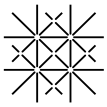

University of Basel

Single-Player Chess as a Planning Problem

Bachelor thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence Research Group
https://ai.dmi.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Augusto Blaas Corrêa

Ken Rotaris
ken.rotaris@stud.unibas.ch
2016-057-523

07.06.2022

Acknowledgments

On January 9th 2022, I first contacted Prof. Dr. Malte Helmert with the request to write my thesis with the Artificial Intelligence Research Group, and I'm grateful for not only the opportunity he gave me but also the generous topic suggestions, which included my wish to write a thesis in a chess-related topic. I also want to express my gratitude to my Supervisor, Augusto Blaas Corrêa, without whom this thesis would not have been possible and who always provided invaluable feedback, pointers, and suggestions in our scheduled weekly meetings and with prompt feedback via email. Lastly, I thank my family and friends, who have supported me in many ways.

Abstract

This thesis will look at Single-Player Chess as a planning domain using two approaches: one where we look at how we can encode the Single-Player Chess problem as a domain-independent (general-purpose AI) approach and one where we encode the problem as a domain-specific solver. Lastly, we will compare the two approaches by doing some experiments and comparing the results of the two approaches.

Both the domain-independent implementation and the domain-specific implementation differ from traditional chess engines because the task of the agent is not to find the best move for a given position and colour, but the agent's task is to check if a given chess problem has a solution or not. If the agent can find a solution, the given chess puzzle is valid.

The results of both approaches were measured in experiments, and we found out that the domain-independent implementation is too slow and that the domain-specific implementation, on the other hand, can solve the given puzzles reliably, but it has a memory bottleneck rooted in the search method that was used.

Table of Contents

1 Introduction

1.1 Running Example

Consider the planning problem presented in Figure 1.1. This puzzle is known as Guarini Problem and is one of the oldest chess puzzles, dating from 1512. The problem involves four knights which are positioned at the four corners of a 3x3 chessboard. The goal is to make minimal number of moves so that the two white knights end up on the squares where the two black knights are placed and vice versa. A knight can move by going two squares in either a horizontal or vertical direction, afterwards, it can turn 90 degrees left or right and move one more square forwards.

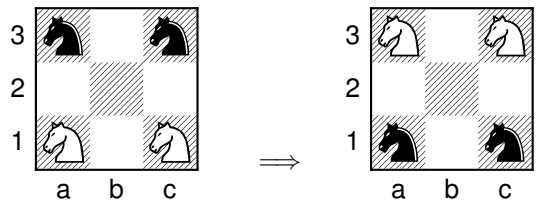

Figure 1.1: Guarini Problem: two white and two black, positioned at the four corners of a small 3×3 chessboard (Left: starting position, Right: goal position).

The Guarini Problem (Figure 1.1) can be abstracted into an undirected graph where the nodes represent the squares, and the edges represent the possible knight moves. We can also visualize the squares at which a knight currently resides with colours (black for black knights and green for white knights in Figure 1.2).

It may be hard to make sense of the graph on the left in Figure 1.2 but if we rearrange the nodes we can simplify the graph to look like the one on the right side in Figure 1.2. After the rearrangement all nodes are still connected in the same way as before; we simply moved the nodes positioned at the four corners to the opposite corner (exchange c1 and a3 and so forth). Notice that the node in the middle has no edges and could therefore be disregarded. Since two knights cannot occupy the same square, it is impossible to have two knights on the same node. Moreover, it implies that the knights cannot cross, which means every knight has to move in the same direction. We can conclude that it is only possible to

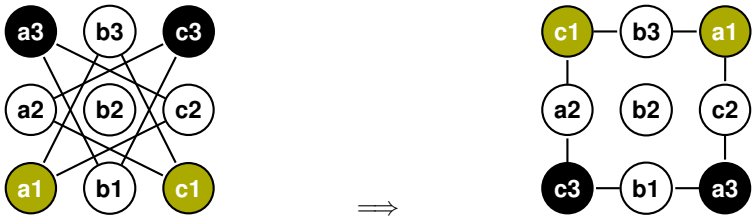

Figure 1.2: Guarini Problem as an undirected graph with black nodes representing the position of the black knights and green nodes representing the position of white knights).

solve the puzzle if the four knights rotate their position clockwise or anti-clockwise. Since we need four of these clockwise or anti-clockwise rotations to achieve the defined goal state, we end up needing a minimum of $4 * 4 = 16$ moves to solve the Guarini Puzzle [19, p.2].

Finally, this graphical solution has to be translated into a step by step plan representing the movements of the knights on the 3x3 chessboard. Fortunately, in this case the translation is straightforward, and an AI planning system could therefore generate a minimal plan for the Guarini Problem that looks as follows (counter-clockwise solution with 16 moves):

1st counter-clockwise rotation:	$a3 \rightarrow c2, a1 \rightarrow b3, c1 \rightarrow a2, c3 \rightarrow b1$
2nd counter-clockwise rotation:	$c2 \rightarrow a1, b3 \rightarrow c1, a2 \rightarrow c3, b1 \rightarrow a3$
3rd counter-clockwise rotation:	$a1 \rightarrow b3, c1 \rightarrow a2, c3 \rightarrow b1, a3 \rightarrow c2$
4th counter-clockwise rotation:	$b3 \rightarrow c1, a2 \rightarrow c3, b1 \rightarrow a3, c2 \rightarrow a1$

Many correct plans solve the Guarini Problem even if we impose the additional constraint of taking turns. There are infinitely many plans which solve the problem because, in any given position, there is no limit on how many times the knight can move back and forth. However, there are only a finite amount of solutions which have a length of only 16 moves. The objective of an AI planner is to find such a plan. It does not need to be the shortest solution, but the planner aims to get as close to the optimal solution as possible without taking too much time. From this we can conclude that the planner needs to be able to somehow balance performance and plan quality depending on the task at hand.

1.2 Overview

This thesis will look at Single-Player Chess as a planning problem. We define "Single-Player Chess" as a game of chess played by only one player who switches colours every turn. The player (also called an agent) has a defined start and goal position. The agent's objective is only to play legal chess moves to reach the given goal position. The thesis has two main parts: one where we look at how we can formulate the Single-Player Chess problem as a PDDL domain and use a domain-independent planner and the second part where we formulate the problem as a domain-specific solver. Lastly, we will compare the PDDL approach (domain-independent and, therefore general-purpose AI) with the domain-specific state-space[1] search algorithm by doing some experiments and comparing the results of the two approaches.

[1] A complete definition of what a state-space is and how it applies to the Guarini Problem can be found in Appendix A

1.3 Motivation

The general purpose PDDL planner approach and the domain-specific state-space search approach (implemented in Java) are designed to be used to check the validity of a chess puzzle at hand. The implementations differ from traditional chess engines such as Stockfish, where the agent plays against an opponent and has to find the best response to an opponent's last move. Our implementation does not have an opponent and does not need to come up with the single best move but instead needs to find out if the given goal position is reachable with a finite number of moves and provide an action plan. Our implementation can therefore be used to determine if a given chess puzzle is a valid puzzle or not.

With it is inherent complexity, chess is also an excellent candidate to test the limits of a general-purpose PDDL planner. This thesis aims to find out if a PDDL planner can handle the complexity of chess and how it compares to a domain-specific implementation. We want to find out where things fail and where the bottlenecks are.

2
Background

2.1 What is planning

Planning is an abstract and explicit reasoning process which consists of choosing and organizing the actions that can achieve a given objective before executing any of them [7, p.25]. When a computational agent performs a planning task, it is selecting which actions to take in order to attain a specific goal [8, p.18].

Classical AI Planning Systems are model-based. They take a description (or model) of the initial situation (or state), the actions available to manipulate the states, and the goal condition. The output of a planning system is a plan that describes the actions which transform the initial state into the goal state.

We can differentiate between two phases during planning: the encoding of the problem where the model is formulated within some language and decoding that language into a solution that involves a search algorithm.

There are many types of planning, but we will only consider classical planning tasks[2] which require a set of restrictive assumptions: The environment must be discrete, deterministic, static and fully observable [8, p.25].

2.2 PDDL

PDDL (Planning Domain Definition Language) is a formal language designed to represent/encode planning models. The PDDL planning formalism is a language which enables us to talk about state-spaces (planning tasks) with a compact encoding of the model at hand. This compact encoding is critical when dealing with a large state-spaces such as chess and is achieved by formulating the model in predicate logic [15][10, p.12]. The decoding of the PDDL file happens through the use of a domain-independent planning system. This planner is a generic tool which is independent of the problem at hand. [1, p.344 & p.367]

[2] A finite-domain planning task $\mathcal{P}$ is formalized as a state transition system and is defined in Appendix B.

2.2.1 Formal definition of PDDL

We define a PDDL planning task in a finite-domain representation as a 4-tuple in Equation 2.1. This definition assumes that all logical formulas are expressed in a first-order language $\mathcal{L}$, which contains a sufficient amount of constant symbols (objects in PDDL terminology), relation symbols (predicates), and variable symbols [13].

$$\mathbf{\Pi} = \langle \boldsymbol{\chi_0}, \boldsymbol{\chi_\star}, \boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{O}} \rangle \tag{2.1}$$

where the components are defined as follows:

- $\boldsymbol{\chi_0}$ is the initial state, which is a finite set of atomic variables. Every atom that is true in the initial state must be explicitly mentioned. All non mentioned atoms are assumed to be false.
- $\boldsymbol{\chi_\star}$ is the goal state which is captured by a closed formula called the goal formula. Like the initial state, the goal state can also consist of atoms, but in addition to those, the goal state can be defined in terms of closed formulas called axioms.
- $\boldsymbol{\mathcal{A}}$ is a finite set of PDDL axioms (more on axioms in Section 2.2.2).
- $\boldsymbol{\mathcal{O}}$ is a finite set of PDDL operators which define how a state of the state-space can be inspected and manipulated. We mainly are concerned with operators such as negation, conjunction, disjunction, implication, and equivalence, which, if used together, can build complex expressions. [13]

2.2.2 Predicates and Axioms

There are built-in binary predicates (such as '='), and there are user-defined predicates. We list the facts that are true in $\boldsymbol{\chi_0}$ and the conjunction of predicates that must be true for $\boldsymbol{\chi_\star}$ to be achieved, respectively. These facts are lists of ground facts, which means that all predicate parameters must be instantiated with objects of the problem [10, p.21].

Predicate variables can also optionally have a type, and if no type is given, they are assumed to be of type '`object`'. The word 'object' is reserved and denotes the top-level type, which contains all objects [10, p.28]. Different types allow certain objects to replace only those variables in a predicate with the same type, meaning we can create subtype predicates. Types are used to restrict what objects can form the parameters of an action [9]. Let us take a look at how predicates are defined.

Listing 2.1: Defining a predicate in PDDL

```
1 (:predicates
2     (<predicate_name> <a_1> ... <a_n> [- <t_1>] {<A_1> ... <A_m> - <t_2>})
3 )
```

Where a_1, ..., a_n represent arguments and t represents the optional type. If the argument is typed (meaning it has been defined to have a type), then the type declaration is mandatory. If more than one typed argument is used, then the argument(s) can be appended after the previous type(s) (as indicated with the curly brackets in Listing 2.1).

