Institut für Programmstrukturen
und Datenorganisation (IPD)

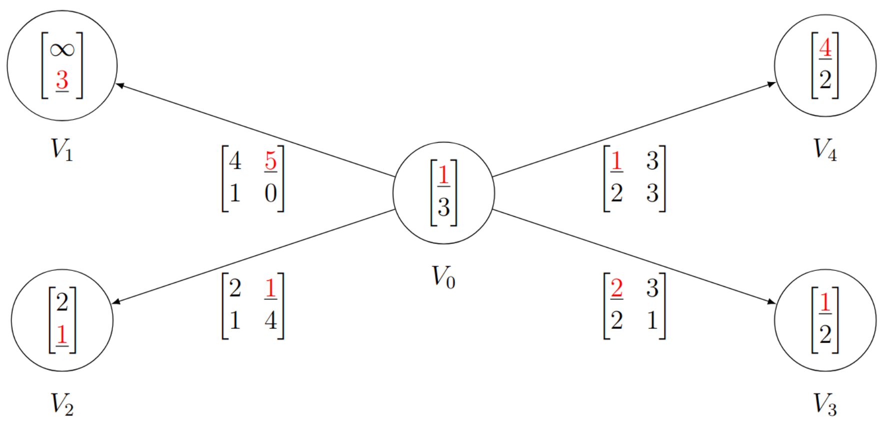
Lehrstuhl Prof. Dr.-Ing. Snelting

Development of a library for solving and analyzing PBQP

Bachelorarbeit von

Max Baumstark

an der Fakultät für Informatik



$\begin{bmatrix}\infty\\3\end{bmatrix}$
V_1
$\begin{bmatrix}4&5\\1&0\end{bmatrix}$
$\begin{bmatrix}1\\3\end{bmatrix}$
V_0
$\begin{bmatrix}1&3\\2&3\end{bmatrix}$
$\begin{bmatrix}4\\2\end{bmatrix}$
V_4
$\begin{bmatrix}2\\1\end{bmatrix}$
V_2
$\begin{bmatrix}2&1\\1&4\end{bmatrix}$
$\begin{bmatrix}2&3\\2&1\end{bmatrix}$
$\begin{bmatrix}1\\2\end{bmatrix}$
V_3

Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M.Sc. Sebastian Graf
Abgabedatum:	22. Februar 2019

Abstract

Partitioned Boolean Quadratic Problems (PBQP) are a promising approach for solving SSA-based register allocation problems. They unite assignment and coalescing into a single optimization problem, which allows finding a better solution for the register allocation problem. Despite their advantages PBQP see little use, partially due to a lack of a standalone library for handling and solving PBQP instances. All existing PBQP libraries are either tied into larger frame works, incomplete or have bugs. We create a standalone library to handle, solve and visualize PBQP. We also implement different solving algorithms and then compare and evaluate these different PBQP solving algorithms based on their performance and solution quality. As part of this we introduce quadratic programming as a mean to obtain an optimal solution for a PBQP instance.

Partitioned Boolean Quadratic Problems (PBQP) sind ein vielversprechender Ansatz zur Lösung von SSA-basierten Registerzuteilungsproblemen. Sie vereinen den Zuweisungsschritt und den Verschmelzungsschrits des Registerzuteilungsprozesses in ein einzelnes Optimierungsproblem, was uns ermöglicht bessere Lösungen für diesen Registerzuteilungsprozess zu finden. Trotz ihrer Vorteile werden PBQP nicht weitläufig benutzt, unter anderem, da keine alleinstehende Bibliothek zum Handhaben und Lösen von PBQP existiert. Alle existierenden PBQP-Bibliotheken sind entweder Teil größerer Frameworks, unvollständig oder haben Bugs. Wir entwerfen und implementieren eine Bibliothek zum Handhaben, Lösen und Visualisieren von PBQP-Instanzen und implementieren verschiedene PBQP Lösungsalgorithmen. Wir vergleichen diese verschiedenen Lösungsalgorithmen anhand ihrer Laufzeit und der Qualität der von ihnen produzierten Lösungen. Hierbei führen wir Quadratische Optimierung als neuen Ansatz zum Finden von optimalen Lösungen ein.

Contents

1 Introduction

Over the past decades we have seen a constant race towards faster and faster computation. Great advancements have been made in various computer related fields with one of them being compiler optimization. Compiler optimization attempts to optimize runtime performance of applications, thus creating faster applications. One of the ways to achieve this is by improving the compiled programs register allocation, the assignment of the many program variables to the limited amount of CPU registers. Finding an ideal assignment is NP-hard [1], but when compiling an application we have to find a solution in polynomial time, which makes this a very interesting research topic with lots of potential.

Part of solving register allocation problems are among others assigning registers to variables in a step called assignment and reducing the amount of unnecessary move operations between registers in a step called coalescing. Typically these two steps are done separately from each other. An initial solution for the assignment is retrieved through a coloring problem and then improved as part of the coalescing. A problem here is that the coloring problem used does not take the coalescing aspect into account at all, so the degree to which the solution can be improved by the coalescing step is limited. This can be improved by using an optimization problem which unites coalescing and assignment, the Partitioned Boolean Quadratic Problem (PBQP). The PBQP is a class of NP-complete optimization problems and allows us to find better solutions for register allocation problems in similar time[2].

PBQP also find usage in other fields, for example during selection of primitives in Deep Neural Networks [3], but we will focus on their usage in register allocation. In particular we will work based on SSA-based register allocation and its implementation in libFirm, which is a compiler being developed by KIT[4]. Despite their superior ability to solve register allocation problems PBQP are not used very widely in register allocation. Aside from a lack of awareness a major reason for this limited use is the lack of a standalone application or library for creating, handling and solving PBQP instances.

All existing complete PBQP solvers which we are aware of are part of larger frameworks, tied into those frameworks and can't be used on their own. Additionally there are only very few and very limited analysis or visualization tools for dealing with PBQP, thus making debugging of programs working with PBQP unnecessarily hard. In this thesis we present pbqp-papa a standalone C++ library for solving, analyzing and visualizing PBQP instances. The goal is to provide a configurable and extensible framework which provides future developers all the tools they need to work with PBQP.

This includes:

- Object oriented graph representation
- C-API for high-performance applications like compilers
- Saving PBQP instances to and loading them from JSON files
- Multiple solver implementations
- Builtin support for infinite entries
- PBQP instance visualization as graph in svg files

No such framework exists yet, because applications opting for usage of PBQP have no reason to create a standalone library as direct integration into their code base is usually easier and quicker for them.

Utilizing our framework we also explore different solving approaches and evaluate their viability based on their performance and the quality of the solutions they create. Research on these different PBQP solving approaches can be applied to obtain solutions of higher quality or to obtain solutions of the same quality in less time. These improvements translate into higher quality solutions in register allocation and thus directly to less move operations in compiled code and faster execution time for compiled applications.

