd annual ACM Symposium on Theory of Computing (STOC 2010)*, pages 687–694, 2010.

[76] Tokuzo Kiyohara, Scott Mahlke, William Chen, Roger Bringmann, Richard Hank, Sadun Anik, and Wen-Mei Hwu. Register Connection: A New Approach to Adding Registers into Instruction Set Architectures. *SIGARCH Computer Architexture News*, 21:247–256, 1993.

[77] Ton Kloks. *Treewidth: Computations and Approximations*. Lecture Notes in Computer Science. Springer, 1994.

[78] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy Code Motion. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 224–234. Association for Computing Machinery, 1992.

[79] Donald E. Knuth. Structured Programming with go to Statements. *ACM Computing Surveys*, 6(4):261–301, December 1974.

[80] Yusuke Kobayashi and Ken-ichi Kawarabayashi. Algorithms for finding an induced cycle in planar graphs and bounded genus graphs. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2009)*, pages 1146–1155. ACM-SIAM, 2009.

[81] M. R. Kramer and J. van Leeuven. The complexity of wire-routing and finding minimum area layouts for arbitrary VLSI circuits. *Advances in Computing Research*, 2:129–146, 1984.

[82] Philipp K. Krause. lospre in linear time. Unpublished manuscript.

[83] Philipp K. Krause. Optimal Placement of Bank Selection Instructions in Polynomial Time. In *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*, M-SCOPES '13, pages 23–30. Association for Computing Machinery, 2013.

[84] Philipp K. Krause. Optimal Register Allocation in Polynomial Time. In *Compiler Construction - 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013. Proceedings*, volume 7791 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2013.

[85] Philipp K. Krause. The Complexity of Register Allocation. *Discrete Applied Mathematics*, 168(0):51 – 59, 2014.

[86] Philipp K. Krause. Bytewise Register Allocation. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, pages 22–27. Association for Computing Machinery, 2015.

[87] Philipp K. Krause and Lukas Larisch. The tree-width of C. Unpublished manuscript.

[88] Dénes König. *Theorie der endlichen und unendlichen Graphen*. Chelsea, 1935.

[89] Steffen L. Lauritzen and David J. Spiegelhalter. Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems. *Journal of the Royal Statistical Society, Series B*, 50(2):157–224, 1988.

[90] Jonathan K. Lee, Jens Palsberg, and Fernando M. Q. Pereira. Aliased register allocation for straight-line programs is NP-complete. *Theoretical Computer Science*, 407(1-3):258–273, 2008.

[91] Minming Li, Chun Jason Xue, Tiantian Liu, and Yingchao Zhao. Analysis and Approximation for Bank Selection Instruction Minimization on Partitioned Memory Architecture. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '10, pages 1–8. Association for Computing Machinery, 2010.

[92] Johann B. Listing. *Vorstudien zur Topologie.* Vandenheock und Ruprecht, 1847.

[93] Tiantian Liu, Minming Li, and Chun Jason Xue. Joint Variable Partitioning and Bank Selection Instruction Optimization on Embedded Systems with Multiple Memory Banks. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ASPDAC '10, pages 113–118. IEEE Press, 2010.

[94] Tiantian Liu, Alex Orailoglu, Chun Jason Xue, and Minming Li. Register Allocation for Embedded Systems to Simultaneously Reduce Energy and Temperature on Registers. *ACM Trans. Embed. Comput. Syst.*, 13(3):50:1–50:26, December 2013.

[95] Daniel Lokshtanov, Daniel Marx, and Saket Saurabh. Slightly superexponential parameterized problems. In *22st ACM–SIAM Symposium on Discrete Algorithms (SODA 2011)*, pages 760–776, 2011.

[96] James F. Lynch. The equivalence of theorem proving and the interconnection problem. *SIGDA Newsletter*, 5(3):31–36, 1975.

[97] Frédéric Mazoit. A single exponential bound for the redundant vertex theorem on surfaces. *CoRR*, abs/1309.7820, 2013.

[98] Karl Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927.

[99] Yuan Mengting, Wu Guoqing, and Yu Chao. Optimizing Bank Selection Instructions by Using Shared Memory. In *Embedded Software and Systems, 2008. ICESS '08. International Conference on*, pages 447–450. IEEE Press, 2008.

[100] Matthias Middendorf and Frank Pfeiffer. On the complexity of the disjoint paths problem. *Combinatorica*, 13(1):97–107, 1993.

[101] Bojan Mohar and Carsten Thomassen. *Graphs on Surfaces.* Johns Hopkins series in the mathematical sciences. Johns Hopkins University Press, 2001.

[102] Etienne Morel and Claude Renvoise. Global Optimization by Suppression of Partial Redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[103] V. Krishna Nandivada and Rajkishore Barik. Improved Bitwidth-aware Variable Packing. *ACM Transactions on Architecture and Code Optimization*, 10(3):16:1–16:22, September 2008.

[104] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100. Association for Computing Machinery, 2007.

[105] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms.* Oxford Lecture Series in Mathematics And Its Applications. Oxford University Press, 2006.

[106] Takao Nishizeki, Jens Vygen, and Xiao Zhou. The edge-disjoint path problem is NP-complete for series-parallel graphs. *Discrete Applied Mathematics*, 115(1-3):177–186, November 2001.

[107] David John Pereira. *Isothermality: making speculative optimizations affordable.* PhD thesis, 2008.

[108] Fernando M. Q. Pereira and Jens Palsberg. Register Allocation Via Coloring of Chordal Graphs. In *In Proceedings of APLAS'05, Asian Symposium on Programming Languages and Systems*, pages 315–329, 2005.

[109] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.

[110] Philippe Rinaudo, Yann Ponty, Dominique Barth, and Alain Denise. Tree Decomposition and Parameterized Algorithms for RNA Structure-Sequence Alignment Including Tertiary Interactions and Pseudoknots. In Ben Raphael and Jijun Tang, editors, *Algorithms in Bioinformatics*, volume 7534 of *Lecture Notes in Computer Science*, pages 149–164. Springer, 2012.

[111] Edward M. Riseman and Caxton C. Foster. The Inhibition of Potential Parallelism by Conditional Jumps. *IEEE Transactions on Computers*, C-21(12):1405–1411, Dec 1972.

[112] Neil Robertson and Paul D. Seymour. Graph Minors. I. Excluding a Forest. *Journal of Combinatorial Theory, Series B*, 35(1):39–61, 1983.

[113] Neil Robertson and Paul D. Seymour. Graph Minors. III. Planar Tree-Width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[114] Neil Robertson and Paul D. Seymour. Graph Minors. II. Algorithmic Aspects of Tree-Width. *Journal of Algorithms*, 7(3):309–322, 1986.

[115] Neil Robertson and Paul D Seymour. Graph Minors. V. Excluding a Planar Graph. *Journal of Combinatorial Theory, Series B*, 41(1):92–114, August 1986.

[116] Neil Robertson and Paul D. Seymour. Graph Minors. XIII. The Disjoint Paths Problem. *Journal of Combinatorial Theory, Series B*, 63(1):65–110, 1995.

[117] Neil Robertson and Paul D. Seymour. Graph Minors. XXI. Graphs with unique linkages. *Journal of Combinatorial Theory, Series B*, 99(3):583–616, 2009.

[118] Neil Robertson and Paul D. Seymour. Graph Minors. XXII. Irrelevant vertices in linkage problems. *Journal of Combinatorial Theory, Series B*, 102(2):530–563, 2012.

[119] Neil Robertson, Paul D. Seymour, and Robin Thomas. Quickly excluding a planar graph. *Journal of Combinatorial Theory, Series B*, 62:323–348, 1994.

[120] Hein Röhrig. *Tree Decomposition: A Feasibility Study.* Diplomarbeit, 1998.

[121] Petra Scheffler. A Practical Linear Time Algorithm for Disjoint Paths in Graphs with Bounded Tree-width, 1994. Technical report, Fachbereich 3 Mathematik, Technische Universität Berlin.

[122] Bernhard Scholz, Bernd Burgstaller, and Jingling Xue. Minimizing Bank Selection Instructions for Partitioned Memory Architectures. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 201–211. Association for Computing Machinery, 2006.

[123] Bernhard Scholz, Bernd Burgstaller, and Jingling Xue. Minimal Placement of Bank Selection Instructions for Partitioned Memory Architectures. *ACM Transactions on Embedded Compututing Systems*, 7:12:1–12:32, 2008.

[124] Bernhard Scholz and Erik Eckstein. Register Allocation for Irregular Architectures. *SIGPLAN Notices*, 37(7):139–148, June 2002.

[125] Bernhard Scholz, R. Nigel Horspool, and Jens Knoop. Optimizing for Space and Time Usage with Speculative Partial Redundancy Elimination. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '04, pages 221–230. Association for Computing Machinery, 2004.

[126] Alexander Schrijver. Finding k disjoint paths in a directed planar graph. *SIAM Journal on Computing*, 23(4):780–788, 1994.

[127] Alexander Schrijver. *Combinatorial Pptimization. Polyhedra and Efficiency. Volume A.* Springer, Berlin, 2003.

[128] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A Generalized Algorithm for Graph-Coloring Register Allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288. Association for Computing Machinery, 2004.

[129] Yinglei Song, Russell Malmberg, Liming Cai, Chunmei Liu, and Fangfang Pan. Tree Decomposition Based Fast Search of RNA Structures Including Pseudoknots in Genomes. In *In Proceedings of 2005 Computational System Bioinformatics Conference*, pages 223–234. IEEE Computer Society, 2005.

[130] Sriraman Tallam and Rajiv Gupta. Bitwidth Aware Global Register Allocation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 85–96. Association for Computing Machinery, 2003.

[131] André L. C. Tavares, Quentin Colombet, Mariza A. S. Bigonha, Christophe Guillon, Fernando M. Q. Pereira, and Fabrice Rastello. Decoupled graph-coloring register allocation with hierarchical aliasing. In *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '11, pages 1–10. Association for Computing Machinery, 2011.

[132] Mikkel Thorup. All Structured Programs Have Small Tree Width and Good Register Allocation. *Information and Computation*, 142(2):159–181, 1998.

[133] William T. Tutte. How to draw a graph. *Proceedings of the London Mathematical Society*, 13:743–767, 1963.

[134] Frank van den Eijkhof, Hans L. Bodlaender, and M. C. Arie Koster. Safe Reduction Rules for Weighted Treewidth. *Algorithmica*, 47(2):139–158, February 2007.

[135] Jens Vygen. NP-completeness of some edge-disjoint paths problems. *Discrete Applied Mathematics*, 61(1):83–90, 1995.

[136] Klaus Wagner. *Graphentheorie*. Bibliographisches Institut, 1970.

[137] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27:1013–1030, October 1984.

[138] Reinhold P. Weicker. Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules. *SIGPLAN Notices*, 23:49–62, August 1988.

[139] Jingling Xue and Qiong Cai. A Lifetime Optimal Algorithm for Speculative PRE. *ACM Transactions on Architecture and Code Optimization*, 3(2):115–155, June 2006.

[140] Atsuko Yamaguchi, Kiyoko F Aoki, and Hiroshi Mamitsuka. Graph Complexity of Chemical Compounds in Biological Pathways. *Genome Informatics*, pages 376–377, 2003.

[141] Mihalis Yannakakis and Fanica Gavril. The maximum k-colorable subgraph problem for chordal graphs. *Information Processing Letters*, 24(2):133–137, 1987.

[142] Jizhen Zhao, Dongsheng Che, and Liming Cai. Comparative Pathway Annotation with Protein-DNA Interaction and Operon Information via Graph Tree Decomposition. In *Pacific Symposium on Biocomputing'07*, pages 496–507, 2007.

[143] Jizhen Zhao, Russell L. Malmberg, and Liming Cai. Rapid ab initio RNA Folding Including Pseudoknots Via Graph Tree Decomposition. In Philipp Bücher and BernardM.E. Moret, editors, *Algorithms in Bioinformatics*, volume 4175 of *Lecture Notes in Computer Science*, pages 262–273. Springer, 2006.

[144] Hucheng Zhou, Wenguang Chen, and Fred Chow. An SSA-based Algorithm for Optimal Speculative Code Motion under an Execution Profile. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 98–108. Association for Computing Machinery, 2011.

Index

∅, 27

abstract syntax tree, 38
assembler code, 38

bag, 32
bank selection instruction, 23, 96
bounded tree-width, 34
branch set, 30

C, 78–79
calculation set, 112
Chaitin, 40, 122, 140
cheap solution, 51–53, 56, 60, 61
chromatic number, 31
circuit, 151
clique, 29
clique-width, 34, 45
coalescing, 40
code generator, 38
compatible coloring, 99
compiler, 21–22, 37–40, 77–160
complete graph, 29
complexity of register allocation, 40, 125–130, 149–160
concentric cycles, 50
conflict graph, 40, 125, 151
connected graph, 29
consecutive segments, 59–61
Contiki, 88, 119, 131, 134
contracting an edge, 30
control-flow graph, 39, 78, 79, 98, 111, 123, 125, 150
Coremark, 131, 134, 145
coremark, 88, 144
cost function, 98, 125
covered edge, 100
cw, 34
cycle, 29

degree, 28
Dhrystone, 131, 134, 144, 145
dhrystone, 88
directed graph, 27
disc-with-edges embedding, 64, 66, 68, 69
disjoint paths problem, 20–74
DPP, 45
DRA, 151
drawing, 32

edge, 27
edge around a terminal, 64
edge contraction, 30
embedded C, 104
empty set, 27
endpoints of a linkage, 47
endpoints of a path, 29
endpoints of an edge, 27
equivalent segments, 52

face, 32
finite graph, 27
forget node, 34
fpt, 20, 152
FUZIX, 88

Game Boy, 130
global common subexpression elimination, 23, 109
graph, 27
graph isomorphism, 28
graph minor theorem, 19
grid, 29
grid minor theorem, 20

hardness of register allocation, 40, 129, 130, 149–160
HC08, 130
horizontal edge, 29

I, 51
iCode, 38
in-degree, 28

independent set, 29
induced subgraph, 28
infinite face, 32
infix, 57
instance of the disjoint paths problem, 46
integers, 27
intermediate code, 38
intersection graph, 31
introduce node, 34
invalidation set, 112
irrelevant vertex, 19, 20, 45, 46, 61
iteration statement, 79, 84

join node, 34
jump statement, 79

lazy code motion, 24, 109
leaf node, 34
lexical analyzer, 38
life set, 111
linkage, 47
linker, 38
live-range, 40, 123, 125, 150, 151
live-range splitting, 128
loop, 27
lospre, 109–120
LR35902, 130

machine code, 38
maximum independent set, 129
memory bank, 96, 98, 99
minor, 30
model, 29
multigraph, 28, 53

$\mathbb{N}$, 27
named address space, 103, 104
natural numbers, 27
nice tree-decomposition, 34, 100, 125
node, 27

ORA, 151
out-degree, 28
outerplanar graph, 32

parametrized complexity theory, 20
parametrized problem, 151
parametrized reduction, 151
partial redundancy elimination, 24, 109
partition, 27
partitioned memory architecture, 23, 96
path, 28
path-decomposition, 32
path-width, 33
pattern of a linkage, 47
peephole optimizer, 38
pic, 97, 98
planar drawing, 32
planar embedding, 32
planar graph, 32
plane graph, 32
power set, 27
prefix, 57
preprocessor, 37
product of sets, 27
program, 98, 150
pw, 33

$\mathbb{R}$, 27
Rabbit, 97, 130
real numbers, 27
redundancy elimination, 38, 109–120
register, 39, 130, 134, 140, 142, 144
register aliasing, 40
register allocation, 39–40, 121–160
register allocator, 39–40, 121–147
register preferences, 40
register pressure, 120
restriction, 27

S08, 130
safety, 112, 118
SAT-circuit, 151
SDCC, 22, 77, 88, 103
Sega Master System, 97
segment, 50
segment dual graph, 53
segment graph, 52
segment type, 52, 60, 61
segment types, 56
selection statement, 78, 84
series-parallel graph, 32
short-circuit evaluation, 79
simple path, 29
size of a grid, 29
SMS, 97
spilling, 39
square grid, 29
STM8, 142

structured program, 78, 157
subdivided grid, 50
subdivided untidy grid, 59
subgraph, 28
suffix, 57
supergraph, 28
syntax analyzer, 38

Thorup, 80–81, 130
three-address-code, 38
tight concentric cycles, 50–53, 56, 59–61
tongue tip, 53
transparent node, 98
tree, 29
tree-decomposition, 19, 21, 32–36
tree-width, 19–21, 33
triangle, 29
tw, 19, 33
two-terminal graph, 32

undirected graph, 27
untidy grid, 59
use set, 111

valid assignment, 123
vertex, 27
vertex coloring, 31
vertical edge, 29
vital linkage, 47

W[1], 20
W[2], 20
W[P], 20, 152
W[SAT], 20, 152
Wagner's conjecture, 19
weft, 151
weighted circuit satisfiability, 152
Whetstone, 88, 144, 145
width of a tree-decomposition, 33

XP, 20, 152

$\mathbb{Z}$, 27
Z180, 96, 130
Z80, 97, 130, 131