We can have three types of predicates, namely 'static predicates', 'fluent/normal predicates' and 'derived predicates'. Static predicates do not change their value after they are defined in χ_0. Fluent predicates occurring in χ_0 or simple effects of operators in $\mathcal{O}$ are called fluent predicates. They can change their value via the '`:effect`'-section if an action is applicable. The derived predicates are the most interesting ones because they allow us to make our PDDL implementation independent of the chessboard size. A derived predicate which is in the head of an axiom in $\mathcal{A}$ is called a derived predicate. Derived predicates derive their value due to evaluating a logical expression over other predicates so instead of having a stored-value, derived predicates are computed anew whenever a state is generated. Axioms are logical formulas that assert relationships between propositions that happen within a situation (in contrast to action definitions, which establish relationships between successive situations). We can even define recursive derived predicates if we define an recursive base case which will be very useful for implementing Single-Player Chess in PDDL. [13][5][9][10]

2.2.3 Running Example in PDDL

PDDL domain and problem definitions are usually stored in two separate files with the extension ".pddl" but the PDDL language does not demand this (although many planners require it). Let us stay with our running example from Figure 1.1 and take a look at how we would encode this problem in PDDL.

2.2.3.1 The Guarini Problem PDDL domain file

A PDDL domain description is a description of the actions we can take to change the state-space (moving a knight). It also stores all information about the used predicates. Let us take a closer look at the complete PDDL domain file in Listing 2.2.

A domain.pddl file starts with '`define (domain Guarinis_domain)`' which is the specification of the domain's name. This is followed by the '`:requirements`'-section, which indicates to the planner which features of the PDDL language the domain is going to use. In our running example we are using '`:negative-preconditions`' ('`not`' negations) and '`:typing`'.

Since we used the '`:typing`' precondition we can add a '`:types`' section where two types are used; One '`- figure`' type to represent the type of a chess piece, the second type is '`- location`' which is used to describe the coordinates of the board where we encode both ranks and files as 'numbers' $1-N$ with the prefix 'n' ($n1-nN$). A square is identified by a tuple of two of these 'numbers'. We could also have encoded every square with its exact coordinate ('$a1$' instead of combination '$n1n1$', for example), but then possible moves of the pieces have to be listed explicitly, which makes our modelling much more rigid and less compact.

Following these, a list of all used predicates must be given under the '`:predicates`' section. The parameters (indicated by the prefix '`?`') are given in the form of variables, and if these variables are typed, the type is also specified ('`(at ?figure - figure ?file ?rank - location)`'). These predicates will be initialized as binary facts in the problem-file in Listing 2.3 (see subsection 2.2.3.2).

Finally, we can append all of our actions. An action can be defined with the line '`:action`

<action-name>'. The action has to define a list of (typed) '**:parameters**', the '**:precondition**'s (what has to be valid in order for the action to be applicable), and the '**:effect**' (what happens if the action is chosen). In the precondition subsection we are only allowed to use conjunctions and disjunctions of positive (add) or negative (delete) predicates. [17, p.46] [10, p.14]. We now take a look at Listing 2.2.

Listing 2.2: 'domain.pddl' file which models the Guarini Problem

```
1  (define
2      (domain Guarinis_domain) ;------------------------>Domain name
3      (:requirements :negative-preconditions :typing) ;->Requirements list
4      (:types figure location) ;------------------------>List of all types
5      (:predicates  ;-------------------->Predicates section (list of all predicates
           )
6          (at ?figure - figure ?file ?rank - location)
7          (diff_by_one ?file ?rank - location)
8          (diff_by_two ?file ?rank - location)
9          (empty_square ?file ?rank - location)
10     )
11     ;---------------------------------------------------->Actions are listed here:
12     (:action knight_move ;----------------------------->First action
13         :parameters ( ;-->Parameters subsection
14                      ?knight - figure
15                      ?from_file ?from_rank ?to_file ?to_rank - location)
16         :precondition (and ;------->Preconditions subsection
17            (empty_square ?to_file ?to_rank) ;---------->Given square is empty
18            (at ?knight ?from_file ?from_rank) ;-------->Knight isat position
19            (or ;---------------------------------------->Knight move logic...
20                (and  ;--------------------------------->two files, one row:
21                    (diff_by_Two ?from_file ?to_file) ;->File +/- 2
22                    (diff_by_One ?from_rank ?to_rank)) ;>Rank +/- 1
23                (and  ;--------------------------------->two rows, one file:
24                    (diff_by_Two ?from_rank ?to_rank) ;->Rank +/- 2
25                    (diff_by_One ?from_file ?to_file))));File +/- 1
26         :effect (and ;----------->Effect subsection (Effect of the applied action)
27                      (not (at ?knight ?from_file ?from_rank))
28                      (at ?knight ?to_file ?to_rank)
29                      (not(empty_square ?to_file ?to_rank))
30                      (empty_square ?from_file ?from_rank))
31     )
32 )
```

All possible states in the '`Guarinis_domain`' (3x3 chessboard) and the transitions between these states are caused by the '**:action** `knight_move`'. The preconditions '`(empty_square ?to_file ?to_rank)`' and '`(at ?knight ?from_file ?from_rank)`' ensure that the knights' destination file is empty and that there actually is a knight at the origin square. These terms will be initiated in the problem file in Listing 2.3. The same is true for the predicates affected by the '**:effect**'s section. The line '`(`**not** `(at ?knight ?from_file ?from_rank))`' sets the '`at`' predicate to it is negated value (not operator), and line 28 ensures that the '`at`' fluent predicates' fact which corresponds to the new square is set to true for that knight. This models the knight moving from one square to another. [10, p.20]

A similar process happens in lines 29 and 30 for the empty squares. The interesting part is the precondition for the movement itself (lines 19-25). The eight squares a knight can move to can be simplified by breaking the knight movement down to the underlying logic: the knight can either move two files '**and**' one row (lines 20-22) '**or**' two rows '**and**' one file (lines 23-25).

2.2.3.2 The Guarini Problem PDDL problem file

The problem description entails χ_0 and $\chi_\star$ as well as the actual variables to be replaced in the action descriptions [17, p.46]. We can take a look at the complete problem file in Listing 2.3.

Listing 2.3: 'problem.pddl' file which models the Guarini Problem

```
1  (define
2      (problem Guarinis_problem) ;->Problem name
3      (:domain Guarinis_domain) ;-->Reference to domain file
4      (:objects ;------------------->Object section
5          n1 n2 n3 - location ;---->Example usage: 'n1 n1' pair = 'a1' square
6          b_knight_1 b_knight_2 w_knight_1 w_knight_2 - figure
7      )
8      (:init ;---------------------->Initialization section (Start state)
9          (at b_knight_1 n1 n3) ;-->Knight starting positions...
10         (at b_knight_2 n3 n3)
11         (at w_knight_1 n1 n1)
12         (at w_knight_2 n3 n1)
13         (diff_by_one n1 n2) ;---->Difference by one...
14         (diff_by_one n2 n1)
15         (diff_by_one n2 n3)
16         (diff_by_one n3 n2)
17         (diff_by_two n1 n3)
18         (diff_by_two n3 n1)
19         (empty_square n1 n2) ;--->Empty squares...
20         (empty_square n2 n1)
21         (empty_square n2 n2)
22         (empty_square n2 n3)
23         (empty_square n3 n2)
24     )
25     (:goal ;---------------------->Goal state section
26         (and
27         (at w_knight_2 n1 n3) ;-->Knight goal positions...
28         (at w_knight_1 n3 n3)
29         (at b_knight_2 n1 n1)
30         (at b_knight_1 n3 n1))))
```

The problem description (in Listing 2.3) generates all objects (lines 5 and 6) and lists all facts that are true in χ_0 (lines 9-23) and facts that must be true in $\chi_\star$ (lines 27-30). Notice that in the '`:goal`'-section there is an '`and`' present which is not needed in the '`:init` '-section. All not enumerated combinations of the '`at`' predicate are set to false by default. The tuple '$n1n3$' corresponds to the 'a3' square in our logic because we defined the 'at' predicate as '`(at ?figure - figure ?file ?rank - location)`' and refer to the '`?file` and '`?rank` variables in a consistent way throughout the code. '`n1 n3`' are number replacements because the naming conventions in PDDL require the first character to be an alphabetic character. To make these number replacements behave like numbers, we need lines 13-23, where we define the difference between two of these 'numbers' as a predicate. This is why we can say '`(diff_by_Two ?from_file ?to_file)`' in the domain file to express a file movement by two squares.

The generated valid output plan when we give both the domain and problem file to a PDDL planner looks as follows:

Listing 2.4: Output plan-file for the PDDL modeled Guarini Problem

```
(knight_move w_knight_2 n3 n1 n1 n2) ; This valid output solution can now finally
(knight_move w_knight_2 n1 n2 n3 n3) ; be translated into the following readable
(knight_move w_knight_2 n3 n3 n2 n1) ; action plan:
(knight_move w_knight_2 n2 n1 n1 n3)
(knight_move w_knight_1 n1 n1 n2 n3) ; wN1 a1b3, bN1 a3c2, bN2 c3b1, bN1 c2a1,
(knight_move w_knight_1 n2 n3 n3 n1) ; bN2 b1a3, wN2 c1a2, wN2 a2c3, bN2 a3c2,
(knight_move w_knight_1 n3 n1 n1 n2) ; wN1 b3c1, bN1 a1b3, wN1 c1a2, bN1 b3c1,
(knight_move w_knight_1 n1 n2 n3 n3) ; wN2 c3b1, wN1 a2c3, wN2 b1a3, bN2 c2a1
(knight_move b_knight_2 n3 n3 n1 n2)
(knight_move b_knight_2 n1 n2 n3 n1) ; where 'bN' means 'black knight' and 'wN'
(knight_move b_knight_2 n3 n1 n2 n3) ; means 'white knight'. The numeric value
(knight_move b_knight_2 n2 n3 n1 n1) ; following this prefix (either 1 or 2)
(knight_move b_knight_1 n1 n3 n2 n1) ; corresponds to the index of the knight.
(knight_move b_knight_1 n2 n1 n3 n3)
(knight_move b_knight_1 n3 n3 n1 n2)
(knight_move b_knight_1 n1 n2 n3 n1) ; cost = 16 (unit cost)
```

2.3 State-Space Search algorithm

By searching forward from the initial state, many planning algorithms try to construct a sequence of actions to reach the goal state. We will briefly explore one of these forward state-space search algorithms called 'breadth-first search' since it was implemented as a first approach to test the Single-Player Chess problem implementation.

2.3.1 Breadth first search

Breadth-first search is an uninformed search strategy which can be used if all actions have the same cost (uniform cost function) and if the root is expanded first. Both requirements are fulfilled by the rules of chess. Breadth-first search starts by expanding all the successors of the root node. Afterwards, all the successors of those successors are expanded and so on. This process is a systematic search strategy, and therefore it is complete (even on an infinite state-space) meaning it will find a solution if one exists. The algorithm is also cost-optimal because when it is generating nodes at depth d, all the nodes at depth $d-1$ have already been generated, so if there had been a solution with fewer actions, the algorithm would have found it.

Chess has an average branching factor of 35 [18] which is very large. This is a problem due to the generated nodes remaining in memory, so both the time and space complexity are $\mathcal{O}(b^d)$, where b is the branching factor and d is the depth level where the solution was found [1, p.76].

2.3.1.1 Best-First Search

Russell and Norvig put it like this: "In general, exponential-complexity search problems cannot be solved by an uninformed search for any but the smallest instances." [1, p.77] and this is why we will now look at an informed search algorithm. This informed Best-First Search algorithm is presented as pseudo-code in Listing 2.5. This algorithm helps us decide

which node from the set of nodes that have been reached but not yet expanded (this set is called 'frontier' or 'open list') we have to expand in each iteration. The algorithm thereby chooses the node with a minimum '`f()`' value out of the frontier in each iteration. This node is returned if its state is a goal state (line 7) and otherwise expanded (line 8) with the '`EXPAND`' function to generate further child nodes. These new child nodes are added to the frontier, but only if the state has not been reached before. If the state can now be reached with a path with a lower cost than any previous path, the node is re-added (line 10). The algorithm has two return statements. One return statement is placed in line 7, which is triggered if the algorithm finds a goal state. The other return statement is located in line 13, which is triggered if the algorithm moves out of the while loop. That means its frontier was empty, which in turn means we never found a goal state and the algorithm found no solution. By using different functions for '`f()`', we get different algorithmic behaviours.

Listing 2.5: "The best-first search algorithm, and the function for expanding a node." (from the book 'Artificial Intelligence: A Modern Approach' by Russel)

```
1  function BEST-FIRST-SEARCH(problem,f()) ;returns a solution node or failure
2   node ← NODE(STATE=problem.INITIAL)
3   frontier ← a priority queue ordered by f(), with node as an element
4   reached ← a lookup table, with one entry with key problem.INITIAL and value
        node
5   while not IS-EMPTY(frontier) do
6    node ← POP(frontier)
7    if problem.IS-GOAL(node.STATE) then return node
8    for each child in EXPAND(problem, node) do
9     s ← child.STATE
10    if s is not in reached or child.PATH-COST<reached[s].PATH-COST then
11     reached[s] ← child
12     add child to frontier
13  return failure
14
15 function EXPAND(problem,node) ;yields nodes
16  s ← node.STATE
17  for each action in problem.ACTIONS(s) do
18   s' ← problem.RESULT(s,action)
19   cost ← node.PATH-COST+ problem.ACTION-COST(s,action,s')
20   yield NODE(STATE=s',PARENT=node,ACTION=action,PATH-COST=cost)
```

We have not yet defined what a '`node`' is. A node is a data structure to keep track of the search tree and contains four components:

- '`node.STATE`' which is the state the node is representing (in our case bitmaps)
- '`node.PARENT`', which is the parent node from which the current node was expanded.
- '`node.ACTION`', which is the action used on the parent's state to generate the current node (in our case we can represent the action with a string but we will see in chapter 4 that we use a bitmap for this containing the difference to the previous state).
- '`node.PATH-COST`', which is the total cost from the initial state up to the current node.

To represent the nodes, the most suitable data structure is a queue. A queue enables us to efficiently check if the frontier is empty, get the topmost node of the frontier and add new nodes to the frontier.

A lookup table (e.g. a hash table) can be used to store the reached states, where each key represents a state, and each value represents the node for that state. Because we keep track

of all the previously reached states. The algorithm is able to detect all redundant paths and only retain the best path to each state. The algorithm is complete for tree structures (which is the case with chess) and optimal (since we have uniform action costs) [1, p.73].

2.3.2 Instantiations of Best-first Search

The best-first search algorithm has three primary instantiations: 'greedy-best-first search', 'A* search' and 'weighted A* search' where the only difference is the evaluation function '`f()`' which comes up with a value by which the priority queue is ordered. Greedy-best-first search uses only the heuristic of a state to determine this value (h(n.state)) while A* search adds the path cost to come up with that value (g(n)+h(n.state)). Lastly, there is the weighted A* search instantiation which is a variation of A* with the only difference that we multiply the heuristic value with a freely choosable weight $w \in \mathbb{R}_0^+$ with w usually ≥ 1 [16, p.].

3

Single-Player Chess as a PDDL problem

3.1 Problem Encoding

This section will first look at the Single-Player Chess PDDL domain file and then go over the corresponding PDDL problem file.

3.1.1 Domain File

We already introduced a possible implementation of a domain-file to encode our running example in Listing 2.2 and we will now expand upon this implementation to understand the full implementation of the Single-Player Chess problem.

3.1.1.1 Types

The first thing we need to do after defining the head of the PDDL file is to declare the types. In the context of chess, three types make sense to consider: locations on the chessboard, colours and figures. In our PDDL model, we divide the types further. We can see the full used type declaration in Listing 3.1.

Listing 3.1: Types section of the Single-Player Chess domain file.

```
1 (:types  ;Notice: We did not use a colour type
2          figure location - object ;----------------------> board objects
3          pawn knight bishop rook queen king - figure ;--> piece types
```

To encode the board positions we will follow the example from Chapter 2 Listing 2.3 as seen in Listing 3.1 in line 2.

Next is the figure representation. There are six different pieces in chess (pawn, knight, bishop, rook, queen, and king), and each of these figures exists in 2 variations: the 'colours' black and white. The representation of these six pieces is straightforward. As it is implemented in Listing 3.1 in line 3, they are of type '`- figure`', which in turn is declared to be of the top-level type '`- object`' (line 2). This is used to limit the applicability of actions to distinct figures. We also precomputed these figure types as a static predicate in order for the type to be accessed efficiently. When checking if a figure is of type pawn, for example, we can simply use the static predicate '`(is_pawn ?pawn - pawn)`'.

Lastly, we need to encode the colours. The straightforward approach would be to declare a type '`- colour`', which is a subtype of the figure. There main reason why we did not use the type '`- colour`' was that we would have introduced a non-strict hierarchy. If we, for example, have a knight instantiation called '`w_knight1`' then is of type '`- figure`' and has the subtype '`- knight`'. Suppose this knight in addition, has colour type '`- white`', which is also a subtype of '`- figure`'. In that case, the knight has two super types on the same hierarchy layer (colour and knight), which is not a strict hierarchy and can cause problems when we want to check which type of object it is. The answer to this would be to declare the type '`- colour`' as a subtype of the type '`- knight`', but we will see why this is not necessary next.

Since we still need to be able to differentiate between black and white figures, we implemented static predicates called '`(is_white ?figure - figure)`' and '`(is_black ?figure - figure)`' and then instead of having to use an existing operator, we can use the same_color derived-predicate from Listing 3.2 to check if given pieces have the same colour or not. We can even further optimize and precompute this and implement the '`same_color`' as a static predicate to avoid repeated computation.

Listing 3.2: Derived predicate which checks if two given pieces have the same color

```
1 (:derived (same_color ?figure1 ?figure2 - figure)
2         (or (and(is_white ?figure1)
3                 (is_white ?figure2))
4             (and(is_black ?figure1)
5                 (is_black ?figure2))))
```

In a domain file, predicate declarations follow the type declarations. However, if we followed this ordering, this thesis would get dry and long, so instead it is more useful to pick one figure and explain how its movement is encoded and cherry-pick some non-intuitive predicates.

We will now have a look at how the rook movement is encoded. At the end of Section 3.1.1.2 we will look at how bishop- and queen movements follow the same pattern. We already saw how a knight movement could be encoded on a small example in Chapter 2, Listing 2.2. The king movement can be seen as a special case of the queen movement because the king can also move in all directions but only by one square with the additional restriction that he cannot move into check. The only piece missing will be the pawn and we will take a closer look at the pawn movement in Section 3.1.1.6.

3.1.1.2 The Rook

The rook can move over the whole board in the horizontal and vertical direction if no piece is blocking its movement. To encode this vertical and horizontal movement, we use recursive derived predicates to check whether a given square is vertically or horizontally reachable and use that as one of the preconditions to move a given rook. We now take a look at the '`vert_reachable`' recursive derived predicate in Listing 3.3.

The recursive derived predicate in Listing 3.3 checks the vertical movement. In vertical movements, the file stays the same (lines 3 and 12). Lines 2-8 represent the base case of

Listing 3.3: Vertical reachability derived predicate

```
1  (:derived (vert_reachable ?from_file ?from_rank ?to_file ?to_rank - location)
2          (or (and (vert_adj ?from_file ?from_rank ?to_file ?to_rank) ;-->base case
3                   (= ?from_file ?to_file)
4                   (or
5                     (empty_square ?to_file ?to_rank) ;empty
6                     (exists(?figure - figure) ;capturable
7                         (and(at ?figure ?to_file ?to_rank)
8                             (not(occupied_by_same_color ?figure ?from_file
                                  ?from_rank))))))
9              (exists(?next_rank - location) ;-->recursive case
10                  (and
11                      (empty_square ?to_file ?next_rank)
12                      (= ?from_file ?to_file) ;same file
13                      (vert_adj ?from_file ?from_rank ?to_file ?next_rank)
14                      (between ?from_rank ?next_rank ?to_rank)
15                      (vert_reachable ?from_file ?next_rank ?to_file ?to_rank)))))
```

the recursion, and lines 9-14 are the recursive case. The base case condition consists of a conjunction of two parts. One part (line 5) represents that a rook can move to a specified square if it is an empty square. The conjunction in lines 5-8 represents the case that there is a figure of the opposite colour on the destination square (meaning the rook can capture that piece).

We will now move to the recursive case in the derived predicate in Listing 3.3. This recursive part checks if there '**exists**' a next location (called '?next_rank') such that the square corresponding to that next location is fulfilling the conjunction of four preconditions which now follow. Since the base case of the recursion already considers whether or not a destination square is reachable (either an empty square or an occupied square with a capturable opponent figure placed on it), we only need to check if all the squares before that base case are empty in the recursive case (line 11). To make sure that every square between the origin square of the rook and the destination square is taken into account, we have the recursive derived 'between' predicate, which checks if the '?next_rank' is between the origin square and the destination square. The fourth and last condition is the recursion itself which simply recursively "calls" the derived predicate 'vert_reachable' with a new '?from_rank' which is now the '?next_rank'. This call ensures that all the above conditions are fulfilled for every square between the start and goal location of the rook movement.

We can now use the recursive derived predicate from Listing 3.3 together with its counterpart 'horiz_reachable' (which does the same but in the horizontal direction) to implement the rook movement shown Listing 3.4. Although, at first glance, the rook movement looks quite different from the knight movement in Listing 2.2, it still has quite some similarities. We are now using these two recursive derived 'reachability' predicates (Listing 3.4 lines 12-18) as preconditions for the rook movement action.