Even though PBQP are a much wider class of problems, this thesis focuses around their application in SSA-based register allocation and its implementation in libFirm. The use cases the library is tailored around and the motivation for this paper to begin with are entirely based on libFirm and its usage of PBQP. All PBQP instances used for performance evaluation were created based on register allocation problems in libFirm. As a result findings regarding performance of different solvers may only apply to a limited degree to PBQP in general because all instances examined are of the specific structure register allocation problems have. PBQP instances created by other use cases may have wildly different node structure or density, thus influencing both the performance of different solving algorithms and the quality of their solutions.

2 Basics

2.1 Partioned Boolean Quadratic Problem

2.1.1 Formal form

The Partitioned Boolean Quadratic Problem (PBQP) is a NP-complete[2] optimization problem. It is formally defined as [5]:

$$\begin{aligned} \mathsf{min} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} \vec{x}_i^T C_{ij} \vec{x}_j \\ \text{s.t.} \quad & \vec{x}_i^T 1 = \vec{1} & \forall i = 1...n \\ & \vec{x}_i \in \{0,1\}^{d_i} & \forall i = 1...n \end{aligned}$$

$\vec{x}_i$ are binary vectors in which all entries are 0 except for one entry, which is 1. Finding a solution for the problem means determining which entry is 1 for each $\vec{x}_i$. d_i may vary for different i, but for all i $d_i \geq 1$ is required. C_{ij} are cost matrices of size $d_i \times d_j$, which define a cost for every combination of selections in $\vec{x}_i$ and $\vec{x}_j$. Cost matrices may contain values of any number type.

The goal is to find a selection which minimizes the total cost over all cost matrices.

Cost matrices may also contain ∞ as a value, which, instead of assigning a cost, completely forbids a selection. Usage of ∞ may make finding a solution entirely impossible, for example if all entries in a row or column of a cost matrix are ∞.

2.1.2 Graph form

While functional this formal definition of PBQP is rather complicated. It can not be visualized easily and would use a lot more space and time than needed if implemented this way. For example a vector of binary variables $\vec{x}_i$ in which one entry is 1 and the others are 0 represents a selection and we could instead just save the index of the selection as integer. Additionally there is no need for C_{ij} to exist or be considered if all of their entries are 0. Removing these unnecessary edges and going from binary selection arrays to just an integer representing the selection leads us to a representation as directed graph.

We can transform a PBQP instance from its formal form into a graph form like this: Each $\vec{x}_i$ is turned into a node $\vec{V}_i$. Every $\vec{V}_i$ has a cost vector $\vec{m}_i$, which is initialized as $\vec{0}$ when converting a PBQP instance from its formal form. $\vec{m}_i$ are no longer binary, but may contain any number or ∞.

Each cost matrix C_{ij} is assigned to an edge $\vec{E}_{ij}$ going from $\vec{V}_i$ to $\vec{V}_j$.

Finding a solution works the same way in this form, for each node one entry has to be selected. The difference is that the total cost to minimize does not only include the cost implicitly selected in the edge cost matrices C_{ij}, but also the cost entry directly selected in $\vec{m}_i$:

$$\begin{aligned} \textsf{min} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} \vec{x}_i^T C_{ij}\vec{x}_j + \sum_{i=1}^{n} \vec{x}_i^T \vec{m}_i & \\ \text{s.t.} \quad & \vec{x}_i^T 1 = \vec{1} & \forall i = 1...n \\ & \vec{x}_i \in \{0,1\}^{d_i} & \forall i = 1...n \end{aligned}$$

This form allows reducing the problem without affecting the quality of the created solution by removing certain edges. Formally removing an edge E_{ii} means setting all values of C_{ij} to 0, but in the graph representation and its implementation we can just remove E_{ij} entirely once its cost is moved into V_i or Vj. This makes finding a minimal solution easier.

An edge E_{ii} is a cycle on the node $\vec{V_i}$ and can be removed by adding the diagonal of their cost matrix C_{ii} to $\vec{m}_i$. Also given two edges E_{ij} and E_{ji}, E_{ji} can be removed by adding C_{ji}^T to C_{ij}.

From here on any PBQP instance will always assumed to be in graph form as this form was used in the implementation of the solver. It is much easier to understand and visualize.

2.2 Solving PBQP

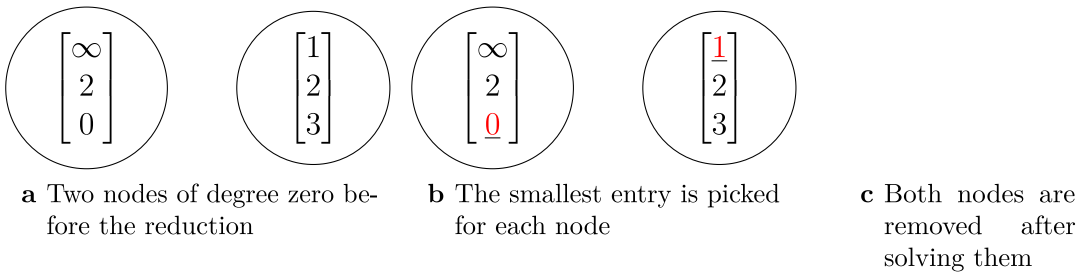
PBQP instances are solved by reducing the problem until the solution of the remaining problem is trivial. The following section explaining this process is based on Buchwald et al.[2]. Reductions remove nodes and edges to simplify the problem until the entire graph is empty. After that the reduction steps taken are iterated over in reverse order and for every step the node which was removed is assigned a selection. The selection assigned usually depends on the selection already assigned to adjacent nodes. All nodes which existed when the reduction was applied, except for the ones removed by the reduction, are guaranteed to already have a selection assigned by the time the removed nodes have to be solved in the back propagation phase. Reductions are either optimal or heuristic.

2.2.1 Optimal reductions

Optimal reductions remove single nodes and their incident edges. They guarantee that a minimal solution of the reduced PBQP instance also allows construcing a minimal solution of the original PBQP instance [2]. If a PBQP instance can be fully solved using only optimal reductions, the solution created will also be optimal. This is only the case for sparse graphs though, for example for PBQP instances whose graph is a tree an optimal solution can be found in linear time using only optimal reductions. There are three optimal reductions, which can be applied to nodes of degree 0, 1 and 2. The direction of the incident edges does not influence when these reductions can be applied, because the direction of an edge can be inverted by transposing its cost matrix. $C_{ij}^T = C_{ji}$

2.2.2 R0 reductions

A node of degree 0 can be removed immediately. Its solution is its smallest entry, which can be determined directly during back propagation. A simple example can be seen in Figure 2.1.



$\begin{bmatrix}\infty\\2\\0\end{bmatrix}$
$\begin{bmatrix}1\\2\\3\end{bmatrix}$
$\begin{bmatrix}\infty\\2\\\underline{0}\end{bmatrix}$
$\begin{bmatrix}\underline{1}\\2\\3\end{bmatrix}$
a Two nodes of degree zero before the reduction
b The smallest entry is picked for each node
c Both nodes are removed after solving them

Figure 2.1: Solving two nodes of degree zero using RO

2.2.3 R1 reductions

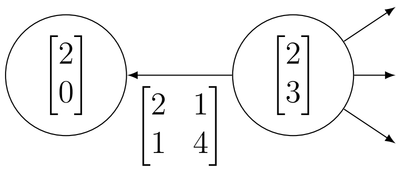
A node V_i of degree 1 which is incident to the edge E_{ij} and adjacent to the node V_j can be removed together with its edge by calculating the minimal cost created in C_{ij} and $\vec{m}_i$ for each possible selection in V_j. This cost is added to each V_j and during backpropagation a solution for V_i is selected based on the solution of V_j.

For all a in $[1...|d_j|]$, meaning for every possible selection in V_j, we find the minimal selection in V_i, the b for which

$$\begin{aligned} min \quad & \vec{x}_i^T C_{ij} \vec{x}_j + \vec{x}_i^T \vec{c}_i \\ & \vec{x}_{ja} = 1 \\ & \vec{x}_{ib} = 1 \end{aligned}$$

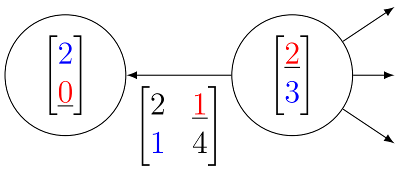
and add this minimum to $\vec{m}_j a$, the a-th entry of $\vec{m}_j$. The b calculated for each a is saved and used during backpropagation. If $\vec{x}_{ja} = 1$ in the solution, then $\vec{x}_{ib} = 1$. This means if the a-th entry was selected in V_j, then the b-th entry in V_i has to be selected.

Figure 2.2 shows an example application of R1 on the node V_0. If the first entry in V_1 were to be selected, the total cost would be minimized by selecting the second entry in V_0 for a total cost of 5. We remember to select the second entry in V_0 if the first entry in V_1 is selected during back propagation and set m_{11} to 5. Analogue we set m_{12} to 4 and remember the first entry in V_0 for it. After reduction only a modified V_1 is left.



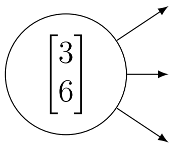
$\begin{bmatrix}2\\0\end{bmatrix}$
$\begin{bmatrix}2\\3\end{bmatrix}$
$\begin{bmatrix}2 & 1\\1 & 4\end{bmatrix}$

a A node of degree one and its adjacent node before the reduction



$\begin{bmatrix}2\\0\end{bmatrix}$
$\begin{bmatrix}2\\3\end{bmatrix}$
$\begin{bmatrix}2 & 1\\1 & 4\end{bmatrix}$

b For each selection in the adjacent node we find the selection which minimizes the total cost



c The cost of the cheapest selection is added to each entry of the adjacent node and the node itself is removed

Figure 2.2: Solving a node of degree one using R1

2.2.4 R2 reductions

A node V_i of degree 2 which is incident to edges C_{ij} and C_{ik} and adjacent to the nodes V_j and V_k can be removed by calculating the minimal cost for each possible selection combination in V_j and V_k and saving this cost in a newly created edge $E*_{jk}$. We create its cost matrix $C*_{jk}$ like this:

$$\begin{aligned} C*_{jkbc} = min \quad & \vec{x}_i^T C_{ij} \vec{x}_j + \vec{x}_i^T C_{ik} \vec{x}_k + \vec{x}_i^T \vec{c}_i \\ \vec{x}_{ia} &= 1 \\ \vec{x}_{jb} &= 1 \\ \vec{x}_{kc} &= 1 \end{aligned}$$

For all $b \in [1, [|\vec{x}_j|], c \in [1, [|\vec{x}_k|]$ we get a a for which the total cost is minimal. Analogue to R1 reductions, for every b, c combination the a selected is the selection to pick for V_i based on the selection in V_j and V_k during back propagation. The created cost matrix $C*_{jk}$ contains the entire cost of V_i and its incident edges. Adding the edge $E*_{jk}$ allows removing V_i and its edges. If an edge E_{jk} already exists, $E*_{jk}$ can be added to its cost matrix C_{jk} instead of creating a new edge. If an edge E_{kj} already exists, $C*_{jk}^T$ can be added to C_{kj} instead. An example application of R2 can be seen in Figure 2.3

2.2.5 Heuristic reductions

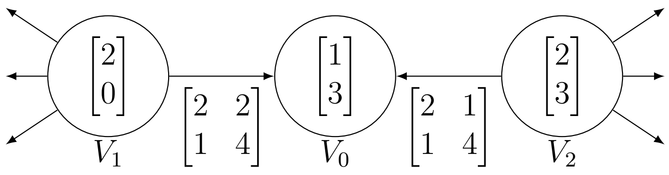
Heuristic reductions do not guarantee minimal solutions. They can even make solvable PBQP instances unsolvable. They allow us to find good solutions in linear time though, even with PBQP being NP-complete. They are applied whenever no more optimal reductions are possible. The only heuristic reduction used here is early-decision RN, it is applied to nodes of degree three or higher. Early-decision RN selects a solution for a node by calculating a local minimum based on adjacent nodes and incident edges and then removes the node. Ideally the node to which early-decision RN is applied has a high degree, so more nodes in total are taken into account and better decisions are made. For a given node V_i with adjacent nodes V_j ... V_k, for which all edges point away from V_i (if necessary edges can be turned by transposing their cost matrix), RN determines its selection by calculating:

$$\text{min} \sum_{j}^{k} m_{ia} + m_{jb} + C_{ijab} \qquad a \in \{1, d_i\}, b \in \{1, d_j\}$$

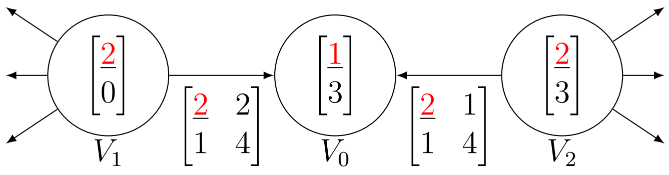
Opposed to early-decision RN, which determines a selection for the node during reduction, late-decision RN also exists. Late-decision RN just removes the node during reduction and decides on a selection during backpropagation. The solutions created by late-decision RN are worse than the ones created by early-decision RN though[2], so only early-decision RN is used here and any mentions of RN in the following refer to early-decision RN. An example application of RN can be seen in Figure 2.3

2.2.6 PEO

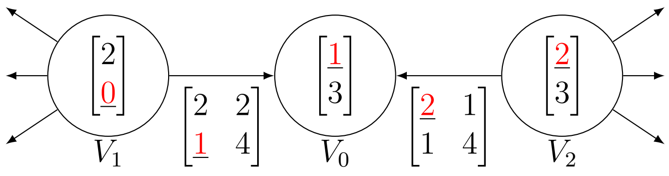
PBQP instances created by register allocation problems will often contain many ∞ entries. When given a solvable PBQP instance RN may make decisions which make finding a solution impossible. To ensure that a solution can always be found a perfect elimination order (PEO) is used. A PEO defines an order on all nodes of the graph and guarantees that RN will not make a solvable problem unsolvable as long as the order in which RN is applied is the order of the PEO. PEOs remain valid throughout applying optimal reductions. Nodes that were already removed by optimal reductions can simply be skipped in the PEO.[2]



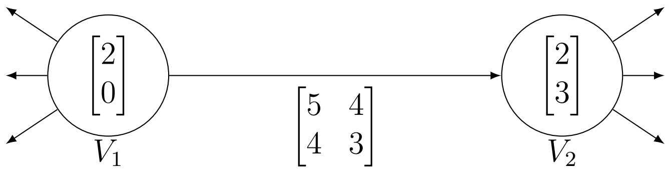
a A node of degree two and its adjacent nodes before the reduction



b For each selection combination in V_1 & V_2 we find the selection in V_0 which minimizes the total cost, two examples for this are shown here

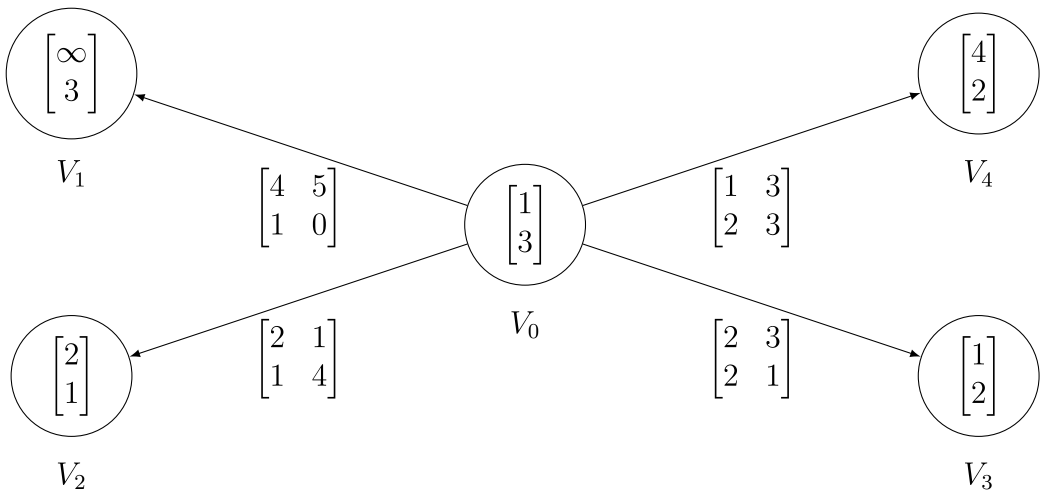


c Another selection combination in V_1 & V_2

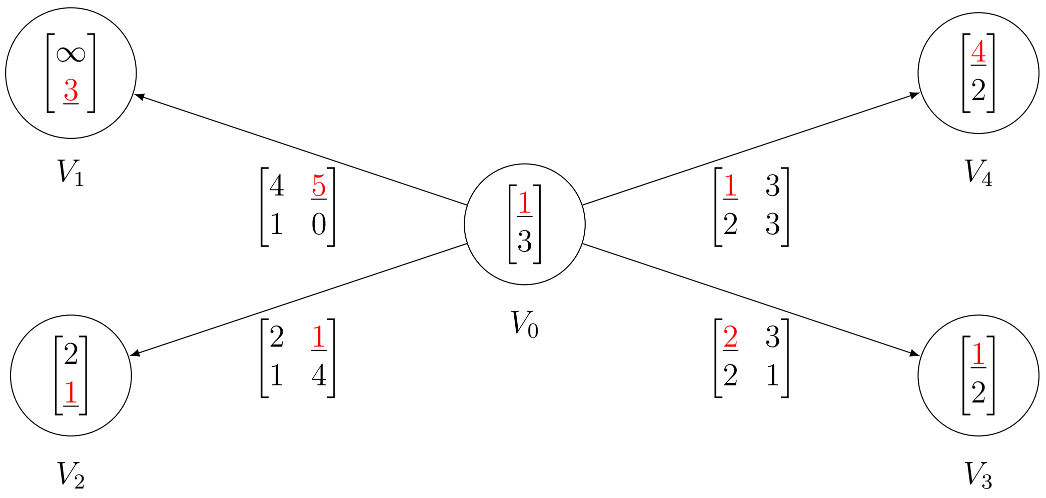


d The cheapest cost created by V_0 and its incident edges for each selection combination in V_1 & V_2 is put into a new edge and $V_0$is removed

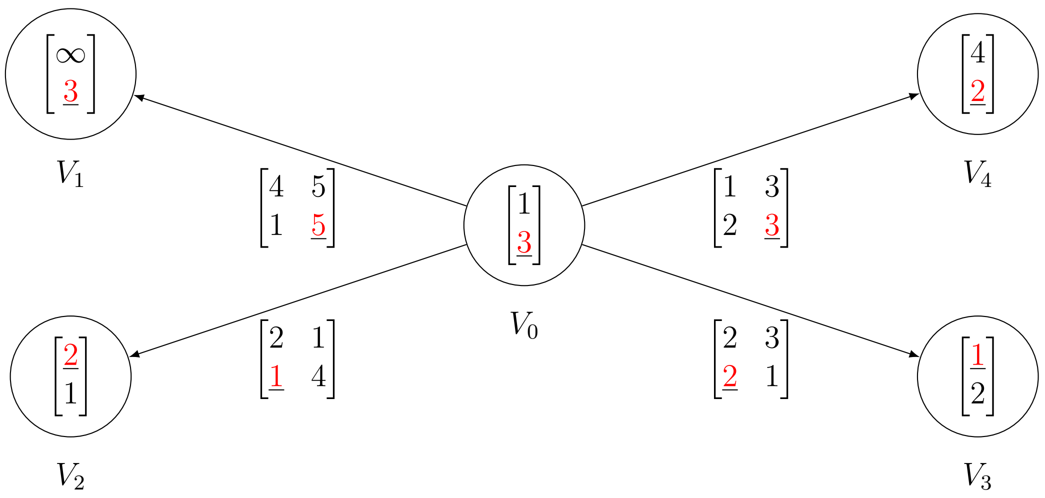
Figure 2.3: Solving a node of degree two using R2



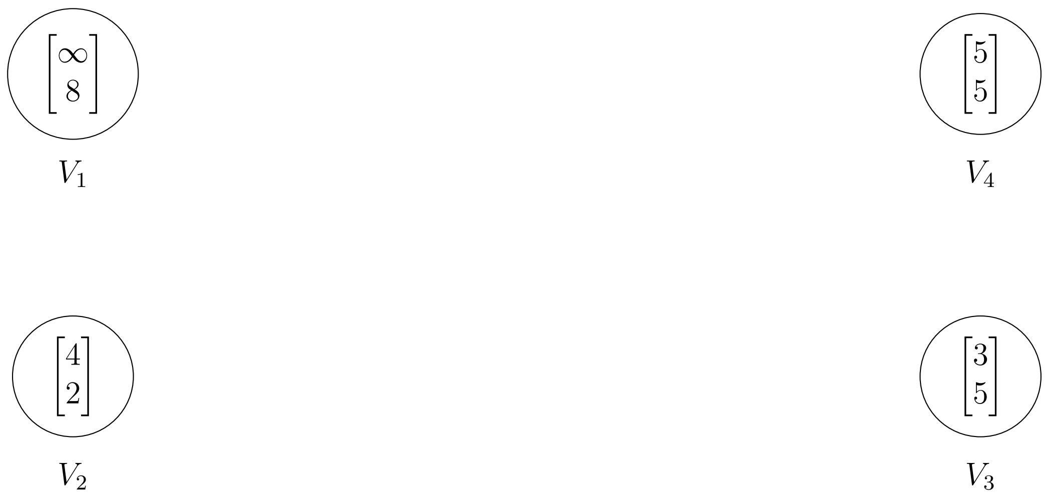
a A node of degree four before the reduction



b For each selection in V_0 we determine the local minimum, resulting in a total cost of 19 for the first selection



c For each selection in V_0 we determine the local minimum, resulting in a total cost of 22 for the second selection



$\begin{bmatrix}\infty\\8\end{bmatrix}$
V_1
$\begin{bmatrix}5\\5\end{bmatrix}$
V_4
$\begin{bmatrix}4\\2\end{bmatrix}$
V_2
$\begin{bmatrix}3\\5\end{bmatrix}$
V_3

d The first entry is selected for V_0, V_0 is removed and the costs for V_0s edges are added to their incident nodes

Figure 2.3: Solving a node of degree four using early decision RN

3 Design and Implementation

We implemented pbqp-papa[6], a library which allows solving PBQP instances using multiple solvers and different kinds of debugging and visualization for PBQP instances.

3.1 Library design

3.1.1 Programming language choice

Existing PBQP solvers like the one in libFirm which is called KAPS (Karlsruhe PBQP Solver) are implemented in C, but we chose C++ as language for this project, because it offers high level constructs like classes and templates. These constructs provide us with the means neccessary to create an expandable and more modern library, while avoiding the code duplication and limitations KAPS suffers from. C++ also has lots of useful data structures and algorithms in its std name space and allows easily integrating external libraries to automate tests, handle JSON or generate images. C++ was chosen over other object oriented language, because it lets us improve performance and memory usage due to its manual memory management and enables us to easily create an interface which is usable from C code. Both of these are very important for usage of the created library in compiler environments like libFirm.

3.1.2 General design

In our implementation PBQP instances are represented in their graph form. A graph consists of a set of pointers to nodes and a set of pointers to edges. Even though the graph only holds pointers to these edges and nodes they are still owned by the graph in the sense that only the graph can create nodes or edges, every node or edge can only be part of one graph at once and a graph will delete all of its nodes and edges when it is deleted itself. This behavior prevents graph components from leaking memory. An edge consists of a pointer to its source node, a pointer to its target node and a matrix containing the costs associated with the edge. A matrix is simply a 2-dimensional array with arbitrary height and length holding values. A node consists of a vector holding its cost, an integer index and a list of the edges incident to the node. Matrices, nodes, edges and graphs are all implemented as templates, where the template type is the type of value which is being held in the cost matrices/vectors.

This allows users of the library to put any numeric type in its cost matrices as long as basic operations like addition and comparison are defined for this type.

Graph instances are always guaranteed to be simple, meaning they have neither multiple edges nor loops. Attempting to create either of those is legal and will be taken into account for the problem instance, but no new edge will be created. For loops the diagonal of the cost matrix will be added to the cost vector of the node the loop is on. For multiple edges we add the cost matrix of the newly created edge to the cost matrix of the already existing edge. If the new edges direction is opposite to the existing ones we also transpose the cost matrix of the new edge beforehand.

Single reduction steps always return an instance of a subclass of the DependentSolution interface. DependentSolution requires subclasses to implement a solve() and a revert() method. The solve() method takes a solution instance as parameter in which all nodes existing after the reduction are already solved. The method is expected to add the correct selection for all nodes it reduced to the provided solution, possibly using the existing selections for other nodes. The revert() method takes a graph instance as parameter and reverts the reduction applied, it guarantees that if it is given the graph in its state after the reduction it will modify the graph back to its original state before the reduction. The combination of these two methods and their generalization in a superclass allows polymorphic treatment of reduction results. It makes it so the solving of a PBQP instance in its back tracking phase doesn't have to know at all which reductions were actually applied, it simply needs to call the solve() method for a list of DependentSolutions. Additionally new reductions can be implemented, integrated and swapped out easily.

Reduction steps never entirely delete nodes, but instead set a 'deleted' flag on the node to true. This allows removing nodes from the graph without having to worry about invalidating other pointers to the same node which may be kept by other reductions involving the same node or for examples PEOs. These can handle removed nodes accordingly after a simple check of the nodes 'deleted' flag.

To support infinite entries the library also offers a template type called 'InfinityWrapper'. Any numeric type that would be accepted as template type for a graph instance is also accepted as template type for an InfinityWrapper instance. An InfinityWrapper wraps a value of its template type and implements the standard arithmetic operations required to use it as data type in a PBQP graph instance. Every template type also has a special value which is used to represent infinity and the implemented arithmetic operations properly support this infinite value. For example $5+\infty = \infty+5 = \infty$ and $\infty \geq X$ is true for all$X$. Usually the special value used is the maximum value the number can represent. This choice minimizes the chance of ever reaching the special value unintentionally. A reduction or any other use case doesn't have to differentiate between a Matrix<T> and a Matrix<InfinityWrapper<T>>, it can just use the C++ built in operators and it'll work properly either way, because InfinityWrapper overrides those. InfinityWrapper thus centralizes all handling of infinity and its edge cases. Reductions can be implemented without any checks for infinity and if they are instantiated with a normal numerical template type which can not be infinite, no time is wasted doing any checks for infinite values or modified arithmetic operations. On the other side when using InfinityWrapper as a template type, checks for infinity and proper arithmetic calculations involving it are used automatically.

We also added support for exporting PBQP instances to JSON files and importing PBQP instances from those JSON files. This allows us to easily compare and benchmark different solving approaches or extract interesting problem instances. Previously KAPS could only solve problems that were provided to it directly through ongoing compilation in libFirm, there was no way to solve a specific problem.

Asserts are used widely throughout the library for consistency checks and to validate parameters. These checks ease debugging and ensure correctness, but can also be removed at compile time to avoid wasting time on them in production environments.

3.2 Solving PBQP with linear/quadratic programming solvers

Instead of solving PBQP instances in their graph form through reductions they can also be solved by turning them into a quadratic or linear programming problem and solved by an off-the-shelf solver. This is a promising alternative approach for retrieving minimal solutions of a PBQP instance. Linear/quadratic programming solvers will always output the ideal solution, which is very useful to have as comparison for evaluating the quality of our heuristic solutions. Additionally commercial linear/quadratic programming solver are highly optimized and can solve or simplify the PBQP instance in ways other graph form oriented solvers could not, thus possibly surpassing existing brute force implementation in speed. We turned PBQP instances into both linear programming and quadratic programming problems and solved them using Gurobi.

3.2.1 Solving PBQP through quadratic programming

To solve a PBQP instance with a Graph G and nodes V_n through quadratic programming we have to convert it into a single quadratic expression. Finding a minimal solution for the created expression is then done by an off-the-shelf solver.

Before we can define the expression to minimize we need to define helper variables. For every node V_i with a cost vector $\vec{m}_i$, d_i binary variables x_{ia} are created out of which one is 1 and the others are 0 each:

$$\begin{aligned} x_{ia} &\in \{0,1\} & \forall i \in [1,n], a \in [1,|d_i|] \\ \sum_{a=1}^{|d_i|} x_{ia} &= 1 & \forall i \in [1,n] \end{aligned}$$

We also need to forbid selections picking infinite costs in either cost vectors or matrices:

$$\begin{aligned} x_{ia} &= 0 && \text{if} \quad m_{ia} = \infty \\ x_{ia} * x_{jb} &= 0 && \text{if} \quad C_{ijab} = \infty \end{aligned}$$

The expression to minimize is now:

$$\min \quad \sum_{i=1}^{n} \sum_{a=1}^{d_i} x_{ia} * m_{ia} + \sum_{i} \sum_{j} \sum_{a}^{d_i} \sum_{b}^{d_j} x_{ia} * C_{ijab} * x_{jb}$$

Node costs create linear summands, but the edge cost matrices each add a quadratic summand.

3.2.2 Solving PBQP through linear programming

PBQP can also be expressed fully as a linear problem. Analogue to the definition as quadratic problem we define variables x_{ia}:

$$\begin{aligned} x_{ia} &\in \{0, 1\} && \forall i \in [1, n], a \in [1, |d_i] \\ \sum_{a=1}^{d_i} x_{ia} &= 1 && \forall i \in [1, n] \end{aligned}$$

and we forbid infinite selections in nodes:

$$x_{ia} = 0 \quad \text{if} \quad m_{ia} = \infty$$

Previously the matrix costs added quadratic summands to the the final expression. To avoid this we define a binary variable y_{jkbc} for every entry in every edge cost matrix C_{jk} of size b x c. For every cost matrix C_{jk} all the variables y_{jkbc} created are 0, except for one which is 1. The entry selected in the matrix also has to match the entries selected in its adjacent nodes. This is enforced by ensuring each row sum equals the binary variable with the same index in the source node of the edge and each column sum equals the binary variable with the same index in the destination node of the edge.

$$\begin{aligned} & y_{jkbc} \in \{0, 1\} \quad \forall C_{jk} \in G, \ \forall b \in [1, |\vec{m}_j|], \ \forall c \in [1, |\vec{m}_k|] \\ & \sum_{c=1}^{d_k} y_{jkbc} = x_{jb} \ \forall C_{jk} \in G, \ \forall b \in [1, |\vec{m}_j|] \\ & \sum_{b=1}^{d_j} y_{jkbc} = x_{kc} \ \forall C_{jk} \in G, \ \forall c \in [1, |\vec{m}_k|] \end{aligned}$$

With the y_{jkbc} we can also use a linear form to disallow infinite entries in cost matrices:

$$y_{jkbc} = 0 \quad \text{if} \quad C_{jkbc} = \infty$$

Finally we can define the linear function to minimize as a sum of products of binary variables and cost vectors or matrix values:

$$min \quad \sum_{i=1}^{n} \sum_{a=1}^{|\vec{m}_i|} x_{ia} * m_{ia} + \sum_{j} \sum_{k} \sum_{b=1}^{|\vec{m}_i|} \sum_{c=1}^{|\vec{m}_j|} y_{jkbc} * C_{jkbc}$$

3.3 Bruteforcing PBQP

Opposed to applying heuristic reductions once optimal reductions are no longer possible it is also possible to bruteforce the problem to find an optimal solution. We implemented two different means of brute forcing and evaluated their performance. The first one iterates over all possible selections in an order determined by a Gray code. The second one uses Branch and bound to skip expensive solutions as early as possible.

3.3.1 Bruteforcing PBQP in Gray code order

A solution for a PBQP with n nodes can be defined as a vector $\vec{S}$ where

$$|\vec{S}| = n$$
$$S_i \in [1, |m_i|] \quad i \in [1, n]$$

In this form we can start with $\vec{S} = 0$ and increment this solution by incrementing S_n by 1. If $S_n = |m_n|$, then S_n is set to 0 and S_{n-1} is incremented. Inductively applying this allows iterating over all possible solutions, effectively we're just counting up a number, but every digit with index i has its own base $|m_i|$.

Naive brute forcing can use this to iterate over all possible selections, recalculate the cost of the entire PBQP for every selection and eventually find an optimal solution. This is very inefficient though, because incrementing the solution usually only changes very few selections. For $d_j = 2$ only one selections changes for half of the increments, for a quarter of the increments only two selections change etc.. Our approach can thus be improved by only recalculating the cost of the nodes which had their selections changed and their incident edges.

We can do better by using a Gray code order to only recalculate one node per increments. For this we first initialize multiple arrays of helpers variables, each containg a value for every node:

```
unsigned int currentSelection [length];
bool trend [length];
unsigned short limits [length];
void initGrayCode() {

  unsigned int index = 0;
  for(Node node : graph.getNodes()) {
    currentSelection [index] = 0;
    //start by increasing everything
    trend [index] = true;
    limits [index] = node.getVector().getLength();
    index++;
  }
}
```

Using this we can then increment the Gray code by calling the following method. It iterates over all possible solutions, while only changing one selection every time.

```
unsigned int incrementGrayCode() {
  for(unsigned int i = 0; i < length; i++) {
    if(trend[i] {
      //increasing
      if(currentSelection[i] == limits [i]) {
        //trend is switched and iteration continues
        trend[i] = false;
      }
      else {
        //just increment
        currentSelection[i]++;
        return i;
      }
    }
    else {
      //decreasing
      if(currentSelection[i] == 0) {
        //trend is switched and iteration continues
        trend[i] = true;
      }
      else {
        currentSelection[i]--;
        return i;
      }
    }
  }
}
```

This method allows calculating the total cost of all possible solutions with minimal effort. It returns the index of the node which was changed, which allows finding the total cost of the current solution easily, because the previous selection of the node changed is known and the total cost of the previous solution is known. Subtracting the cost created by the changed node and its edges with the previous selection and adding the cost created with the new selection to the previous total cost gives the new total cost. Note that the bigger d_i is, the smaller becomes the improvement this approach offers compared to a naive brute forcing one.

A problem of this approach is that it can not handle infinite entries in cost matrices. This is due to the way the next solution is calculated based on modifying the current one. If the current solution has an infinite cost, then subtracting the cost added by a specific node will not return a valid result. Another possible issue is the fact that this algorithm will always try all solutions. High costs for specific selections in a node may make these selection obviously non-optimal, but the Gray code will still try all possible combinations involving them. This problem can be addressed by using Branch and Bound instead.

3.3.2 Bruteforcing PBQP using Branch and Bound

Using Branch and Bound we want to find an optimal solution, while looking at as few solutions as possible. To do so we first create a list of all nodes by applying a BFS algorithm to our PBQP, starting at an arbitrary node. The order in the list is the order in which the BFS found the nodes.

We then pick the first selection for every node and calculate the total cost for this solution. This cost is the starting minimum. From here on we apply a standard Branch and Bound algorithm to traverse our tree and find a smaller solution.

```
void findMinimalSolution(Node node, unsigned int
   currentKnownMinimum, unsigned int costSoFar, unsigned int
   * selections) {
  for(unsigned int i = 0; i < node.getVector().getLength();
     i++) {
    unsigned int updatedCost = calculateUpdatedCost(
       costSoFar, selections, node, i);
    if (updatedCost >= currentKnownMinimum) {
      //skip right away if we already know it's more
         expensive
      continue;
    }
    else {
      //iterate over the child nodes in the tree created via
         BFS
      selections [node.getIndex()] = i;
      //recursively find minimal cost
      Node next = node.getNextInBFSList();
      findMinimalSolution(next, currentKnownMinimum,
         updatedCost, selections);
    }
    }
}
```

The arguments it starts with are the first node of the BFS order, the minimal cost created using the first trivial solution, a starting cost of 0 and a selection which is 0 for every node.

Using the list based on the BFS to iterate over all nodes is not necessary for the algorithm to work, but doing so improves its speed. The order ensures that for almost all nodes for which a selection is being tried, at least one adjacent node already had a selection assigned. This applies for all nodes except for the root node. Additionally the BFS makes all other nodes adjacent to the current one more likely to already be selected or to be selected shortly after than it would be in a random order. This is good, because more pairs of adjacent nodes already being selected lets us determine more edge costs. More determined edge costs mean we get a bigger sum and more of the entire PBQP is taken into account, thus allowing us to recognize non-optimal solutions earlier and to skip them earlier.

4 Evaluation

4.1 Performance of linear/quadratic programming solvers compared to traditional PBQP solver

We implemented conversion of PBQP instances into both a linear and quadratic programming format and solved them using Gurobi. Gurobi is a commercial solver which is widely used for solving linear programming and quadratic programming problems. We compared the solutions produced by Gurobi via quadratic programming, by Gurobi via linear programming, by KAPS and by our library. We compared them under the aspect of solution quality, meaning the total cost across all edges and nodes in the solution they provided and under the aspect of solving time.

For this purpose we created a sample set of PBQP instances by compiling a SPEC2000 benchmark using libFirm and dumping all PBQP instances created during the compilation process to JSON files using our library and its C-Interface. Out of these dumped PBQP instances we picked multiple sample groups, each representing a different order of magnitude in terms of the PBQP instances node count. The groups picked can be seen in Figure 4.1

Node count	Sample size	Edge count	Edges/node
41	13	31	0.75
190	15	443	2.33
761	6	2195	2.88
1792	2	6528	3.64

Figure 4.1: Sample PBQP instance groups used for performance evaluation. Node and edge counts are averaged across the entire group and rounded to the nearest integer. Edges/node is rounded to the second decimal

Due to the massive differences in rarity and solving time for problems of bigger size the size of the sample groups is not identical. All problems used are within a 10 % margin of the average node count of their group though.

We solved each sample group n times, measured the total time taken and divided this total time by both the amount of problems within the same group and by n to obtain the average amount of time taken per PBQP instance solving. This benchmark was done on a machine with a quad-core i5-4670 running Ubuntu 18.04. The results can be seen in Figure 4.2. The amount of runs n decreases for larger sample groups due to the massively increased time required to solve them. The two larger sample groups were not solved using linear programming, due to the approaches exponential runtime. After over 12 hours the linear programming approach had not completed a single run for the group of average node size 761.

Our implementation behaves slower than the one provided in KAPS, but this difference becomes smaller with bigger PBQP instances. We assume that this is the case due to the increased overhead, which the object oriented implementation in pbqp-papa has.

The solving time for Gurobi using linear programming is massively increased compared to the solving time using quadratic programming. We assume that this is the case due to the increased amount of binary variables used in the linear programming approach. The quadratic programming approach uses one variable for each entry in each cost vector while the linear programming approach uses one variable for each entry in each cost vector and each cost matrix. This means for a graph with n nodes, vectors of length d and m edges the quadratic programming approach uses $O(n * d)$ binary variables while the linear programming approach uses $O(n * d + m * d^2)$ binary variables. The problem instances used here all have cost vectors with a length of 16 and thus 256 entries in every cost matrix. For a mediocre sized graph with 100 nodes and 200 edges this already results in an increase of factor 33 in binary variables of the linear programming approach compared to the quadratic one.

We also compared the quality of the solutions obtained from the different solvers for each sample group. Due to all solvers used being entirely deterministic we only had to solve every sample group once. For each sample group and each solver we calculated the sum of the costs of all obtained solutions and divided it by the amount of PBQP instances within the sample group. This gave us the average cost determined for each sample group. The results can be seen in Figure 4.3.

The solutions obtained by both approaches using Gurobi are identical, because they are both optimal. While using Gurobi in production is not feasible, due to its long run time for larger PBQP instances, it is very useful for evaluating the quality of our heuristics compared to the optimal solution. We can see that our heuristic solution becomes increasingly worse compared to the optimal one with bigger PBQP

Node count	n	KAPS	Gurobi Linear	Gurobi Quadratic	pbqp-papa
41	20	0.689ms	40.5ms	18.38ms	1.420ms
185	20	6.577	334.7s	0.827s	10.631ms
761	5	28.220ms	-	128s	40.492ms
1792	2	91.070ms	-	8299s	120.825ms

Figure 4.2: Solving time taken for sample PBQP instance groups on average per PBQP instance. Solving times are rounded to the last decimal shown

instances. For small instances there is no difference at all though.

The solution quality of KAPS and pbqp-papa are identical as well, which makes sense as they implement the same algorithm.

Node count	KAPS	Gurobi Linear	Gurobi Quadratic	pbqp-papa
41	0.692	0.692	0.692	0.692
185	449.6	225.3	225.3	449.6
761	76148	-	69613	76148
1792	272626	-	167315	272626

Figure 4.3: Solution quality across sample problem groups per PBQP instance on average

4.2 Branch and Bound

We implemented Branch and Bound as described in 3.3.2 and attempted to solve PBQP instances with it. We were unable to do so though, because smaller PBQP instances generated based on SSA-based register allocation vary very little in their cost distribution. Most of the PBQP instances in the sample group with the least nodes had an optimal cost of 1 and mostly coloring edges. This is very bad for Branch and Bound, because its effectiveness relies on bad solutions identifying themselves as such as early as possible. That is not the case here though, so the Branch and Bound algorithm has to iterate over almost all approximately 16^n possible solutions, assuming n nodes with a vector length of 16 each. For a small PBQP instances with 40 nodes with already ends up being 2^{160} possible solutions, an amount that can not be checked within feasible time

Even though larger PBQP instances have more variation in their costs and thus are more favorable for Branch and Bound structure wise, solving them was still not possible. With an increasing node count the NP-hard nature of the problem increases the amount of solutions to look at much faster than finding solutions gets easier due to more cost variety.

4.3 Findings made using debugging tools

There has been a PBQP instance, which KAPS could not solve. It was unclear whether this was due to an algorithmic problem or an implementation fault in KAPS. We implemented a solver dissecting the reduction and solving process step by step to find possible faults and using it we were able to solve this PBQP instance, thus determining that the fault lies within KAPS' implementation. This also reconfirms the findings of Buchwald et al.[2] regarding the early application of optimal reductions not influencing the validity of the PEO and thus the ability to find a solution.

5 Conclusion

5.1 Related work

Scholz and Eckstein.[7] first introduced the idea to use PBQP for register allocation. They used a linear heuristic and did not employ SSA form[2].

Hames and Scholz[8] further refined this approach with a new heuristic approach and a branch-and-bound algorithm for optimal solutions[2].

Buchwald et al.[2] applied PBQP to SSA-based register allocation and implemented KAPS and libFirms usage of PBQP based on it. They implement and compare different heuristics and establish early decision RN as the preferred one.

Scholz[9] offers an implementation for solving PBQP using a linear heuristic or a brute force approach, but the available source code is incomplete.

LLVM[10] has an experimental implementation of a PBQP solver using heuristics as part of their framework.

Frieler[5] developed a parallelized PBQP solver using X10. They do not support infinite entries in cost matrices or PEOs and used randomly generated PBQP instances for benchmarking purposes. Due to this and the lack of a C interface to connect to a compiler their implementation is not suited for our use case in SSA-based register allocation.

5.2 Future work

Parallelization in graph based solving algorithms as explored by Frieler[5] is not taken into account at all here due to time limitations. It could provide significant improvements to the run time of solvers. Parallelization would be possible both by separating the graph of a PBQP instance into its connected components and by simultaneously applying ideal reductions to nodes not sharing any adjacent nodes.

Additionally more diverse testing data would enable us to evaluate the performance and quality of our solvers in a more general sense towards PBQP. All PBQP instances used here are register allocation problems in libFirm, which all have similar density and similar cost structure. The majority of edges are coloring edges which have ∞ on the diagonal and 0 in all other entries. These type of edges only affect solvability, but not quality of the solution. All edges which are not coloring edges are affinity edges originating from coalescing properties. The content of these does vary, but in most cases they are still filled with only a combination of 0 and a single other integer. More diverse entries in the edges cost matrices would allow for much better evaluation of solvers solution quality.

A more domain oriented approach for implementing solving algorithms could be beneficial as well. This thesis uses libFirm mostly as a black box to generate the PBQP instances used. We make no connection between the instructions being optimized and the PBQP instance to solve, we only work based on the PBQP instance generated by KAPS. Making this connection could grant insight into the structure and generation of generated PBQP instances though and thus improve understanding of the PBQP instance. This could possibly lead to improved solving algorithms and maybe even allow reusing good solutions for common patterns reappearing during compilations.

Finally a direction worth exploring is solving PBQP instances with different PEOs or without them at all. Both our implementation and KAPS rely entirely on the existence of a PEO to ensure a solution can be found when infinite entries are present. They also treat KAPS implementation for generating a PEO, which is only based on coloring and does not take affinity edges into account at all, as a black box. Developing an improved PEO algorithm for generating PEOs which also take affinity edges into account would likely lead to improved solution quality. Additionally it could be possible to leave PEOs behind entirely and develop solving algorithms not using an external order for the application of RN. Instead these would have to ensure on their own that a solution can always be found.

5.3 Sum up

We developed a standalone library for creating, analyzing and solving PBQP instances. Using this library we evaluated the performance of our library in comparison to KAPS and showed that albeit slightly slower our implementation can compete in speed while offering a massively improved and more feature rich API. We used our library to analyze the performance and solution quality of different solving approaches. As part of this we introduced solving of PBQP as a quadratic programming problem, which provides optimal solutions within acceptable time for smaller PBQP instances. We showed that formulating and solving PBQP instances as a linear programming problem is not a good choice, because it performs significantly worse than the quadratic programming approach. Finally we showed that a Branch and Bound approach is not feasible for retrieving optimal solutions of PBQP instances generated from SSA-based register allocation.

Bibliography

[1] F. Bouchez, A. Darte, C. Guillon, and F. Rastello, "Register allocation: What does the np-completeness proof of chaitin et al. really prove? or revisiting register allocation: Why and how," in *Languages and Compilers for Parallel Computing* (G. Almási, C. Caşcaval, and P. Wu, eds.), (Berlin, Heidelberg), pp. 283–298, Springer Berlin Heidelberg, 2007.

[2] S. Buchwald, A. Zwinkau, and T. Bersch, "Ssa-based register allocation with pbqp," in *Compiler Construction* (J. Knoop, ed.), vol. 6601 of *Lecture Notes in Computer Science*, pp. 42–61, Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-19861-8_4.

[3] A. Anderson and D. Gregg, "Optimal DNN primitive selection with partitioned boolean quadratic programming," *CoRR*, vol. abs/1710.01079, 2017.

[4] KIT, "libfirm." `https://pp.ipd.kit.edu/firm/index.html`, Feb. 2019. [Online; accessed 14-February-2019].

[5] C. Frieler, "Entwicklung eines parallelen pbqp-lösers mit x10," Apr. 2012.

[6] M. Baumstark, "pbqppapa." `https://github.com/maxopoly/pbqp-papa`, Feb. 2019. [Online; accessed 22-February-2019].

[7] B. Scholz and E. Eckstein, "Register allocation for irregular architectures," *SIGPLAN Not.*, vol. 37, pp. 139–148, June 2002.

[8] L. Hames and B. Scholz, "Nearly optimal register allocation with pbqp," in *Modular Programming Languages* (D. E. Lightfoot and C. Szyperski, eds.), (Berlin, Heidelberg), pp. 346–361, Springer Berlin Heidelberg, 2006.

[9] B. Scholz, "libfirm." `http://www.complang.tuwien.ac.at/scholz/pbqp.html`, Feb. 2004. [Online; accessed 21-February-2019].

[10] LLVM, "Llvm pbqp implementation." `https://llvm.org/doxygen/namespacellvm_1_1PBQP.html`, Feb. 2019. [Online; accessed 21-February-2019].

Erklärung

Hiermit erkläre ich, Max Baumstark, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift