Pebbles in Motion

Polynomial Algorithms for Multi-Agent Path Planning Problems

Author:
Pat Mächler
patrick.maechler@stud.unibas.ch

Supervisor:
Gabriele Röger
Prof. Malte Helmert

Submitted by 20th August 2012 to

Artificial Intelligence,
Department of Mathematics and Computer Science,
University of Basel

UNI
BASEL

in partial fulfillment of the requirements for
the degree of *Master of Science in Computer Science*

Abstract

Multi-Agent-Path-Finding (MAPF) is a common problem in robotics and memory management. *Pebbles in Motion* is an implementation of a problem solver for MAPF in polynomial time, based on a work by Daniel Kornhauser from 1984. Recently a lot of research papers have been published on MAPF in the research community of Artificial Intelligence, but the work by Kornhauser seems hardly to be taken into account. We assumed that this might be related to the fact that said paper was more mathematically and hardly describing algorithms intuitively. This work aims at filling this gap, by providing an easy understandable approach of implementation steps for programmers and a new detailed description for researchers in Computer Science.

Contents

Chapter 1

Introduction

In recent years there has been an increasing number of publications in journals and conferences in Artificial Intelligence and Robotics on the topic of solving Multi-Agent Pathfinding (MAPF) problems. In this chapter we will introduce this problem along with the problem of pebble motion and how they relate to each other. We discuss their application areas and certain properties for their computational complexity.

We close this chapter by describing the aim of this work and the further structure of this report.

1.1 MAPF and Pebble Motion Problems

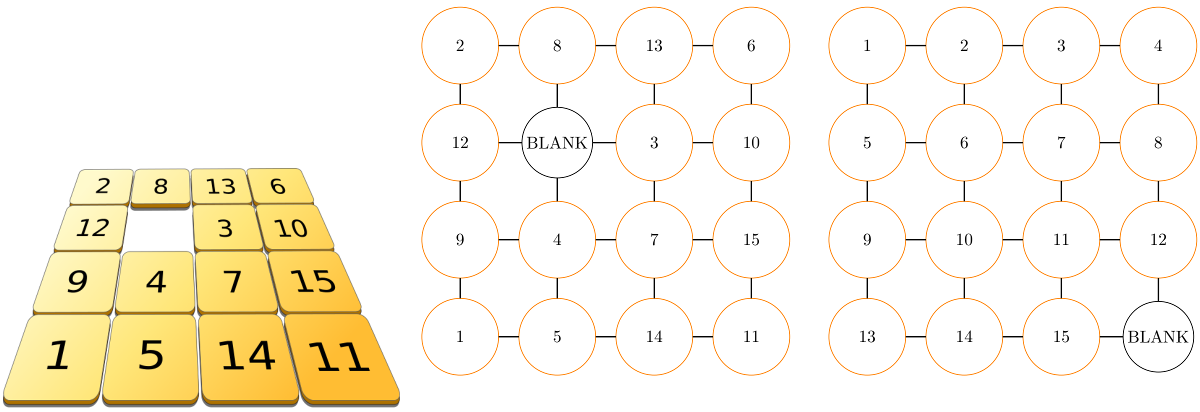
In this section we describe the MAPF and pebble motion problems and how they relate to each other. We will see that pebble motion problems can be seen as the equivalent of a major subclass of MAPF problems.

1.1.1 Pebble Motion Problems

A pebble motion problem is given by a graph, where each vertex is either occupied by a pebble that should reach a goal position (distinct from all other pebbles) or unoccupied ("blank"). In order to achieve this all pebbles can move sequentially along the edges of the graph from an occupied vertex to a blank vertex. The problem is solved if all pebbles have reached their goal positions.

A widely known pebble motion problem in recreational puzzles is that of the 15-puzzle.

It has been shown that for any given problem in this area finding the best solution with a minimal number of steps is NP-hard [9], i.e. it is not feasible to compute the optimal solution with an increasing



(a) Classical variant. (b) Translated into a graph. (c) In goal configuration.

Figure 1.1: The classical 15-puzzle, its transformation into a graph and displyaying the goal configuratio.

number of vertices and pebbles, thus solving suggestions are trying to generate a solution with few steps in reasonable polynomial calculation.

Research in pebble motion problems was carried out primary by mathematicians around the eighties. The work by Wilson [14] published 1974 was a major step by indicating an efficient solving technique on biconnected graphs, which was later refined in 1984 by Kornhauser, Miller and Spirakis whose work [3][4] gives a mathematical description for an algorithm that solves with a generalization for all types of such graphs.

1.1.2 Multi-Agent Pathfinding

An often described application for MAPF problems are multiple robots in a storage room that should reach a certain distinct goal position in a non-competitive way.

Thus the abstract definition of a MAPF problem is that there is a space occupied by a number of agents whose aim is to reach a certain distinct goal position. The problem is solved if all agents have reached their goal positions.

Almost all solving approaches divide the space into a discrete graph (or assume it is already discrete) where the agents are put onto the vertices and then a plan is to be created to reach the goal in a low number of steps, as finding an optimal solution is NP-hard as well [2].

1.1.3 Relation Between MAPF and Pebble Motion Problems

MAPF could be thought of a generalization of pebble motion problems: In pebble motion problems we usually assume that there is a central planner who aims to minimize a sequential execution of moves on the graph.

In MAPF solving research there are mainly two distinctions made

1. whether the agents can move in parallel or only sequential as well
2. if there is a centralized all-knowing planning agent or if the agents plan on their own depending on their knowledge

It can be intuitively seen that the problem of sequential, centralized planning MAPF is directly equivalent to the pebble motion problem. However the recent research has not yet really reflected the works on pebble motion from 1974 nor 1984 ; likely reasons for this were

- that the research was carried out by rather different research communities,
- at different decades and
- furthermore the archival proceedings publication by Kornhauser, Miller and Spirakis [4] is quite sketchy, often refering to a “final version” that never appeared. The actual details are described in Kornhauser’s master’s thesis [3].

1.2 Aim of this work

The aim of this work is to provide a step-by-step explanation of the approach for solving pebble motion problems as described in 1984 by Kornhauser, Miller and Spirakis [3], with an emphasis placed on the pebble motion problem cases of biconnected graphs with 1 blank vertex, as this is the most interesting one.

A major movitivation is further to provide a freely available implementation of that work.

1.3 Further structure of this paper

In chapter 2 we will introduce some approaches to solving MAPF and pebble motion problems in brevity.

In 3 we will explain in detail the ideas of the works published in 1984 [3][4] by Kornhauser, Miller and Spirakis.

In chapter 4 we will provide some code fragments to translate the ideas of just mentionned works into a more algorithmic description, thus improving the preconditions for other implementations.

In chapter 5 we will close with a discussion and steps for future work.

Chapter 2

Background

In this chapter we will describe some recent approaches to solving MAPF problems quite briefly. The group-theoretic approaches for solving centralized-planned, sequential MAPF problems (on which this work is based upon) by Wilson from 1974 [14] and refined by Kornhauser et al. in 1984 [3][4] are started to be described in detail in 3.

In 2008 Peasgood et al. [8] presented a centralized planning algorithm in linear time for trees, where a solution is guaranteed iff there are less agents than leaves in the tree.

In 2009 Wang and Botea [13] defined the *class of SLIDEABLE* MAPF problems (which are based on a grid) along with their algorithm *MAPP* to solve it in low-polynomial time for most but not all cases of SLIDEABLE. An extended version of *MAPP* from 2011 [12] by Wang and Botea can solve a larger subclass of SLIDEABLE problems than the one from 2009.

In 2011 Khorshid et al. [2] presented their tree-based agent swapping strategy *TASS*, which is a centralized polynomial-time algorithm for a class of trees that can find solutions on more densely occupied as in the paper by Peasgood et al. (e.g. for 4 blanks and 966 agents). Furthermore they presented a Graph-to-Tree Decomposition (*GDT*) algorithm (e.g. usable to transform SLIDEABLE into trees).

The *Push and Swap* algorithm from 2011, by Luna and Bekris [6] [7] is complete for all graphs with at with at least two unoccupied nodes.

However looking in detail at all these works it is clear that many findings have already been covered by Wilson [14] or Kornhauser et al. [3]. The works by Luna and Bekris, as well as by Peasgood et al. rely on observations that were already contained in Kornhauser's thesis, eventually to be interpreted as instantiations of the general algorithm by Kornhauser et al. for the case with at least two unoccupied vertices. Khorshid et al.specify conditions which are sufficient for a solvable, but are just special cases of Kornhauser's et al. criteria when considering only trees, as Kornhauser gives a more precise analysis, specifying sufficient and necessary conditions.

According to our observation the researchers of these works regard the identification of tractable subclasses of MAPF problems for non-optimal solving as an interesting open problem. Wang and Botea [12] empasized that they

> have identified conditions for a class of multi-agent path planning problems on grid maps that can be solved in polynomial time

Khorshid et al. wrote [2]

> We have demonstrated that this algorithm runs in polynomial time [...]. This work is just one step in classifying problems which can be solved in polynomial time.

Luna and Bekris stated [6]

> In comparison to existing complete alternatives, the proposed method provides completeness for a much wider problem class.

In contrast to the statements above from recent research work in AI and Robotics on MAPF, the work of Kornhauser et al. from 1984 [3] [4] describes a centralized planned solving procedure in polynomial-time for all solvable, sequential MAPF problems which creates solutions with upper and lower bounds of

$O(n^3)$ moves required on graphs with n vertices. Thus to the best of our knowledge this work has neither been ever fully implemented[1], nor taken fully into account in recent research; the only notable exception beeing Surynek who already pointed out in early 2009 [10] the strong connection between research on MAPF problems and the earlier work on pebble motion problems, as well as providing further improved (but rather briefly explained) algorithms *BIBOX* [10] and *BIBOX*$-\Theta$ [11] for solving the specific class of problems on biconnected graphs with one blank explicitely based on the earlier works by Wilson [14] from 1974 and the improved procedure by Kornhauser et al. from 1984 [3].

[1] i.e. including their insights into the decomposition of connected graphs

Chapter 3

Mathematical Description

In this chapter we will explain in detail the ideas of the work published in 1984 [3][4] by Kornhauser, Miller and Spirakis relevant to solving pebble motion problems, i.e. we omit the last part of their work, which is refering to insights on the diameter of permutation groups.

We start by introducing useful mathematical notations and then proceed with an overview of the basic ideas of the group-theoretic proofs for pebble motions problems.

3.1 Basic Mathematical Definitions

In this section we are going to introduce some basic definitions of mathematical structures that are crucial in order to understand the ideas that were published by Kornhauser et al.

3.1.1 Graphs and Paths

As graphs are building the main underlying structure of pebble motion problems, we start by giving a formal definition for these before we can start to formalize the problems as such in the next subsection.

Definition 1 (Graph) *A (simple, undirected)* graph $G = (V, E)$[1] *consists of*

1. *a set of* vertices *(or nodes)* V
2. *a set of* edges E *whose elements are 2-element subsets of* V.

Given a vertex $v \in V$ *its* adjacent vertices *(or directly neighbored vertices), the subset consisting of all attached vertices to* $v \in V$ *is denoted by* $adj(v)$, *i.e.* $adj(v) = \forall u \in V, \{u, v\} \in E \equiv u \in V_v$. *The* valence *(or* degree*)* $val(v)$ *of a vertex* $v \in V$ *is an integer denoting the number of edges attached to it, i.e.* $val(v) \equiv |adj(v)|$.

A graph $G = (V, E)$*is a* bipartite graph *iff there are disjoint vertex sets* X *and* Y *with* $V = X \dot\cup Y$ *and for all* $\{v_x, v_y\} \in E, (v_x \in X$ *and* $v_y \in Y)$ *or* $(v_y \in X$ *and* $v_x \in Y)$.

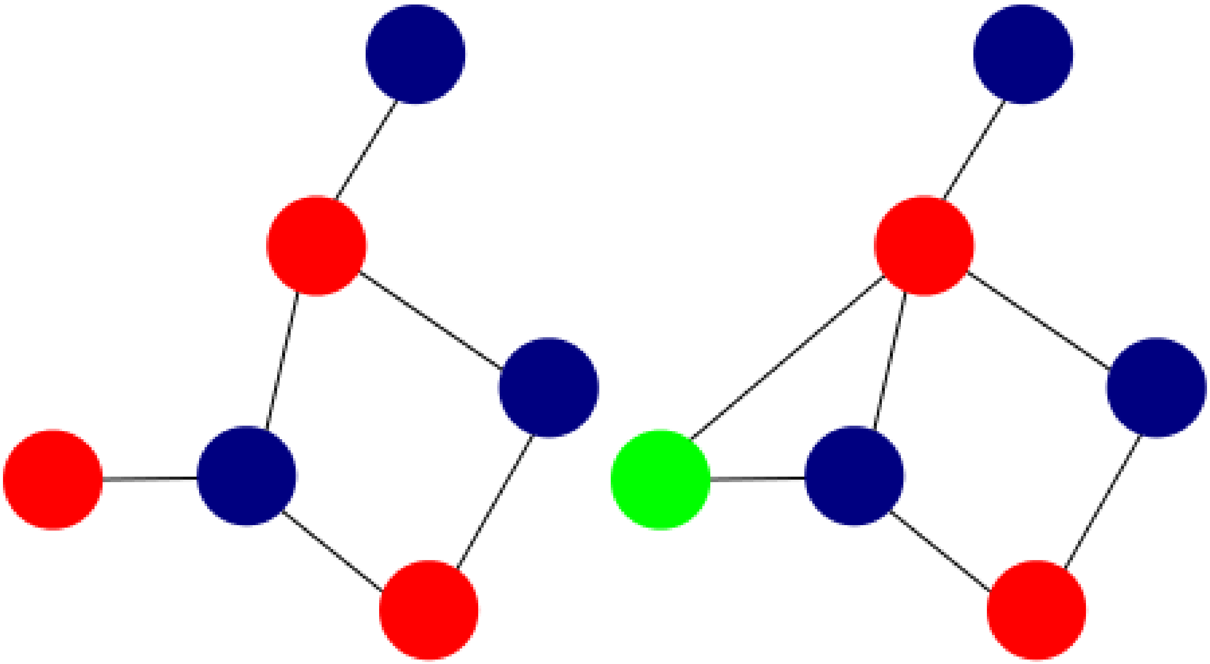
Further we define paths on a graph which will turn out useful later for various purposes.

Definition 2 (Path) *A* path p *on a graph* $G = (V, E)$ *from some vertex* v_0 *to some vertex* v_n *is a sequence* $p = [v_0, \ldots, v_n]$ *whose elements* $v_i \in V, \forall i, 0 < i \leq n, \{v_i, v_{i+1}\} \in E$.

A simple path *has the requirement that no vertex is visited more than once i.e. the path* $[v_0, \ldots, v_n]$ *in the graph* $G = (V, E)$*has the property that* $\forall i, j, 0 \leq i < j < n, v_i \neq v_j$.

A closed path $[v_0, v_1, \ldots, v_n]$ *on a graph* $G = (V, E)$*has the property that the start and ending vertex are identical, i.e.* $v_0 = v_n$. *Otherwise it is an* open path.

[1] We will solely refer to graphs with unweighted, undirected edges in this work.



(a) A bipartite graph. (b) A non-bipartite graph.

Figure 3.1: Examples of a bipartite and a non-bipartite graph.

A cycle path *is a path that satisfies the requirements for a* closed path *and a* simple path *(with the exception that $v_0 = v_n$).* [2]

Given a path $p = [v_0, \ldots, v_n]$ we write $p^{-1} = [v_n, \ldots, v_0]$ to denote the reversed order of elements of the path sequence p.

Let us denote a handle *of a graph G as a simple path h on G from a vertex v to a distinct vertex w, where for all vertices $u \in h, u \neq v \neq w, val(u) = 2$ and for vertices $val(v) \geq val(w) > 2$. The vertices of $u \in V$ with $val(u) = 2$ are denoted as* internal vertices of the handle.

Let us similarly denote to handles an arm *of a graph G from a vertex v to a distinct vertex w, the difference beeing that that $val(v) = 1$ or $val(w) = 1$.*

One application of paths is to define the set of connected graphs.

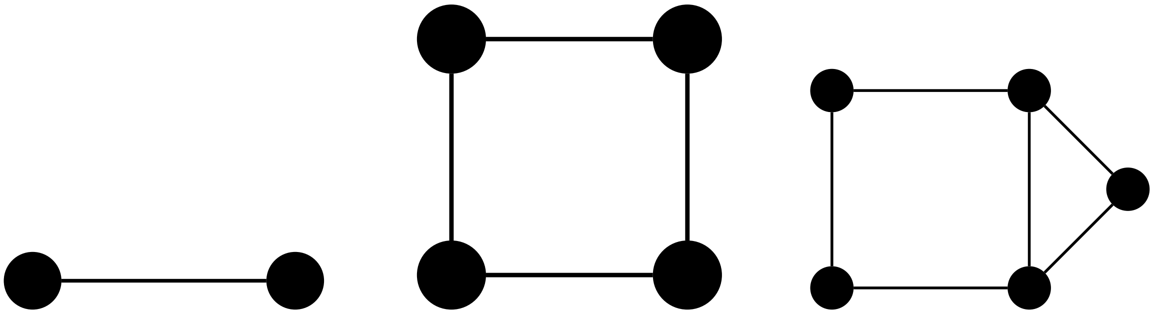
Definition 3 (Connected and Disconnected Graphs) *A graph $G = (V, E)$belongs to the set of* connected graphs *iff for any two vertices $v_1, v_2 inV$ there is a path from v_1 to v_2 in G. Otherwise the graph belongs to the set of* disconnected graphs.

A disconnected graph $G = (V, E)$can be easily decomposed into a finite set of connected graphs, denoted as connected subgraphs of G. *Use Starting with an arbitrary $v \in V$ an empty set W where , using $adj(()V)$, a set. We will refer to this decomposition function as $conG(G) = \{G_0 = (V_0, E_0), G_1 = (V_1, E_1), \ldots, G_g = (V_g, E$ If $G = (V, E)$is aleady a connected graph $conG(G)$ results simply into the set with the graph G itself as its only element, i.e. $conG(G) = \{G = (V, E)\}$ An algorithmal description for $conG(G)$ is found at 4.2.1.*

We can now further define the betti number[3] *of a graph by the function $betti(G) = |V| - |E| + |conG(G)|$ (recall $|conG(G)| = 1$ for a connected graph). Informally the betti number describes the number of "loops" in a graph.*

[2]In most scientific literature on graphs such a path is simply called a *cycle*. However we will later proceed with a small introduction into permutation theory, where there is also the notion of *cycle permutations*, which are mathematically not equivalent to *cycle paths* on graphs, but also informally often referred by the term *cycle* in literature; for clarity reasons we decided thus to use the term *cycle path* throughout this work when speaking about paths on graphs that satisfies their requirements.

[3]To be mathematically more precise: we denote in this work with the function "$betti(G)$" an applied definition for graphs of the function $betti_1(S)$. There is a set of *betti functions* $\{betti_0(S), betti_1(S), betti_2(S), \ldots\}$ where $betti_i(S)$ is solely applicable to a topological space S as argument with $dimension(S) = i + 1$. $betti_1(S)$ is applicable to a graph $G = (V, E)$as defined in this paper which is a planar topological space with $dimension(G) = 2$. As only this case is relevant for this work we refrain from going more into details of the *betti functions*.



(a) A trivial biconnected or T_0 graph (b) A polygon or T_1 graph (c) A T_2 graph

Figure 3.2: Examples of biconnected graphs classified into disjoint subsets by their betti number

We now define the biconnected graphs B (a subset of connected graphs) and further decompose it into disjoint subsets T_X.

Definition 4 (Biconnected Graphs with T_X) *For a graph $G = (V, E)$and $v \in V$ we define a surjective mapping $removeVertex(G, v)$, whose application results in a graph G_r where v is fully removed from G i.e. $removeVertex(G, v) = G_r(V \setminus v, E \setminus \{\{v, u\}, \forall u \in adj(v)\}$.*

A graph $G = (V, E)$belongs to the subset of biconnected graphs *iff by choosing an arbitrary vertex $v \in V$ the result of $removeVertex(G, v)$ is not a disconected graph. An alternative definition is that G is biconnected iff for $v_i, v_j \in V$ there is a* simple cycle path $[v_i, \ldots, v_j, \ldots, v_i]$ *on G.*

Let B denote the set of all biconnected graphs and decompose it into disjoint subsets $B = T_0 \dot\cup T_1 \dot\cup T_2 \dot\cup T_3 \dot\cup \ldots$, where the index denotes the respective betti numbers of the graphs i.e. a graph $G \in B, i \geq 0, G \in T_i$ iff $betti(G) = i$.

The subset T_0 is further denoted trivial biconnected graphs *whose graphs consist of a single edge and two vertices i.e. $|V| = 2, |E| = 1$.*

The subset T_1 is further denoted as polygons*, whose graphs consists of a single "loop" i.e. $\forall v \in V val(v) = 2$.*

Given a handle $h = [v, u_1, u_2, \ldots u_u, w]$ in graph G we define a surjective mapping $removeHandle(G, h)$ whose application results in a graph G_h where all internal vertices of the handle h have been removed from G, i.e. $removeHandle(G, h) = removeVertex(removeVertex(\ldots removeVertex(G, u_u) \ldots, u_2), u_1)$.

3.1.2 Pebble Motion Problems

We now can formally describe a pebble motion problem.

Definition 5 (Pebble Motion Problem and Moves) *A* pebble motion problem *is formally described as $\Pi = (G, P, a, z)$, where*

- *$G = (V, E)$is a graph*
- *$P = \{p_0, p_1, \ldots, p_k\}$ is a set of pebbles with $|P| \leq |V|$*
- *a is the initial pebble configuration, a mapping of P to V (see below)*
- *z is the goal pebble configuration.*

A pebble configuration *of* Π *is a mapping as $c \colon P \to V$ is an injective function from P to V. The image of c consists of the* occupied vertices *which we will write as V^c_+. $V^c_- = V \setminus V^c_+$ denotes the set of all* blank vertices *(or* blanks*). Let further denote C_Π the set of* all pebble configurations *of* Π.

A move *m from vertex $v \in V$ to vertex $winV$ on graph G is denoted as an ordered pair $m = (v, w)$. Note that each* move *$m = (v, w)$ has an inverse move $m^{-1} = (w, v)$. We define a partial function* execMove: $(V \times V) \times C_\Pi \rightarrow C_\Pi$. *It maps moves and configurations to configurations if the move is applicable on the configuration c; otherwise* execMove *is undefined:*

$$execMove((v,w),c) = \begin{cases} c\left[c^{-1}(v) \mapsto w\right] & if\ v \in V_+^c, w \in V_-^c, \{v,w\} \in E \\ \perp & otherwise \end{cases}$$

where $c\,[p \mapsto v]$ *denotes the configuration which is equivalent to* c *except that pebble* p *is mapped to vertex* v.

For a move sequence $M = [m_1, m_2, \ldots, m_n]$ *we define an analogous partial function* execMoveSeq(M, c) *iff the moves in* M *are consecutively applicable starting on configuration* c*, as otherwise* $execMoveSeq(M, c)$ *would be undefined.* $execMoveSeq([m_1, m_2, \ldots, m_n], c) = execMove(m_n, execMove(m_{n-1}, \ldots execMove(m_1, c) \ldots))$

A move sequence M *is a* solution *of* Π *if* $execMoveSeq(M, a) = z$.

We define the function $generatePebbleMS(p)$ *which creates the move sequence* $M = [(v_0, v_1), (v_1, v2), \ldots, (v_{n-1}, v_n)]$ *given a path* $p = [v_0, v_1, v_2, \ldots, v_{n-1}, v_n]$

A move sequence $M = [(v_1, w_1), (v_2, w_2), \ldots, (v_n, w_n)]$ *is* simple *if for* $1 \leq i < n, v_i = w_{i+1}$. *Such a simple move sequence can also be seen as moving a "blank" along a path on* G. *Hence we can specify such a move sequence alternatively by the trace of this blank, i.e.* $\langle M \rangle = \langle w_1, w_2, w_3, \ldots, w_n, v_n \rangle = \langle w_1, v_1, v_2, \ldots, v_{n-1}, v_n \rangle$.

We define the function $generateBlankMS(p)$ *which creates the simple move sequence* $\langle M \rangle = \langle v_0, v_1, v2, \ldots, v_{n-1}, v_n \rangle$ *given a path* $p = [v_0, v_1, v_2, \ldots, v_{n-1}, v_n]$

3.1.3 Permutations, Cycles, Transitivity and Conjugations

For the next section to understand it is a preliminary to define the notion of Permutations and structures that are build upon them in order to further explain the basic algorithm. Note that the definitions in this subsection are to be found in a similar form at the master's thesis

Definition 6 (Permutations amd Cycles) *A* cycle h *is a bijective mapping from a finite set* Q *to itself. It is denoted by* k *distinct elements* $h = \begin{pmatrix} e_1 & e_2 & \ldots & e_k \end{pmatrix}, \forall 1 \leq i \leq k, e_i \in Q$ *The application of a cycle* $h(Q)$ *maps an element denoted in* h *to the element denoted in* h *on it's right side except for the last element that is mapped to the first element and all elements* $\in Q$ *not denoted in* h*, i.e.* $\forall 1 \leq i < k, e_i \mapsto e_{i+1}, e_k \mapsto e_1$*).*

An cycle which maps k elements is denoted as k-cycle*. A 2-cycle is also denoted as* transposition *or* swap.

A permutation *is an analogous mapping to a cycle. A permutation is written as*

$$P = \begin{pmatrix} e_1 & e_2 & \ldots & e_{n-1} & e_n \\ f_1 & f_2 & \ldots & f_{n-1} & f_n \end{pmatrix}$$

By applying a permutation P *on a set* Q *the elements on the upper row are mapped to the elements on the lower row i.e.* $\forall 1 < i < n, e_i \mapsto f_i$

Any permutation can also be written in cycle notation*, a product i.e. a sequential application of k-cycles* $P = \begin{pmatrix} e_1 & f_1 \end{pmatrix} \begin{pmatrix} e_2 & f_2 \end{pmatrix} \ldots \begin{pmatrix} e_{n-1} & f_{n-1} \end{pmatrix} \begin{pmatrix} e_n & f_n \end{pmatrix}$ *A permutation (or cycle) which maps no element to another is denoted as* identity permutation*.*

P *is an* even permutation *if it can be expressed by an even number of transpositions. Otherwise* P *is an* odd permutation*. It follows that any k-cycle is an even permutation if it k is odd, and an odd permutation if k is even.*

Let us denote a permutation group by $P_n = (Q, P)$

- Q *a set,* $|Q| = n$
- P *a set of permutations on* Q

$S_n = (Q, P = S)$ *denotes the* symmetric group *where* S *as the set of all possible permutations on* Q.

Let $A_n = (Q, P = A)$ *denote the* alternating group *a subgroup of* S_n *with* A *as the set of all even permutations on* Q.

Given a permutation group, we now define the notion of transitivity.

Definition 7 (Transitivity) *A subgroup $P_n = (Q, P)$ of S_n is said to be* k-transitive *iff for any two subsets $A, B \subset Q$ with $|A| = |B| = k$ there is a sequence of permutations $\in P$ whose consecutive application maps A to B, i.e. for $\forall A = \{a_1, a_2, \ldots, a_k\}, B = \{b_1, b_2, \ldots, b_k\} \subset Q \exists T_{AB} = [p_1, p_2, \ldots p_t], 1 \leq i \leq t, p_i \in P, T_{A,B}(A) = p_t(p_{t-1}(\ldots p_1(A)\ldots)) \mapsto B$.*

For a subgroup P_n to be k-transitive it is sufficient, if there is a fixed subset $C \subset Q, |C| = k$ to which any $A \subset Q, |A| = k$ can be mapped to by a permutation sequence $T_{A,C}$. As permutations are bijective, there is also an inverse permutation sequence $T_{C,A}$ whose application maps $C \mapsto A$ for any $A \subset Q$. Thus for any two subsets $X, Y \subset Q$ with $|X| = |Y| = k$ to map X to Y we use a consecutive application of $T_{X,C}$ and $T_{C,Y}$, i.e. $T_{C,Y}(T_{X,C}(X)) \mapsto Y$

It is obvious that the symmetric group S_n is n-transitive.

It is also easy to show that the alternating group $A_n = (Q, A)$ must be $(n-2)$-transitive as either $\begin{pmatrix} e_1 & e_2 & \ldots & e_{n-2} & e_{n-1} & e_n \\ f_1 & f_2 & \ldots & f_{n-2} & f_{n-1} & f_n \end{pmatrix}$ or $\begin{pmatrix} e_1 & e_2 & \ldots & e_{n-2} & e_{n-1} & e_n \\ f_1 & f_2 & \ldots & f_{n-2} & f_n & f_{n-1} \end{pmatrix}$ are an even permutation and are thus $\in A$.

We then proof that A_n can be created be the at most $n-2$ 3-cycles.

Definition 8 (Generation of permutation groups) *A* permutation group *can be created for any finite set $F = f_1, f_2, \ldots, f_k$ of permutations on n elements . The set G created by consecutive (and repetitive) applications of any element in F is clearly a group, denoted as the* permutation group generated by *F. We denote $G(F) = G$ for this operation.*

We can show that the alternating group $A_n = (Q, A)$ can be generated by the set of all 3-cycles on n elements denoted as Z_n and is expressible as a sequence of at most $n-2$ 3-cycles. As all 3-cycles are even permutations, $G(Z_n)$ must be a subgroup of A_n. In order to show that $G(Z_n) \equiv A_n$ take any permutation $A \ni a \begin{pmatrix} e_1 & e_2 & \ldots & e_{n-1} & e_n \\ f_1 & f_2 & \ldots & f_{n-1} & f_n \end{pmatrix}$. Let $Z_n \ni p_i = \begin{pmatrix} e_i & f_i & g_i \end{pmatrix} 1 \leq i \leq n$ denote a corresponding 3-cycle which maps e_i to f_i, f_i to some g_i and g_i to e_i. If p_1 does not map e_2 to f_2 we multiply p_1 with $p_2 = \begin{pmatrix} p_1(e_2) & f_2 & g_2 \end{pmatrix}$ where we choose p_2 such that $g_2 \neq f_1$ (i.e. an element that has already been correctly mapped by p_1 previously). If necessary (i.e. iff for some $i, e_i \neq f_i$ already), we continue to consecutively choose a practical p_i (i.e. $g_i \notin \{f_1, \ldots, f_{i-1}\}$ and multiply it with our p_{i-1}. This product results in a permutation which already maps all f_i correctly up to the point when we reach $i = n-2$. However the elements f_{n-1} and f_n must already be correctly mapped at this point, as if they would be not correctly mapped, we would need an odd permutation to fix this mapping, which contradicts the assumption that $a \in A_n$.

Note: By a similar induction it can be shown that S_n can be decomposed in a product of at most n-1 2-cycles.

We now define the conjugation mapping and some of it's properties which will be crucial for solving pebble motion problems efficiently.

Definition 9 (Conjugation) *A* conjugation *is a mapping of the elements of a permutation to another. Given 2 permutations S and T the conjugation of permutation S by T is defined as applying $T^{-1}ST$. This basically applies permutation S to the elements of T as if they were replacing the elements in S. If S was a k-cycle the conjugation of S is again a k-cycle.*

Combining conjugation mappings and the property of transitivity we get the important insight: ***Given any permutation group P_n of which is known that it is k-transitive and that it contains a k-cycle S, then P_n contains all k-cycles.***

3.2 The Approach by Kornhauser

In this section we will informally describe the approach from the master thesis by Kornhauser et al. [3], which is partially based on the previous work by Wilson 1974 [14].

As discussed Wilson showed how to solve pebble motion problems on biconnected graphs with one blank.

Kornhauser et al. showed how to improve Wilson's work on biconnected graphs with generating solutions with a lower number of steps. Furthermore they introduced a generalized solving procedure for pebble motion problems on all types of undirected graphs. In the upcoming subsections we will explain this procedure.

3.2.1 Introductory Overview

The pseudocode listing below gives a first introductury overview over the primary procedures for solving a pebble motion problem $\Pi = (G, P, a, z)$. It is similar to the description of the "("main theorem) found on page 34 of Kornhauser's thesis [3]. For brevitiy of this overview we assumed that Π is solvable, thus left out error handling and other minor details.

```
solveAPebbleProblem (Problem) {
       foreach conProblem of Problem
              solveAConProb(conProblem);
}
//---------------------------------------------------
solveAConProblem(conProblem) {
       rearrangePebbles(conProblem)
       if conProblem is some special problem
              solveASpecialProblem(conProblem)
       else
              foreach subProblem of conProblem
                     solveSubProblem(subProblem)
}
//---------------------------------------------------
solveASubProblem(subProblem){
       if subProblem has only one blank
              solveAWilsonCase(subProblem)
       else
              solveMultiBlankProblem(subProblem)
}
//---------------------------------------------------
solveAWilsonCase(subProblem) {
       solvingCycles = cycleCalc(subProblem)
       if graph of subProblem is a G2
              solveAG2(subProblem,solvingCycles)
       else {
              standardCycle=getStandardCycle(subProblem)
              foreach cycle in solvingCycles
                     conjugateAndExecute(subProblem,cycle,standardCycle)
       }
}
```

Listing 3.1: Pseudo procedure overview

For the four procedures defined in this listing we describe their main functionality now in brevity; a detailed description is discussed in the upcoming subsections.

We start by the procedure `solveAPebbleProblem` which we call for any pebble motion problem $\Pi = (G, P, a, z)$`Problem`. We there decompose the graph G of `Problem` into a set of connected subgraphs by $conG()$ and create a pebble motion problem `conProblem` for each connected graph, which are solved by calling `solveAConProblem` on each of them.

In `solveAConProblem` we first call `rearrangePebbles(conProblem)` so that the same vertices become occupied by a as in the goal configuration then we

1. either sort out some problems with certain special graphs which require different solving algorithms which are then called (`solveASpecialProb`)

2. or we try try to dissect the connected graph of `conProblem` further into subProblems and iteratively solving them by calling `solveASubProblem`.

In `solveASubProblem`

1. in case there is only 1 blank in the subProblem we call the solving procedure `solveAWilsonCase`

2. otherwise we call the solving procedure `solveMultiBlankProblem`

In `solveAWilsonCase` we calculate 3-cycles `solvingCycles` to solve the problem first; then depending if the graph belongs to a special class or not, we either execute these by a calling shortcut function `solveaG2` or call a procedure `getStandardCycle` which results in a known 3-cycle move sequence `standardCycle` that will be used to execute all `solvingCycles` by conjugatng each of them into it by `conjugateAndExecute`.

3.2.2 Procedure `solveAPebbleProblem`

Let `Problem` be a pebble motion problem $\Pi = (G, P, a, z)$with a graph $G = (V, E)$.

If G is a disconnected graph the pebbles are confined to the vertices that are connected to each *connected subgraph of G*. This does not change anything on the solvability of the problem, as either the problem was solvable before the separation of G into a set of connected subgraphs $conG(G)$ or not.

We thus separate a $\Pi = (G, P, a, z)$into a set with $c = |conG(G)|$ of connected subproblems $\Pi^C = \{\Pi_1, \ldots \Pi_c\}$ with the respective connected graphs G_i and pebble sub-configurations a_i, z_i assigned to each $\Pi_i \in \Pi^C$ and proceed by solving each Π_i with the procedure for connected graphs. A possible implementation algorithm for this decomposition is described in **??**.

If all subproblems Π_i were solvable this results in a respective move sequence M_i which combined result in the overal solution move sequence M for Π. If not all subproblems Π_i were solvable, Π was overall not solvable.

3.2.3 Procedure `solveAConProblem`

Let `Problem` be a pebble motion problem $\Pi = (G, P, a, z)$with a connected graph $G = (V, E)$. Let c denote the current pebble configuration at any time.

Same Vertices Occupied As in Goal Configuration

We execute `rearrangePebbles` on the connected graph problem Π in order to overcome cases where $V^a_+ \neq V^z_+$. This step may not only for solve trivial problems, but $V^a_+ = V^z_+$ is also a necessary precondition for later procedures.

For this step we define a short-hand terminology

Definition 10 (Occupy states of vertices) *Given a pebble motion problem* $\Pi = (G, P, a, z)$*that is in the solving process and thus has the current pebble configuration* c*.*

Let us denote a vertex $v \in V$ *as beeing in the* correct state *iff*

- $v \in V^c_+$ *and* $v \in V^z_+$
- or $v \in V^c_-$ *and* $v \in V^z_-$

otherwise as beeing in the wrong state*.*

Let us denote another vertex $w \in V$ *as beeing in the* prefered state for v *iff*

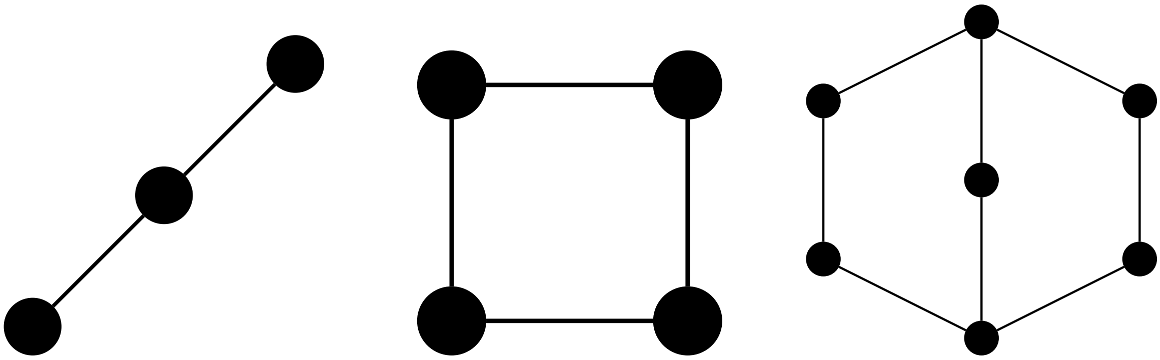
- $w \in V^c_+$ *and* $v \in V^z_+$
- or $w \in V^c_-$ *and* $v \in V^z_-$

The algorithm is informally described by Kornhauser on page 13ff: We create a minimal spanning tree (MST) G_{MST} out of the graph as temporary data structure with the Kruskal algorithm [5] and put all vertices in a queue Q, which constantly rearranges all remaining vertices depending on their valence (i.e. if take out from the queue we first get the leafs vertices, before we get the inner vertices).

We take a vertex v out of queue Q. If v is in the correct state, we prune v out of the G_{MST} by $removeVertex(G_{MST}, v)$ and take the next vertex out of Q.

If v is in the wrong state we search on the graph G_{MST} a simple path p to another vertex w that has the prefered state for v with least distance [4].

[4] No explicit search algorithm is indicated by Kornhauser, but it is obvious that a breadth-first-search (BFS) is to be prefered.



(a) A chain graph (b) A polygon or T_1 (c) A Θ_0 graph

Figure 3.3: Examples of graphs that need a special solution handling

1. If $v \in V_-^c$ we push the pebble mapped to w to v by *execMoveSeq*(*generatePebbleMS*(p), c)
2. If $v \in V_+^c$ we push the blank mapped to w to v by *execMoveSeq*(*generateBlankMS*(p^{-1}), c)

Afterwards v will be in the correct state. We prune v out of the G_{MST} by *removeVertex*(G_{MST}, v) and take the next vertex out of Q.

We proceed until the queue Q is empty.

Details on a possible implementation are given in **??**

Sorting Out Special Problems

There are certain special graphs which require a different solving procedure than the rest, namely

- fully occupied graphs
- chain graphs
- Polygons
- Θ_0 [5]

Also see figure 3.3. We will discuss them now one by one.

Fully Occupied Graphs A $\Pi = (G, P, a, z)$has a fully occupied graph if $|P| = |V|$. The problem is unsolvable if the pebble configurations $a \neq z$. Otherwise the problem was solved from the beginning.

Chain Graphs A $G = (V, E)$is denoted as *chain graph* if it satisfies $\exists x, y \in V, x \neq y, val(y) = val(x) = 1, \forall v \neq x \neq y \in V, val(v) = 2$. The most simple chain is the trivial, biconnected component T_0. The pebbles are confined to line of vertices and can't exchange their order. If a pebble motion problem $\Pi = (G, P, a, z)$has a chain graph the problem was either already solved by the procedure `rearrangePebbles` or the problem Π is not solvable.

Polygons If in pebble motion problem $\Pi = (G, P, a, z)G$ is a polygon, i.e. a T_1 graph, the pebbles are confined to the line of vertices, similar to the chain case. The problem Π is solvable iff the current pebble configuration c is a cyclical rearrangement of the goal configuration z.

[5] this graph was denoted as T_0 by Kornhauser, Miller and Spirakis, which may confuse with the trivial biconnected graph T_0; we thus denote it as Θ_0

Θ_0 graph A $G = (V, E)$is denoted as Θ_0 if it is the T_2 graph shown in figure 3.3. This biconnected graph does not contain any 3-cycle. As will explained Θ_0 is not suitable for the solution via procedure `solveAWilsonCase`. Given the relative low number of vertices it is easiest to calculate a lookup table of all possible pebble configurations C_Π on Θ_0 along with connecting moves to reach a goal configuration z.

Creating Subpuzzles

Let $B = |V_-^c|$ denote the number of blanks for Π. Given B we see that $p \in P$ is often confined to a certain part of G.

Similar to a disconnected graph we can divide the connected graph G into a set $\{G_1, \ldots, G_b\}$ of so-called *maximal biconnected components* by using an algorithm described in 1973 by Hopcroft and Tarjan [1], where each G_i in the set is a biconnected graph. We denote this algorithm by the function $biconG(G)$ which gives us said set.

Given B and $biconG(G)$ we are going to create a set of subproblems $\Pi^S = \{\Pi_1, \ldots \Pi_s\}$ to be solved separately, similar to what we did before for disconnected graphs.

The exact procedure is described in Kornhauser's thesis [3] on page 28ff and will not be redescribed in this work, as we only mentionned it for completeness. As pointed out in 1.2 we are most interested in explaining the solutions of pebble motion problems of biconnected graphs with $B = 1$.

We remain by saying that after this step we will execute the procedure `solveASubProblem` for each $\Pi_i \in \Pi^S$.

3.2.4 Procedure `solveASubProblem`

In case there is only one blank in a resulting subProblem, G must be a biconnected graph. Thus we execute the solving procedure `solveAWilsonCase`. Else we execute the solving procedure `solveMultiBlankProblem` for which we will not go into further detail. The procedure is similar to the *Push and Swap* algorithm by Luna and Bekris [6] [7].

3.2.5 Procedure `solveAWilsonCase`

We are now at the point on which we want to put the emphasis on 1.2: biconnected graphs with 1 blank.

Decomposition of Configuration Permutation into Three-Cycles

Recall from the definition of *Generation of permutation groups* that the alternating group A_n is created as a product of 3-cycles and any permutation $\in A_n$ can be easily decomposed.

If $V_+^c = V_+^z$ then there is a permutation $Y_{c,z}$ which maps c to z. So if $Y_{c,z} \in A_n$ we can decompose $Y_{c,z}$ into a number of 3-cycles; if $Y_{c,z}$ is odd we would have to change the current configuration $c \mapsto d$ by applying an odd permutation so we get $Y_{d,z}$ as an even permutation. To apply an odd permutation to the pebble configuration while keeping the precondition $V_+^d = V_+^z$ is only possible if G is not-bipartite. Thus all bipartite graphs with an odd permutation in Y are not solvable.

We then decompose Y into a list of 3-cycles. See for a possible implementation.

Executing Calculated Three-Cycles

Recall the important insight given in the definition of *conjugation*:

> **Given any permutation group P_n of which is known that it is k-transitive and that it contains a k-cycle S, then P_n contains all k-cycles.**

So if G is 3-transitive we then only need to obtain a move sequence for one known 3-cycle in G to execute all 3-cycles in it; i.e. the ones from the list which we previously calculated.

Acquiring a Move Sequence for a Standard Three-cycle The standard Kornhauser solving assumes that the blank vertex in the goal configuration $v \in V^z_-$ has $val(v) \geq 3$. We discuss in 4.3.2 how to overcome this assumption, by temporarily moving the blank to such a vertex and adapting the goal configuration for this algorithm.

For all T_2 graphs the move sequence for one standard 3-cycle s is easily obtained by cycling the pebbles on the 2 loops in a certain manner [6], except for the special Θ_0 graph which we already handled previously. As we can add a handle to any T_2 graph to create a T_3 graph of it and so on, we can also reduce any T_X to a T_2 graph by removing handles, while avoiding the creation of Θ_0 , thereby having an inductive procedure for acquiring s on all T_X graphs.

Acquiring Three-Transitivity Furthermore it turns out that allmost all graphs in T_2 are easily seen to be 3-transitive if one of their handles consists of more than 4 vertices: We just turn the required pebbles into it correspondingly. There are two graphs in T_2 where all 3 handles have less than 4 vertices:

- The T_2 graph consisting of 5 vertices (we call it an element of the biconnected graph subset G_2, which we handle differently, discussed below)
- The T_2 graph consisting of 4 vertices, but which does not need 3-transitivity as there are only 3 pebbles mapped to it and it is non-bipartite, which implies that we need at maximum one 3-cycle which can be directly executed.

All other biconnected graphs $T_X, X > 2$, can be shown to be at least 3-transitive.

The Solving Procedure In order to calculate a solution move sequence for a biconnected graphs with 1 blank that is not Θ_0 and 3-transitive on vertices $h = (a, b, c)$ we

1. get the permutation for the current pebble configuration
2. if the permutation is odd and G is non-bipartite we apply an odd permutation on the pebble configuration and get the new even permutation
3. we decompose the permutation in a product list of 3-cycles L
4. we acquire the move sequence for the standard 3-cycle s which is easy for all T_2 graphs; if the $betti(G) > 2$ we can acquire the standard 3-cycle as well by "virtually reducing handles" until we have a T_2 subgraph
5. we execute each 3-cycle of $l \in L$ by l moving into h, then to s, then "execute" s, move the pebbles again h and from there to l ("moving" via conjugation

Procedure `solveAG2`

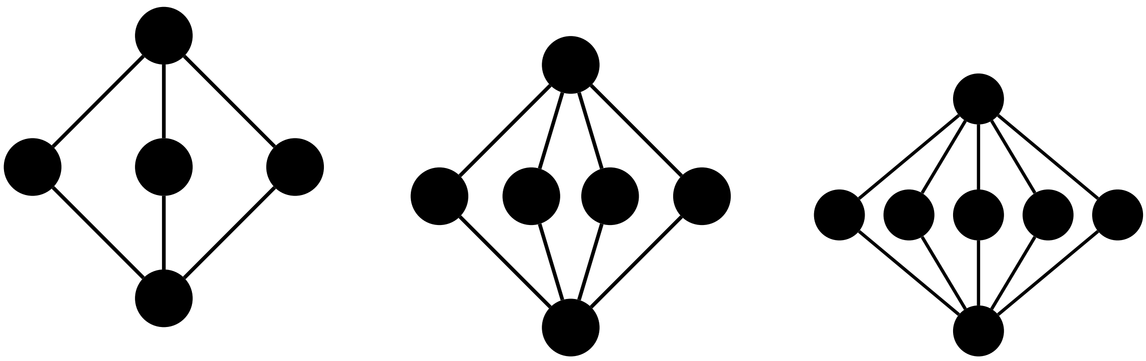
This is a slight variation which we propose from the procedure as described by Kornhauser, as we assume that this generates shorter move sequences in general than the standard procedure.

Let us define the class of G_2 graphs.

Definition 11 (G2 graphs) *A biconnected graph $G = (V, E)$belongs to the subset of G_2 iff $\forall v \in V val(v) = 2$, except 2 distinct vertices x and y with $val(x) = val(y) = 3$ and $adj(x) = adj(y)$.*

If we assume that vertex vx is blank, we can immediately see that it is easy to execute any 3-cycle $(v_1 \quad v_2 \quad v_3)$ on a $G = (V, E) \in G_2$ without . Either vertex vy is part of the 3-cycle (the trivial case) or else we use the simple move sequence $\langle M \rangle = \langle vx, v_3, vy, v_2, vx, v_2, vy, v_1, vx, v_2, vy, v_3 \rangle$ that is equivalent to the 3-cycle we aimed to achieve. An implementation is found at 4.3.5.

[6] in the upcoming chapter we do not provide any pseudocode for this step, as all the cases can be easily implemented taking the descriptions of Kornhauser's master's thesis on page 19-21 [3]



(a) The simplest G_2 with 3 internal vertices

(b) G_2 4 internal vertices

(c) G_2 5 internal vertices

Figure 3.4: Examples of G_2 on which permutation cycles can be directly executed

Chapter 4

Code fragments

This chapter introduces some code fragments for clarification. Throughout this chapter pseudocode in Java-style is used.

4.1 Basic Data Structures

We assume there are already data structures for sets, arrays, queues and common primitive data types like integers, floats and strings present. Furthermore we refer to Edge, Vertex, Pebble, Permutation, KCycle and Move as if they were implemented as classes.

If an Object is said to have assigned the value *NULL*, it is implied that this represents either an invalid state or a special state.

4.1.1 Graph Representation

An undirected graph is then represented by a class called Connections whose signature is as follows

```
class Connections{
        //class fields
        public Edge{} edges;
        public Vertex{} vertices;
        //class constructor
        public Connections(Integer verticeCount);
        //class methods
        public Edge getEdge(Vertex v1, Vertex v2);
        public Vertex[] getVertices(Edge e);
        public Boolean areConnected(Vertex v1, Vertex v2);
        public void addConnection(Vertex v1, Vertex v2);
        public void removeConnection(Vertex v1, Vertex v2);
        public Vertex[] getNeighboursOf(Vertex j);
        public Integer valence(Vertex j);
        public Vertex[] findSimplePath(Vertex v1, Vertex v2);
        public Vertex[] findSimplePath(Vertex v1, Vertex v2, Vertex{}
            forbiddenVertices);
}
```

Listing 4.1: Class Connections

4.1.2 Pebble Problem Representation

A pebble configuration can be represented by using an array of Pebble having the length of the verticeCount where there is either a Pebble or a *NULL* value. Together with the class Connections we present the method signature of the class Graph

```
class Graph {
	//class fields
	public Pebbles[] pebbles;
	public Connections graph;
	//class constructor
	public Graph(Integer verticeCount);
	//class methods
	public Boolean vertexShouldBeOccupied(Vertex v);
	public Boolean vertexIsEmptyAt(Vertex v);
	public Boolean vertexIsCorrectlyOccupied(Vertex v);
}
```

Listing 4.2: Class PebbleGraph

4.2 Algorithms For All Problem Cases

In this section we present algorithms that are applicable for all problem cases.

4.2.1 Calculate Connected Graphs Separately

This is a code fragment for calculating disconnected graphs as in the procedure `solveAPebbleProblem`.

If we assume that the graph may not be fully connected we start by calculating fully connected graph components and solving them separately. We use a common approach by putting all vertices in a list

```
Graph[] findConnectedGraphs(Graph graph){
	PebbleGraph[] disconnectedGraphs=[];
	Vertex{} remainingV = graph.vertices;
	while(remainingV is not empty){
		Vertex v = remainingV.first();
		PebbleGraph newG = new PebbleGraph(graph.vertices.length);
		findConnectedGraphFromAVertex(graph, v, newG);
		remainingV.removeAll(newG.vertices);
		disconnectedGraphs.add(newG);
	}
	return disconnectedGraphs;
}

Graph findConnectedGraphFromAVertex(Graph graph, Vertex v, Graph newG){
	Vertex{} openV = {v};
	do {
		v = openVertices.first();
		open.remove(v);
		closed.add(v);
		newG.pebbles[v]=graph.pebbles[v];
		Vertex{} neighbours = graph.getNeighboursOf(v);
	for (Vertex w : neighbours){
			newG.graph.addConnection(v, w);
		}
		neighbours.removeAll(closed);
		open.addAll(neighbours);
	} while (open is not empty);
	return g;
}
```

Listing 4.3: calculating connected graphs from disconnected one

4.2.2 Calculating Arms and Handles

It turns out to be useful to calculate a list of all arms and handles of a connected graph G once, respectively classify G as polygon or chain just after it's creation by the algorithm above.

If a graph $G = (V, E)$is neither a chain, nor a polygon, each edge $e \in E$ is part either part of a handle or an arm of G.

A possible algorithm to do this all at once is by putting all existing edges $e \in E$ in a queue Q.

1. We initalize an empty set x of edges. We take the first edge $e = s$ out of Q, add e to x, get one of both vertices $v \in e$. We start building a simple path p by putting v into it.
2. If $val(v) = 2$ we continue to "walk into this direction" by updating the current edge e and vertice v accordingly and adding v to p.
   (a) If we find that the current edge e is the edge we started with, we have found G beeing a polygon and stop the execution.
   (b) If $val(v) < 2$ we may have found either a part of an arm of G or G beeing a chain. If $val(v) > 2$ we may have found either a part of a handle or arm of G. In both cases we "walk into the other direction" by assigning assign $p = p^{-1}$, edge $e = s$ and vertex v with the other vertice of e (i.e. the one $\notin p$.
3. We then continue similar to step (2) until we find $val(v) \neq 2$. For brevity let us denote the first and last vertex in p with a and b.
   (a) If $val(a) = val(b) = 1$ we have found G to be a chain and stop the execution.
   (b) If $val(a) \neq 1 \neq val(b)$ we have found p to be a handle of G
   (c) Else we have found p to be an arm of G
4. If we found an arm or a handle we add it to a respective lists and remove all edges in x from Q. We do this until Q is empty.

```
findHandles(Graph graph) {
        Handles[] = [];
        Arms   [] = [];
        Edge{} unusedEdges = E of G;
        while (unusedEdges not empty){
                Edge  startE=unusedConnections.next();
                Edge  currentE=startE;
                Edge{} visitedE={};
                Path  path[] = [];
                walkIntoDirection(0);
                while (valence(currentV)==2){
                        if (currentE==startE)
                                exit(G is a polygon);
                        walkIntoDirection(0);
                 }
                 path = path.reverse();
                 currentE=startE;
                 walkIntoDirection(1);
                 while (valence(currentV)==2)
                        walkIntoDirection(1);
                 if (valence(currentV))==1)
                        if (valence(path[0])==1)
                                exit(G is a chain);
                        else
                                Arms.add(path);
                 else
```

```
            if (valence(path[0])==1)
                Arms.add(path);
            else
                Handles.add(path);
        unusedEdges.removeAll(visitedE);
    }
}
walkIntoDirection(Integer i){
    Vertex currentV = getVertices(currentE)[i];
    path.add(currentV);
    visitedE.add(currentE);
    if (getEdge(adj(currentV)[0],currentV)==currentE)
        currentE = getEdge(adj(currentV)[1],currentV)
    else
        currentE = getEdge(adj(currentV)[0],currentV)
}
```

Listing 4.4: algorithm to find arms and handles of a graph

4.2.3 Moving Pebbles into the Goal Configuration

We introduce the signature of a class LeafQueue which is filled with a minimum spanning tree (MST) created via the original graph via the Kruskal algorithm. From the queue we then sequentially prune each leaf vertex out of the MST as discussed for procedure `rearrangePebbles` from the previous chapter.

```
class LeafQueue{
    //class constructor
    public LeafQueue(Connections tree);
    //class methods
    public Vertex popNextLeaf();
}
```

Listing 4.5: Class LeafQueue

The following listing is describing the actual procedure. The Kruskal algorithm for generating the MST is not explicitly listed, but assumed to be present as library call.

```
Move[] rearrange(Graph problem) {
    Move[] solutionMoves=[];
    Connections mst=KruskalAlgorithm.getMST(problem.graph);
    LeafQueue verticeQ = new LeafQueue(mst);
    while (verticeQ is not empty) {
        Vertex nextV = verticeQ.popNextLeaf();
        if (not graph.vertexIsCorrectlyOccupied(v))
            solutionMoves.add(fixVertex(problem, mst, v));
        for (Vertex w:mst.getNeighboursOf(v))
            mst.removeConnection(w, v);
    }
    return solutionMoves;
}

Move[] fixVertex(Graph problem, Connections mst, Vertex v) {
    Move[] partialSolution=[];
    Boolean lookFor = problem.vertexShouldBeOccupied(v);
    Integer vc = problem.graph.countVertices;
    Vertex{} openV = {};
    Vertex[] predecessors = new Vertex[vc];
```

```
        predecessors.set(v, v);
        do {
                Vertex{} neighbours = mst.getNeighboursOf(v);
                neighbours.removeAll(predecessors);
                for(Vertex w:neighbours)
                        if (predecessors[w]==null)
                                predecessors[w]=vertex
                openV.addAll(neighbours);
                v = openV.pop();
        } while (problem.vertexIsEmptyAt(v) == lookFor);
        Vertex previousV=predecessors[v];
        do {
                if (lookFor)
                        partialSolution.add(new Move(v,previousV);
                else
                        partialSolution.add(new Move(previousV,v);
                Vertex tmp = previousV;
                v = previousV;
                previousV = predecessors[tmp];
        } while (previousV is not equal to v);
        return partialSolution;
}
```

Listing 4.6: arranging pebbles to end position

4.3 Algorithms for Biconnected Graphs with One Blank

In this chapter we discuss algorithms that are useful in case we need to solve a non-trivial biconnected graph with only 1 blank.

4.3.1 Fixing Odd Permutations on Non-Bipartite Graphs

This listing shows pseudocode for how to fix an odd permutation on non-bipartite graphs. We first check the parity of the currently induced permutation by the pebble configuration. If it is odd we can then acquire an odd cycle for non-bipartite graphs; else we result with *NULL* and the problem is not solvable. We then show how to apply the odd cycle to get into a new pebble configuration where an even permutation is induced.

```
Move[] assertEvenPermutation (Graph graph)
        if not permutationIsEven(graph.pebbles) {
                Vertex[] oddCycle = getOddCycle(graph.connections);
                if (oddCycle==NULL)
                        exit(odd Permutation on bipartite graph);
                else
                        return fixToEvenPermutation(graph,oddCycle)
        }
}

boolean permutationIsEven(Pebbles[] permutation){
        Integer parity = 0;
        Integer l= permutation.length;
        Boolean[] v = [];
        int j=l-1;
        while(j>=0){
                v.add(false);
```

```
                j--;
        }
        j=l-1;
        while(j>=0) {
                if(v[j]){
                parity++;
            } else if (permutation[j]!=NULL) {
                int x = j;
                do {
                                x = this.get(x);
                                v.set(x,true);
                } while (x!=j);
                }
            j--;
        }
        return (p%2==0);
}

Vertex[] getOddCycle(Connections graph){
        Integer l = graph.Vertex.length
        Boolean[] colour = new Boolean[l];
        Boolean[] visited = new Boolean[l];
        Vertex[] path   =  new Vertex[l];
    Vertex[] cycle  = NULL;
    for (Vertex v : G.Vertex)
                if (!visited[v])
                        dfs(graph, v);
        return cycle;
}

void dfs(Connections G, Vertex v) {
        visited[v] = true;
        for (Vertex w : G.getNeighboursOf(v)) {
                if (cycle not empty)
                        return;
                if (!visited[w]) {
                        path[w] = v;
                        colour[w] = !colour[v];
                        dfs(G, w);
        } else if (colour[w] == colour[v]) {
                        for (Vertex x = v; x != w; x = path[x])
                                cycle.add(x);
                        cycle.add(w);
                }
        }
}

Move[] fixToEvenPermutation(Graph graph, Vertex[] oddCycle) {
        Move[] m = [];
        Vertex blank = graph.pebbles.indexOf(NULL);
        Vertex exchange = oddCycle[0];
        Vertex[] exchangePath = graph.connections.findSimplePath(blank, exchange);
        m.add(exchangePath);
        oddCycle.needAtBeginning(blank);
        m.add(oddCycle);
        m.add(exchangePath.reverse());
```

```
        return m;
}
```

Listing 4.7: fixing odd permutations

4.3.2 Having the blank at a vertex with $val(v) > 2$

This step is not explicitly described in the Kornhauser thesis citeKHT, but imminent in order to acquire a solution as the blank should be residing at a vertex v with $val(v) > 2$.

If this is not the case we look at which vertex v the blank is residing and apply a apply a breadth-first-search to find the closest vertex w with $val(w) > 2$, resulting in a path p from v to w.

We then push the blank mapped at v to w by $execMoveSeq(generateBlankMS(p),)$ and furthermore modifiyng the goal configuration z accordingly to z_b, so that we can execute the standard algorithm.

Afterwards we reverse the operations just described by $execMoveSeq(generateBlankMS(p^{-1}),)$ and re-modifiy the goal configuration from z_b back to z accordingly.

4.3.3 Calculating Three-Cycles

This is an implementation the decompositon of an even permutation into a list of 3-cycles.

```
KCycle[] cycleCalc(Graph graph) {
        Permutation p = new Permutation(graph.pebbles);
        KCycle[] solution=[];
        Vertex{} availableV=graph.getNonEmptyVertices();
        Permutation p2 = Permutation.identity(p.length);
        for (int steps = 0; steps < p.length - 2; steps++) {
                int oldIndex = p2.indexOf(p[steps]);
                availableV.remove(steps);
                if (oldIndex == steps) {
                        continue;
                }
                int n = availableV \ {oldindex,steps};
            KCycle s = [n,steps,oldIndex];
            p2.apply3cycle(s.reverse());
            solution.add(s);
        }
        return solution.reverse();
}
```

Listing 4.8: calculating 3-cycles out of a permutations

4.3.4 Executing a Three-cycle by Conjugation into the Standard Cycle

This fragment shows how to execute any 3-cycle $(v_1 \quad v_2 \quad v_3)$ by conjugation, given a handle in the graph that provides 3-transitivity.

```
//for simplicity we assume that handles have been ordered so that the blank is at
    the bottom

Graph graph = problemToSolve.graph;
Vertex[] handleToUse = graph.connections.getLongesHandle();
blank = graph.pebbles.indexOf(NULL);
KCycle pushDown = pushPebbleDownPositions(localGraph);
Move[] pushDownSequence = addSequenceFromCycleNotation(blank, pushDown);

```

```
Move[] conjugateIntoCycle(KCycle cycleToAchieve, KCycle standardCycle, Move[]
    standardCycleMoveSequence, ){
        Move[] m = [];
        Move[] standard = getMovesIntoHandle(standardCycle);
        Move[] cycle = getMovesIntoHandle(cycleToAchieve);
        m.add(cycle);
        m.add(standard.reversed());
        m.add(standardCycleMoveSequence);
        m.add(standard);
        m.add(cycle.reversed());
        return m;
}

Move[] getMovesIntoHandle(KCycle verticePositionsToMove){
        Graph localgraph = graph.clone();
        Move[] m = [];
        Pebbles[] pebblesToMove = [];
        for (Integer i = verticePositionsToMove.size() - 1; i >= 0; i--) {
                Integer pos = verticePositionsToMove[i];
                Pebble p = localGraph.pebbles[pos];
                pebblesToMove.add(p);
        }

        Vertex to = handleToUse[0];

        for (Pebble pebble : pebblesToMove) {
                if (not first pebble) {
                        localGraph.executeMoveSequence(pushDownSequence);
                        m.add(pushDownSequence);
                }
                pebble = pebblesToMove.get(i);
                Vertex from = localGraph.pebbles.indexOf(pebble);

                if (to == from) //if already at correct position
                        continue;

                // searching a move sequence from->to not passing handleToUse
                if (handleToUse.contains(from))
                        from = movePebbleOut(localGraph, verticePositionsToMove,
                            handleToUse, from, m);

                Vertex{} forbiddenVertices.add(handleToUse);
                MoveSequence todo = getMoveCycle(localGraph, from, to,
                    forbiddenVertices);

                while (localGraph.pebbles.indexOf(pebble) != to) {
                        localGraph.executeMoveSequence(todo);
                        m.add(todo);
                }
        }
        m;
}

KCycle pushPebbleDownPositions(Connections graph, Vertex[] handleToUse) {
                Move[] pushDownSequence = getMoveSequenceFromCycleNotation(blank,
                    pushDown);
```

```
            Vertex from = handleToUse.get(0);
            Vertex to = handleToUse.get(1);
            Pebble pebble = graph.pebbles[from];
            return getMoveCyclePositions(graph, from, to, blank);
}

KCycle getMoveCyclePositions(Connections graph, Vertex from, Vertex to, VerticeSet forbiddenVertices) {
      forbiddenVertices.remove(to);
      forbiddenVertices.remove(from);
      forbiddenVertices.add(blank);
      VerticeList path1 = graph.findSimplePath(from, to, forbiddenVertices);

      forbiddenVertices.add(path1);
      forbiddenVertices.remove(to);
      forbiddenVertices.remove(blank);
      VerticeList path2 = graph.findSimplePath(to, blank, forbiddenVertices);

      forbiddenVertices.add(path2);
      forbiddenVertices.remove(blank);
      forbiddenVertices.remove(from);
      VerticeList path3 = graph.findSimplePath(blank, from, forbiddenVertices);

      KCycle cycleX = new KCycle();
      for (int j = 1 ; j < path3.size(); j++)
            cycleX.add(path3[j]);
      for (int j = 1; j< path1.size(); j++)
            cycleX.add(path1[j]);
      for (int j = 1; j< path2.size() - 1; j++)
            cycleX.add(path2[j]);

      return cycleX;
}

protected Move[] getMoveCycle(Connections g, Vertex from, Vertex to, VerticeSet forbiddenVertices) {
      KCycle cycleX = getMoveCyclePositions(g, from, to, blank, forbiddenVertices);
      return addSequenceFromCycleNotation(blank, cycleX);
}

Integer movePebbleOut(KCycle verticePositionsToMove,Vertex from, Move[] toAdd){
            Move[] sMoveOut = [];
            Pebble pebble=graph.getPebbleAt(from);

            Move[] handleTurn=pushDownSequence;
            Integer j = handleToUse.length-handleToUse.indexOf(from)-1;
            for (Integer i=0;i<j;i++){
                  graph.executeMoveSequence(handleTurn);
                  toAdd.add(handleTurn);
            }
            Vertex to=graph.pebbles.indexOf(pebble);

            //create reversal move sequence for reversing the handle turn later on
            MoveSequence handleTurnReversals=toAdd.createReversedMoveSequence()
```

```
                  ;

                for (Vertex exchangeV : graph.connections.vertices){
                        if (!pushDown.contains(exchangeV) && exchangeV!=blank) {
                                break;
                Vertex{} vForbidden={};
                vForbidden.add(handleToUse);
                vForbidden.remove(blank);
                vForbidden.remove(handleToUse.get(0));
                Move[] exchange=getMoveCycle(graph,to,exchangeV,vForbidden);

                while(graph.pebbles[exchangeV]!=pebble){
                        graph.executeMoveSequence(exchange);
                        toAdd.add(exchange);
                }
                graph.executeMoveSequence(handleTurnReversals);
                toAdd.add(handleTurnReversals);

                return pNew;
        }
}
```

Listing 4.9: executing a cycle by conjugation into a standard cycle

4.3.5 Executing a Three-cycles directly on G_2 graphs

This snippet shows a slight variation by us, given a G_2 graph on how to execute any 3-cycle $(v_1 \quad v_2 \quad v_3)$ directly on it without conjugation.

```
Move[] findCycleMovesOnG2(KCycle cycleToAchieve, Graph graph) {
        Vertex blank = Pebbles.indexOf[NULL];
        Vertex inbetween = Graph.connections.getNeighboursOf(blank)[0];
        Vertex top = Graph.connections.getNeighboursOf(inbetween)[0];
        if (top == blank)
                top = graph.connections.getNeighboursOf(inbetween)[1];

        if (cycleToAchieve contains top) {
                KCycle clockwise = {1, 2, 3};
                KCycle counterclockwise = {3, 2, 1};
                switch(p) {
                        case 0: cycleToAchieve.apply3cycle(clockwise); break;
                        case 1: break; //nothing to do
                        case 2: cycleToAchieve.apply3cycle(counterclockwise); break;
                }
                Move[] sequence = addSequenceFromCycleNotation(blank, cycle);
        } else {
                Move[] sequence = {
                                        blank, cycle[2], top, cycle[1],
                                        blank, cycle[1], top, cycle[0],
                                        blank, cycle[1], top, cycle[2]
                };
        }
        return sequence;
}
```

Listing 4.10: executing cycles on G2

Chapter 5

Discussion

In this paper we have presented an algorithm first proposed in 1984 by Kornhauser [3] which can solve pebble motion problems, an equivalent to sequential, centralized-planned MAPF problems in polynomial-time that produces sequences of moves with $O(v^3)$ and also provided pseudocode with an emphasis placed on the pebble motion problem cases of biconnected graphs with 1 blank vertex and provided a further proposal to reduce the number of solution moves required in cases of G_2 graphs.

We further created an implementation in Java for the procedure as described by Kornhauser to quite a large extend which we soon want to publish freely available for the public.

Due to the time constraints for this thesis we were unable to test out the performance of our implementation to the full extend by experimentation; it is our intend to rectify this as soon as possible.

Given the quite tractable procedure by Kornhauser for generating sequential moves solutions, we may suggest that future research in MAPF could focus on how to translate sequential move sequences into parallel move sequences.

Acknowledgements

This project is my master thesis in Computer Science at the Artificial Intelligence (AI) Group, Department of Mathematics and Computer Science, University of Basel.

During that time I was supported by Gabriele Röger a senior researcher at the group, as well as by Prof. Malte Helmert, lead of the AI group. I would like to express explicitely my gratitude to both of them.

Both of them gave me constantly helpful inputs concerning possible pathways and dead-ends and motivated me to put my focus on certain aspects of the work. Without them I might not have come close to finishing this project.

Furthermore I'd like to express thanks to my girlfriend Karin and some other of my relatives for further psychological support as I was carrying out this work.

Bibliography

[1] John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, June 1973.

[2] Mokhtar M. Khorshid, Robert C. Holte, and Nathan R. Sturtevant. A polynomial-time algorithm for non-optimal multi-agent pathfinding. In *SOCS'11*, pages –1–1. Symposium on Combinatorial Search, 2011.

[3] Daniel M. Kornhauser. Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1984. NOTE: Do not confuse with proceedings publication by the same name.

[4] Daniel M. Kornhauser, Gary L. Miller, and Paul Spirakis. Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In *Proceedings of the 25th Annual Symposium onFoundations of Computer Science, 1984*, SFCS '84, pages 241–250, Washington, DC, USA, 1984. IEEE Computer Society.

[5] Joseph B. Jr. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[6] Ryan Luna and Kostas E. Bekris. Efficient and complete centralized multi-robot path planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-11)*, San Francisco, CA, 25-30 Sept. 2011.

[7] Ryan Luna and Kostas E. Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In *International Joint Conferences in Artificial Intelligence (IJCAI-11)*, pages 294–300, Barcelona, Spain, 16-22 July 2011.

[8] Mike Peasgood, Christopher M. Clark, and John McPhee. A complete and scalable strategy for coordinating multiple robots within roadmaps. *Robotics, IEEE Transactions on*, 24(2):283 – 292, april 2008.

[9] Daniel Ratner and Manfred K. Warmuth. Finding a shortest solution for the n × n extension of the 15-puzzle is intractable. In *AAAI*, pages 168–172. American Association for Artificial Intelligence, 1986.

[10] Pavel Surynek. An application of pebble motion on graphs to abstract multi-robot path planning. In *Proceedings of the 2009 21st IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '09, pages 151–158, Washington, DC, USA, 2009. IEEE Computer Society.

[11] Pavel Surynek. A novel approach to path planning for multiple robots in bi-connected graphs. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 3613 – 3619, may 2009.

[12] Ko-Hsin C. Wang and Botea Adi. Mapp: a scalable multi-agent path planning algorithm with tractability and completeness guarantees. *Journal of Artificial Intelligence Research (JAIR)*, pages 55–90, 2011.

[13] Ko-Hsin C. Wang and Adi Botea. Tractable multi-agent path planning on grid maps. In *Proceedings of the 21st international joint conference on Artifical intelligence*, IJCAI'09, pages 1870–1875, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[14] Richard M. Wilson. Graph puzzles, homotopy, and the alternating group. *Journal of Combinatorial Theory, Series B*, 16(1):86 – 96, 1974.

Listings

List of Figures