However, there are a few aspects still missing to explain the full rook movement action. Let us first look at how we can encode turn-taking since we want to alternate between white and black moves. This is implemented with a simple fluent predicate called 'white_s_turn', which is set to true or false depending on if it is white or black, which moves. We need to incorporate this turn-taking as a precondition for the rook's movement because, for example, white rooks should only be able to move if it is the whites' turn. We incorporate

this precondition with the derived predicate called 'my_turn' (Listing 3.4 line 10), which simply checks if the given chess figure is white and that it is a white turn (or if the given piece is black and it is a black turn). The colour of a piece is retrieved via the precomputed static predicates 'is_white' and 'is_black' mentioned in Section 3.1.1.1.

Listing 3.4: Rook movement in PDDL

```
1  (:action rook_move
2          :parameters (?rook - rook ?from_file ?from_rank ?to_file ?to_rank -
               location)
3          :precondition (and
4                                (valid_position) ;-->only continue if position is valid
5                                (at ?rook ?from_file ?from_rank)
6                                (or
7                                   (empty_square ?to_file ?to_rank) ;-->empty
8                                   (and ;-->capturable piece at destination
9                                       (not(empty_square ?to_file ?to_rank))
10                                      (not(occupied_by_same_color ?rook ?to_file
                                            ?to_rank))))
11                               (myturn ?rook) ;--> turn taking derived predicate
12                               (or
13                                   (and ;-->vertical movement
14                                       (= ?from_file ?to_file)
15                                       (vert_reachable ?from_file ?from_rank ?to_file
                                            ?to_rank))
16                                   (and ;-->horizontal movement
17                                       (= ?from_rank ?to_rank)
18                                       (horiz_reachable ?from_file ?from_rank ?to_file
                                            ?to_rank))))
19         :effect (and
20                        (not(at ?rook ?from_file ?from_rank))
21                        (forall (?figure - figure) ;-->check if we captured a figure
22                           (when
23                               (and;-->precondition of the when operator
24                                   (at ?figure ?to_file ?to_rank) ;piece at
                                        destination
25                                   (not_same_color ?rook ?figure) ;piece has opposite
                                        color
26                               )
27                               (and ;-->effect of the when operator
28                                   (not (at ?figure ?to_file ?to_rank))
29                                   (removed ?figure)
30                                   (not(is_on_board ?figure)))))
31                        (at ?rook ?to_file ?to_rank)
32                        (not(not_moved ?rook)) ;-->important for castling rights
33                        (last_piece_moved ?rook) ;-->important for en-passant
34                        (when (white_s_turn) ;-->taking turns
35                           (not(white_s_turn)))
36                        (when (not(white_s_turn))
37                           (white_s_turn))
38                        (empty_square ?from_file ?from_rank)
39                        (not(empty_square ?to_file ?to_rank))
40                        (not(valid_position)))) ;-->declare position as invalid (!!)
```

The only missing precondition that has not been mentioned so far is that the current state must be a valid position (line 4). We will see that only positions which are deemed valid are accepted as a goal state in Section 3.1.2.2. We also set the 'valid_position' fluent predicate to false after every action (line 40) except the action which checks if the current state is a valid state (see Section 3.1.1.3).

We will now look at the effect section of our rook move action. It has all the same effects as the knight effects in Listing 2.2, but there are a few additional effects such as updating the mentioned '`white_s_turn`' fluent predicate to its opposite value (lines 34-37). We also need to check if an opponent's piece has been captured by the rook movement and update the predicates related to that figure. This is done with a combination of the '**`forall`**'-operator and the '**`when`**'-operator and can be seen in Listing 3.4 in lines 21-30[3].

Lastly, the lines 32 and 33 are used for making en-passant (history needed) and castling moves possible (more on that later on).

We can modify this rook movement to also work for the bishop. What we need instead of the '`vert_reachable`' and '`horiz_reachable`' predicates is a derived predicate called '`diag_reachable`'. Since moving on the diagonal involves changing both the rank and the file, we must ensure that the bishop stays on the same diagonal. Otherwise, the bishop can move in zigzag like moves. To ensure this we have a recursive derived predicate called '`same_diag`' which checks if the given pair ⟨'`?next_file`', '`?next_rank`'⟩ is between the given origin square and destination square and therefore ensures that the '`?next_file`' variable is on the same diagonal. We can go one step further and precompute this derived predicate, which is what we did in our implementation.

Now that we understand the implementation of the rook movement, the king and queen movements are straightforward (with the exception of taking checks into consideration but we will talk more about this in Section 3.1.1.3). The king movement is implemented with a conjunction of the eight surrounding squares a king can move to with the restriction of squares, which puts the king in check (Section 3.1.1.3 will talk about this). The queen movement is a conjunction of the rook movement and the bishop movement.

The castling precondition is checked by tracking whether or not the king or rooks moved (with a fluent predicate). If the king and one of the rooks have not moved, the rook can reach the king (only empty squares in between), and no piece is targeting any of the squares from the origin location of the king up until the destination location of the king, then we allow castling. We will not go into more detail about this and encourage the reader to have a look at the code ('**`:action`** `castling`').

3.1.1.3 Red Zone

The red zone is used to determine if a given king of a given colour is either currently in check or will be placed in check if he moves to a specified position. To calculate the red zone, we check if there '**`exists`**' any opposite colored figure (compared to the '`?king`') which can reach the specified square. We need to check this for every figure and we do so within a disjunction.

[3] The reason we do not have different actions for capturing and moving a rook is that it is error-prone if we have the same actions in two different variations. Taking the final implementation and implementing movements and captures separately would be interesting to compare with the current implementation, but not something we focused on in this thesis, and the resulting PDDL domain file will have almost twice as many actions, resulting in an even longer file.

Listing 3.5: Red Zone PDDL code where only the rook attacking rays are checked

```
(:derived (red_zone ?king - king ?kt_file ?kt_rank - location)
  (or
    (exists(?rook - rook ?c_file ?c_rank - location)
        (and
            (is_on_board ?rook) ;optimization if there are no rooks present on the
                 board
            (at ?rook ?c_file ?c_rank)
            (not_same_color ?king ?rook)
            (or
                (and
                    (= ?kt_file ?c_file) ;vertical movement
                    (vert_reachable ?c_file ?c_rank ?kt_file  ?kt_rank)
                )
                (and
                    (= ?kt_rank ?c_rank) ;horizontal movement
                    (horiz_reachable ?c_file ?c_rank ?kt_file ?kt_rank))))
    );------->end of rook attacking ray calculation
    (...) ;-->here we place all other exists conditions (including the king)
    ))
```

The red zone derived predicate in Listing 3.5 consists of a disjunction of '**exists**'-operators. Only the possible destination squares (attacking rays) of the rook are presented in Listing 3.5, but we can append the movements for all other figures (including the king - since the kings cannot get closer than one square). The rook attacking rays are calculated by simulating the rook movement reachability predicate shown in Listing 3.3. There is, however, one special case which occurs if we do not slightly adjust the reachability predicate. In the position depicted in Figure 3.1, we coloured all squares that are marked by our current red zone implementation in red.

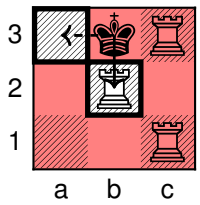

Figure 3.1: 3x3 chessboard position with only one legal move (capturing the rook on b2).

Notice that the rooks in Figure 3.1 on the squares $c1$ and $c3$ protect each other, including them in the red zone. However, The king on $b3$ protects the square on $a3$ because the current implementation does not differentiate between kings and normal pieces. This leaves the king free to walk onto the apparently protected (by his own 'shadow') $a3$ square, but this is an illegal move. There is only one legal move for the king to take in this position: capturing the rook on $b2$. What we can do to fix this is to copy our three reachable derived predicates ('`diag_reachable`', '`vert_reachable`' and '`horiz_reachable`') and add the appendix '`_red`' to indicate that they are used only to calculate the red zone. We then adjust these red zone predicates so that kings are ignored in the recursive case if they are of the same colour. What we get is a red zone derived reachability predicate that sees through kings of the opposite colour which is precisely what we need to fix the problem from Figure 3.1.

3.1.1.4 Check, Mate, Stalemate and Absolute Pins

A check occurs whenever the king is positioned in a red zone. A mate position occurs if two conditions hold: One, the king is in check and unable to escape this check because all reachable squares are red zones, and two, none of the same coloured pieces can move, meaning the check cannot be avoided by the defender by either capturing the attacker or blocking its attack with a piece. We will encode this subproblem such that we reach an unsolvable state after a mate or stalemate position is reached. This results in mating positions only being allowed as a goal state of a game but not as an intermediate step.

Similarly, a stalemate position is a position where the king is not in check, he must move (it is his turn and no other piece of the same colour with legal moves) but the king is unable to move to another square because all reachable squares are red zones. With a stalemate, there are not any actions that can happen so a stalemate is an unsolvable state.

An absolute pin happens when a defending piece cannot move without exposing the same coloured king to a check. The pinned piece can only move between the king and the attacker (including capturing the attacker). What we decided to do is to create a new fluent predicate called '`(valid_position)`' which is set to true in the starting position (in case the given starting position is also the goal position) and declared that the fluent predicate must be a true fact in the goal state. We also set the position to be '`(not (valid_position))`' after every action (Listing 3.4 in line 40). The result is a "locking mechanism" where after every action, the position is deemed an invalid position, and no further actions can be taken, nor can the newly created state be accepted as a goal position. After the position is "locked", we check if the given position is valid, and if it is, we set '`valid_position`' to true, and all actions are unlocked again. To unlock all actions, we need another action which has the precondition that the current state is invalid, and we can see this action in Listing 3.6.

Listing 3.6: Unlocking action which checks if the current state is a valid state or not.

```
1  (:action check_if_last_move_was_valid ;check same colored king
2      :parameters (?figure - figure ?from_file ?from_rank - location)
3      :precondition (and
4                          (not(valid_position))
5                          (at ?figure ?from_file ?from_rank)
6                          (myturn ?figure)
7                          (opposite_king_not_in_check ?figure)
8                     )
9      :effect (and
10                     (valid_position)
11             )
12 )
```

The action in Listing 3.6 checks the state of the board from the position of colour which is next to play (Listing 3.6 line 5 and 6). The precondition for a move to be invalid is that it is the turn of colour A and that the king of colour B is in check. If colour A moves a pinned rook out of the way and exposes his own king, for example, this is an invalid position, and since the current turn (which was switched after the action was applied) is colour B, the opposite king (king of colour A) is in check, and the position is deemed invalid since the '`check_if_last_move_was_valid`' cannot be applied. This precondition in Listing 3.6 (line 7) allows for mate and stalemate positions to be accepted as goal states since, in a mating

position, the king of the not-mated colour is not in check[4].

Because we chose an action to check if a position is valid or not, the size of the output plan will be increased. In the results of the experiments in Chapter 5 we filter out the unlocking actions. Also, the output plan only shows the user the chess moves in standard chess notation, leaving out the unlocking actions.

The question that arises is why we encoded this mechanism as an action and not just incorporate it into the effects section of every action. What we can try to do instead of the "locking mechanism" is to add a precondition where the same coloured king (as the moving colour) must not be in check. But if we do this, then we have the problem that an invalid action is detectable only after an invalid action has already been taken, and if the resulting invalid state is a goal state, then this invalid state can get accepted as a solution, meaning with this implementation we cannot prevent invalid states completely. We also tried to implement a valid move checking mechanism using a derived predicate, but we could not make it work. The reason why this derived predicate implementation failed was that the effect of an action can only be observed after it has been made, meaning we can only check if the last move was valid, but we cannot predict if the action we are about to take will be valid once it has been taken. We need a resulting state of action to be marked invalid somehow for the planner to be unable to accept the newly generated state. But if we try to accomplish this labelling by setting a label like '`valid_position`' to false after every action, then we are not able to set that label to true again since we do need an effect section of an action to change a state. We are not entirely convinced that it is impossible to implement the described validation of a state as a derived predicate, but we could not accomplish it.

3.1.1.5 Blocking Checks and Capturing Checking Pieces

We now take a look at Figure 3.2 where the only valid move for black is to block the attacking ray of the white rook on square $a1$ by moving its (black) rook to square $a2$.

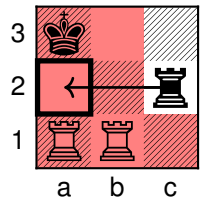

Figure 3.2: In this position it is blacks' turn and the only valid move for black is to move the rook from $c2$ to $a2$.

[4] The problem with this implementation is that we can no longer have positions where there is not at least one king and one opponent piece present on the board since the '`check_if_last_move_was_valid`' action only sets the predicate '`valid_position`' if the 'opposite_king_in_check' condition holds. If there is no opponent piece, then the precondition cannot hold and therefore, the '`valid_position`' predicate is never set to true, so the position is deemed unsolvable even though it could potentially be solvable. However, this is not a problem in classical chess positions since at least the opponent king must be present on the board in a valid chess position. With this in place, a valid classical chess starting position is actually enforced.

To solve this subproblem, what first comes to mind is implementing something similar to the red zone approach. We can check if there exists a figure which has an attacking ray aimed at the king, and if so, check if there exists a defending figure which can intercept that attacking ray by moving between its king and the attacking piece, effectively blocking the attack (the attacking ray of the white rook on square a1 in this example). The problem with this is that this is not a full solution because there is a particular case which would not be taken care of by this approach: If there are two attacking pieces aiming at the king. The defender can block one of those attacks and still be in check, meaning the blocking move would be illegal. The same is true if one of the attackers is captured by the defender. We'd need a counting function and a numeric predicate, but the planner we are using does not support functions. Luckily we do not actually need to worry about any of this because the valid move implementation described in Section 3.1.1.3 already handles this by declaring all child states of a parent position like the one depicted in Figure 3.2 as invalid states if the resulting state is not a state where the rook is blocking the attacking ray of the rook on square $a1$.

3.1.1.6 Pawn movements

The only missing movement we have not explored is the pawn movements. There are a total of eight actions that capture all the possible pawn moves, the actions are: '`pawn_promotion_queen`', '`pawn_promotion_rook`', '`pawn_promotion_bishop`', '`pawn_promotion_knight`', '`pawn_capture`', '`pawn_move_one`', '`pawn_move_two`' and '`en_passant`'.

The regular pawn move must ensure that the pawn only moves in one direction. Black rooks can only move by descending the rank number, and white pawns move by increasing the rank number. These two conditions are implemented with a pre-computed static predicate called '`plusOne`' (for white pawns) and '`minusOne`' (for black pawns). We can check what colour the pawn has and apply the correct precondition accordingly in the action '`pawn_move_one`'. Another precondition is that the white pawns cannot reach the last line on the board (because there the pawn will promote) and are also not allowed to be positioned at the first line of the board (the reverse is true for black pawns). We implement this by having another precondition which consists of the last and first rank on the board called '`(last_pawn_line ?to_file ?to_rank - location)`'.

For the pawn promotions, we have four almost identical actions, with the only difference being that another piece is positioned on the board depending on the action the planner chooses. We also need to differentiate between capturing into promotion and moving into promotion because if we capture into promotion, then we also need to remove that captured piece from the board as we do in all other movement actions (except regular pawn movements because those actions can not capture a piece which is also why there are so many different actions to model the pawn movements if compared to other movements).

Let us also have a brief look at how the en-passant action works. The preconditions for an en-passant move are that there exists an opposite-coloured pawn which made a double move in the last turn, and the target is positioned next to the attacking pawn (on either side horizontally). To find out if the piece moved in the previous move, we keep track of the last moved piece with a fluent predicate called '`last_piece_moved`', which is updated in the

effect section of every action. To check whether or not the pawn made a double move, we first ensure that every pawn can only make a double move once. If a pawn makes a double move with the action '`pawn_move_two`', then we update the fluent predicate '`(double_moved ?pawn - pawn ?file ?rank - location)`' for that pawn and can now simply check if a pawn made a double move. The main precondition for a double pawn move is that the pawn must be positioned at the second rank for white pawns and positioned at the second last rank for black pawns. We will not go into more details about pawn movements but instead encourage the reader to look at the implementation in the GitHub repository PDDL domain file.

3.1.2 Problem File

3.1.2.1 Preprocessing

From here, we could go two roads: one where we expect the user to populate the PDDL problem file by hand and the other road where this is done for the user. We've decided to go the second road and enable the user to provide the planner with only an initial board position and a goal board position given as FEN-codes[5] and they will be converted to a PDDL problem file. Also, the user can define any board size in the header of the file 'population_generator.py' and defining the matching FEN codes of the same board size. If the sizes do not match, an error handler is informing the user about it.

3.1.2.2 Goal State Encoding

There are mapping problems when we have figures that occur multiple times on a chessboard when we try to generate a precomputed goal state, and it is not trivial to map the instantiation label of a figure in the initial position correctly to a figure in the goal position.

Suppose we have two white bishops on the board which we name '`w_bishop_1`' and '`w_bishop_2`'. Now we define a start state where '`w_bishop_1`' is on square '`n1 n1`' and '`w_bishop_2`' is on the square '`n1 n2`'. Because we take a chessboard with unlabeled pieces as input (the initial state FEN code) and since we already labelled the starting state bishops with '1' and '2', the algorithm needs to take this into account and label the goal state pieces in accordance since bishops cannot change their square colours when they move. So, if the white black-squared bishop (white bishop positioned on a black square) '`w_bishop_1`' is labelled as the second bishop (which is white-squared) in the goal state, then the goal state will be unreachable. The same goes for all other figures except the king because he can only occur once.

[5] The Forsyth-Edwards Notation (FEN) is a standard notation used to describe the current chessboard position. In addition, the FEN code has optional parameters containing the active colour, castling rights, possible en-passant targets, the halfmove clock and the fullmove number. We do not expect these additional parameters as user inputs because the algorithm looks at the board position to determine if castling, for example, is possible. This requires the assumption that the given starting position has not moved the rook(s) nor the king(s) if the rook(s) or the king(s) are positioned in the starting position. One might imagine starting positions where a king moved one time back and forth which means he moved and is therefore not eligible for castling rights anymore but the algorithm will still deem the position to have castling rights because the position is evaluated so and there is no history taken into account. This would be easy to fix but the there were other priorities that needed attention before the deadline of this thesis.

We could solve this problem when it occurs with bishops by always labelling the white squared bishops with uneven numbers and the black squared bishops with even numbers. With this in place, we could avoid this problem to some extent, but this solution is not correct for two main reasons. One reason is that pawn promotions are not considered since they can produce multiple bishops on the chessboard. Another reason is that a figure which occurs numerous times on the chessboard is not guaranteed to be the same figure as in the starting position because they may exchange their positions.

If, for example, we have two rooks '`rook_b1`' and '`rook_b2`' in our initial position and we now need to label two rooks in the goal position, then we cannot know for sure which rook moved to what position if we do not have a plan yet. The planner can find a solution in this case because the rooks can exchange their positions by making a few extra moves[6]. But as soon as pawns do something other than moving forwards[7], we run into the same mapping problem again where the planner cannot find solutions to solvable problems.

The solution to this subproblem is to state that there is some figure in the goal state at the destination square, but we do not define its label. We do this by using the static predicate which defines the figure and colour type for every figure, as seen in Listing 3.7 (lines 5 and 6). To safely do this[8], we also need to specify which squares are empty in the goal position, which means we need to keep track of empty squares as we saw with the knight movement of the Guarini Problem in Chapter 2 in Listing 2.2.

Listing 3.7: PDDL derived predicate which lets us define a goal state without explicit mentioning of the figure labels.

```
1 (:derived (black_rook_at ?file ?rank - location)
2     (exists(?figure - figure)
3         (and
4             (at ?figure ?file ?rank)
5             (is_rook ?figure)
6             (is_black ?figure))))
```

3.2 Problem Decoding

The decoding of the PDDL problem file happens through the use of a domain-independent planning system. In this thesis, we will make use of state of the art domain-independent classical planning system called 'Fast Downward' [12] to decode the implemented PDDL input files.

3.3 Optimizations

There are some derived predicates which are precomputable and others which are not. The latter are not precomputed because either the number of instantiations is too large

[6] The planner will not find a solution in all cases. We can, for example, imagine positions where at least one of the rooks cannot move.

[7] If the pawns capture a piece, for example, then the pawn numbering (let us say left to right and top to bottom) falls apart, and the planner will not be able to find a solution.

[8] We could also use a static predicate that indicates if a figure has been removed and use it in the goal definition, but then we'd only move the mapping problem one step ahead since we still need to label the pieces which were removed, which is what we tried to avoid in the first place.

or because they really are not precomputable since they depend on the current state of the world. In addition to the already mentioned optimizations we can also precompute the remaining derived predicates which are: '`vert_adj`', '`horiz_adj`', '`diag_adj`', '`same_diag`', '`between`', '`plusOne_nTimes`' and '`minusOne_nTimes`'. These precomputations are implemented (including the ones mentioned in Section 3.1.2.1). However, making the changes resulted in some weird behaviour and the thesis deadline did not allow to find out the problem which is why we reverted these changes.

If we take a look at the output section of the Fast Downward planner [12] then we will realize that for any plan, the majority of time will be spent "computing negative axioms". We tried some alternatives to deal with it, but they also did not work out.

3.4 Statistics of the Domain File

We want to end this chapter on a statistical note so here are a few numbers about the implemented PDDL domain file:

Table 3.1: Statistics of the PDDL domain file ('#' stands for 'number of')

Name	**Amount**
#types	8
#actions	18
total #predicates	61
#static predicates	19
#fluent predicates	9
total #derived predicates	33
#recursive derived predicates	6
#non-recursive derived predicates	27

To get an idea of the difficulty of solving our PDDL Chess implementation, we compare it to the Blocks World domain [4]. The Blocks World problem is NP-hard and can be implemented with five fluent predicates (clear, on-table, arm-empty, holding, on) and four actions (pickup, putdown, stack, unstack)[9]. Our implementation of Single-Player Chess in contrast is using a total of 61 predicates out of which 29 are derived predicates and out of those six are recursive derived predicates.

Let us also have a look at the Sliding Tile Puzzle which is PSPACE-complete [11]. The Sliding Tile Puzzle can be implemented with six predicates (tile, position, at, blank, inc, dec where "inc" and "dec" encode addition and subtraction of positions) and four actions (move-up, move-down, move-left, move-right)[10].

Both the Blocks World domain and the Sliding Tile Puzzle are already hard for some planners so we expect the chess domain to be even more challenging for the planner.

[9] Mentioned Blocks World implementation: https://github.com/gerryai/PDDL4J/tree/master/pddl/blockworld
[10] Mentioned Sliding Tile Puzzle implementation: https://github.com/SoarGroup/Domains-Planning-Domain-Definition-Language/blob/master/pddl/slidetile.pddl

4

Single-Player Chess using a Domain Dependent State-Space Search Solver

In this chapter, we will have a look at the implementation of the Single-Player Chess Problem as a domain-dependent search algorithm.

4.1 Problem Encoding

We decided to go with the popular approach to encode the chessboard as bitboards (see Section 4.1.1) in the language environment of Java.

4.1.1 Bitboards

Bitboards are a type of data structure designed for the efficient encoding of game boards as sets of bits. In a bitboard, each cell is assigned a bit, indicating whether or not a piece is present, requiring only a fraction of the processing and memory power. With bitboards, common operations can be carried out using fast bitwise manipulations. Another advantage to bitboards is that bitwise operations can be applied to all board cells simultaneously. The cells of an 8×8 chessboard can be conveniently packed into a single 64-bit long integer. The drawback of this method is that the 8x8 board size is hard coded. [2]

4.1.1.1 Representing a Chessboard Position

Since we have twelve pieces (six black pieces and six white pieces), we need $12*64$ bits to represent a given state of a chessboard. We look at how we can encode the starting position of the Guarini Problem (left image in Figure 1.1). To do so, we use an imaginary[11] byte containing nine bits to represent the nine board squares. We need two of these (imaginary) 9-bit bytes since we have two types of figures (black knights and white knights). In Java, the bits are interpreted in big-endian[12], and we follow this logic. The top left square ($a3$) in our running example is represented with '000000001' and the bottom right square ($c1$)

[11] We could implement this by using a 32-bit integer and only caring about the first nine bits.
[12] With big-endian, the most significant byte of the data is placed at the byte with the lowest address.

is represented with '100000000'. The resulting byte array representing the state of the board will contain the two entries '000000101' (for black knights) and '101000000' (for white knights).

4.1.1.2 Encoding Nonsliding Piece Movements

The encoding of the movements needs to be reflected in the bitboards and is accomplished with bitwise operators '>>>' (signed shift right) and '<<' (shift left). We take a look at how we can encode the movements of a King in Figure 4.1.

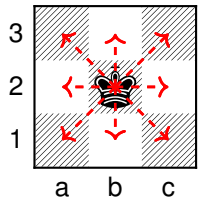

Figure 4.1: King movements in all eight directions (as a reference point)

First, we set a 9-bit byte to '000010000' to represent the king being in the middle of the board. To represent a movement, we use shift operators: the operation '000000001 << 1' results in the output '000000010' which amounts to a movement to the right on the chessboard (from square '$a3$' to '$b3$'). To move the king to the left by one square, we shift to the right with the operator '>>>'.

To accomplish an up and down movement of the king on the board, we shift the bitmap left and right by the board size. For the king to move to the '$b3$' square (upwards movement), all we need to do is shift the 9-bit byte by three to the left ('>>>'). To represent diagonal movement we have to add and subtract 1 depending on the move being diagonal or anti-diagonal. We can generalize movements to a $n * n$ board by multiplying the board size with the number of squares we want to advance our piece by (N) and shift the bitboard by $n * N$ but we need to be aware of board overflows since the bit is continuous and does not have inherent rows and ranks.

4.1.1.3 Encoding Sliding Piece Movements

Sliding pieces add a lot of complexity to the chess game because sliding piece attacks depend on other pieces, which may block the attack in a particular direction[13]. This additional complexity is also reflected in the amount of code necessary to compute them versus the amount necessary to compute nonsliding piece movements. The popular approach to generate sliding piece movements is to use the 'XOR-trick' [6], but we decided against this approach because many of the calculations still have to be done separately when we are using this trick and we found a way to consider everything at the same time. We used

[13] For the non-sliding pieces the legal capturing destinations are calculated straight forward and deemed as a red zone

two methods which calculate the sliding piece attacks and movements. While doing so, the method is gathering all information needed to calculate the red zones (which eliminate king movements), pinned pieces, where these pinned pieces are allowed to move to, protected pieces which the king is not able to capture, squares which the defender can block to get out of check and we can also count how many pieces are attacking the king[14].

Because a bitmap can contain multiple figures of the same type (there are two black rooks, for example), the function needs to first separate those figures into separate bitmaps for it to be able to loop over those isolated pieces. The index of the current isolated attacker piece is calculated by shifting the bit and counting how many times we had to change to the right in order for the bitmap to be equal to the long '1'. With that index, we can now calculate what the represented row and columns would be on a chessboard by using the formula: '$index/board\_size$' and cast it into an integer. To calculate the row, we use the formula '$index \pmod{board\_size}$'.

Now that we know the file and rank of an attacker piece, we can move the piece in every direction until it reaches the end of the board. We do this by simulating the piece's movement in all four directions. This is very efficient since the loops will, in total, only execute '$2 * (board\_size - 1)$' times (possibilities in both directions minus the square on which the attacker piece is currently positioned)'. We can make it even more efficient by checking for a collision with the same coloured pieces at the current square or checking for a collision with the opposite-coloured piece at the previous square. In both cases, we can break the loop prematurely because if we know that the current square to be checked has the same colour, we do not want to set the bit. Similarly, if we know the previous checked location (either $k - 1$ or $k + 1$ depending on the direction of the attacking ray) was an opposite-coloured piece, we already set the bit and now know we must stop since we cannot continue the piece movement. Otherwise, we can just continue shifting the attacker bitmap to the right and adding the resulting bitmap to the result bitmap (squares which can be attacked). This also saves us time since we do not need to do an additional removal of those unreachable squares.

To detect the red zones we mark all squares up to the location where we detect either a capturable opposite colored piece or one square before the same colored piece as a red zone where the king is not allowed to move to. We must also consider that the king is not allowed to block his own movement like we saw in Chapter 3 so what we do is initiate an inner loop which is activated if the current square is occupied by an opposite colored king and this inner loop marks all squares from the king position up to the edge of the board as additional red zones. These red zones are saved in an additional bitmap. Because we can differentiate between our own and the opponents' pieces, we can save the attacked pieces (checks) and the protected pieces (king cannot capture) in different bitmaps. This also saves us a lot of computation power when calculating the valid moves in a second step. The algorithm also takes into account that if more than one piece is attacking the same colored king, then the only valid move is a king move for that color. This means that the algorithm

[14] Counting the number of check giving pieces is relevant because whenever a king is in check, then the same coloured pieces can either block the attacking ray or capture the attacking piece but if multiple attackers check the king, then the only valid move is to move the king.

can collect all necessary information to calculate the valid moves directly while the piece movements are calculated. In a second step the algorithm filters out all non valid moves and we are left with only valid moves. The only thing that is left to do is to extract the individual movements from the bitmaps.

4.2 Problem Decoding

To be able to keep this section short, the code for the Best-First Search has been written in such a way that it resembles pseudocode from Listing 2.5 to a large extent, so we will not go over it here in detail but invite the reader to have a look at the code instead. A difference from the pseudocode in Listing 2.5 to our implementation is that we do not save the actions as a string but rather use the bitmap difference to the previous node. Calculating the actions is very expensive compared to the bitwise operations because we use strings to represent the movement ('O-O' for kingside castling for example). To save computation power, we decided to save the difference from the parent node to the current node with a bitmap representing that difference and when the algorithm has found a solution, we can simply invert the solution path and build the action strings representing the corresponding actions. This way we only calculate the string actions for the solution path.

We implemented the Breadth-first search algorithm, the Greedy-best-first search algorithm, the A* algorithm and the weighted A* algorithm.

4.2.1 Heuristic

As a heuristic, we count the number of pieces that are out of order compared to the goal state. We encoded two optimizations into the heuristic function where we return an infinite value in certain cases. If this infinity value is reached, the state is deemed to be unsolvable, and the search recognizes this by aborting the search and returning false. The infinite value is set as the heuristic value in the following cases:

1. If the number of pawns of a specific colour (whose turn it is) present in the current state is smaller than the number of missing pieces of that same colour, we deem this state unsolvable.
2. The second case where we assign an infinite value is if the pawn in the last row of colour c is further ahead than the pawn in the last row of the goal state, then we also deem the given state as unsolvable pawns cannot move backwards.

The way we implement the heuristic is also goal-aware since if the current state has no pieces that are out of order, the heuristic will return a value of 0.

Lastly, we will have a look if the heuristic we are using is consistent. To test this we need to show that the triangle inequality holds: $h(s) \leq cost(a) + h(s') \; \forall \; s \xrightarrow{a} s'$ where s and s' are states of the state space and a is an action. Since in our implementation we are dealing with uniform costs, we can simplify this equation to $h(s) \leq 1 + h(s')$ and rewrite this as $h(s) - h(s') \leq 1$. This means that it is not allowed for the heuristic value to drop by more than one by applying any action. Suppose we have a state s with two opposite coloured pieces on the board which is one state away from the (goal) state s' with $h(s') = 0$. To

reach state s' from s, white needs to capture a black piece. Our heuristic will estimate that the goal state is two steps away ($h(s) = 2$) because there are two pieces out of place (the uncaptured black piece and the white piece which is capturing that black piece). However, the goal state is just one move away meaning the heuristic value drops by more than one, violating the triange inequality. More generally: As we move from one state to another in the state-space, the heuristic will not monotonically decrease in value meaning the heuristic is inconsistent. This heuristic is also not admissible as it can overestimate the distance to the goal state. Since the heuristic is not admissible, the A* search algorithm does not guarantee optimality (although it is implemented) [1, p.88].

5 Experiments

In all following experiments, we set a time limit of 10 minutes and a memory limit of 8 gigabytes. The Computer on which this implementation was tested runs on Ubuntu 20.04, has 16GB of Memory and an Intel Core™ i7-7500U CPU 2.70GHz × 4 processor. The code for the PDDL model as well as for the domain dependent solver can be found on the GitHub repository of this thesis[15]. The PDDL model was solved using the Fast Downward planner [12]. The reported time in the PDDL experiments is measured from planner execution time until a plan is returned by the planner and does not involve any precomputations. In the domain dependant solver experiments the reported time is started right before the root node is generated and ends when a solution is returned. All plans which were generated in the following experiments are valid plans.

5.1 PDDL Implementation: Performance

We start from the classical chess starting position (FEN code: '$rnbqkbnr/pppppppp$/8/8/8/8/$PPPPPPPP/RNBQKBNR$') and the goal position is given with the FEN code '$rnbqkbnr/pppppppp$/8/8/4P3/8/$PPPP$1$PPP/RNBQKBNR$'. When we run the planner on this problem, the planner runs out of memory while computing "Computing negative axioms...". The memory limit was reached at the 4 minutes and 32 seconds mark.

Because our PDDL implementation can handle a board size of arbitrary dimensions, we will set the board size to be 4x4 in this next experiment and place the kings on the board. The FEN position for the initial state is 'k1K1/4/3r/RR2' and looks similar to the 3x3 position presented from Chapter 3 in Figure 3.2, except that there is another king on the board, and the position is stretched out to a 4x4 board (it is also black's turn to move). In Table 5.1 we continue from this situation by making a few moves and finally ending at a mating position (row 7). The first row in In Table 5.1 represents our first goal state. The instance in row i+1 is constructed to be one move 'further' than i. Therefore, with every row, the number of moves needed to reach the defined goal state increases. We are not taking a closer look at the exact output plans of every row, but we indicate the number of

[15] https://github.com/kentaris/Bachelor-Thesis_Single-Player-Chess

moves the output plan has in the column 'Moves' if a valid solution is found.

Table 5.1: PDDL planner results: initial position with kings on a 4x4 chessboard

Row	**Time**	**Moves**	**Goal State**
1	185s	1	k1K1/4/r3/RR2
2	176s	2	k2K/4/r3/RR2
3	176s	3	3K/k3/r3/RR2
4	176s	4	3K/k3/rR1/R3
5	175s	5	k2K/4/rR1/R3
6	175s	6	k2K/4/RR1/4
7	180s	12	3K/k3/RR1/4

If we have a look at the amount of 'Moves' and the row number then we can see that both match up until row 6 where the amount of moves required is 6. Row 6 is a mating position (black king is mated). We gave the planner one more move after this mating position because we expected it to return that there is no valid plan for this position but it turns out there is, it is just longer. Here the planner found a valid plan with 12 moves to reach this position.

In this next experiment we expand 4x4 chessboard used in experiment four to a 5x5 chessboard. We use a similar initial state as in our previous experiment (initial FEN: k2K1/5/5/4r/RR3) where the only valid move for black is to move the black rook on e5 to a2 to block the check (goal FEN: k2K1/5/5/r4/RR3). When running this experiment, the planner aborts due to the time limit being reached. The planner was "Computing negative axioms..." when the 600 seconds limit was reached and the planner aborted.

If we continue our trend of increasing our board size and we try our experiment on a 6x6 chessboard, we have the exact same situation where the planner is not able to find a solution within 10 minutes.

If we again increase our board size to be of size 7 (initial FEN: k2K3/7/7/7/7/6r/RR5 where the only legal move for black is to block the check with its rook: k2K3/7/7/7/7/r6/RR5). Instead of running out of time like in the previous experiment, the memory limit was reached after 81 seconds while the planner was "Computing negative axioms... ". When we ran the same experiment on a 8x8 board (Initial FEN: k2K4/8/8/8/8/8/7r/RR6, Goal FEN: k2K4/8/8/8/8/8/r7/RR6) the same the planner also runs out of memory (after 108 seconds and also while computing negative axioms).

We said in Chapter 3 in Section 3.4 that to find a solution for problems such as the Blocks World problem or the Sliding Tile problem is already quite hard for the solver and we also saw that our PDDL chess implementation uses a multiple of the predicates used in those two problems. The main complexity in our chess implementation comes from the 'red_zone' derived predicate. In this experiment, we show this by creating positions on an 8x8 chessboard where the 'red_zone' derived predicate is set to always return false. We are not using any kings to avoid invalid positions because once the 'red_zone' derived predicate

only returns false the king is free to move everywhere (including into checks) which makes room for invalid positions.

Table 5.2: PDDL planner results of states without kings

Row	**Time**	**Moves**	**Goal State**
1	111s	1	rnbq1bnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQ1BNR
2	109s	2	rnbq1bnr/ppppp1pp/8/5p2/4P3/8/PPPP1PPP/RNBQ1BNR
3	109s	3	rnbq1bnr/ppppp1pp/8/5P2/8/8/PPPP1PPP/RNBQ1BNR
4	119s	4	rnbq1b1r/ppppp1pp/5n2/5P2/8/8/PPPP1PPP/RNBQ1BNR
5	114s	5	rnbq1b1r/ppppp1pp/5n2/5P2/8/3P4/PPP2PPP/RNBQ1BNR
6	117s	6	r1bq1b1r/ppppp1pp/2n2n2/5P2/8/3P4/PPP2PPP/RNBQ1BNR
7	111s	7	r1bq1b1r/ppppp1pp/2n2n2/5P2/8/3P4/PPPB1PPP/RN1Q1BNR
8	125s	9	r1bq1b1r/ppp1p1pp/2n2n2/3p1P2/8/3P4/PPPB1PPP/RN1Q1BNR
9	114s	12	r1bq1b1r/ppp1p1pp/2n2n2/3p1P2/8/3P1Q2/PPPB1PPP/RN3BNR
10	115s	11	r1bq1b1r/ppp1p1pp/5n2/3p1P2/3n4/3P1Q2/PPPB1PPP/RN3BNR
11	116s	14	r1bq1b1r/ppp1p1pp/5n2/3Q1P2/3n4/3P4/PPPB1PPP/RN3BNR
12	117s	12	r1b2b1r/ppp1p1pp/5n2/3q1P2/3n4/3P4/PPPB1PPP/RN3BNR
13	115s	14	r1b2b1r/ppp1p1pp/5n2/3q1P2/3n4/1P1P4/P1PB1PPP/RN3BNR
14	129s	18	r1b2b1r/ppp1p1pp/5n2/4qP2/3n4/1P1P4/P1PB1PPP/RN3BNR
15	139s	26	r1b2b1r/ppp1p1pp/5n2/4qP2/3n4/1PPP4/P2B1PPP/RN3BNR
16	154s	24	r1b2b1r/ppp1p1pp/5n2/1n2qP2/8/1PPP4/P2B1PPP/RN3BNR
17	149s	26	r1b2b1r/ppp1p1pp/5n2/1n2qP2/8/1PPP1N2/P2B1PPP/RN3B1R
18	135s	35	r1b2b1r/ppp1p1pp/5n2/4qP2/8/1PnP1N2/P2B1PPP/RN3B1R
19	134s	35	r1b2b1r/ppp1p1pp/5n2/4qP2/8/1PNP1N2/P2B1PPP/R4B1R
20	144s	36	r1b2b1r/ppp1p1pp/5n2/5P2/8/1PqP1N2/P2B1PPP/R4B1R
21	194s	30	r1b2b1r/ppp1p1pp/5n2/5P2/8/1PBP1N2/P4PPP/R4B1R
22	216s	35	r4b1r/ppp1p1pp/5n2/5b2/8/1PBP1N2/P4PPP/R4B1R
23	325s	36	3r1b1r/ppp1p1pp/5n2/5b2/8/1PBP1N2/P4PPP/R4B1R
24	172s	33	3r1b1r/ppp1p1pp/5B2/5b2/8/1P1P1N2/P4PPP/R4B1R
25	153s	38	3r1b1r/ppp1p2p/5p2/5b2/8/1P1P1N2/P4PPP/R4B1R
26	281s	40	3r1b1r/ppp1p2p/5p2/5b2/7N/1P1P4/P4PPP/R4B1R
27	193s	46	5b1r/ppp1p2p/5p2/5b2/7N/1P1r4/P4PPP/R4B1R
28	163s	50	5b1r/ppp1p2p/5p2/5b2/7N/1P1B4/P4PPP/R6R
29	203s	44	5b1r/ppp1p2p/5p2/8/7N/1P1b4/P4PPP/R1R5
30	156s	50	7r/ppp1p2p/5p1b/8/7N/1P1b4/P4PPP/R1R5
31	timeout	-	7r/ppp1p2p/5p1b/8/7N/1P1b4/P4PPP/R2R4

We can see in Table 5.2 that the planner found solutions up to 50 moves, and the amount of time needed to find a solution is only increasing slightly (on average) with the row numbers. We can also see that in move 31, the planer surpassed the 10-minute limit to find a solution. We did try to give the planner different goal states in this position, but in almost all variations, the planner reached the 10-minute mark (while computing negative axioms). If we choose action 'P a2a3' as the '$i + 1$'th move (in row 31), for example, we would reach a solution in 282 seconds, but that is as far as we managed to push it.

5.2 Domain-Dependent Implementation: Performance (8x8)

In this experiment, we test the performance of our implementation from Chapter 4, and we are using the greedy best-first search variation. To make this experiment comparable to the last one we will use similar instances as in the previous experiment with the exception that we will include both kings. We will see that one of the states includes a position where the king is in check. The FEN codes will not be shown because they can be generated by looking up the same row in Table 5.2 and adding both kings to that FEN code. In Table 5.3

Table 5.3: Domain-Dependent planner results on a 8x8 chessboard (I)

Row	**Time**	**Moves**	**Nodes Expanded**
1	1ms	1	20
2	2ms	2	40
3	3ms	3	121
4	3ms	4	171
5	11ms	6	1365
6	52ms	6	9746
7	14ms	10	1866
8	16ms	8	2267
9	-	-	-

When looking for the goal state in Table 5.3 row 9, the search ran out of memory, and no solution was found. The memory timeout occurs at the 1 minute and 53 seconds mark. We found out that by using another search algorithm, we were able to combat this problem, meaning the reason is likely to be that there are too many states the algorithm has to consider which have the same heuristic value as the goal state, so the algorithm's choices are not guided by the heuristic value anymore within this subset of states with similar heuristic values. Our analysis of the problem confirmed this. However, we were able to find a solution using the A* algorithm, which brings us to our next experiment.

In this next experiment, we replicate experiment seven with the A* search algorithm instead of the greedy best-first search algorithm. The used heuristic is not admissible, but the results seemed good in practise. We now take a look at Table 5.4.

Table 5.4: Domain-Dependent planner results on a 8x8 chessboard (II)

Row	**Time**	**Moves**	**Nodes Expanded**
1	2ms	1	20
2	2ms	2	40
3	5ms	3	221
4	6ms	4	260
5	18ms	5	2043
6	16ms	6	1458
7	42ms	7	6377
8	54ms	8	9767
9	80ms	9	17148
10	83ms	10	18018
11	573ms	11	234'446
12	763ms	12	304'938
13	2s 166ms	13	1'180'648
14	4s 305ms	14	2'253'356
15	-	-	-

As we can see in Table 5.4, we run out of memory in row 15. This is not because the algorithm would not be able to find a solution at this depth but because the given goal state in row 15 is a state which involves an invalid move ('P c2c3' while white king is in check). It should be mentioned that there may still be a solution to this problem. We will now choose a different state in row 15 by moving the white knight in front of the king to block the black queen attack in row 15 and continue with new moves from there in Table 5.5.

Table 5.5: Domain-Dependent planner results on a 8x8 chessboard (III)

Row	**Time**	**Moves**	**Nodes Expanded**
15	7s 424ms	15	4672735
16	-	-	-

In Table 5.5 we see that the algorithm starts to run out of memory. This happens at the 1 minute 56 seconds mark.

As a final experiment, we tried to find a solution to the famous "Opera Game"[16] played between Paul Morphy and Duke Karl in 1858. With the greedy best-first variation, we were able to get a valid solution of length 22 when given (as a goal state) the sixth move of the Opera Game (B f1c4). A solution to this problem was found within 102 milliseconds with 31'681 nodes expanded. Using the A* variation on the same goal state, we get a valid solution of length 12 within 4 seconds and 123 milliseconds with 2'193'133 nodes expanded. As expected, the A* search is not the answer to the algorithm running out of memory, but A* could be used as an alternative approach in cases where greedy best-first search is unable to find a solution. The difference in expanded nodes per second between the A* search and

[16] https://www.chessgames.com/perl/chessgame?gid=1233404

the greedy best-first search is because of a base overhead for all problem instantiations. Furthermore, the more states are searched, the better the laptop (on which we are testing this implementation) can make use of SIMD instructions operating on vectors of 64-bit fields, which means we can do bit-wise operations on multiple bit-boards at the same time [3].

6
Conclusion

When we look back at the PDDL experiments, we can see that the red zone is the bottleneck predicate preventing us from having a PDDL implementation which can tell us whether a given chess puzzle is solvable or not if there is a king on the board. The implementation can be easily modified to generate problem-solvers for a knight tour[17] or a tour of any chess piece for that matter. One can also get creative and solve multi-figure tours. However, when a king is present on the board, the domain-independent PDDL approach is not fast enough to be used on an 8x8 chessboard, as we saw in Experiment two. The implementation can, however, be used on a 3x3 board. This is sufficient to solve chess problems such as the Guarini Puzzle but not enough to create interesting puzzles involving two kings. On a 4x4 chessboard, we have the same problem, but a 4x4 chessboard is enough to hold space for two kings; however, if we want to play regular chess on a 4x4 board, then the moves will be limited as we are only interested in non-repetitive moves unlike in problems such as the Guarini Puzzle. There are optimizations with which we could push the limits of the implementation to maybe work on bigger chessboards, but the time restrictions of this thesis did not allow them to make it into the experiments. The results of the PDDL implementation reflect our expectations because PDDL is used for domain-independent problems and can therefore not be optimized the way we can optimize a domain-specific implementation.

As a result of the optimized data structure (bitmaps) and various other optimizations we were able to implement, our domain-specific approach is much faster and can handle 8x8 board positions, as we saw in Experiments seven to nine. However, the implementation has a memory bottleneck, limiting the number of nodes we can expand. It would have been interesting to see how far we could push the limits of the domain-dependent approach if we handled this memory bottleneck, but time constraints did not allow that.

From our experiments, we can conclude that the greedy best-first search works well when we have a goal state that is close enough to the initial state. Furthermore, in Experiment Eight, we saw that the A* search is not the most efficient approach to our goal of determining

[17] In a knight's tour, the planner has to find a sequence of steps such that the knight visits every square exactly once on a chessboard.

if a given chess puzzle is solvable or not, but it can find a solution in some cases where the greedy best-first search fails. The drawback of A* is that it expands more nodes and therefore takes more time, and it also is not optimal like we saw in Chapter 4. One way to work around this is to use a different heuristic function (Manhattan distance, for example). What we also could do instead of using the A* algorithm is to use multiple heuristic values and have the best of all worlds by defining the composite heuristic value for a state to be $h(n) = \max_{i \in [n]} h_i$ where $h_1, ..., h_n$ are different heuristics are different heuristics [1, p.100]. Due to time constraints, we were not able to implement and test this.

Bibliography

[1] *Artificial intelligence: a modern approach*. Pearson, Prentice Hall, Upper Saddle River, NJ, 2020. ISBN 0134671937.

[2] Cameron Browne. Bitboard methods for games. *ICGA journal*, 37(2):67–84, 2014.

[3] Robert D. Cameron, Thomas C. Shermer, Arrvindh Shriraman, Kenneth S. Herdy, Dan Lin, Benjamin R. Hull, and Meng Lin. Bitwise data parallelism in regular expression matching. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 139–150, 2014. doi: 10.1145/2628071.2628079.

[4] Stephen Chenoweth. On the NP-Hardness of Blocks World. In *AAAI 1991*, pages 623–628, 1991.

[5] Drew, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins SRI, Anthony Barrett, Dave Christianson, et al. PDDL— the planning domain definition language. *Technical Report, Tech. Rep.*, 1998.

[6] Isenberg Gerd. Efficient generation of moves and controls of sliding pieces. https://www.chessprogramming.org/Efficient_Generation_of_Sliding_Piece_Attacks. (visited on 16.05.2022).

[7] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Elsevier, 2004. ISBN 9780080490519.

[8] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016. ISBN 9781107037274.

[9] Adam Green and Jan Dolejsi. Planning.wiki: The domain. https://planning.wiki/ref/pddl/domain. (visited on 06.05.2022).

[10] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. *An introduction to the planning domain definition language*, volume 13. Morgan Claypool Publishers, 2019. ISBN 9781627057370.

[11] Robert Hearn and Erik Demaine. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1-2):72–96, 2005.

[12] Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[13] Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.

[14] Malte Helmert. Foundations of artificial intelligence - state-space search: State spaces. https://ai.dmi.unibas.ch/_files/teaching/fs22/ai/slides/ai05.pdf, 2022. (visited on 03.05.2022).

[15] Malte Helmert. Foundations of artificial intelligence - automated planning: Planning formalisms. https://ai.dmi.unibas.ch/_files/teaching/fs22/ai/slides/ai34.pdf, 2022. (visited on 03.05.2022).

[16] Malte Helmert. Foundations of artificial intelligence - state-space search: Greedy BFS, A*, weighted A*. https://ai.dmi.unibas.ch/_files/teaching/fs22/ai/slides/ai16.pdf, 2022. (visited on 13.05.2022).

[17] Peter Kissmann. *Symbolic search in planning and general game playing.* PhD thesis, Universität Bremen, 2012.

[18] Sabarinath Mohandas and M Abdul Nizar. Ai for games with high branching factor. In *2018 International CET Conference on Control, Communication, and Computing (IC4)*, pages 372–376. IEEE, 2018.

[19] John J. Watkins. *Across the Board: The Mathematics of Chessboard Problems.* Princeton University Press, Jul 2012. ISBN 9780691154985.

A

A.1 State Space definition

A state-space (or transition system) is the set of all possible configurations a given system can be in and can be formalized as a 6-tuple:

$$\boldsymbol{\Sigma} = \langle \boldsymbol{\mathcal{S}}, \boldsymbol{\mathcal{A}}, \boldsymbol{cost}, \boldsymbol{\mathcal{T}}, \boldsymbol{s_0}, \boldsymbol{s_\star} \rangle \tag{A.1}$$

with the following components:

- $\boldsymbol{\mathcal{S}}$ is a finite set of states s in which the system may be. In our example problem from Figure 1.1 $\boldsymbol{\mathcal{S}}$ is equal to the set of all possible combinations of the four knights where they are placed on the outer ring (middle square is empty) of the board.
- $\boldsymbol{\mathcal{A}}$ is a finite set of actions a that the actor can perform. In our example $\boldsymbol{\mathcal{A}}$ is equal to the the possible knight moves in every position which gives us 8 possible actions out of which exactly two are applicable actions in every given position.
- $\boldsymbol{cost}$ is a finite set of action costs with $\mathcal{A} \rightarrow \mathbb{R}_0^+$ and $cost(a_i)=\sum_{i=1}^{n} cost(a_i)$. The cost function is what we want to minimize. If the cost function is not given explicitly, then $cost(s,a)=1$. In the Guarini-Problem, the most logical choice of an action cost is the cost function which assigns 1 to every action.
- The transition relation $\boldsymbol{\mathcal{T}} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ with $\mathcal{T}$ is deterministic in $\langle s,a \rangle$ meaning that it is forbidden to have both $s \xrightarrow{a} s'_1$ *and* $s \xrightarrow{a} s'_2$ *with* $s_1 \neq s_2$ *and* $s,s' \in \mathcal{S}$ being states with $s \rightarrow s'$ (s' is a successor of s meaning when the transition relation $\mathcal{T}$ is applied to a state, it results in a successor). A state transition is defined as the triple $\langle s,a,s' \rangle \in \mathcal{T}$ which describes the transition from state s to state s' with action a.
- $\boldsymbol{s_0} \in \mathcal{S}$ is the initial state. In the Guarini example the start state is given in figure 1.1 (left side board).
- $\boldsymbol{s_\star} \subseteq \mathcal{S}$ is a finite set of goal states [14]. In the Guarini example the goal state is given in figure 1.1 (right side board).

B

B.1 Planning Problem Definition

A finite-domain planning problem $\mathcal{P}$ can be formalized as a state transition system (also called a classical planning domain or state-space) [10, p.12 & p.25].
It can be formalized as a 5-tuple:

$$\boldsymbol{\Sigma} = \langle \boldsymbol{\mathcal{V}}, \boldsymbol{s_0}, \boldsymbol{s_\star}, \boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{O}} \rangle \tag{B.1}$$

with the following components:

- $\boldsymbol{\mathcal{V}}$ is a finite set of state variables $v \in \boldsymbol{\mathcal{V}}$. We can distinguish state variables into three types: statics, fluents and derived variables (more on them in Section 2.2.2).
- $\boldsymbol{s_0}$ is the initial state (which is a state over $\boldsymbol{\mathcal{V}}$). In the Guarini example, the start state is given in Figure 1.1 (left side board).
- $\boldsymbol{s_\star}$ is the goal state (which is a partial variable assignment over $\boldsymbol{\mathcal{V}}$). In our running example (Guarini Problem), the goal state is given in Figure 1.1 (right side board).
- $\boldsymbol{\mathcal{A}}$ is a finite set of axioms (over $\boldsymbol{\mathcal{V}}$). An axiom is defined as a triple $\langle cond, v, d \rangle$ where *cond* is defined as a partial variable assignment (called condition), v is a derived variable and d is the resulting value (called derived value) for v.
- $\boldsymbol{\mathcal{O}}$ is a finite set of operators which define how a state of the state-space can be inspected and manipulated. We can build complex expressions when combining different operators. [13]

A solution is defined as a path through the state-space[4] starting from the initial state ($\boldsymbol{s_0}$) to the goal state ($\boldsymbol{s_\star}$) [1, p.108].

Notice also: In Definition 2.1 we reduced Definition B.1 by $\boldsymbol{\mathcal{V}}$ and the reason why we remove $\boldsymbol{\mathcal{V}}$ is because *"PDDL tasks use first-order concepts such as schematic operators whose variables can be instantiated in many different ways"* [13, p.10].

University of Basel

Faculty of Science

Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis:

Name Assessor: _____

Name Student: _____

Matriculation No.: _____

With my signature I declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Place, Date: _____ Student: _____

Will this work, or parts of it, be published?

No

Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: _____

Place, Date: _____ Student: _____

Place, Date: _____ Assessor: _____

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis