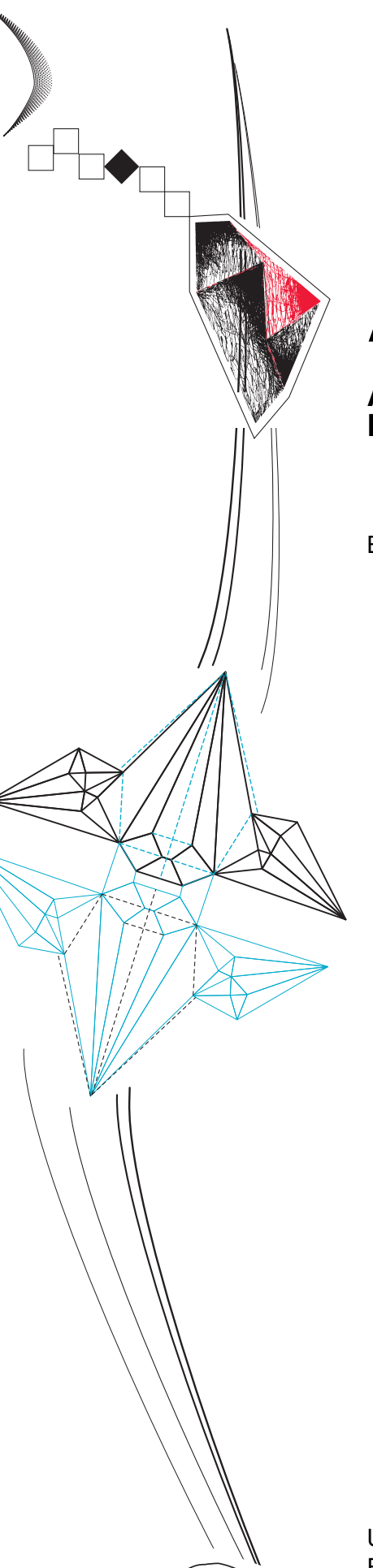


March 26, 2020

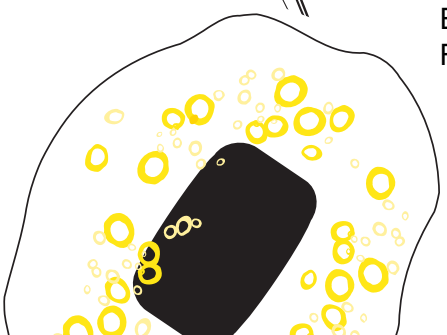
ABEONA

A GENERALIZED FRAMEWORK FOR STATE SPACE EXPLORATION ALGORITHM COMPOSITION

Bram Kamies

COMMITTEE
prof.dr. A. Rensink
dr. S. Wang

University of Twente, Enschede, The Netherlands
Electrical Engineering, Mathematics & Computer Science (EEMCS)
Formal Methods & Tools (FMT)



UNIVERSITY OF TWENTE.

Acknowledgements

Before all, I would like to thank my graduation committee and most importantly Arend Rensink for giving me guidance and feedback throughout the project.

Besides the committee I would also like to thank Ilonka Grosman, Roos Kamies and Aimée Zoë Walst for giving me the emotional support to continue studying and finishing what I started. Of course there are more people to thank, and thank them I will. I know that with this thesis I would have made my dad proud. His alzheimers already chipped away a lot of what he used to be, so he may no longer be able to understand what I have achieved here.

My master thesis has been a rollercoaster of a ride, with ups and downs. The content was interesting, introducing me to many aspects of computer science as well as testing my abilities over and over again. Those were the fun times, where school was challenging and fun. The downs were not necessarily part of school directly but are still tied into my experience of my time at the University of Twente. In the past years I have learned a lot, not only about computer science but also about becoming more of a responsible person. I will never regret my decision to follow my passion and study at UTwente.

At the time of writing the world is gripped by the coronavirus (COVID-19). With quarantine impacting the wrap-up of my master study, it certainly has taken the wind out of the sails for the final presentation. Still, we must go on and rest assured that an adequate celebration will occur regardless. The future is full of uncertainty and this can be overwhelming, but I am a much stronger person than I was when I first stepped foot into Enschede. I hope to give back to others as I have received from them; in experience, knowledge and comfort through tough times.

- Bram Kamies, 26 March 2020

Abstract

In this thesis, we propose a novel design for implementing state space exploration (SSE) algorithms through the composition of components. The goal of this design is to offer a generalisation for users of SSE algorithms to use in their implementations. The advantage of the compositional approach is the ability to tweak, add and remove components from any given SSE algorithm built with the compositional framework. This satisfies the use-case in which users need to tweak and configure their solution and would have otherwise had to resort to modifying the original implementation source code.

The framework supports a compositional approach and we validate its utility. Following an analysis of existing SSE algorithms we first derived a skeleton pipeline for the execution of SSE algorithms and a set of components necessary to implement them. To validate the design, we integrated the implementation into two existing solutions for SSE based exploration. The integration shows how the framework contributes to existing solutions. Moveover, the integration expands the capabilities of the original solution. Second, we compared the performance overhead of the framework to that of other solutions. This includes comparing the performance to a plain implementation as well as to the performance of the existing solutions in the integrations. Third, we provided various demo applications with the developed framework implementation to showcase the state-agnostic property of the framework. This shows that the framework is widely applicable to any preexisting representation of states the user may have. Finally, we performed a user study to obtain insight into the user-friendliness of the framework for new users.

We were able to implement the design into a working framework prototype. The results of the performance comparison shows that the design's pipeline creates a performance overhead. This is clear in the comparison with a tailor-made algorithm.
The user studies gave positive feedback; participants indicated they found the modularity beneficial. While the new framework does not outperform existing solutions, it is on-par in terms of time-complexity in several scenarios.

Table of contents

1 Introduction

This report is the result of a master thesis project for the University of Twente. The project is centered around state space exploration (SSE), with a focus on GROOVE as a case study. The problem in question is the considerable cost of implementing new SSE algorithms while in reality the new work only requires a few minor changes to the original implementation logic. To offer a new method of implementing SSE algorithms we devised a novel approach whereby an exploration algorithm is composed of several reusable components.

The work presented here was preceded by a research topics report (**Appendix A**), the results of which were used as part of the implementation of our work. The most important part being the algorithm analysis, where we performed an initial decomposition for a set of algorithms.

The study of computer science encompasses many different fields, with the science dedicated to the means and abilities by which we are able to control, describe and transform data and processes. Computer science overlaps in places with mathematics and physics. Within this study are several areas that leverage the technique of SSE, which is the focus of this work. The fields this work is most closely related to are artificial intelligence, model checking and design space exploration. This work contributes to these fields by developing a novel compositional approach to algorithm implementation.

SSE is a method used in computer science to facilitate searching through a large space (domain) made up of states (values). The state space is represented by a directional graph in which the states (nodes) are connected by transitions (edges) describing how one state may be transformed into another. As this technique is comprehensive and applicable for a great range of different purposes, it is widely used in different fields. The following section details a few areas in which SSE is being used, how it is being applied in those fields and what practical applications it has been used in.

Model checking

The method of model checking [1] focusses on verifying properties and statements about Kripke structures [2]. A common application of this is to perform verification of (concurrent) software. Such verifications provide guarantees about the software along all possible paths of execution. These guarantees are then then used to ensure the absence of bugs and the adherence of certain safety protocols. Especially in safety critical systems such as those in the medical sector (e.g. MRI-machines), such guarantees are difficult to obtain with traditional testing techniques, while they are necessary for the systems to be used safely. The difficulty of obtaining such guarantees comes from the high complexity of the system, proving the correctness of many processes running in parallel within large systems is costly.

Several model checking toolchains exist, such as LTSmin [3], mCRL2 [4], JPF [5] and GROOVE [6].

Planning

When we talk about the planning field we specifically mean the artificial intelligence field where solver programs generate plans to solve combinatorial and scheduling problems. One of the methods for building planners uses SSE to build partial solutions to a problem until a full solution is found. Planning problems require their answers to be in the form of a plan which can be enacted from the starting conditions and which will result in fulfilling the predefined goal conditions. The International Planning Competition (IPC) developed the Problem Domain Description Language (PDDL [7]) to help standardize the input format for planning systems. PDDL has since then been widely adopted and it is the standard model for describing planning problems.

Planners such as FF [8], FD [9] and BFWS ([10], [11]) are well known planners in the field of artificial intelligence.

Design space exploration
Embedded systems design employs a technique called Design Space Exploration (DSE [12]). DSE automatically evaluates design options. In DSE, a model for a system design is defined where only the high-level function of the system is defined. This high level abstraction allows the decision making to be delayed in terms of implementation details. Each point in the design at which a choice can be made between different implementations forms a choice-point. The possible combinations of choices at these choice-points forms the design space. This design space can grow very large as it increases exponentially with each new choice-point. DSE aims to automatically explore the design space and find suitable implementations for the high-level model (also called a synthesis of the system) from which the most appropriate implementation can then be chosen.

DSE is essential for large scale systems, as they can be designed in a more efficient fashion than if they were designed manually. With large sets of choices, the number of possible designs can easily reach the likes of millions or billions of designs.

Mathematical programming is among the common techniques of SSE mentioned in [13] as well. This usage is most closely related to this work as it is based on a branch-and-bound algorithm [14]. In [15] the proposed algorithm improves DPLL, an algorithm based on depth-first-search (DFS [16]). DFS itself is a well-known SSE search strategy.

1.1 Motivation

Existing SSE tools are focussed on implementing a specific set of exploration algorithms and optimizations for executing state space searches. This gives users a reliable feature-set and an optimized performance for the techniques within the tools' domain. The dedication to specific state representations and exploration algorithms restricts users to conform to the input languages that their tools understand. Previous efforts to generalize these tools often focused on expanding the range of input languages they understand. This gives users with an existing model more opportunities to use their models directly with tools that can cooperate with more diverse input languages. However, the generalization of existing tools does not offer extension of the capabilities of said tools; the available feature-set remains the same.

Take for example a user that wants to test different search strategies on a particular model or problem. For them to use the different algorithms that exist they would need to find conversions to the models that the tools implementing those algorithms accept. When the users requirements change and they need an algorithm that is available in a specific tool but with an optimization that the tool does not support then the user is left to make the required source-code changes themselves, or ditch the tool for another.

A modular approach to the feature-set of a SSE strategy would save users from fully reimplementing all exploration algorithm details themselves and only require them to design a module which applies the behavior they need.

For existing tools such a modification is also possible, but is not scalable in the same way a module would be. When the number of requirements changes over time (as they tend to do), the user needs to implement these modifications across a code base each time. Without a module system to guide and map these modifications to the existing tool, they could easily become entangled. Changing requirements can result in additions to code, but can lead the user to remove previous changes to the algorithm. The entanglement of code forms a major problem as code changes often depend on each other, causing malfunctions when requirements contradict one another. Additionally, when the user decides to switch to a different tool, then some of their changes may need to be ported over if the new tool does not support all requirements out of the box.

Thus we propose that a modular framework which supports exploration algorithms would benefit the many fields in computer science which utilize SSE. Developers and researchers would be able to use the framework to build new tools that provide a more flexible environment to their end-users. Additionally, if possible the framework can be made flexible enough to also target existing solutions, to extend their feature-set by integrating with their search strategy.

Implementing such a framework is not trivial, as the application of SSE occurs in many different ways. In model checking, for example, the verification of a formula over a state space may terminate the exploration as soon as a trace that violates it is found while in DSE a certain number of solutions may be collected. There is no clear way of dividing a search strategy into meaningful components.

1.2 Research questions

Based on the motivation, our goal is:

> Create a generalized SSE framework that provides the user with fine-grain control over the algorithm through composition of search strategies using modules, allowing them to utilize different and configurable SSE based approaches on a problem they are facing.

This poses the following research questions:

Can a general framework for building search algorithms be designed? (main question)
Our main question to answer is whether the framework can actually be designed and made. We want to figure out what it takes to create a SSE framework that is able to remain flexible and modular to meet our end-users constraints. This is our main research question. For the design of the framework, we have the following sub-questions to answer about the framework:

Does the general framework support implementation of existing exploration algorithms through composition?
The intention to use modularization and composition does question the nature of exploration algorithms, whether a composition can be achieved and what components would exist in a compositional approach.

Can specific components of algorithms be integrated into other algorithms?
Besides the modular approach offering great flexibility to users and developers of exploration algorithms, there may also be an opportunity for reuse. The problem we face when implementing a SSE algorithm is that we need to build it from the ground up when existing solutions are not suitable. The framework aims to solve this partially by offering more flexibility in a new solution, but the sharing of the resulting components would help future authors to not even need to implement their own custom modules if there already exists one for them.

How well does the framework integrate with existing applications of state space exploration?
Designing a novel framework offers a new approach to users that are trying to solve a new problem but offers little contribution to users of SSE algorithms that have already built a solution using some other tool. To extend an olive branch to users of other tools we want to look into the possibility to deploy the framework in existing solutions.

How does the performance of the framework compare to that of existing state space exploration implementations?
A generalized approach which aims to be domain-independent will not be able to utilize optimizations that knowledge of that domain has to offer. As such it is likely that the framework has a certain overhead cost associated with it compared to existing solutions. We want to measure the performance of a general framework and compare it to existing solutions to offer some insight into the trade-offs.

Is such a general framework easy enough to work with while remaining sufficiently general?
Because the framework will be most like a toolkit (code library) rather than a standalone tool (e.g. a command-line or graphical program), it is useful to look at the user experience of the new framework. Knowing the pain-points will highlight where the design may fall short in meeting user expectations and challenging topics may point out a steep learning curve in the knowledge required to operate the framework for new users.

1.3 Methodology

In our preparation we did an analysis of several exploration algorithms, from the analysis we were able to derive the general pipeline which each algorithm uses. We also found the components necessary to encapsulate each of the algorithms and what components were unique to each algorithm.

Before implementing the framework we chose our design methodology from well known programming practises to help ensure that our final implementation would support all our requirements and concerns. The programming practises as well as some complexity analysis helped us optimize performance ahead of time. During development of the framework we also implemented demo applications which are able to demonstrate the generality, flexibility and features of the framework.

For performance testing of the framework we used a controlled environment and made efforts to equalize the resource availability among the test runs. We utilized the Java Microbenchmark Harness (JMH) to facilitate the benchmarking environment. Each benchmark performed 5 repeats for which the results were then averaged out.

After the implementation of the framework we targeted two existing tools utilizing SSE, GROOVE and PDDL4J, for integration. With the integration our goal was to demonstrate the ability to use the framework. Specifically, its modular and compositional features. The integration with the PDDL4J library was also subject to a benchmark comparison to observe the possible overhead the framework brings with it.

Additionally we ran a small user study to obtain feedback about the framework. This includes testing the user friendliness of the framework to new users as well as observing the behaviour of users when exposed to the new compositional approach to algorithms.

The report is laid out as follows: first we detail the required background information to understand the work in **Chapter 2**. Then in **Chapter 3** we mention existing work that is related to what we are doing here. With **Chapter 4** we explain our design and implementation of the framework. Next we talk about the performance of our implementation, the integrations and the user study in **Chapter 5**. Finally we form our conclusion in **Chapter 6** followed by the future work in **Chapter 7**.

1.4 Products

4.4.1 Framework

This work produces the Abeona framework, a java library implementing the modular and composition based framework for executing SSE algorithms. We refer to this work in some cases as the Abeona project.

4.4.2 Demos

The framework's source code[1] comes with a range of demo applications which implement various problems and utilize the framework to solve them. Each demo is designed to use a specific set of features of Abeona to demonstrate how said features are used. The demo applications are packaged with the source code of the abeona code repository.

4.4.3 Integrations

The integration with GROOVE will expand its existing search strategies with a modular strategy that can be fine-tuned by the user. Additionally, the GROOVE developers can extend it with additional modules to further expand the flexibility of the new search strategy. The GROOVE integration is made in a separate code repository[2] from the abeona code repository.

The integration with PDDL4J will provide existing users of PDDL4J with a new search strategy which is modular. Users have fine-grained control over the search strategy without having to fully implement all of its details. The implementation of the integration is part of the abeona code repository as a demo package.

4.4.4 Contributions

This thesis contributes to several fields, as the product is designed to be general and widely applicable. The contributions listed here are based on the goals of the project as well as the products of the work itself.

- The framework will offer authors of existing tools a toolkit to introduce component based exploration strategies to their existing codebase and infrastructure.
- The framework will be a toolkit for users that require a custom made solution based on SSE.
- The framework will be a benchmark tool to researchers who manage and run a wide range of benchmarks for SSE algorithms, offering the benefit of implementing them through a single codebase.
- The framework will be an educational tool for students to implement various search strategies. In addition, it will help them understand the underlying characteristics those algorithms embody.
- The integrations have been used to extend two existing tools (GROOVE and PDDL4J) with new capabilities

[1] Abeona repository: https://github.com/maritaria/abeona
[2] GROOVE integration code: https://github.com/maritaria/abeona-groove

2 Background

This chapter explains the background information used throughout the rest of the report. The topics are as follows: **Section 2.1** explains the terminology of SSE, **Section 2.2** goes into detail on model checking and SSE is used in the field. **Section 2.3** explains the link between SSE and the planning domain, also mentioning PDDL and giving an example of this. **Section 2.4** gives a small overview of the principles behind object-oriented programming. Finally, **Section 2.5** explains aspect-oriented programming.

2.1 State space exploration

State spaces contain all possible states a system might occur in, including invalid or error states. States within the state space are connected when a state can transition into another state in that space. The connections between states are directional, it is not given that the mutation that follows from a transition could be undone. The structure of a state space is represented mathematically as a directed graph. The states of the state space can be mapped directly to nodes in the graph and the transitions between states can be represented with directional arrows in the graph. An example of a state space is shown in **Figure 2.1**, here the state space contains six states (S0 through S5). We will use this example state space to explain state space related terminology in this section.

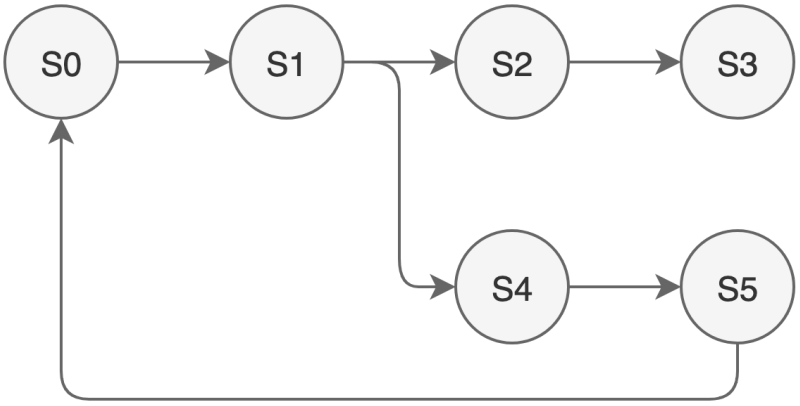


Figure 2.1: Example state space with six states (S0-S5) and various transitions between them.

A state is represented in the graph with a node, but a state itself is a mathematical entity. Across various applications of SSE different internal structures for states are used. Transitions are in their simplest form a tuple containing the source state from which the transition starts, the target state the transition points to and a label that identifies the transition. It is possible for multiple transitions to exist between the same two states while representing different mutations. In **Figure 2.1** the transitions are not labelled because there are no two transitions with the same source and target states. For a given state, the transitions which have that state as the source state of the transition are named the “outgoing transitions” of that state.

Known and unknown states

The set of states contained in a state space can be split into two subsets, the known and unknown states. During exploration there is a notion of known-ness where a section of the complete state space is known. A state is known when it is explicitly defined by the user or discovered during exploration. All states that are not known are part of the unknown set of states. In **Figure 2.2** the states from the state space have been labelled as known (green) and unknown (gray). During exploration a state may be encountered that is part of the unknown states set. This is called the discovery of that state and moves it into the set of known states. The discoveries occur when a state is being evaluated. The evaluation of a state consists of enumerating the outgoing transitions of the state and filtering out the transitions which point to an unknown state.

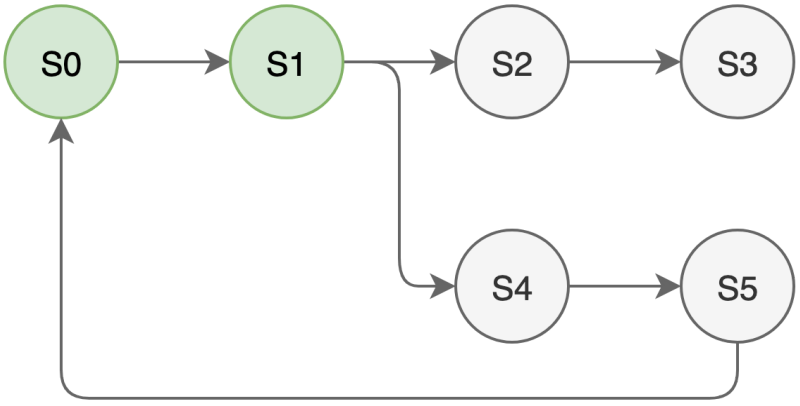


Figure 2.2: The state space with states S0 and S1 labeled as known (green), the remaining states are unknown (gray)

Frontier and heap

State spaces are static structures, that is they do not change during the exploration. Because of this, every state only needs to be evaluated once during exploration. The known states set is split into two subsets, the frontier and the heap. The frontier contains the states which have not yet been evaluated. The heap contains those states which are known and have been evaluated. The discovery of a state places the state into the frontier. In **Figure 2.3** the states from the running state space example are divided into the frontier (blue) and heap (green) sets.

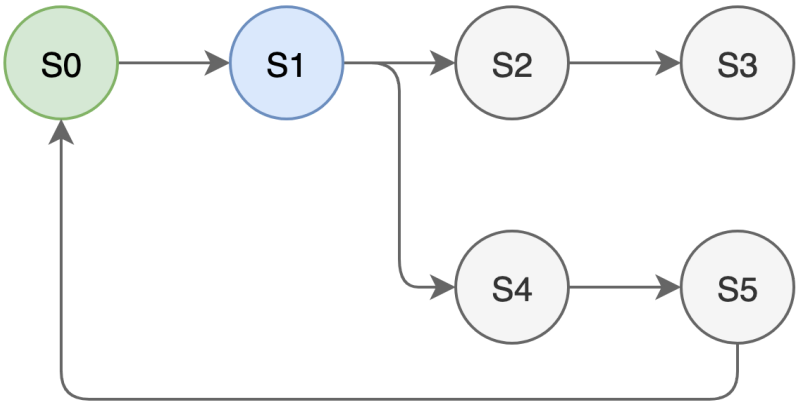


Figure 2.3: The state space with state S1 marked as part of the frontier (blue)

Exploration is a process which iterates over the states in the frontier in order to attempt to increase the set of known states. At every iteration a state is evaluated: this consists of enumerating the outgoing transitions of the state and possibly discovering unknown states from the transitions. In **Figure 2.4** the state state after evaluating S1 is shown, the evaluation caused the state to become part of the heap (green). From S1 the two outgoing transitions pointing to S2 and S4 are found. For both transitions the target states are unknown and thus the S2 and S4 states are discovered and put into the frontier (blue). After the evaluation the frontier contains the 2 discovered states and these will be evaluated in the next iterations.

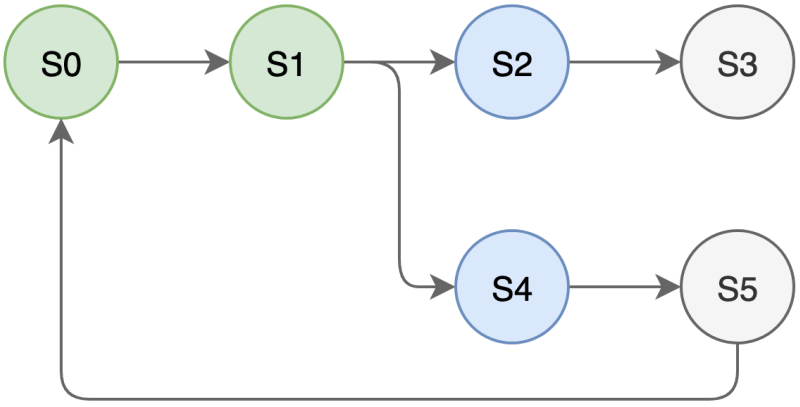


Figure 2.4: The state space after discovering states S2 and S4 (now part of the frontier) when evaluating S1.

Initial states and goal states

Exploration needs to start with at least one state in the frontier. The user of a SSE algorithm is required to give the algorithm an initial state(s). The iteration of state evaluations and discoveries continues as long as there are states left in the frontier set. If exploration ends because the frontier is empty it is called the exhaustion of the frontier. For finite state spaces the frontier is guaranteed to eventually become empty. When the frontier becomes exhausted then all states reachable from the given initial states have been found and evaluated.

Users of SSE do not always need to run the exploration until the frontier is exhausted. Instead, they may be looking for a specific (type of) state, such states are called ‘goal states’. In model checking, finding a state that violates the model being checked on the state space may be sufficient to terminate exploration early. Alternatively, in DSE the exploration is usually run until several goal states (suitable solutions) have been discovered.

2.2 Model checking

In model checking a formula is verified to hold against all states in the state space, making the state space a model of the formula [1]. The formula can describe the behaviour of the system over time, using temporal logic [17]. The two main variations of logic used to describe such properties are linear temporal logic (LTL [18]) and computation tree logic (CTL [19]). A model checker will verify one or more logic expressions against a state space. The state space may be fully defined as some graph but more commonly it is generated from a system specification dynamically and explored using SSE. Model checkers are able to produce counter-examples when they encounter a violation of the formula being checked.

For an example of model checking a property we demonstrate this on a fictional example state space shown in **Figure 2.5**. The image is a small variation on the running example state space from the previous section. In the state space there is a goal state: S3 (marked red). A model checker would be able to verify that S3 is reachable and produce the trace [S0->S1, S1->S2, S2->S3]. Additionally it would also be able to prove that there is no guarantee that S3 will ever be reached, proven by the trace: [S0->S1, S1->S4, S4->S5, S5->S0] (looped infinitely).

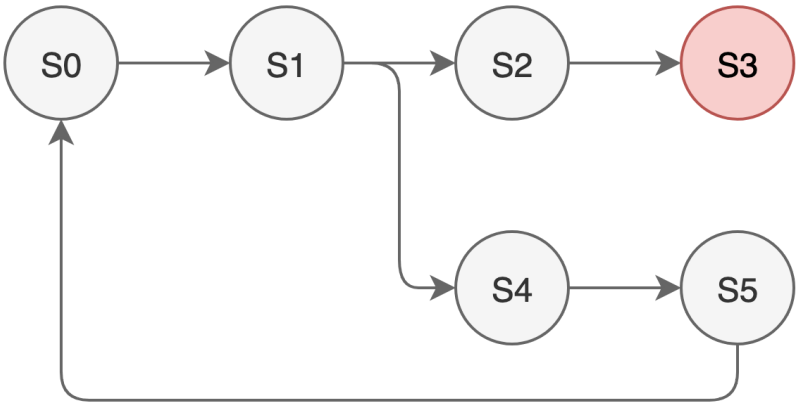
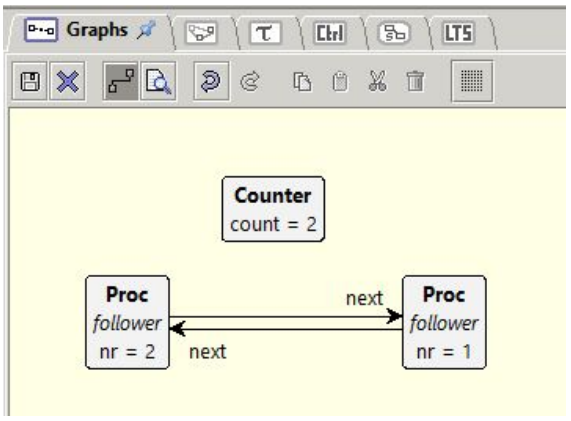


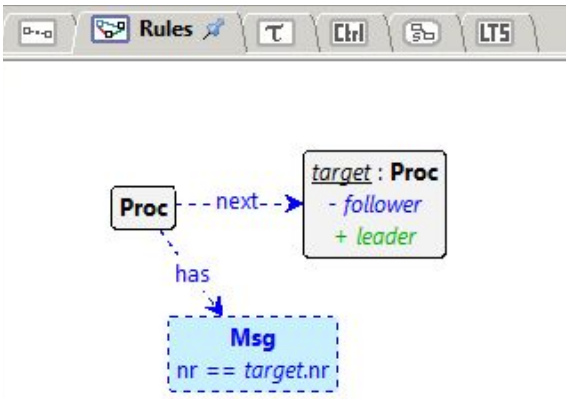
Figure 2.5: An example state space with state S3 highlighted (red) as a state of interest in the system.

2.2.1 GROOVE

The GROOVE toolset [6] is an application with model checking capabilities based on graph transformations. Models of systems in GROOVE, so called ‘grammars’, are expressed almost entirely using graphs. States in the state space are represented using graphs, properties of states are labelled edges from the state instance to the literal value (as a node). In **Figure 2.6-A** the user interface for a (state) graph is shown. Grammars also consist of “rules” defining graph transformations, expressed as graphs. One such rule graph

is shown in **Figure 2.6-B**, here the blue elements indicate a deletion directive, when all elements in the rule match a state instance then this will result in a new state with the blue elements removed from the state instance.

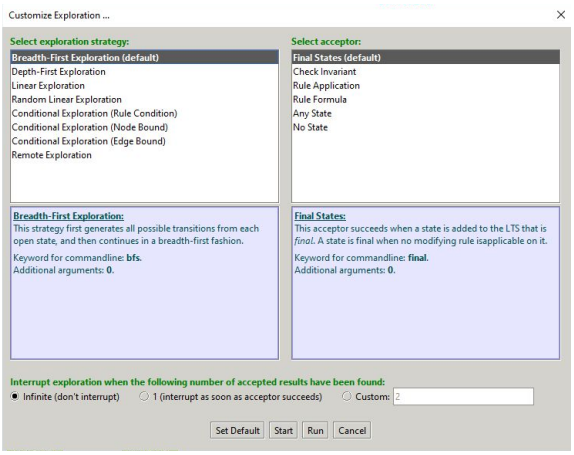


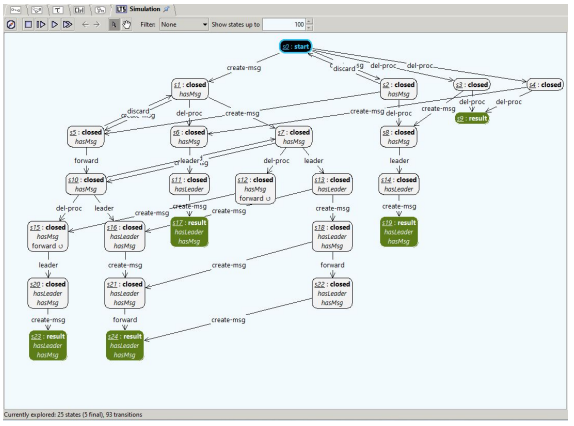


A: Example of a state-graph in GROOVE **B:** Example of a rule-graph in GROOVE

Figure 2.6: Two types of views used in the GROOVE simulator to view and edit parts of a GROOVE-grammar.

The GROOVE tool set contains the GROOVE simulator (shown in **Figures 2.6 and 2.7**). The toolset also comes with various other command-line programs to perform model checking, exploration and other tasks on GROOVE grammars. To perform an exploration of the state space in GROOVE the user needs to select a strategy using the exploration configuration dialog (shown in **Figure 2.7-A**). An exploration configuration consists of three main parts: the strategy, the acceptor and the termination settings). The strategy is chosen from a list of available strategies (top left list). Each strategy may have its own settings to further configure the exploration strategy (shown in bottom right panel). The strategy is in control of the exploration order and extent to which states are explored. The acceptor setting defines what identifies goal states during the search. Each of the acceptors can also have their own settings (shown in the bottom right panel). Finally the termination setting controls how often the acceptor needs to be triggered before the exploration is terminated. After running the exploration the discovered state space is shown in the state space overview (see **Figure 2.7-B**).





A: The exploration configuration dialog in GROOVE

B: The view of an explored state space in GROOVE.

Figure 2.7: Two exploration related views from the GROOVE simulator application.

2.3 Planners

In the field of artificial intelligence, the algorithms which solve planning problems are called planners: they formulate plans. A plan is a description of a series of actions which turn a system or problem from the initial state into an acceptable (goal) state.

Planners often accept the PDDL [7] as an input to specify problems with. The goal of the PDDL language is to describe the domain in which a (set of) problems may present itself. Because of this, a PDDL problem is split into two files: the domain file and the problem file. This division eases the management of various problems within the same domain. The domain specifies the world in which the problem exists. In PDDL the entities which exist in the world are blank and can be extended with domain-specific information using predicates which may or may not hold for those entities. The predicates do not need to actually implement any logic, they simply exist and a state of a PDDL problem tracks which predicates hold for which entities. Besides predicates, the second main part of a PDDL file are the actions. An action defines an executable change to the state of the problem. It is described using preconditions and postconditions (called effects). **Figure 2.8** shows the domain part of a well-known problem for planners, the gripper problem. The problem file of a PDDL problem defines the details specific to that problem instance within the domain. A problem is built on top of the domain specification and instantiates it with a list of objects used, the initial state for the problem and a description of the goal state. For the gripper domain a problem file is shown in **Figure 2.9** which details the items in the rooms that the gripper robot has to move, where they are located initially and where they have to go.

```
(define (domain gripper-domain)
   (:requirements :typing)
   (:types room ball gripper)
   (:constants left right - gripper)
   (:predicates (at-robby ?r - room)
                (at ?b - ball ?r - room)
```

```
            (free ?g - gripper)
            (carry ?o - ball ?g - gripper))
  (:action move
      :parameters  (?from ?to - room)
      :precondition (at-robby ?from)
      :effect (and  (at-robby ?to)
                  (not (at-robby ?from))))
  (:action pick
      :parameters (?obj - ball ?room - room ?gripper - gripper)
      :precondition  (and  (at ?obj ?room) (at-robby ?room) (free ?gripper))
      :effect (and (carry ?obj ?gripper)
                  (not (at ?obj ?room))
                  (not (free ?gripper))))
  (:action drop
      :parameters  (?obj - ball ?room - room ?gripper - gripper)
      :precondition  (and  (carry ?obj ?gripper) (at-robby ?room))
      :effect (and (at ?obj ?room)
                  (free ?gripper)
                  (not (carry ?obj ?gripper)))))
```

Figure 2.8: The domain PDDL-file of the gripper problem

```
(define (problem gripper-problem)
   (:domain gripper-domain)
   (:requirements :typing)
   (:objects rooma roomb - room
             ball4 ball3 ball2 ball1 - ball)
   (:init (at-robby rooma)
          (free left)
          (free right)
          (at ball4 rooma)
          (at ball3 rooma)
          (at ball2 rooma)
          (at ball1 rooma))
   (:goal (and (at ball4 roomb)
               (at ball3 roomb)
               (at ball2 roomb)
               (at ball1 roomb))))
```

Figure 2.9: The problem PDDL-file of a gripper problem

Planning problems bear a close relationship to graph transformation systems (as used by GROOVE). PDDL forms relationships between entities and labels properties of entities using predicates, this can easily be represented in a graph transformation system by leveraging the expressive nature by which graphs are able to encode relationships. As an example the initial state of the problem in **Figure 2.9** can be expressed as a graph, this is shown in **Figure 2.10**. The transformations of such a graph can be performed in accordance with the actions defined by **Figure 2.8**.

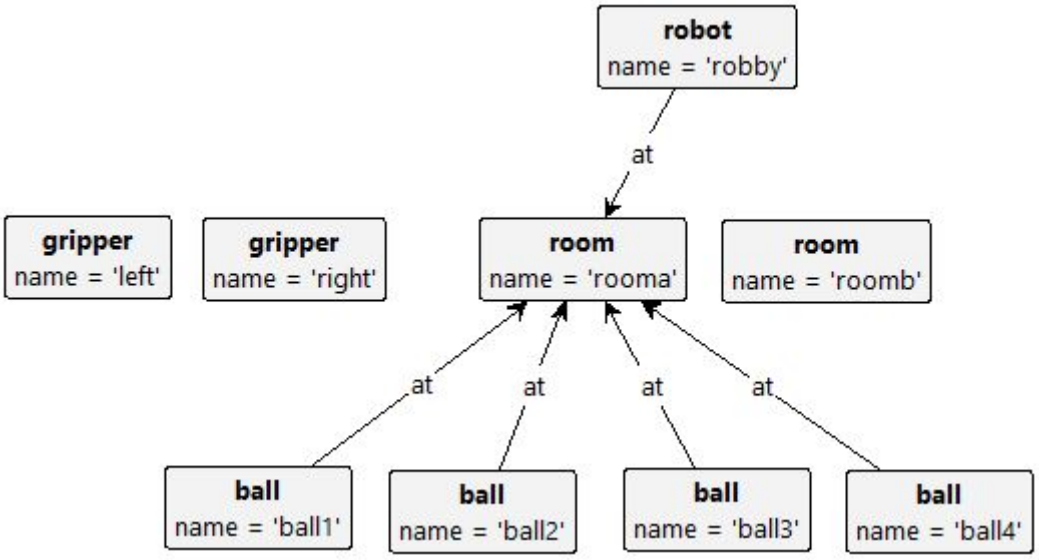


Figure 2.10: The initial problem state from **Figure 2.9**, represented as a graph

2.4 Object-oriented programming

In **Chapter 4** the concept of object-oriented programming is used, so we mention it here briefly. The paradigm of object-oriented programming (OOP [20]) has been a popular language feature in many programming languages over the past two decades. The paradigm enables programmers to structure code in a manner that reflects the real world. This follows from the ability to organise code and data into classes that are able to represent real-world concepts and group the behaviour and data such concepts would contain. Apart from its organizational benefits, the class system also enables several other features of OOP based languages. First, a class in OOP offers encapsulation [21] of the internal details by protecting the state from being changed arbitrarily and instead to be controlled through a predefined API. The state is stored in the class fields and the methods form the mentioned API. Secondly, OOP languages (often) include a way to organise classes into a hierarchy, through inheritance [21]. When a class inherits from another, it places these classes into a hierarchy. Inheriting from a class makes the new class a subclass of the original class. Likewise the original class can be called a parent-class of the new class. Encapsulation is tied into the hierarchy, where parent classes have the ability to limit and control the access subclasses have to the internal state and APIs of the parent class. A specific kind of inheritance, multiple-inheritance, where multiple classes are the parent-class of a new class is not often supported directly by OOP languages, but instead programming patterns are used to overcome this. Finally some languages also allow interfaces to be strongly typed. Such interfaces can be integrated into the hierarchy as well, where a class implements an interface as well as interfaces inheriting from one another.

2.5 Aspect oriented programming

A programming paradigm called aspect oriented programming (AOP [22]) was developed to solve problems where a requirement or programming concern would cut across the boundaries of abstractions. A common example of this is the logging facilities of a code

base: you often want a centralized logging facility without burdening all your code with management of logging resources or infrastructure. In AOP these concerns are isolated into aspects. An aspect represents the concern to implement over the codebase and consists of advice code. Each piece of advice code is made up of a pointcut and executable code. A pointcut defines the join-points (locations in the codebase) that the advice applies to. At those points in the program the advice wraps the original code and has the ability to run behaviour before and after the original code. Additionally, the wrapping enables the advice to modify the inputs, outputs and execution of the original function. In **Figure 2.11** shows the sequence diagram where the execution of some (OOP) function is surrounded by two sections of advice code.

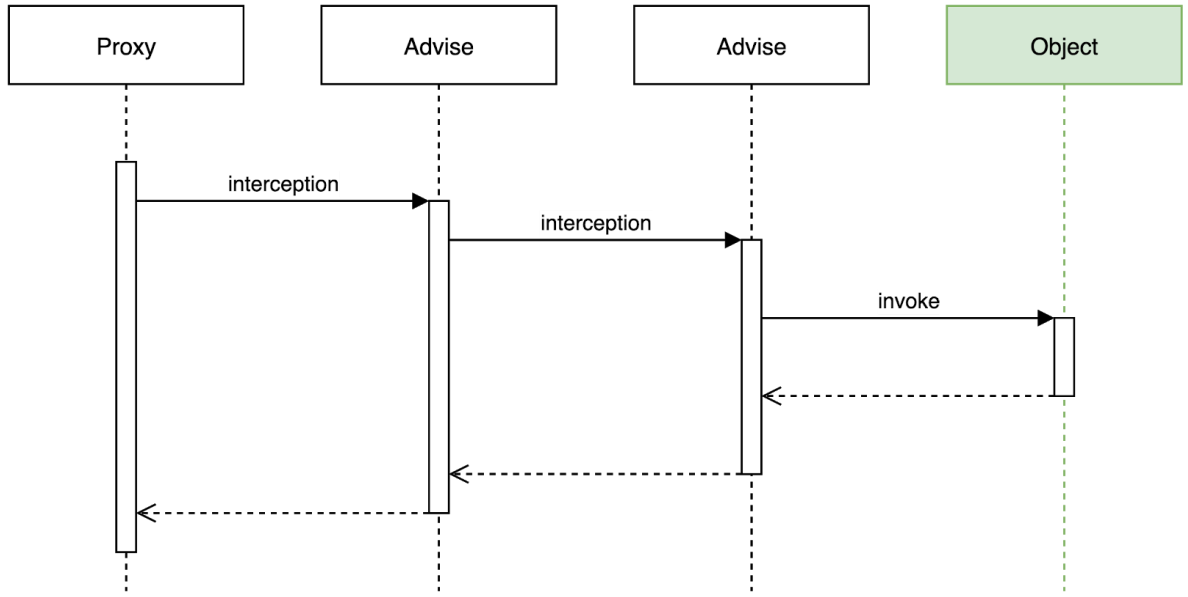


Figure 2.11: Example sequence diagram of an advised method call on some object.

3 Related work

GROOVE is an existing simulator and model checker that can be used when working with graph transformation systems. The editor built into the simulator is a user-friendly and powerful tool for prototyping and testing models rapidly. In terms of exploration capabilities there is a limited set of algorithms available, some can be configured using the rules from the grammar. From the exploration dialog (see **Figure 2.7-B**) the user is able to fine-tune the exploration behaviour to some extent. The underlying codebase could be extended to implement new algorithms but this would require extensive work for each algorithm added. The specialization of GROOVE in graph transformation systems does limit the types of input that the program is able to accept, however the capabilities of graph transformation systems to encode different types of problems is quite powerful. The work of GROOVE to provide a structure in which algorithm implementations are made is not part of the design, merely a by-product. Here we specifically build a new framework from the ground up with the goal of applicability for new and existing solutions alike.

PDDL4J is a code library that provides an easy to use interface for loading PDDL problem files into a java program. The library also includes a framework that implements some exploration algorithms out of the box. The framework supports a generic interface which allows its users to implement new exploration algorithms that solve PDDL encoded problems. The level of abstraction the framework provides is rather simple, users will have to implement new algorithms fully from the ground up and can only really reuse the pddl encoding representation already provided by the framework. That said, the framework does support some generalized heuristic encodings which helps reuse of heuristic functions across different algorithms. This work provides an even more generalized framework for building ‘solvers’ and exploration algorithms. The work on PDDL4J is (by design) tied to the domain of planning, whereas the Abeona project is not and deliberately does not tie itself to any particular domain.

Java PathFinder (JPF [5]) is a model checking tool that operates on java bytecode. It is highly specialized to the java runtime environment, but can be used to model check real java code instead of an abstracted representative model. The implementation uses a dedicated DFS algorithm to explore the state space of (abstracted) java programs. We do not believe that our work would benefit the java pathfinder nor that the JPF has much to offer to this project. We mention it as it is a model checking tool utilizing SSE, although it is heavily tied to the particular SSE strategy it uses.

NuSMV ([23], [24]) is a modular system for model checking. The modularity of the tool is focussed around modularizing the features of the tool, as opposed to our focus on modularizing parts of exploration algorithms. With the modularity of the tool their codebase is modularized at the level of model checking concepts, but doesn’t modularize the underlying concept of SSE.

LTSmin ([3]) is a toolset for generating and exploring state spaces for a wide range of input languages/models. The toolset is based on the conversion of the input languages to fixed sized integer vectors. These vectors are part of the so-called PINS model used throughout the toolset's architecture. In terms of exploration strategies, users can build their own 'algorithmic backends' which perform the exploration algorithm. Between the input converters (reading an input language to the PINS model) and the algorithmic backends the PINS-to-PINS layer resides. In this layer the PINS model is optimized before handing it off to the algorithmic backend for processing. To this extent, the LTSmin toolset offers a wide range of compatible inputs and optimizations. LTSmin and PDDL4J have in common that it offers a ready-to-use input model and state representation. This work lays focus on the expression and implementation of exploration strategies over providing a wide range of input languages out of the box.

In the field of DSE there exist several frameworks and toolchains ([25]–[27]) for automating the design space exploration process. The works in this area pertaining to generalization of those tools often result in model transformations to extend an existing tool to accept inputs from different model types ([28]–[30]). Model input transformation does allow users to utilize more tools for a given input model as well as alternate representations when using a specific tool. However, the generalization fails to provide them with a method to easily alter the existing exploration method to their needs. In the case of the Generic Design Space Exploration framework [28], their approach is based on transformation of input models to MiniZinc [31]. The MiniZinc (constraint problem) models can then be used by a user picked solver. This thesis provides a novel framework that offers solutions for SSE at a very generalized level.

Our work is focussed foremost on providing a generalized framework to implement SSE algorithms with, the novelty being the compositional approach used. To generalize the method we develop it such that it is state-agnostic and users can write their own input readers. With the focus on the actual flexibility of the exploration algorithm we aim to satisfy users who seek to create tailor-made solutions. We recognize that such users are likely to use an existing algorithm but require special optimizations not found in other tools or additional logic to integrate the solution in their environment. Our contribution will satisfy these users with a ready to use API; while the prior mentioned work may require the users to modify the code base or roll their own implementation from scratch.

4 Implementation

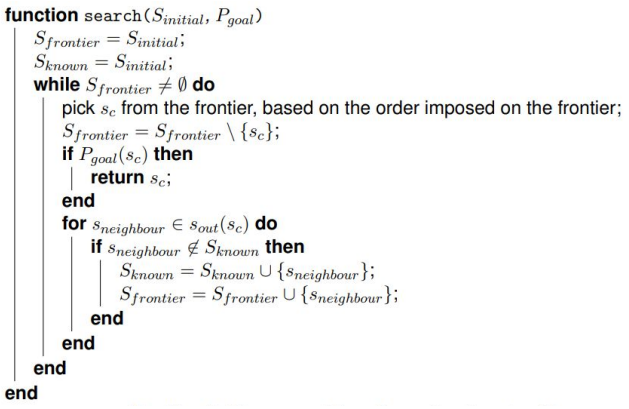
The purpose of this chapter is to detail the implementation of the work, specifically its design. First an analysis will be presented that was done before the main work was made, this analysis was crucial in discovering the requirements for the design. It also helped obtain the proper level of abstraction needed in the design. Second, we will explain the concepts that the framework consists of, as well as how they are put together to form the framework. Lastly, the design methodology is explained. This final part details how the concepts were implemented. In particular, the programming paradigms and techniques that were used are discussed.

4.1 Compositional analysis

Prior to thies thesis, an analysis was done on a series of SSE algorithms. These algorithms were BFS, DFS, Dijkstra's algorithm, A*, sweep-line, genetic algorithm and simulated annealing. They were chosen for being well-known SSE algorithms as well as being general enough to not warrant domain specific knowledge. For the genetic algorithm and simulated annealing the primary interest was to evaluate algorithms that are not directly SSE algorithms, but use a SSE algorithm as part of their functioning. This section reports the major findings and outcomes of the analysis. The full analysis and breakdown is included in **Appendix A**. The analysis focuses on finding the commonalities and variabilities among the algorithms. These findings allowed a common form to be derived that is general enough to support all of the algorithms.

The analysis resulted in a skeleton algorithm which contains the logic shared among all SSE algorithms. This skeleton algorithm is shown as pseudocode in **Figure 4.1**. This skeleton algorithm is not sufficient to perform the mentioned algorithms directly, some have special requirements and additional logic that needs to be added. The specifications for these algorithms is shown in **Figure 4.2**. The table shows a breakdown for each algorithm, listing the frontier order (F_o), presence of frontier constraints (F_c), frontier capacity (F_s), heap usage (heap), the list of user defined elements (user-defined) and finally the points at which custom logic should be inserted (aspects).

The design that follows from the skeleton algorithm and the breakdown utilized AOP to weave the custom logic into the skeleton algorithm. In the final implementation the skeleton is implemented as an event based pipeline.



```
function search(S_initial, P_goal)
    S_frontier = S_initial;
    S_known = S_initial;
    while S_frontier ≠ ∅ do
        pick s_c from the frontier, based on the order imposed on the frontier;
        S_frontier = S_frontier \ {s_c};
        if P_goal(s_c) then
            return s_c;
        end
        for s_neighbour ∈ s_out(s_c) do
            if s_neighbour ∉ S_known then
                S_known = S_known ∪ {s_neighbour};
                S_frontier = S_frontier ∪ {s_neighbour};
            end
        end
    end
end
```

Figure 4.1: Pseudocode for exploration algorithm skeleton, from **Appendix A**

Algorithm	F_o	F_c	F_s	**Heap**	**User-defined**	**Aspects**
BFS	FIFO	-	inf	req.	-	-
DFS	LIFO	-	inf	opt.	-	-
Dijkstra	f_{tc}	-	inf	req.	f_tc	eval-trans
A*	$f_{tc} + H$	-	inf	req.	f_tc, H	eval-trans
Sweep-line	f_p	-	inf	req.	f_p	state-picked
Genetic algorithm	-	X	1	-	c_{gen}, p_{cont}, p_{next}, f_{score}	state-picked
Simulated annealing	-	X	1	-	E	state-picked, post-eval

Figure 4.2: Algorithm breakdown table, from **Appendix A**

The analysis also outlined some concepts that are used in the design, mainly the notion of a ‘query’. A query is a structure which holds all relevant information about an algorithm’s composition, this includes the settings for the columns in **Figure 4.2**.

4.2 Design requirements

Before detailing the design we want to lay out what requirements and approaches were picked to be used during the actual design and implementation phases.

State-representation agnostic
In light of the various areas where SSE may be utilized we do not want to constrain the framework by using a specific state representation model. As such we want to leave the representation model up to the users of the framework.

Composition over inheritance
As the main goal of the research is to devise a framework where composition of algorithms is the main feature, we want to aim to compose as much of the framework as possible out of components. As such we will use the 'composition over inheritance' design/programming pattern as much as possible. The implementation of the Query (defined by **Appendix A**) will require a compositional approach regardless of the rest of the framework design.

Language choice
The language which the implementation will be made in Java, this was an external requirement in order to be able to implement the integrations. The choice for Java was also based on the historic precedent that academic work often utilizes java as object-oriented language to create implementations with. The JVM compatible language Kotlin was also considered, but dropped due to lack of experience of the author.

Open-closed principle
To make the framework as generic as possible we want to combine the principle of "open to extension, closed to modification" with a minimalistic API design. This results in building a foundation for the framework in java interfaces. The semantic difference between an abstract class and a java interface is that java interfaces do not restrict the inheritance interface. In programming literature the usage of inheritance is thought to create tight coupling between the classes and entanglement of their responsibilities.

The minimalistic design of the framework's public API is an effect of using the open-closed principle. The framework's public API should be minimal and restricted. For the users' convenience, extensions on this minimalistic API that are completely separate from the main API are created.

4.3 Framework design

Based on the analysis and the defined requirements, we now present the framework's design. First we will discuss how states and transitions are represented in the system. Then, we explain the underlying systems (events and pipeline) that the framework is built on. Next, we detail the components that make a query. Finally, we round it all up by explaining how the query implementation functions.

Throughout the following sections, UML diagrams are shown of the designed components and their relationships. The designs have been simplified for brevity and relevance.

4.3.1 States and transitions

In Abeona, a state is fully defined by the users, so there is no dedicated type or interface for states. In java, all classes ultimately inherit from the `java.lang.Object` superclass. Abeona uses this as the interface for states. Because of this, users of the framework are required to properly implement the `hashcode()` and `equals(Object)` methods. The hashcode function is required to be able to use hash based storage structures (e.g. `java.util.HashSet`, `java.util.HashMap`). The design of the next-function assumes that state instances are

immutable or at least that the next-function does not try to reuse existing instances by modifying them directly to apply mutations.

Transitions are implemented as a java class that simply holds some data. To enable users to define extra data on a transition it has a `userdata` field that can be populated by the next-function. The UML diagram for the state type and transition class are shown in **Figure 4.3**.

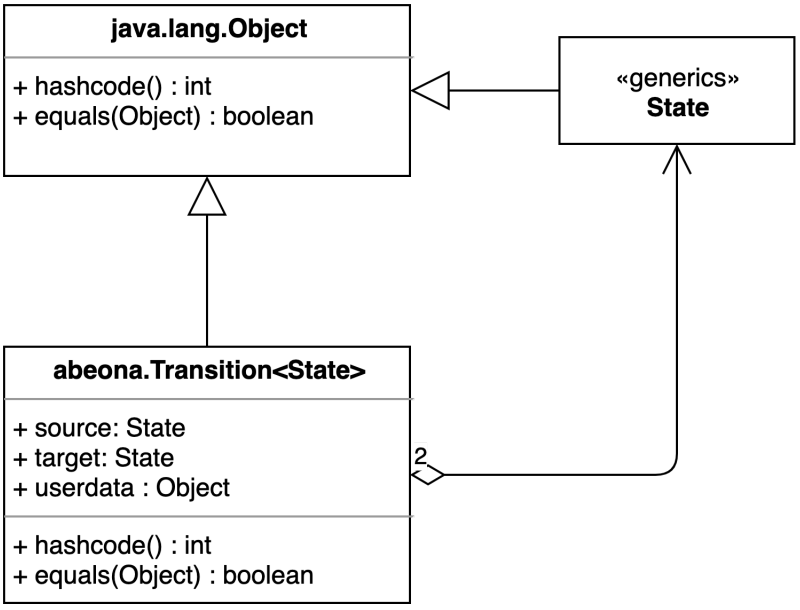


Figure 4.3: The UML diagram specifying the representations for states and transitions.

4.3.2 Events

The foundation of the flexibility the framework offers is drawn from event-based messaging and the “advice” concept from aspect-oriented programming (see **Section 2.5**). Events and advisable functions together form the ‘tappable’ event system. A tappable is a join-point where callbacks can be registered. These callbacks are where users can program custom behaviour for their algorithm. Events and advice tappables are divided because not all exposed parts of the pipeline have a meaningful piece of behaviour that can be advised. **Figure 4.4** shows the hierarchy of the different tappable interfaces. The advice tappable classes are based on the java functional interfaces in the `java.util.function` package. In the case of an event-type tappable this means a callback will be executed when a specific event occurs. This type of tappable does not grant access to the modification of code execution, but rather only to observe it. In other words, the events are intended to observe the behaviour of the system. Events can support feedback by the type of the data provided to the handlers. The advise-type tappables are not modeled after the event-listener pattern. Instead, they implement the advice concept from aspect-oriented programming.

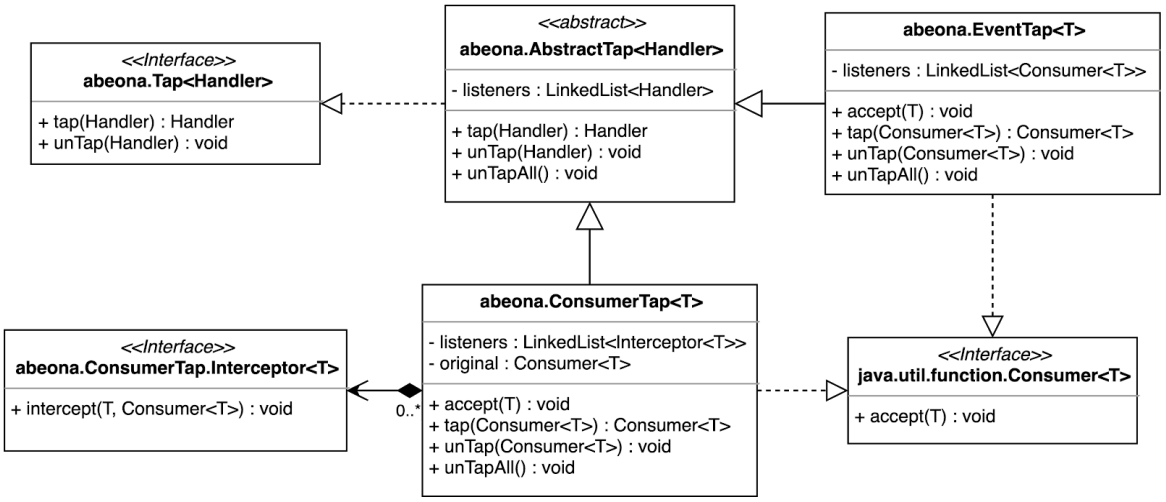


Figure 4.4: The UML diagram for part of the tappable system.

In **Figure 4.4** the EventTap class implements the event-type of tappables. Besides this there are several other taps, they implement various existing functional interfaces from the `java.util.function` package. Only the tappable that implements the `java.util.function.Consumer<T>` interface is depicted. The advice tappables implement a functional interface and wrap an existing function with the same functional interface. Registering handlers, called “interceptors”, advise the existing function. The interceptor’s signature is the same as the original functional interface, except that the interceptor receives one additional parameter. That parameter is an instance of the original function over which the interceptor gains execution control over. When multiple interceptors are registered on the tap, the extra parameter will be the next advice in the chain until the last advice receives the original function. The tappable system provides the flexibility and AOP features that are implemented in the pipeline and are reusable by other components to support AOP advising as well.

4.3.3 Pipeline

Thee the pipeline, which supports the exploration algorithms, is based on the work of [32, Ch. 3] and refined based on the findings from the analysis in **Appendix A**. A flowchart detailing the pipeline behaviour is shown in **Figure 4.5**. The red blocks containing “start” and “end” represent the start and end of the algorithm respectively. Blue blocks indicate an event and the text in the block identifies the type of event being dispatched. Gray blocks are regular flowchart blocks that represent program code or logic. Orange blocks represent advisable sections of behaviour, implemented with the advice-type of tappable classes. Finally, the red box marked “for each” represents a flow of logic which is executed for each item in the collection of transitions that is iterated. The gray circle at the top of the red box indicates the start of the loop and the gray circle at the bottom indicates the end of the looped logic.

The pipeline starts by emitting an event signalling the start of the exploration. Its purpose is to signal to registered behaviours to prepare for exploration, for example a behaviour that

needs to initialize or reset data before exploration. When the algorithm exits an event is dispatched that signals the end of exploration.

After the starting event has been emitted the main loop of the pipeline runs. The main loop implements the process that incrementally expands the set of known states. This loop continues as long as there are states in the frontier on which can be operated. In this loop, logic exists to catch any signals from events and behaviours to terminate exploration manually. In the case of a manual termination signal, the “exploration ending” event is still dispatched.

The main loop performs the following steps on each iteration: (i) choose the next state to evaluate, (ii) add the state to the heap and (iii) evaluate the next state. The next state is chosen by the frontier, though it is also an advisable function call. The second step of adding the state to the heap prevents it from entering the frontier when looping over the outgoing transitions. This would only occur when a self-loop is present. Before the next-function is invoked to generate the outgoing transitions the state is reported for evaluation with an event signal. The evaluation of a state involves looping over all outgoing transitions that the next-function generates. Each transition is reported with another event. This allows behaviours to augment transitions as soon as they have been created. The transition also grants access to the state to which it points. Subsequently, the target state of the transition is checked to be known. This is an advisable check which allows behaviours to impose restrictions on the explorable sections of the state space. If the state is not known (and should be discovered), the discovery event is set into motion, both reporting the transition that led to a new discovery, and inserting the state into the frontier. The insertion into the frontier is the final point at which behaviours can filter out a state from exploration. This concludes the logic to evaluate a single transition. After the for-each loop finishes, the end of evaluation of the state is signaled. After the end of the evaluation signal the main loop has finished an iteration and the frontier empty check is performed again.

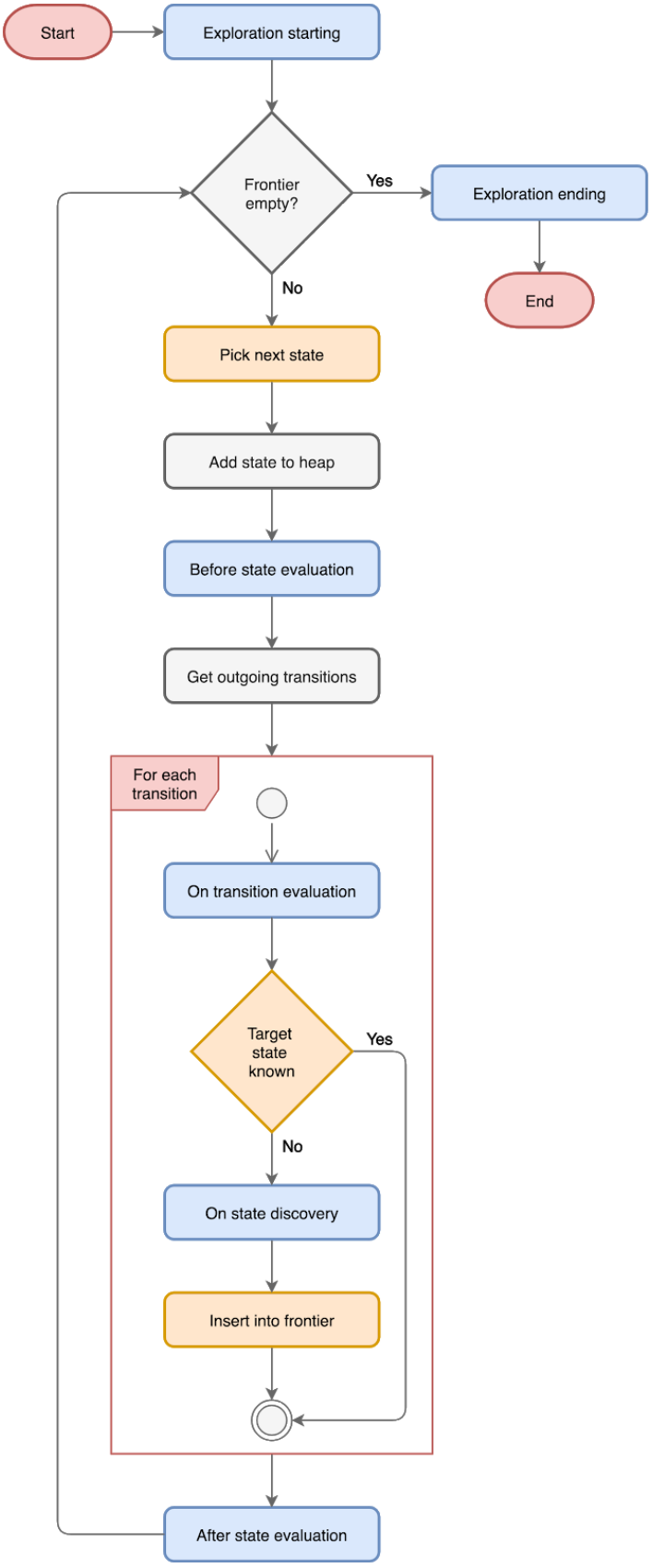


Figure 4.5: A flowchart representing the logic of the ‘exploration’ pipeline with markings for plain logic (gray), events (blue) and advice (orange).

The pipeline has two modes of operation, continuous and step-by-step. The continuous mode performs the logic as described here (and by the flowchart). The step-by-step mode exists for applications where the pipeline may only progress a single step at a time. In this case the inner logic of the main loop is performed once, the exploration signalling has to be managed manually by the user of the framework. This is possible as the design does not implement access controls for triggering events: any code with access to the tappable point can trigger an event.

4.3.4 Frontier

The frontier has two responsibilities: storage and ordering. Efficient storage and maintaining a particular ordering combines these requirements into a single interface. If they were dealt with separately, the implementations could not guarantee compatibility. A hierarchy of interfaces was created (shown in **Figure 4.6**) to capture different types of frontiers. There are different types of frontiers and as such it is most appropriate to capture the most essential functions in the base interface (`Frontier`). The base frontier has only two responsibilities: ingest more states and provide the next state to explore. For the first responsibility we added a method `add(Stream)`, which takes a `java.util.stream.Stream` instance. The implementation of the framework makes a lot of use of the `java.util.stream` package. This is mainly due to the desire to implement infinite-sized frontiers. This is also why the `Frontier` interface does not expose any way to reason about the states stored within the frontier, only whether the next state exists and what it is. To integrate well with the existing java framework, we had `Frontier` extend the `java.util.Iterator` interface as this captures the second responsibility well. To conclude the base interface, we added a method to remove all states from the frontier by. For frontiers that use finite sets of states we created the `ManagedFrontier` interface. This specialization grants more access to (and control over) the states contained within the frontier. A managed frontier is expected to be able to manage the states contained within. A further specialization of the `ManagedFrontier` is the `OrderedFrontier`, which specifies that implementations of the interface are to order the states contained within by some comparator. The comparator is also exposed from the interface. Abeona provides three different types of frontiers out-of-the-box: the `GeneratorFrontier`, `QueueFrontier` and `TreeMapFrontier`. The `GeneratorFrontier` implements only the `Frontier` interface and can be used as a frontier when the next-function creates an infinite stream of transitions for some states. This is possible because states are entered into the frontier as streams and these can be lazely iterated over. The `QueueFrontier` implements a frontier which stores its elements in a Deque (bi-directional queue) which allows for FIFO and LIFO access. The last frontier provided, the `TreeMapFrontier`, is a frontier which uses a TreeMap to hold an ordered set, the user specifies the `java.util.Comparator` to order the items by. As shown in **Figure 4.6** there is one more frontier interface, the `DynamicallyOrderedFrontier`, which is a specific type of frontier which needs to be used when the property that states are ordered by can change during exploration. One such algorithm is the A* algorithm, where the heuristic may lead to finding cheaper paths to known states.

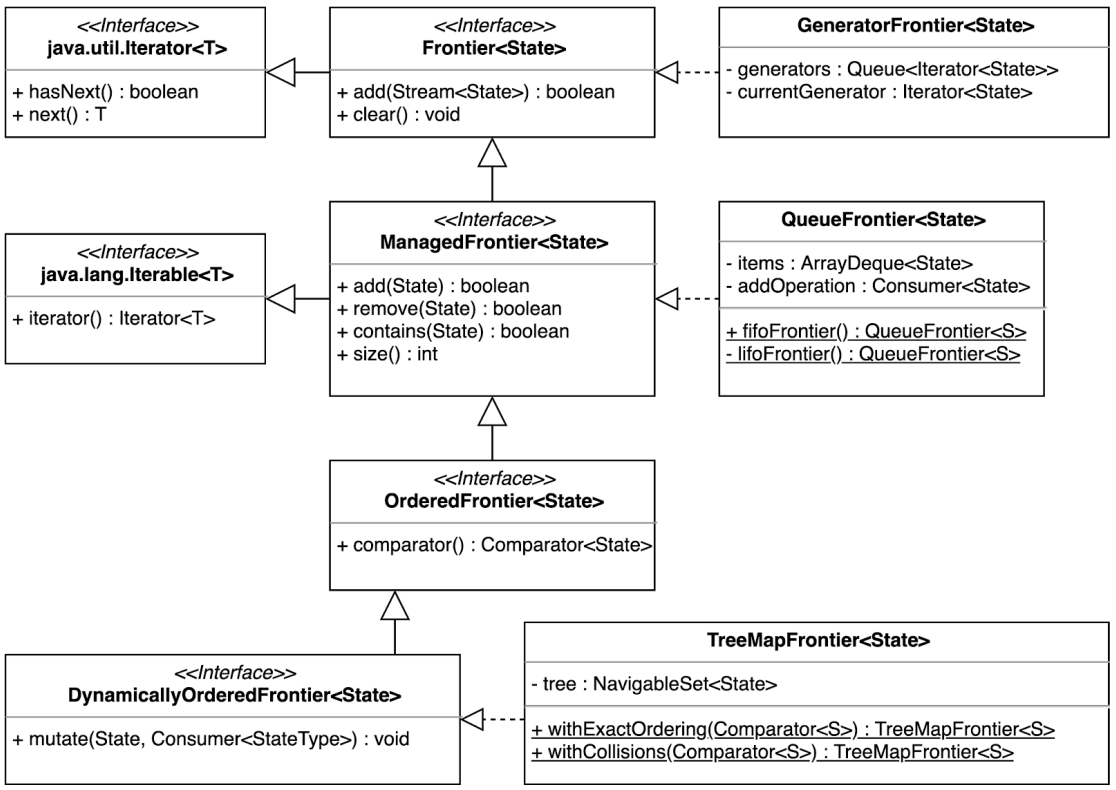


Figure 4.6: The interface hierarchy for frontiers, including the provided interfaces.

4.3.5 Heap

The heaps in Abeona are responsible for identifying known states and should not be confused with the term heap for program memory. We saw the need to support storage mechanisms which cannot reproduce the exact set of states contained within, for example where only hashes of the original states remain. As such we created a small hierarchy of heap interfaces, where the base interface (`Heap`) only supports adding a single state, testing a state for known-ness and clearing all known states. A UML diagram for the heap hierarchy is shown in **Figure 4.7**. The base interface does not support removing states since the interface does not guarantee that the implementation is able to manage or recognize individual states. The recognition aspect can be explained by thinking of a hashing-heap as a possible implementation of the interface. A hashing heap would hash any state added to it and store only the hashes (integers). Such a heap would not be able to provide the remove-operation reliably. For heaps that use storage methods that support management (and iteration) we provide the `ManagedHeap` interface. The two heap implementations that Abeona provides are `NullHeap` and `HashSetHeap`. The former acts as a 'black hole' which never contains any states. The latter is a heap using a `java.util.HashSet` for storing the state instances (and identifying them).

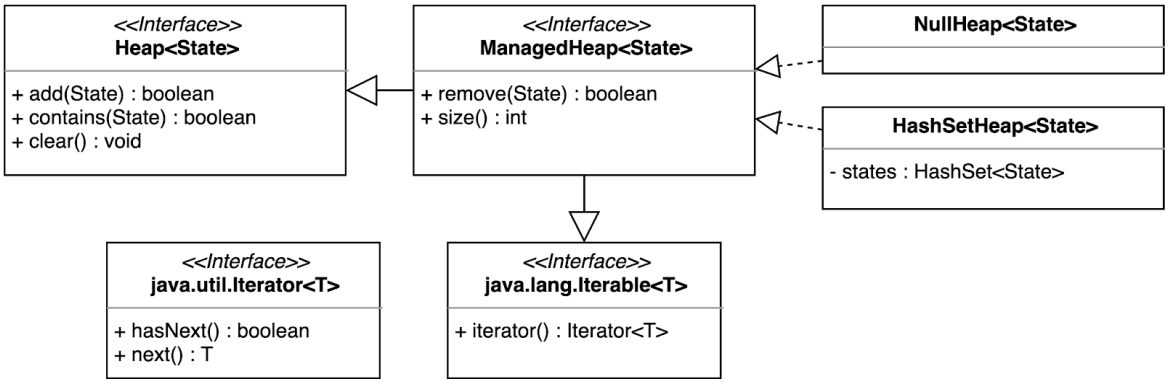


Figure 4.7: The UML diagram showing the heap type hierarchy in Abeona

4.3.6 Next-function interface

The interface of the next-function is a simple functional interface, the function receives a single state as input and is expected to produce a stream of (outgoing) transitions from that state. We added a convenience helper function that generates a next-function when given a function that produces streams of (neighbouring) states for a given state. This is for users who do not need to insert userdata onto transitions. In **Figure 4.8** the relationship between the `NextFunction` interface and the `java.util.function.Function` interface.The user can define a next function by implementing the interface as a class or through a lambda-expression.

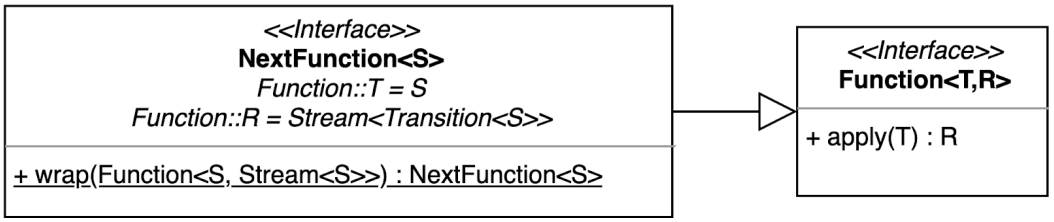


Figure 4.8: A UML diagram showing the structure of the NextFunction interface

4.3.7 Query

Based on the work of **Appendix A**, the `Query` class was designed as a means to perform the composition of algorithm components and behaviours. Besides the composition of components the query also is responsible for implementing the pipeline. The relationship of the `Query` class is shown in **Figure 4.9** with a UML diagram. In the diagram the generic parameter `S` represents the java class for states in the state space.
In the construction of the `Query` the frontier, heap and next-function components need to be provided. After this the query is able to receive additional behaviours through the `addBehaviour()` function. While the behaviours on a query instance can be added or removed, this is not possible for the other types of components (frontier, heap and next-function). A query instance is unable to perform a SSE when any of these components are not present. As such the API has been constrained not to allow for such a query instance to exist. The purpose of the query class and why it encompasses both the composition and the pipeline is that it is able to represent and execute an exploration algorithm. The query

represents an exploration algorithm by the composition of components it holds and it is able to execute that algorithm through the implemented pipeline.

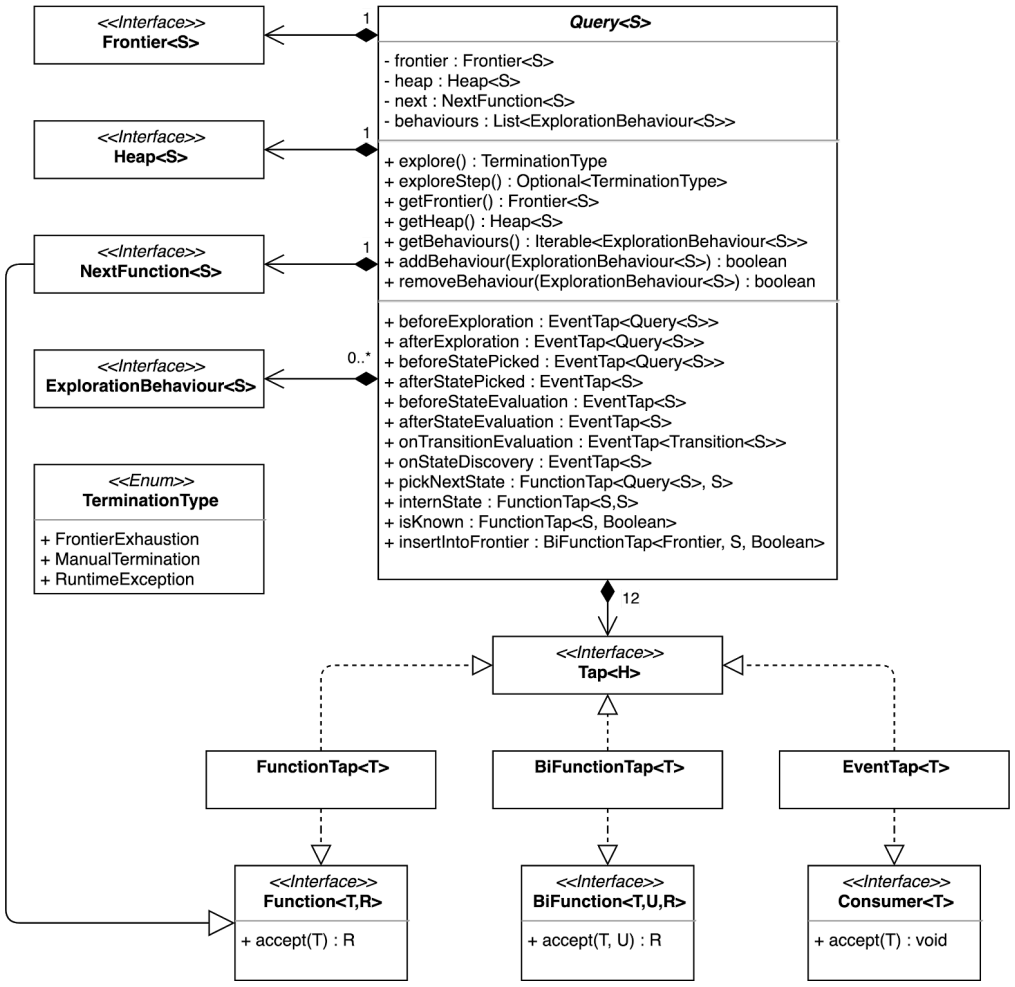


Figure 4.9: The UML diagram of the Query together with its tappable points and relationship to other components.

4.3.8 Behaviours

With the exposed join-points and events on the pipeline there already is sufficient flexibility to implement the algorithms we analyzed as well as additional requirements a user may have on an exploration strategy. However, this is not yet presented in a modular fashion. Composing a query this way would mean each query would have the behaviour of the algorithm manually attached to it as well as any changes required for one or more optimizations the user wants to use. Modularization of the algorithm requirements as well as for each optimization upholds the compositional approach and hides away the implementation details of such ‘components’ into the module. To do this we introduce behaviours, these are parallel to aspects from AOP, as packages of modifications to the pipeline. We defined the base interface of a behaviour to be able to attach and detach to the pipeline. This allows for the ‘module’ to be ‘installed’ and ‘uninstalled’. Behaviours are intended to use the tappable system to install the required callbacks into the pipeline. A behaviour is a component or module that packages all changes to a query instance necessary to implement the intended behaviour.

In **Figure 4.10** the anatomy and relationships of behaviours in the framework is shown, we only show the built-in `IterationCounter` behaviour as part of the hierarchy, but in reality there are 10 more. The block representing the `Query` class is marked as “partial”, the complete representation has already been shown in **Figure 4.9**. The `IterationCounter` behaviour is based on the `AbstractBehaviour` class (a class implementing abstracting registration/unregister logic on query taps). What it does is track a counter for every query it is attached to and track how many main-loop iterations there have been in the pipeline.

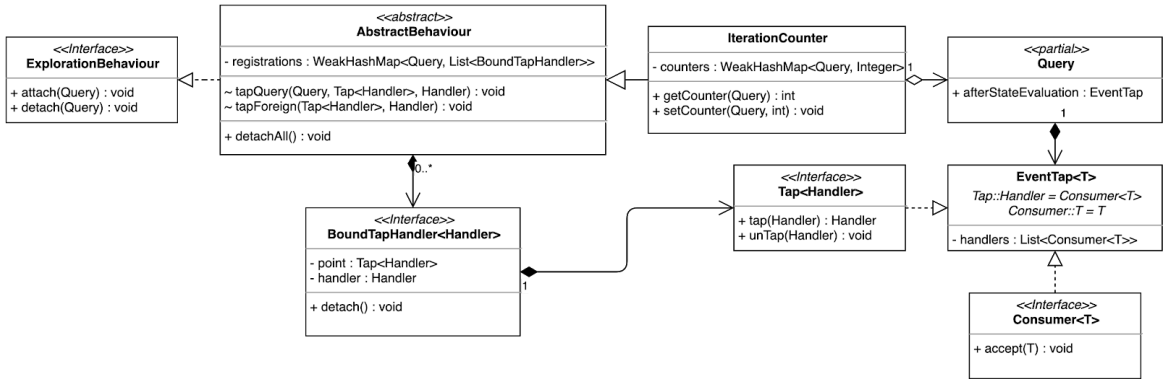


Figure 4.10: A UML diagram showing how behaviours fit into the Abeona framework.

4.4 Optimizations

During the implementation of the framework design optimizations were made preemptively to ensure that the resulting framework implementation has an adequate baseline performance. Data structures were chosen which had optimal time and memory complexity depending on the most common operations of those data structures. An example of this is the utilization of linked lists in the tappable framework, as the most common operations during runtime are addition and iteration on the list of handlers.

As a memory optimization the ability to lazily evaluate transitions was added. For this, the execution of the pipeline was optimized, so that it operates entirely on `java.util.stream.Stream` instances. This approach adds support for infinite streams of outgoing transitions. Stream instances are lazily evaluated during iteration. The `GeneratorFrontier` stores a set of streams and iterates them lazily allowing infinite streams to exist within the frontier.

5. Validation

As part of the validation of our claims for this project we set out to verify specific aspects of the framework in a series of tests. Four types of tests were conducted, each discussed in a separate section of this chapter. **Section 5.1** details performance comparisons between the Abeona framework and existing solutions. **Section 5.2** shows the modularity of the compositional API by comparing query compositions in different problem contexts. **Section 5.3** goes over the existing demo applications which highlight a particular feature of the framework. Finally, **Section 5.4** shows the setup and results of the user studies that were conducted.

5.1 Performance

This section details the results that were collected while testing the framework against other SSE implementations. The framework was compared with three different implementations:

1. A comparison of the framework against a plain implementation of the exploration algorithm in java.
2. A comparison of strategies in GROOVE against implementations of the same strategies with Abeona, inside the GROOVE application.
3. A comparison of strategies in PDDL4J against implementations of the same strategies with Abeona, within the PDDL4J framework.

5.1.1 Maze

A maze-demo is included as mazes are an easy to understand problem that can be implemented using a graph-based representation (see **Figure 5.1**). For the demo application, the graph of the maze is used as a state space and the exploration is used to find a solution trace between a starting and goal position (state) in the maze. The mazes in the demo are encoded as orthogonal cell grids in which each cell can have walls on each of its four sides. The absence of a wall between two adjacent cells implies a move is possible between these two cells. The player state is minimal and only needs to refer to the position in the maze the player currently occupies. The class diagram for the representation of the maze and state space states can be seen in **Figure 5.2**. The maze demo is also used in **Section 5.2.1** and **Section 5.4**.

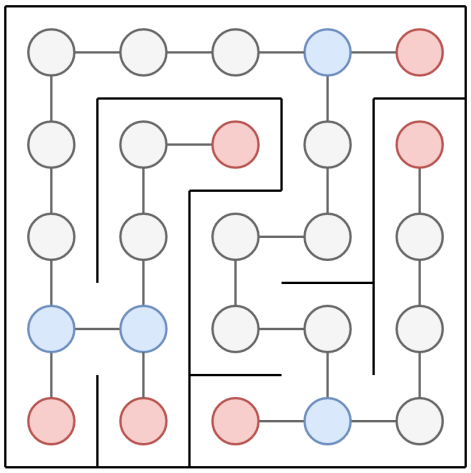


Figure 5.1: A small maze and its graph representation. The nodes are color coded to visualize different types of connections in the maze.

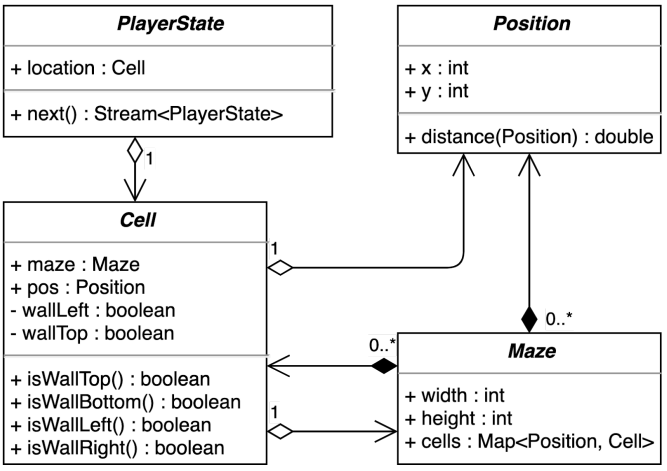


Figure 5.2: The UML diagram for the object-oriented representation for the maze (`Maze` class) and the states in the state space (`PlayerState` class)

The performance was recorded for Abeona queries and plain java implementations of BFS, DFS and A* algorithms. The results for BFS, DFS and A* are shown in **Figures 5.3-5.5** respectively. Each implementation solved square mazes of sizes 8, 16, 32, 64, 128, 265 and 512, in which the size referred to is the length of one side of the maze. The queries and plain implementations were made to terminate when finding the exit of the maze. Each graph shows in blue the performance of the plain java implementation and in red the performance for an Abeona query. In the case of A*, the plain implementation executed in under 0.1ms for the smallest maze, this is the only entry where no data is shown in the graph.

The results from the figures clearly show that there is a constant overhead when using the framework. To some extent this is to be expected as the ‘handcrafted’ code is close to the minimal code required to perform the search and the framework executes additional code for the event and pipeline system. In the case of BFS the handcrafted solution performs significantly worse in the same conditions. We do not clearly know the reason for this, but suspect that slight differences in the implementations may have caused this. The interaction the algorithms have with the frontier and heap are not matched 1 to 1.

The implementations of BFS and DFS for the handcrafted benchmarks are nearly identical and we suspect this is why the DFS results also show a worse performance for the handcrafted method. This said, we were not able to discover the specific reason why the last case has better performance than its predecessors.

In the results of A* the overhead of the behaviour added to implement the algorithm has a distinct overhead compared to the handcrafted solution. For the benchmarks with maze sizes 64 and up the overhead gets a stable offset in the graph. This means that the performance of the Abeona query is a factor slower than the handcrafted method. The overhead seems to be linked to the number of operations, indicating that the implementations are close or equal in time-complexity (with regards to the input size).

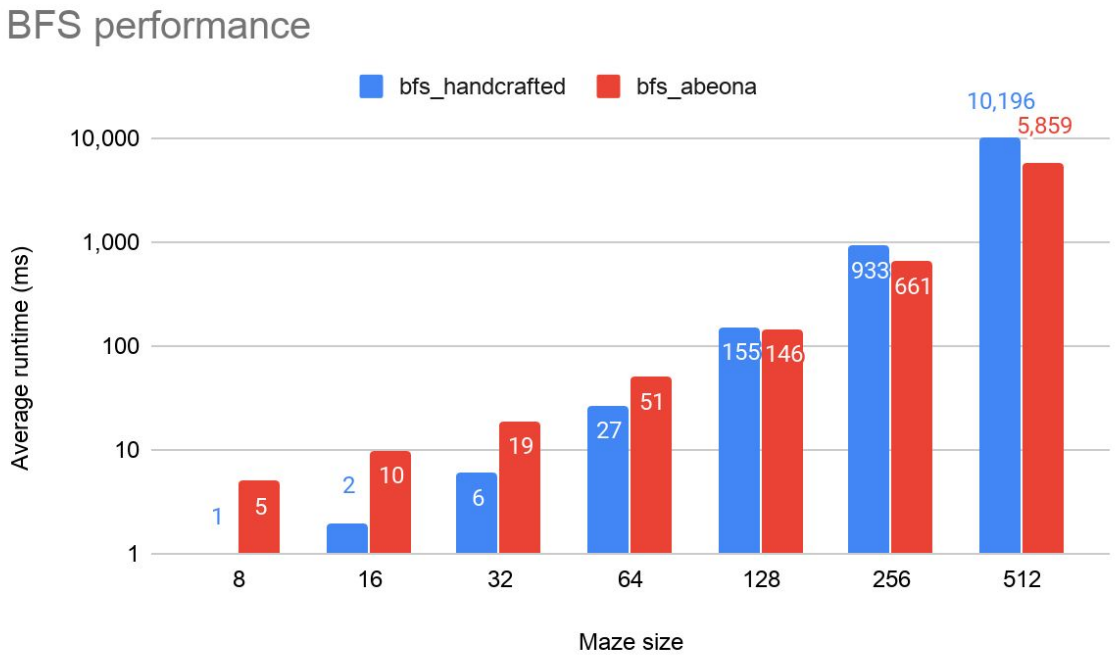


Figure 5.3: Runtime comparison of a handmade BFS algorithm (blue) against a composed query in Abeona (red) for various sizes of mazes.

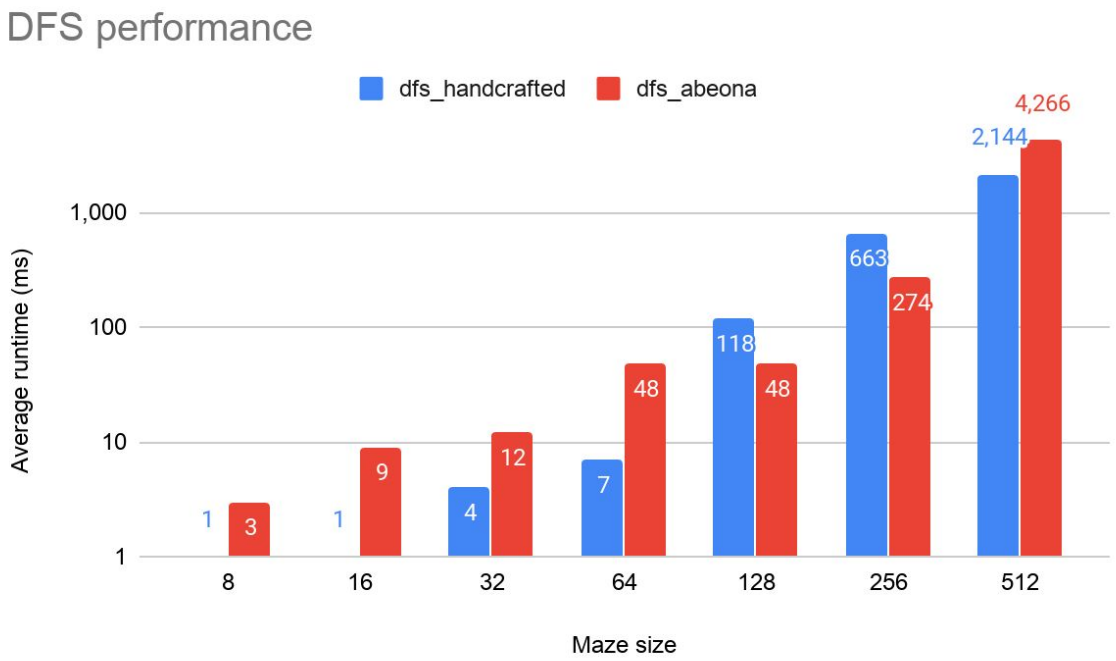


Figure 5.4: Runtime comparison of a handmade DFS algorithm (blue) against a composed query in Abeona (red) for various sizes of mazes.

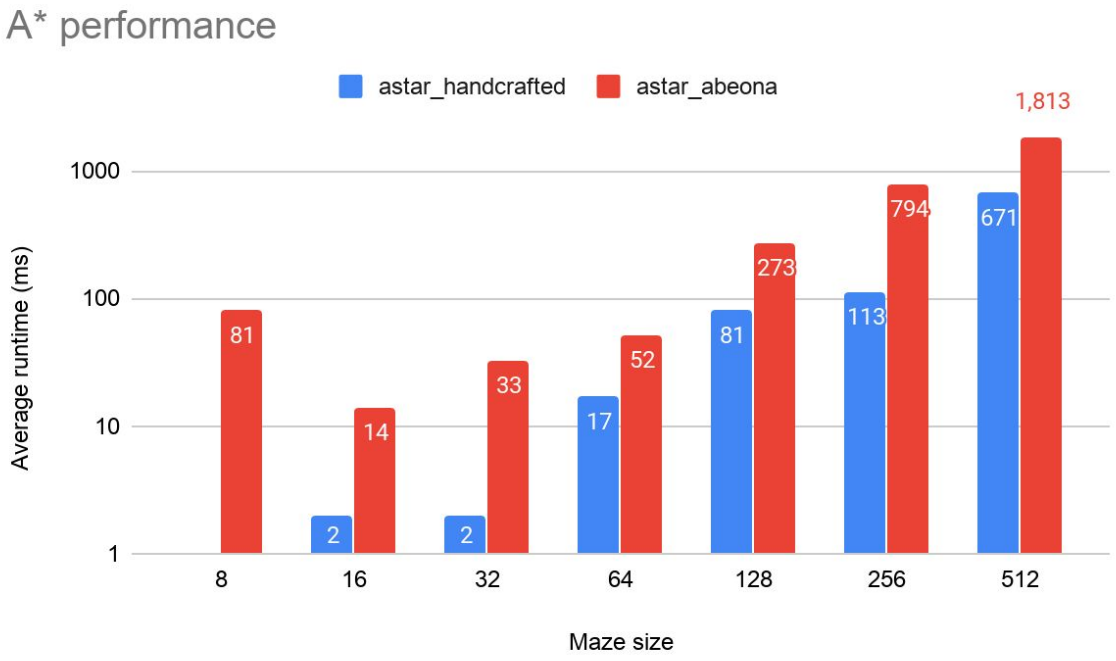


Figure 5.5: Runtime comparison of a handmade A* algorithm (blue) against a composed query in Abeona (red) for various sizes of mazes

5.1.2 GROOVE

The GROOVE integration of Abeona supports the BFS and DFS search strategies, which GROOVE implements as well. The comparison of these strategies is useful to highlight the runtime cost introduced by Abeona in the GROOVE integration. The runtime for these search strategies were measured and are shown here to compare. The “leader-election” grammar that tested the exploration algorithms is available from the GROOVE downloads page[33]. The grammar implements a leader election algorithm for a group of nodes of variable size. For this grammar, six variations were made, with the number of communicating nodes ranging from 1 to 6. The results for BFS and DFS explorations can be seen in **Figure 5.6** and **Figure 5.7** respectively. The results were collected over five execution runs and averaged out. For the BFS strategy there is a noticeable increase in runtime in the last two tests. Upon further inspection of the execution, we found a drastic increase in the number of transitions discovered by the Abeona implementation.

The DFS results do not significantly deviate between the two implementations, they seem to be on-par in performance and time-complexity.

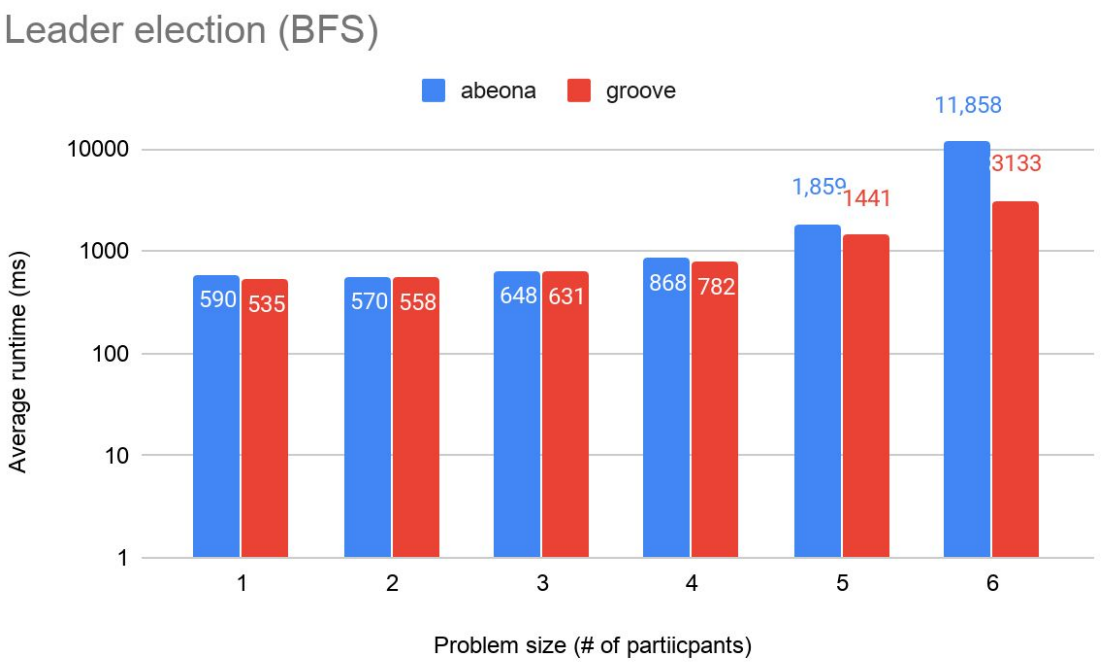


Figure 5.6: The runtime results for BFS exploration performed with existing GROOVE algorithms and the Abeona integration.

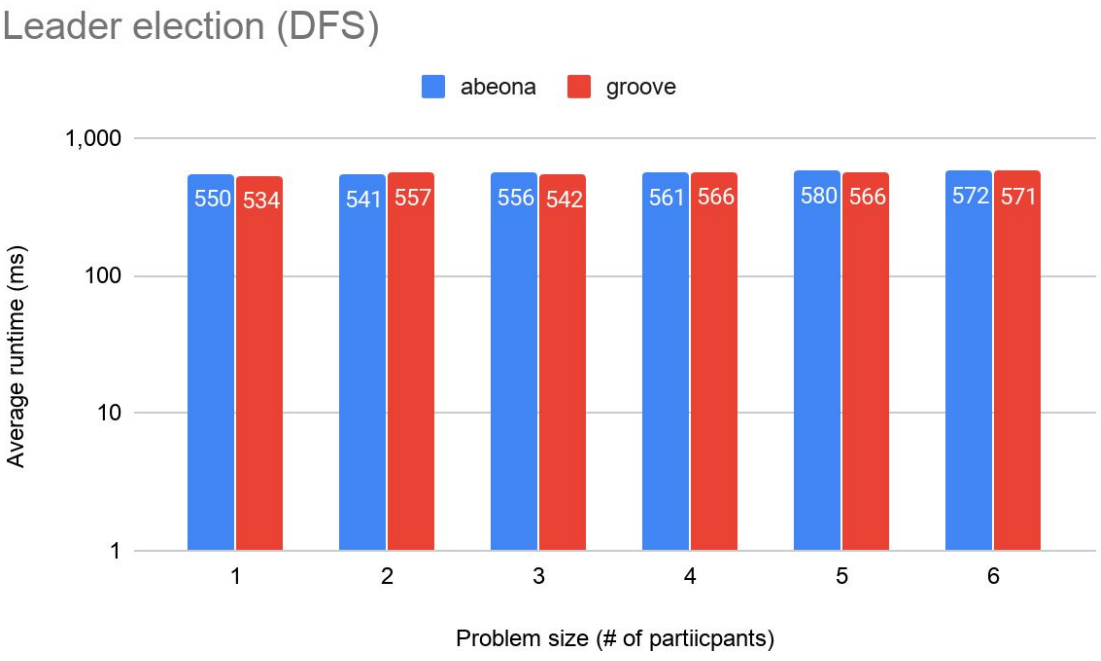


Figure 5.7: The runtime results for DFS exploration performed with existing GROOVE algorithms and the Abeona integration

5.1.3 PDDL4J

The integration with PDDL4J consisted mostly of ‘glue’ code to enable an Abeona query to function within the PDDL4J framework. In **Section 5.3.4** there is a detailed explanation on the bridging ‘glue’ code. The performance comparison of Abeona to PDDL4J consisted of benchmarking the available search strategies in PDDL4J to equivalent search strategies in Abeona. The PDDL4J library provides implementations for the BFS and DFS algorithms. The PDDL4J strategies and the Abeona queries used early termination when finding a suitable plan.

The PDDL4J source code repository[34] includes PDDL files encoding some well known problems. Problems from the ‘blocksworld’, ‘depots’ and ‘gripper’ domains were measured for both the PDDL4J and Abeona implementations for the BFS and DFS search strategies. The average runtime is calculated over five runs, and each run is given a maximum runtime of three minutes. The results of the tests are shown in **Figure 5.8 (A through F)**. The output of the PDDL4J strategies indicated that some executions did not result in a plan. Review of the PDDL4J source code revealed that the search strategies only explore a subset of all outgoing transitions from any given state.

In the blocksworld results the differences between Abeona and PDDL4J have a close relationship where the performance of Abeona seems to be linearly tied to that of PDDL4J. For the last two cases however, this characteristic swaps around. In these two problems the initial setup is ‘more complex’, leading to more options to explore early on.

The problem domains of “depots” and “gripper” suffer heavily from the state space explosion problem. We attempted to also solve the “depots-3” and “gripper-5” problems, but did not receive a solution within three minutes. For the gripper problem the last case did not yield an actual plan while our Abeona implementation did, this was due to the aforementioned shortcoming in the PDDL4J implementations.

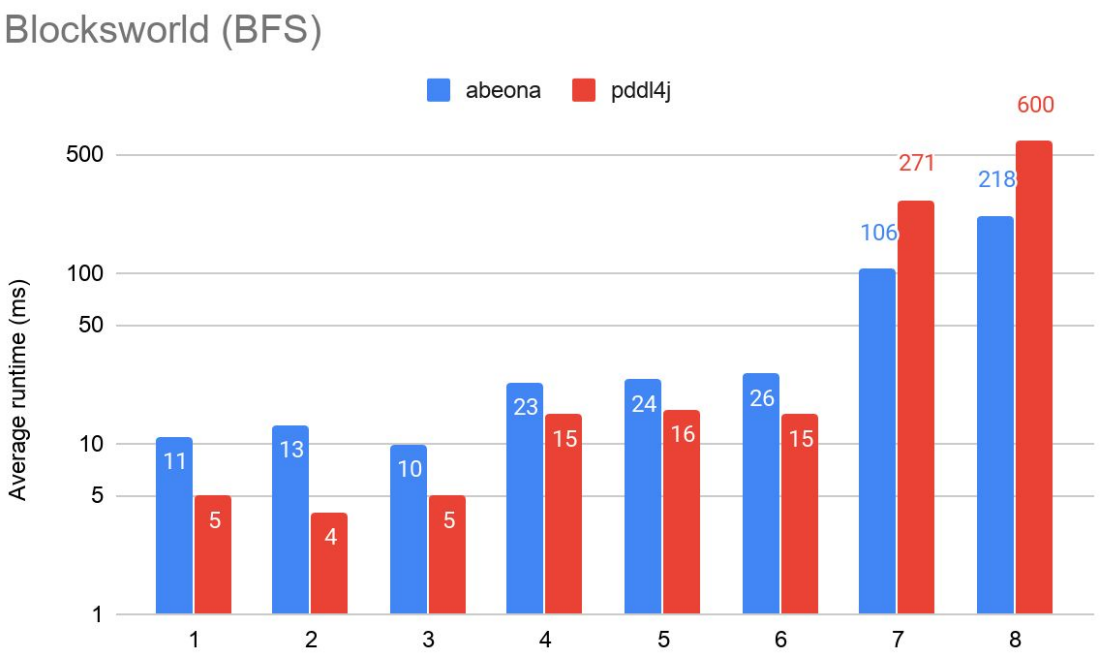


Figure 5.8-A: The runtime results for BFS implementations on blocksworld problems

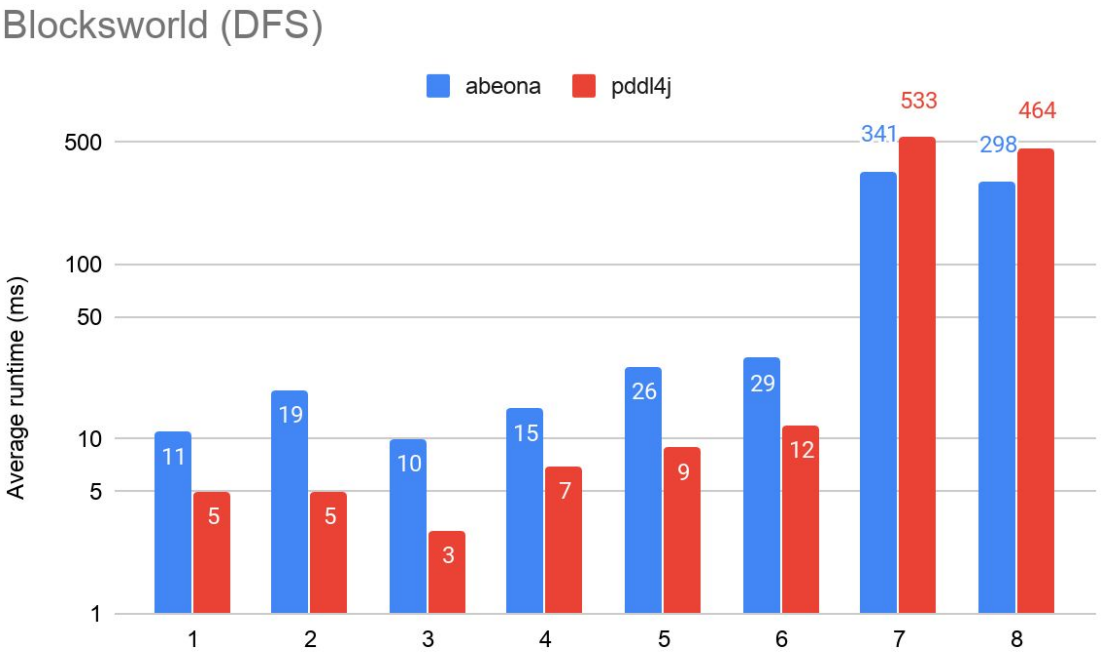


Figure 5.8-B: The runtime results for DFS implementations on blocksworld problems

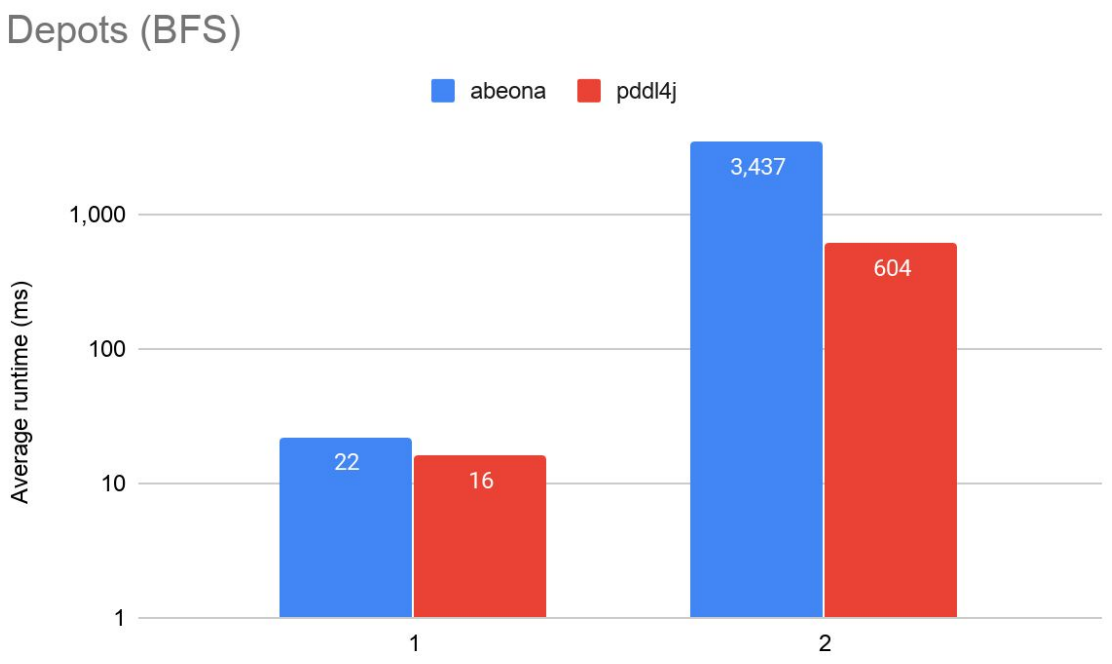


Figure 5.8-C: The runtime results for BFS implementations on depots problems

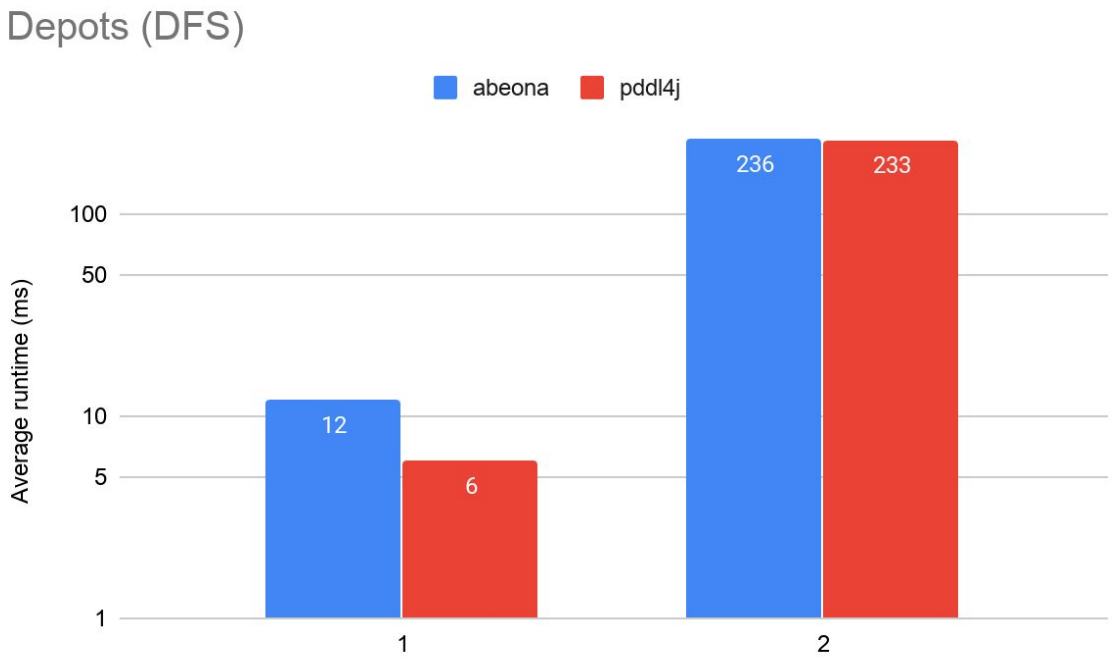


Figure 5.8-D: The runtime results for DFS implementations on depots problems

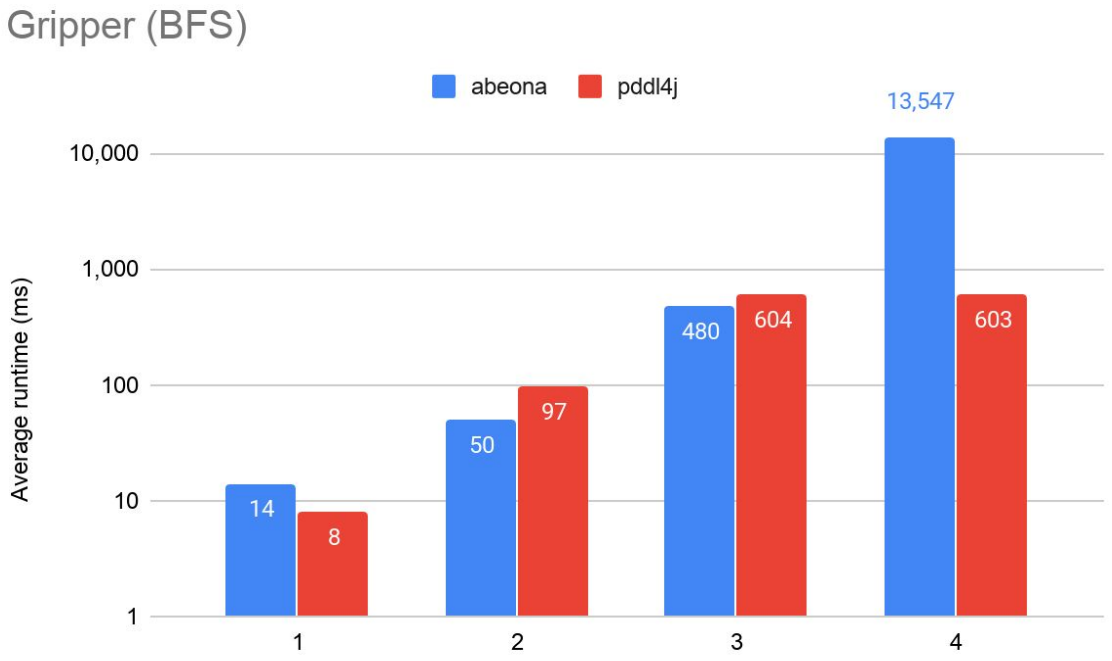


Figure 5.8-E: The runtime results for BFS implementations on gripper problems

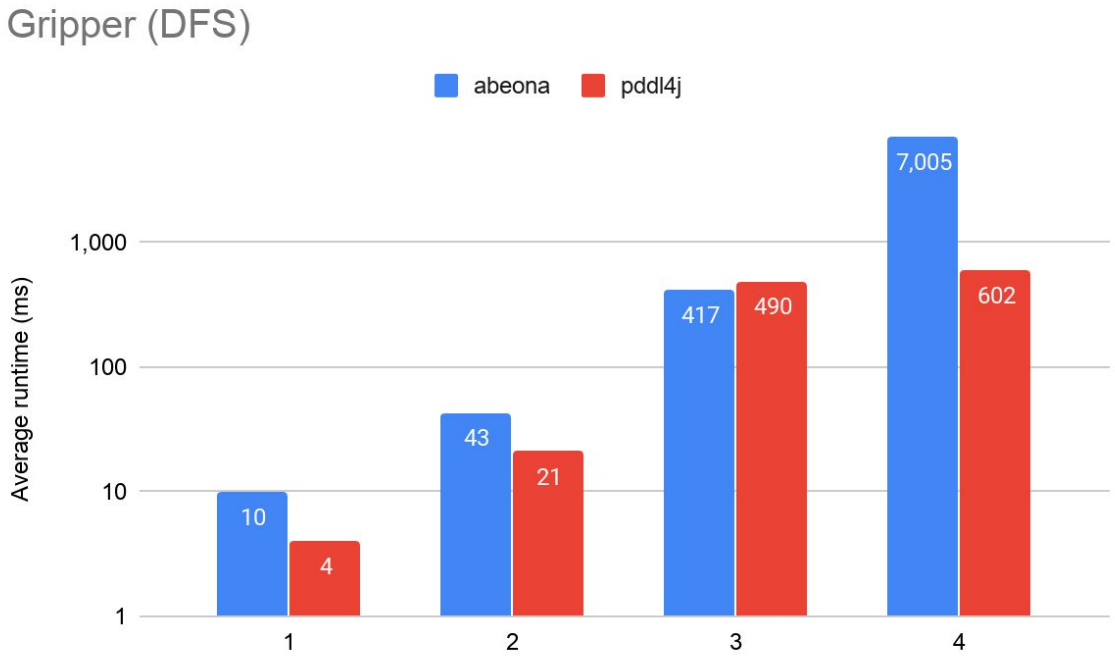


Figure 5.8-F: The runtime results for DFS implementations on gripper problems

5.2 Modularity

The modularity of the framework will be demonstrated over the next few subsections. Changes to queries are shown as the requirements change. Three problems are discussed:

the maze demo (from **Section 5.1.1**), the knapsack problem (a well-known combinatorial problem) and the PDDL4J integration.

5.2.1 Maze

The maze demo was introduced in **Section 5.1.1** already, refer to that section for the technical details on the state representation. To demonstrate the modularity, a query that performs a basic exploration is changed by introducing two requirements. For each requirement the changes to the query composition are shown.

Given the initial composition for a BFS algorithm (see **Figure 5.9**), the query continues exploration until the frontier is exhausted. There are problems where the algorithm can end exploration earlier, for example mazes for which the path to the exit is the desired result. In the demo implementation, there are two settings available to retrieve the goal state position (`END_X` and `END_Y` for the xy-coordinates). The `TerminateOnGoalStateBehaviour` that comes with the Abeona framework can provide the behaviour necessary, it terminates the exploration when state is discovered that satisfies a given predicate. The changes to include this new termination process are shown in **Figure 5.10**, with the changed code highlighted for clarity.

```
static Query<PlayerState> createQuery() {
    final Frontier<PlayerState> frontier = QueueFrontier.fifoFrontier();
    final Heap<PlayerState> heap = new HashSetHeap<>();
    final NextFunction<PlayerState> next = NextFunction.wrap(PlayerState::next);
    final Query<PlayerState> query = new Query<>(frontier, heap, next);
    return query;
}
```

Figure 5.9: The composition of a BFS algorithm

```
static Query<PlayerState> createQuery() {
    final Frontier<PlayerState> frontier = QueueFrontier.fifoFrontier();
    final Heap<PlayerState> heap = new HashSetHeap<>();
    final NextFunction<PlayerState> next = NextFunction.wrap(PlayerState::next);
    final Query<PlayerState> query = new Query<>(frontier, heap, next);
    query.addBehaviour(new TerminateOnGoalStateBehaviour<>(state -> {
        final var pos = state.getLocation().getPos();
        return pos.getX() == END_X && pos.getY() == END_Y;
    }));
    return query;
}
```

Figure 5.10: The composition of a BFS algorithm, that terminates early

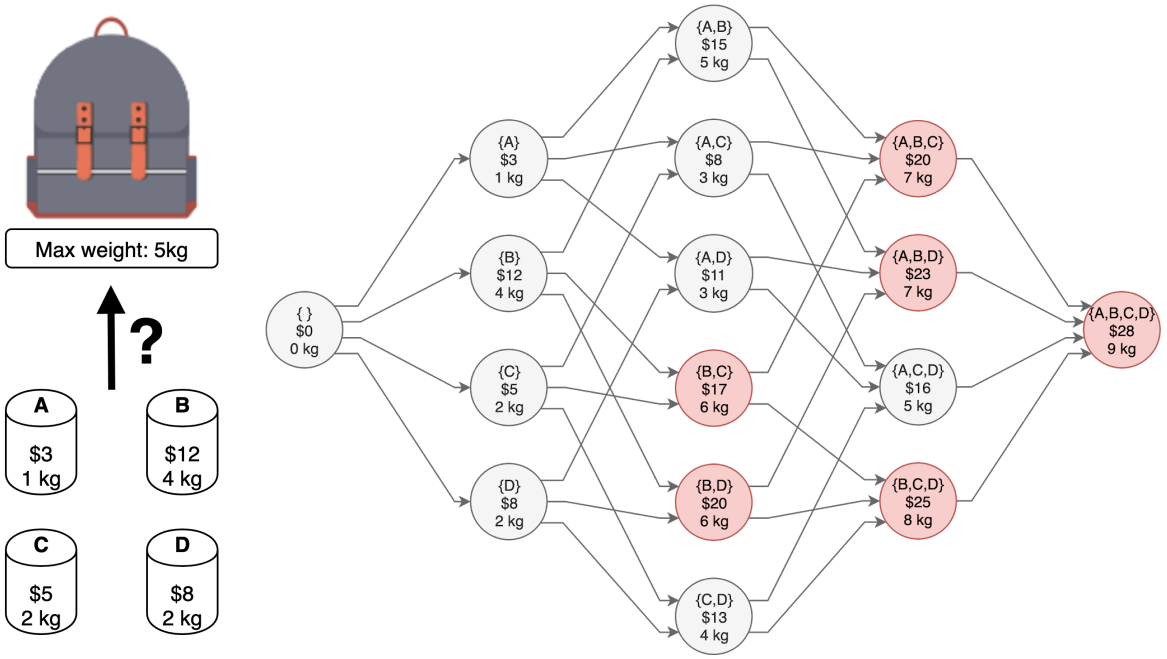
The second change (shown in **Figure 5.11**) is to turn the algorithm into a greedy best-first algorithm which uses a heuristic to order the frontier. In this case the heuristic is based on the horizon distance to the goal, favoring states closest to the goal state. Notably, this change does not affect the previously introduced `TerminateOnGoalStateBehaviour` behaviour.

```
static Query<PlayerState> createQuery() {
    final Frontier<PlayerState> frontier = TreeMapFrontier.withCollisions(
            Comparator.comparingDouble(state ->
                    state.getLocation().getPos().distance(new Position(END_X, END_Y))
            ),
            PlayerState::hashCode);
    final Heap<PlayerState> heap = new HashSetHeap<>();
    final NextFunction<PlayerState> next = NextFunction.wrap(PlayerState::next);
    final Query<PlayerState> query = new Query<>(frontier, heap, next);
    query.addBehaviour(new TerminateOnGoalStateBehaviour<>(state -> {
        final var pos = state.getLocation().getPos();
        return pos.getX() == END_X && pos.getY() == END_Y;
    }));
    return query;
}
```

Figure 5.11: The composition of a greedy algorithm, that terminates early

5.2.2 Knapsack

To touch into the domain of combinatorial problems and dynamic programming, a demo for the knapsack problem was implemented. In this problem a knapsack needs to be filled with some items. Each item has a specific weight and value. The knapsack has a maximum weight it can carry. The problem asks to find the combination of items with the maximum value without exceeding the knapsacks weight limit. We visualize the problem in **Figure 5.11-A** with a knapsack (backpack) that has a maximum weight of 5kg and 4 items with various weights and values. The problem is represented by the contents of the knapsack and this combination of items is mapped to a state space. The transitions between the states represent adding one item to the knapsack. The resulting state space for the example problem is shown in **Figure 5.11-B**. In this figure the states exceeding the knapsacks weight limit are marked in red.



A: The problem context.

B: The corresponding state space, with illegal states marked red. Each state lists the items, total value and total weight.

Figure 5.11: An instance of the knapsack problem, visualized.

The state representation used in the implementation is shown in **Figure 5.12**. The problem context is implemented in the `KnapsackPuzzle` class, which just holds all global information about the problem. The context of the problem is immutable and will not change between states. This allows for the instance to be referenced by the state instances, instead of having to replicate the data onto every state. The states store the items put into the knapsack along with the reference to the puzzle context.

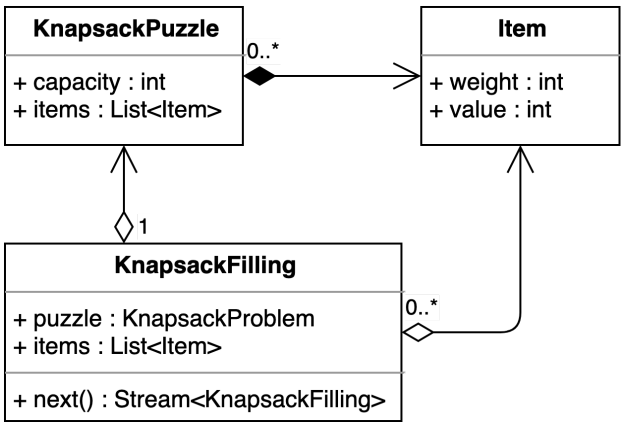


Figure 5.12: The classes implementing the state representation for the state space

To explore the state space, a simple BFS query, such as shown in **Figure 5.13**, can be used. For a more optimized exploration the specialized sweep-line algorithm is suitable for

this case. Sweep-line allows for the heap to be cleared of states that are presumed to not be encountered anymore. A heuristic progress function is used to make this possible. The heuristic function defines the level of progress of the state within the state space. The meaning of this level of progress depends on the context of the state space and is user-defined. The assumption made for sweep-line is that traces through the state space tend not to contain transitions in which the target state has a lower level of progress than the source state.

For the knapsack problem, the state space shown in **Figure 5.11-B** has aligned the states horizontally in "layers". Using this particular encoding, each "layer" in the knapsack problem state space only leads to states in the next layer, never to a previous one.

In the sweep-line algorithm, the states are explored by lowest progression first. A state can be "forgotten" and thus removed from the heap whenever it has a lower progression than any state in the frontier. Sweep-line also has a safety mechanism in case a transition is evaluated that leads to a lower progression state, but this mechanism does not get used for this particular state space. The changes needed to turn the BFS query (**Figure 5.13**) into a sweep-line implementation is shown in (**Figure 5.14**) for clarity on the changes made are highlighted yellow. In the code shown here, the `SweepLineBehaviour` class performs the required modifications to the pipeline for implementing the sweep-line algorithm.

```
static Query<PlayerState> createQuery() {
    final Frontier<KnapsackFilling> frontier = QueueFrontier.fifoFrontier();
    final Heap<KnapsackFilling> heap = new HashSetHeap<>();
    final NextFunction<KnapsackFilling> next = NextFunction.wrap(KnapsackFilling::next);
    final Query<KnapsackFilling> query = new Query<>(frontier, heap, next);
    return query;
}
```

Figure 5.13: The composition of the BFS algorithm for exploring the knapsack solutions state space.

```
static Query<PlayerState> createQuery() {
    final Comparator<KnapsackFilling> progressComparator = Comparator
            .comparingLong(filling -> filling.getItems().count());
    final Comparator<KnapsackFilling> frontierComparator = progressComparator
            .thenComparingInt(KnapsackFilling::totalValue)
            .thenComparingInt(KnapsackFilling::totalWeight);
    final Frontier<KnapsackFilling> frontier = TreeMapFrontier
            .withExactOrdering(frontierComparator);
    final Heap<KnapsackFilling> heap = new HashSetHeap<>();
    final NextFunction<KnapsackFilling> next = NextFunction.wrap(KnapsackFilling::next);
    final Query<KnapsackFilling> query = new Query<>(frontier, heap, next);
    query.addBehaviour(new SweepLineBehaviour<>(progressComparator));
    return query;
}
```

Figure 5.14: The composition of the sweep-line algorithm for exploring the knapsack solutions state space.

The difference sweep-line makes to the knapsack problem is significant. Since transitions never lead to a state in one of the previous layers those layers can be safely removed. For example, running the knapsack problem for 25 available items yields the memory usage shown in **Figure 5.15**. In the graph the size of the heap is plotted against the moment in time during exploration. For exploration, the time axis uses the number of state evaluations as the measuring unit since this is the main loop. The default heap used for “simple” algorithms such as BFS and DFS put one state into the heap during every state evaluation. This is illustrated by the blue line in the graph: the size of the heap is equal to the iterations of the main loop. Meanwhile, the purging of the previous “layer” in the state space is shown by the red line, that resembles a saw-tooth.

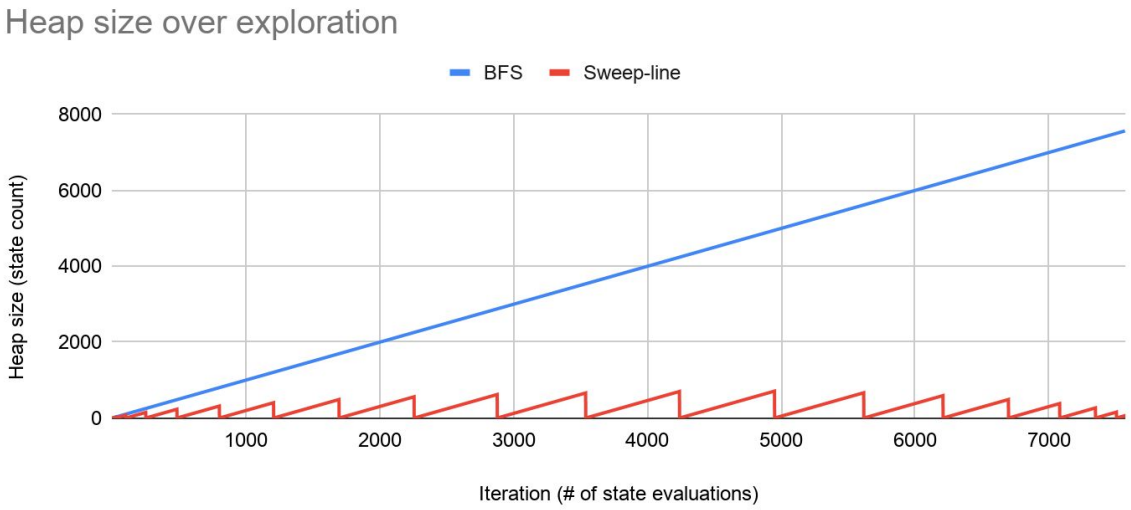


Figure 5.15: The difference between sweep-line and BFS approaches.

5.2.3 PDDL4J

The integration was based on PDDL4J v3.8.2 from the Maven Central Repository. The state representation class of PDDL4J (the `BitState` class) could be directly used. This is possible due to Abeona being state-agnostic by design, as well as the `BitState` class meeting the requirements to be used as a state class type. To allow Abeona to use the state representation, the next-function shown in **Figure 5.16** had to be created. All queries based on PDDL4J state spaces are able to use this same next-function.

```
NextFunction<BitState> createNextFunction(CodedProblem problem) {
    return state -> problem.getOperators()
        .stream()
        .filter(op -> op.isApplicable(state))
        .map(op -> applyEffects(state, op));
}

Transition<BitState> applyEffects(BitState state, BitOp op) {
    final var next = new BitState(state);
    op.getCondEffects()
        .stream()
        .filter(effect -> next.satisfy(effect.getCondition()))
```

```
        .map(CondBitExp::getEffects)
        .forEach(next::apply);
    return new Transition<>(state, next, op);
}
```

Figure 5.16: The next-function factory for the PDDL4J integration

These changes are sufficient to perform explorations based on PDDL4J problems using Abeona queries. In addition, the integration enables the use of an Abeona query within the existing PDDL4J infrastructure. This second part of the integration does not integrate PDDL4J for Abeona users. Instead, it integrates Abeona into PDDL4J. For this to work an Abeona query needs to be executed within the type system of PDDL4J and the `AbstractStateSpacePlanner` class needs to be implemented. This type encapsulates the ability to use SSE for generating plans. The main method of this type is the `search()` function. **Figure 5.17** shows the implementation of this function, allowing Abeona queries to be used to generate PDDL4J plans. The code relies on the compositional nature of behaviours to modify any given query instance and insert the logic required to generate plans. At the end of the search function, the added behaviours are removed, so that they will not influence other executions of the query instance.

```
@Override
public Plan search(CodedProblem codedProblem) {
    // Wipe the frontier of any leftovers
    query.getFrontier().clear();
    // Setup the initial state of the problem
    query.getFrontier().add(Stream.of(new BitState(codedProblem.getInit())));
    // Keep track of a backtrace, but the default of abeona does not store transitions
    // The plan requires the BitOp assigned to transitions, so this behaviour stores the
transitions
    final var backtraceBehaviour = new PddlBacktraceBehaviour();
    query.addBehaviour(backtraceBehaviour);
    // Setup the termination behaviour so we can easily wrap with logic later
    final var goalBehaviour = new
TerminateOnGoalStateBehaviour<>(createGoalPredicate(codedProblem));
    query.addBehaviour(goalBehaviour);
    // Perform the query in a try-finally so we can detach the behaviours after
    try {
        // Execute the query with the wrapper utility
        return goalBehaviour.wrapExploration(query).map(state -> {
            // We now know the goal state, with backtraces we can build the plan
            final var backtrace = backtraceBehaviour.iterateBackwardsTrace(state);
            final var trace = new LinkedList<Transition<BitState>>();
            // Collect the trace into a list so we can traverse it in forward order
            while (backtrace.hasNext()) {
                final var next = backtrace.next();
                trace.addFirst(next);
            }
            // Build the plan
            final var plan = new SequentialPlan();
            for (var step : trace) {
                final var operation = (BitOp) step.getUserdata();
```

```
                plan.add(plan.size(), operation);
            }
            return plan;
        }).orElse(null);
    } finally {
        // Once done we remove the behaviours so we don't leave them lingering around
        query.removeBehaviour(backtraceBehaviour);
        query.removeBehaviour(goalBehaviour);
    }
}
```

Figure 5.17: The implementation for the search function to enable usage of an Abeona query within the PDDL4J planner framework.

5.3 Demos

While building the framework we implemented a series of demo applications which demonstrate the usage and application of the framework to different problem contexts. Each demo has its own unique problem context and as such requires its own state representation to properly represent the state space and problem being faced. Most of the demos are puzzles or toy problems that challenge the framework. The main goal of creating the demos was to highlight the flexibility of the framework to different problem contexts and that the framework leaves the state representation details completely up to the user.

Some of these demos were also used as part of the user studies and the demos were then expanded to include graphical user interfaces to present the states in a more easily understood format.

5.3.1 Wolf, goat and cabbage problem

The wolf, goat and cabbage problem (titled goat-game throughout the codebase) is a simple planning problem. The problem can be formulated as following:

You are a farmer with a wolf, a goat and a cabbage. You are with your belongings on one side of a river and want to travel to the other side together with your belongings. On the riverbed there is a boat that you can use to travel to the other side with enough room to bring one belonging with you. When you are on the other side you must make sure that you do not let the wolf alone with the goat as the wolf will eat the goat. Similarly, you cannot leave the goat alone with the cabbage or it will be eaten by the goat. The starting position of the puzzle is shown in **Figure 5.18**, together with the only two following states that do not result in losing the game. The state space of the game (**Figure 5.19**) is rather small and this is beneficial for users that are unfamiliar with SSE to track the problem easily without getting overwhelmed.

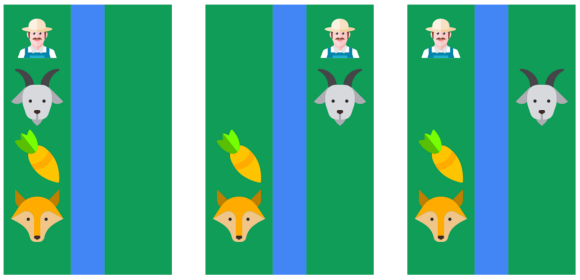


Figure 5.18: The initial state of the game (left) and the two only possible opening moves.

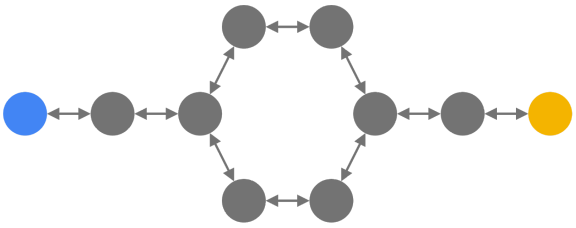


Figure 5.19: The state space of the goat-game, not showing all game-over states.

The state representation created for the game (see **Figure 5.20**) consists of four booleans, each representing the position of the boat (and farmer) or one of the belongings. The objects in the world can be only at one of two positions in the world at any time (left or right of the river) and this is what the boolean values map to. To leverage the OOP paradigm, we also made the state responsible for implementing the next-function as well as some predicate methods to help identify different types of states (invalid and goal states).

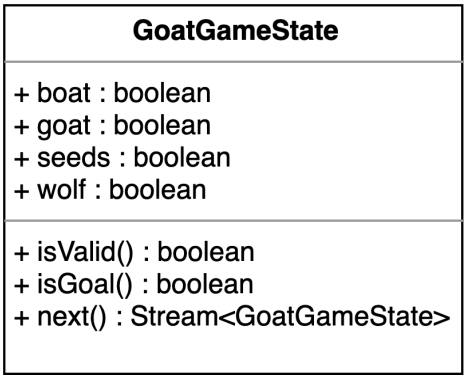


Figure 5.20: The UML representation of the state class used to encode a state in the goat-game demo

The demo comes with an exploration simulator that performs a query on the state space of the goat-game. The simulator includes controls for step-by-step execution of the algorithm that the query composes. The code for composing the BFS algorithm is shown in **Figure 5.21**, this is also the default algorithm used in the demo application. The query composition is able to discover the state space as shown in **Figure 5.19** exactly. This demo was also used in part during the user-studies.

```
static Query<GameState> createQuery() {
    final Frontier<GameState> frontier = QueueFrontier.fifoFrontier();
    final Heap<GameState> heap = new HashSetHeap<>();
    final NextFunction<GameState> next = NextFunction.wrap(GameState::next);
    final Query<GameState> query = new Query<>(frontier, heap, next);
    query.addBehaviour(new LogEventsBehaviour<>());
    return query;
}
```

Figure 5.21: The query definition for exploring the goat-game state space with the BFS algorithm.

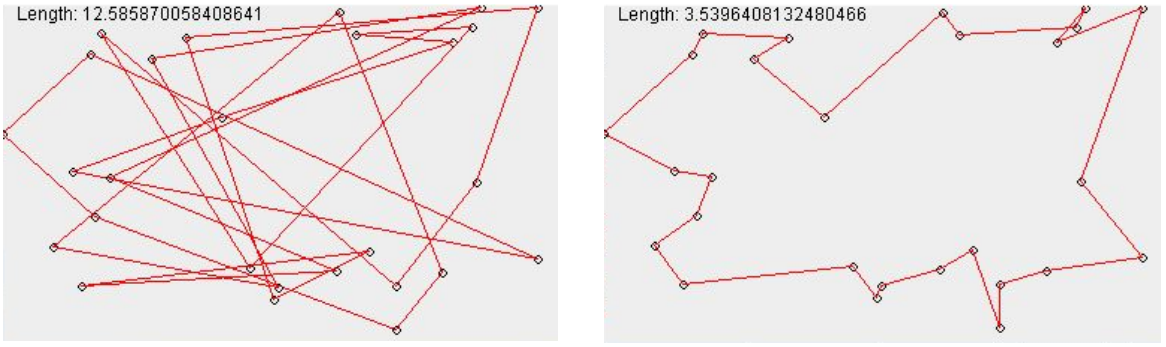
5.3.2 Traveling salesman problem

The traveling salesman problem is a well-known problem in combinatorial problem solving. In this problem a salesman needs to travel to a set of cities, visiting each city only once and ending up at the city they started traveling from. The distances between every pair of cities is known and the goal is to find the shortest path possible for the salesman to travel.

In the analysis in **Appendix A**, possible applications are discussed of SSE as part of other algorithms (named “workflows”). A demo is made for this use-case, which implements simulated annealing within the Abeona pipeline. To implement the randomized selection process we install an advice on the “insertIntoFrontier” join-point. This advice blocks all states from entering the frontier, except one. Because of this, exploration only continues with one outgoing transition when a state is evaluated. The algorithm requires a heap not to be used, as visiting previous states needs to be allowed. This representation of the state space uses a full path across all the cities to represent a single state.

The next-function that is used in the solution provides mutations on the current solution by swapping any two cities in the order of visitations. We presumed this would be most suitable for simulated annealing as it provides a lot of transitions to pick from from any state, to make escaping local maximums more likely to occur. To reduce the runtime of each iteration we utilize the lazy evaluation of the java stream API so each state is generated as it is requested from the stream instance.

The behaviour which advises the “insertIntoFrontier” tappable tracks the temperature of the system during simulation and lowers it over iterations with exponentially smaller decrements. The results of one such run is shown in **Figure 5.22** which ran with a computational budget of 10000 iterations. The length label in the top-left corner in the images shows the sum of the distance travelled along the path.



A: Initial randomly generated solution state **B:** Solution state after 10000 iterations

Figure 5.22: The traveling salesman simulator before and after some iterations for a (randomized) traveling salesman problem

5.3.3 Train network

The train demo is a small application that demonstrates how to utilize the userdata present on transitions. The train network consists of a directed weighted graph with a given start and goal node. The task is to find the cheapest path from the start node to the end node. The cost of the path comes from summing the weights of the edges in the path. In this case we encode the distance between train stations on the transitions and showcase this by composing Dijkstra's algorithm properly. The state space, similarly to the maze demo (from **Section 5.2**) we map the state space directly from a navigational graph and let each state represent a node from this source. A train network graph is built, consisting of stations and connections between the stations. Each connection identifies the distance between the two stations that it connects. A UML diagram for classes that implement this is shown in **Figure 5.23**.

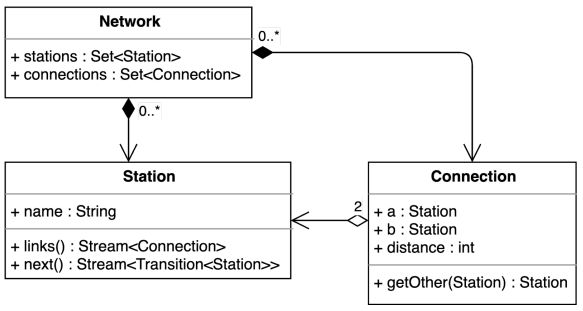


Figure 5.23: The UML diagram for the classes encoding a simple train network used in the demo application.

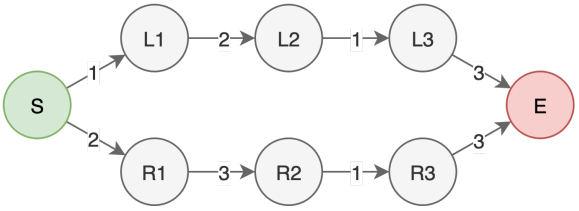


Figure 5.24: The network tested in the demo application, each state is named and the labels on the edges represent the distances between the stations.

The demo is implemented with a runnable unit test that verifies the discovery order of the graph shown in **Figure 5.24**. The unit test verifies that execution of the composed query discovers the states from in the following order: **{S, L1, R1, L2, L3, R2, R3, E}**. This network is set up to not be correctly solvable by either DFS or BFS (which is the behaviour of

Dijkstra's algorithm when all edges have weights of 1), only a correct implementation of Dijkstra's algorithm will solve it correctly.

5.3.4 GROOVE

To integrate with GROOVE we used the existing exploration strategy framework (based on inheritance) to provide abeona as a new option in the exploration dialog. We also put some work into supporting the configuration of Abeona in the dialog, see **Figure 5.25**. The configuration interface of the Abeona strategy first shows two dropdowns to pick the frontier and heap implementation. The user cannot select the next-function implementation because it is specific to the object-model used in GROOVE. After these two dropdowns a list of groupboxes is shown. This list represents the behaviours that will be installed on the abeona query. Each behaviour has its own groupbox with a title describing the behaviour, a checkbox to enable the behaviour and finally additional controls specific to the behaviour. We ensured our implementation can easily be extended in the future to add new behaviours to the list in the interface. The integration is a proof-of-concept and thus the list of behaviours supported is limited and in some parts overlaps with existing capabilities of GROOVE.

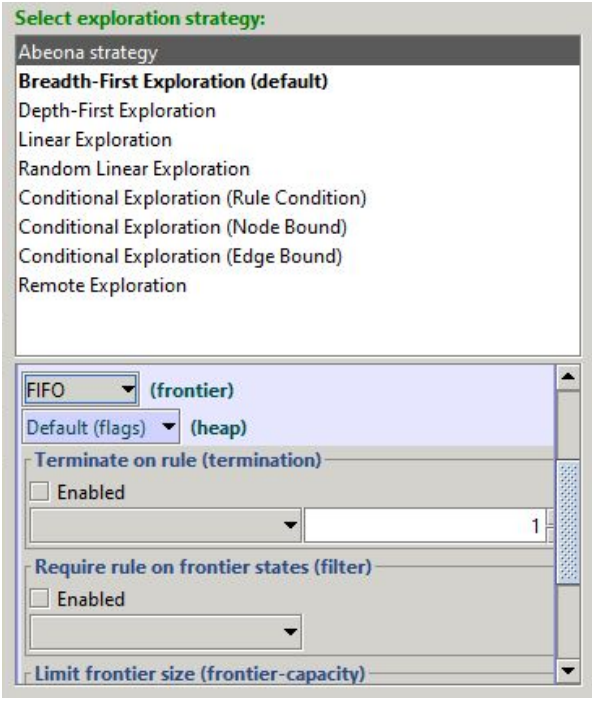


Figure 5.25: The new abeona exploration option in the strategy menu together with the supporting configuration interface.

5.4 User studies

As we have developed a new framework we wanted to get a sense of its usability to new users. For this we performed a usability study aimed at collecting feedback on the API, modularity and ease of use. For the study we set up a series of challenges where the subjects would be asked to solve problems using the Abeona framework. The challenges started off simple, with the 'goat-game' demo and increased in complexity. Each challenge was accompanied by a visual simulator, so the subjects could interact with the exploration.

The final challenge did not have such a simulator, as it did not provide any starting point and gave the subjects complete freedom. This final challenge is included to see how users experience creating a solution using the framework without a starting point. In this final challenge the individual tasks gave small instructions that built on the previous tasks, simulating changing requirements over time.

For the raw results of this study, consult **Appendix B**. As the sample size for the usability study is five, this is too small to make any statistically significant observations. Instead, we will analyze and try to generalize the users' experiences. For this, we focus on shared opinions and feedback as they pertain to each participant's background (skills, knowledge, study).

5.4.1 Tasks

This section details the tasks given to the participants, as well as the intention behind those tasks. The tasks were grouped into 'challenges', where each challenge focussed on a particular problem or demo.

Challenge 1: Wolf, goat and cabbage problem

The first tasks were centered around the "wolf, goat and cabbage" problem introduced in **Section 5.3.2**. This puzzle is simple and has a very small and fixed state space. This makes it easy for the participants to mentally track where any state might be within the state space. For this challenge the goat-game demo was extended with a simulator, which was given to the participants to perform step-by-step explorations through the state space. The interface of the simulator is shown in **Figure 5.26**. The interface features six columns, each showing a list of states. As the user clicks the "next step" button the exploration query performs a single state evaluation and updates the interface. The step-by-step control over the exploration lets the participants become familiar with the concepts of frontier, heap and evaluation. These tasks serve mainly as an introduction to the concepts of SSE and some of the concepts specific to Abeona.

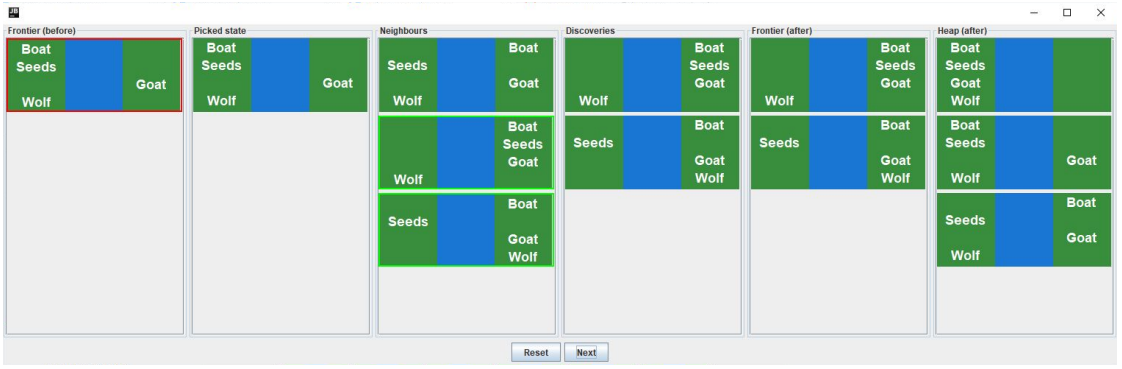


Figure 5.26: Simulator for the goat game challenge

Task list

1. Run the goatgame demo and run an exploration to its completion
    a. On paper, draw the state space that gets explored

    b. Note the order in which the states are discovered
2. Change the algorithm used from breadth-first-search to depth-first-search
    a. How does the order of state discoveries differ?
3. Extend the algorithm to terminate when finding the first goal state. The interface will notify the type of termination to be **ManualTermination**.
    a. How many steps does it take the algorithm to reach the goal state?
4. Change the algorithm back to breadth first search

The first task asks the participant to walk along a single exploration sequence and forces them to visualize the resulting state space. Besides giving the participants an introduction to the concepts, it also allows the researcher to give direct feedback on mishaps and give detailed explanations on the basics. This prevents that questions of the participants about fundamentals would give away answers in later, more difficult challenges.

The second task is similar to the first, and teaches the relationship between BFS and DFS, and the “query” concept. During the first task, participants are given hints to the difference between BFS and DFS, so that they would have a starting point when arriving at the second task. The purpose of the tasks is to observe the interaction of the users with the Abeona framework. This is why the need to investigate the possible solutions is minimized. Rather, the required effective changes are described, to observe how users explore and navigate the API.

The third task is similar to the second in regard to observing the exploration of the API rather than challenging the participants with creatively inventing a solution. The third task adds further changes to the query. The last task plays into the non-overlapping changes that result from these tasks by asking to undo the second task.

Challenge 2: Maze

The maze for the second challenge has a larger state space than the first challenge. This increases the complexity of the problem. Despite their increased complexity, mazes are easily understood visually. The demo application shows the maze, frontier and heap all in a single image. This does warrant an explanation how states map to positions in the state space. The simulator for this challenge is shown in **Figure 5.27**, the state representation is based on the demo described in **Section 5.1.1**, see **Figure 5.2**. The interface uses colors to distinguish the different types of sets in which a state belongs. Unknown states are colored white, states in the heap are yellow and states in the frontier are green. The latest state evaluated is colored red.

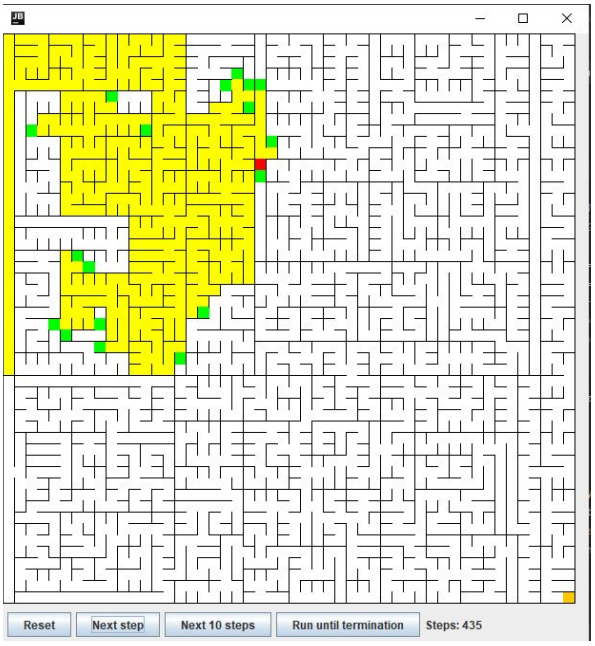


Figure 5.27: The maze demo interface, used in the user study.

Task list

1. Run the default exploration query
    a. Based on exploration order, what algorithm is implemented?
2. Change the query to terminate the search when reaching the goal state
    a. How many steps does it take to find the goal?
3. Change the frontier to an ordered frontier that orders the states based on distance to the goal
4. Try to build a query that finds the goal in the least number of steps
    a. What algorithm did you implement?
    b. How did the algorithm perform?
5. The mazes generated are 'perfect' mazes, meaning they do not contain loops. Because of this it is possible to execute the exploration algorithms without saving any of the states into the heap. Try to implement this

The first two tasks in this challenge reflect largely the same intentions as the first challenge, but on a more complex problem. The third task asks the users to experience using a different type of frontier, specifically the `TreeMapFrontier` specifically. The construction of which requires the usage of the standard java `Comparator` type. Relying on standard java types and features is fine as far as framework design is considered, however what this does to the complexity remains to be seen. With task 4 the participants were given creative freedom to test different approaches to the problem. This reveals novel techniques or approaches that otherwise would be missed due to the researcher's personal bias. Task 5 is a more challenging task, as expertise on the algorithm's requirements is needed to accomplish it.

Challenge 3: Sokoban

Sokoban is another puzzle, more complex than the puzzle used for the previous two challenges, that is easily understood visually. In the puzzle a grid-based world is described, containing walls, open spaces and buttons. The player has a presence in this world, the character is bound by rules, for example the player can only move into open spaces or onto buttons. The task for the player is to move the boxes around in the world such that all buttons in the level become pushed down by a box. The rules that dictate how the player can move and how boxes present a logistical problem to the user.

Sokoban suffers from the state space explosion problem as the number of crates hugely affects the size of the state space. The focus of this challenge is to let the participants utilize the sweep-line algorithm. It is an interesting algorithm as it allows for the heap to have states removed during exploration. The interface used for this challenge can be seen in **Figure 5.28**. It features not only the state column display from the first challenge, but also graphs for the size of the frontier and heap over the exploration steps.

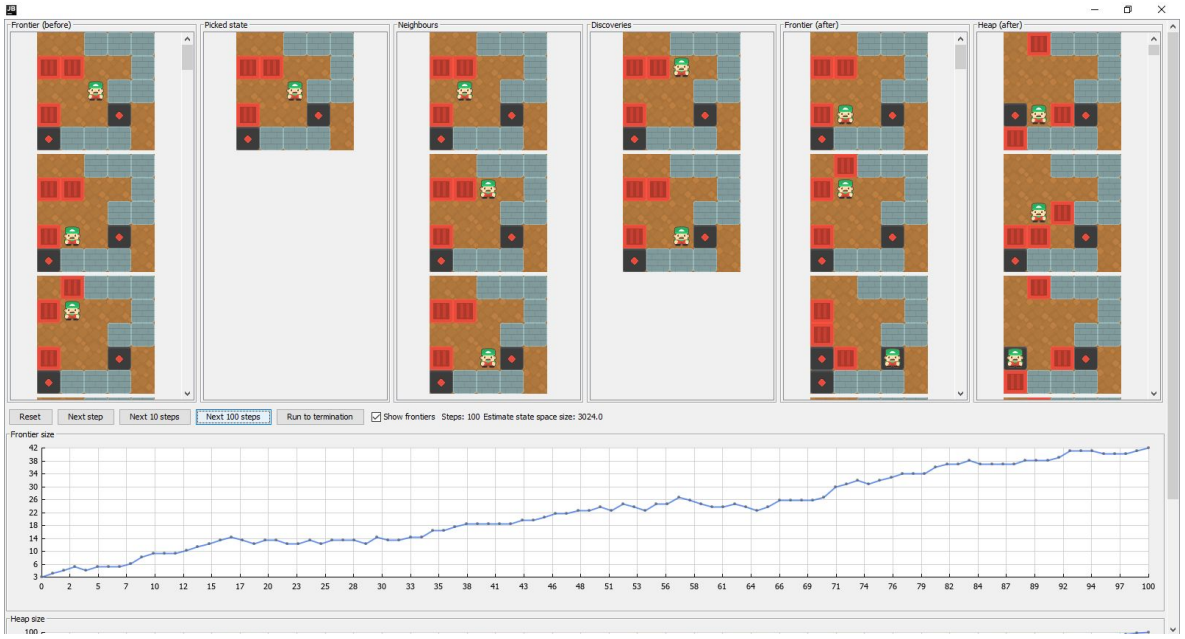


Figure 5.28: Simulator for the sokoban challenge

Task list

1. Run the simulator to become familiar with it and run the default puzzle.
    a. Note the number of states in the frontier and heap present during exploration
2. Change the query to terminate when finding a solution state, states can be tested to be a solution state through the `isSolved()` method
3. Using sweep-line, try to lower the number of states that exist in the heap.
    a. How difficult was it to configure the sweep-line algorithm?
4. Try to lower the number of states that enter the frontier, try and create a behaviour that limits the states entering the frontier.
    a. How many less states are present in the frontier?

b. How did you limit the number of states entering the frontier?
c. Did you have to debug your behaviour? How did you experience this?

The intention was to let the participants utilize an advanced behaviour on their query composition. However, the technical details were too complex to convey and understand in a timely manner. During testing this challenge proved to be too challenging and was often skipped.

Challenge 4: Staircase robot

The problem for this challenge consists of a robot walking up a staircase. The robot (controlled by the player) can move up and down the staircase in specific patterns (e.g. move up 3 steps). The player is not allowed to move the robot such that it steps over the top or to step down while on the ground. This puzzle is simple but in the tasks additional rules are added later on.

This challenge does not start off with a simulator or some prepared code. Instead, this challenge is completely open ended. There was no simulator for this challenge, the applications could only write to the console. The purpose of the open ended aspect was to observe the participants writing their own state representation. The first 4 tasks lead the participants to build up a working implementation of a search strategy and then tasks 5 through 9 introduce additional requirements one at a time. The addition of requirements simulates the changes of requirements over time that is a major challenge in software development and software management.

Task list

1. Create a state class that represents the state of the robot climbing the staircase
2. Implement a next-function on the class, this should return a Stream<T> of the staircase state.
3. Verify that your state representation works by climbing stairs of sizes 5-10
4. Verify that your robot cannot climb stairs of sizes 1-4
5. Add the ability for the robot to climb 7 steps at a time
6. Create a query for climbing a staircase with 1000 steps
7. Add a behaviour to measure the number of state evaluations
8. Try to optimize your query to use the least number of state evaluations to reach the goal
9. Prevent solutions where the robot steps on staircase steps #145, #509 and #733

5.4.2 Analysis

For the raw observations of the above user study, consult **Appendix B**. The following analysis is based on those results.

Changing the frontier from BFS to DFS

In task 2 of challenge 1, the subjects needed to change the algorithm from BFS to DFS and received hints that led them to conclude they needed a different frontier. Asking ourselves what led them to find the correct frontier, we summarize the results as follows:

- *S1* - They had a lot of prior experience with java programming and were familiar with the IDE (intellij) this led them to explore the framework as soon as possible, looking at the class hierarchy and interface implementations. Once they knew what they were looking for they managed to find the method that creates the correct frontier in the source-code of the `QueueFrontier` class.
- *S2* - They received the hint that the frontier order would need to be different and checked for alternate methods on the `QueueFrontier` class and found the method quickly.
- *S3* - They figured out that the exploration order would have to be “filo” and used the autocomplete function to list the alternate methods on the `QueueFrontier`.
- *S4* - They looked into the source code of the `QueueFrontier` and found the method defined there
- *S5* - They received the hint that the frontier would need to be ordered differently, from there they looked into alternate methods on the `QueueFrontier` class using auto completion and found the method there.

This leads to believe that for java-experts the framework is fairly self-explanatory and from the initial example they are able to make a basic modification.

Implement on goal early termination

In each challenge one of the first tasks was to change the query to make it terminate the exploration when a goal state would be found. Here we analyse how that was achieved.

- *S1* - They received no guidance and made several attempts before achieving a solution. They first focussed on the next-function, perhaps in an understanding it was the most controllable source of data. They did look through the list of default behaviours before this attempt but did not register the default termination behaviour in the list. Their successful attempt consisted of clearing the frontier using the `afterStateEvaluation` event. They needed guidance on how to implement an abstract-behaviour.
- *S2* - When looking through the list of default behaviours they did not register the suitable default behaviour. They only saw the list of filenames in the folder view of the source code, they had not taken the time to inspect the documentation thoroughly. When tasked with using the behaviour system or event system they could not figure out how to use them.
- *S3, S4 and S5* - They all had a similar experience in applying the termination behaviour: They found the suitable default behaviour by the class name in the project explorer. They had varying difficulty using the predicate, mostly due to it being a relatively new feature they had not used before. But the purpose was clear from context and/or the documentation.

Observations & conclusions

The experience of S1 and S2 points out that for newcomers the concepts of the events, tappables and behaviours are left unexplained and that the documentation of the functions which expose and consume them are not sufficient by themselves. A possible solution to this

is to provide small guides/tutorials on these systems which explain their usages, how to implement them and how to find/recognize them in the code base.

The experiences of S3, S4 and S5 have shown that for the `TerminateOnGoalBehaviour` is relatively easy to use without much help.

A specific experience S3 had with this behaviour was in task 4 of challenge 2 where he received the advice that the behaviour exposed events that signal when a goal state is found. By that description alone he could not figure out how to use the behaviour in such a way. The major obstacles there were again lacking knowledge in the events system. So one of the guides may have to be how to use events exposed by non-query systems (such as another behaviour).

Using a ordered frontier

In task 3 of challenge 2, the subjects were asked to change the frontier to one which is sorted by a specified metric.

- S1 - They had difficulty finding the correct frontier, using the project explorer to find out what code resides in the “abeona.frontiers” namespace but did not think (at first) that `TreeMapFrontier` would be the required class for the solution. After some exploration of the class/interface hierarchy of the `Frontier` interface it became clear for them that they had to use that class.
- S2 - When applying the `TreeMapFrontier` they looked at the source code and documentation to figure out what it exposed. The first view they had was of the top of the class and this included the constructor. As such he immediately attempted to call it, even though it was private. This sequence of events also occurred with several other subjects.
- S3 - Similarly to S1 there was some difficulty spotting `TreeMapFrontier` as the correct solution, they specifically remarked that the naming of this class was more a hint towards the implementation details than it was a descriptive name of its purpose. First attempts to use it focussed on the constructor instead of the factory methods.
- S4 - Once the `TreeMapFrontier` was found the usage was difficult to deduce because of lacking documentation.
- S5 - They did not notice the `TreeMapFrontier` by name as a suitable solution but found it ultimately by checking the class hierarchy for the `Frontier` interface. The implementation details causing problems for comparators which report states as equivalent caused the subject to think the frontier was wrong.

Observations & conclusions

Based on S1, S3 and S5 we can see that the name of the `TreeMapFrontier` is not very eye-catching and threw off the subjects early on. The `OrderedFrontier` was more often the focus of the subjects, and usually led them back to the `TreeMapFrontier`. Perhaps a naming scheme where `TreeMapFrontier` contains a word closely related to “ordered” or “sorted” would be better suited.

Besides that, the usage of a `Comparator` was also not entirely clear to users immediately. The name comparator is not java-specific itself but its application in java and the framework is. The documentation for the `TreeMapFrontier` did not adequately explain what was required of the comparator and only java-experts knew to use the `Comparator` factory functions. Those factory functions were kept in mind when designing the `TreeMapFrontier` API and as has become clear the documentation may need to hint towards those functions to help new users implement comparators easily.

The `TreeMapFrontier` instantiation was confusing for several subjects, when they first viewed the class they saw the documentation and source code of the top of the class. Since this included the constructor they immediately tried to call the constructor. The documentation of the class may need to specifically mention that it can only be constructed with one of the two factory methods.

Finally, none of the subjects had any information about the comparator edge-case/bug where some states would be considered equivalent based on the equivalence. Perhaps the framework should implement a workaround so they are never exposed to it. Alternatively, the subjects each picked the first occurring factory method they saw from the source code view, simply putting the factory method which implements the workaround first could also nip this problem in the bud.

Executing their own query

Not every subject worked on challenge 4, where they had to build their own state representation as well as use the query object themselves.

- S1 - They looked through the `Query` class to find the methods available and noticed the `explore()` method. We had to remind them that the frontier needs an initial state when on the first run the query terminated immediately. When trying to optimize the query to use a greedy algorithm they tried implementing their own frontier. Because they did not do Challenge 2 the existence of the `TreeMapFrontier` was not known to them. From the documentation alone they were able to figure out how to implement the `Frontier` interface. The fact that there were other more specialized interfaces of `Frontier` (e.g. `ManagedFrontier`) was lost on them, perhaps the documentation on the `Frontier` interface could further clarify this, however they did not carefully read the documentation in the first place so perhaps exploring sub-interfaces of `Frontier` might come naturally.
- S2 - Did not perform this challenge
- S3 - They copied the structure of a `createQuery()` method from the previous examples to build the query in a similar fashion as in prior challenges. When it came to executing the query they were confused why it terminated immediately, they expected the framework to use some initial state, though they did not specify one. They used the source code of one of the examples to figure out how the query object built by them was being used. There they discovered that the initial state has to be inserted into the frontier before exploration.
- S4 - They first wrote their state representation and then built the query code. Their state representation contained methods to create alternate versions of the current

state with some mutation applied, such as moving forward or backward. When building the query code they also copied the `createQuery()` method from a previous example and adjusted it to their needs. The next-function was made with the `NextFunction.wrap()` helper. When explaining the next-function interface they wondered what the wrap function was for and if they needed to implement it. The signature of the function was too confusing (due to the large generics declarations) for them to figure out how the wrap function relates to the next-function immediately.

- S5 - They did not have enough java experience to complete this task on their own, we assisted in writing the code in a timely manner. We also hinted at reusing code from previous exercises to speed up building the query. Their implementation of the state representation used a class method as next-function which was something the other subjects had not done. They decided to do this after we explained how the next-function works and could be implemented with either a lambda or on a class directly. In the approach they used if-statements in the next-function to determine whether some actions are possible or not.

Observations & conclusions

On every first run of the query the subjects did not specify an initial state. This is because none of the queries prior were built with an initial state specified. They all assumed that the framework somehow built an instance automatically. S5 specifically asked whether the `explore()` method could take a state as input and this might be an easy to implement solution to provide a simple API that solves this. Additionally, an error-signal or special termination type may be created for starting queries without an initial state.

A small observation we made is that in the framework, interfaces are used for the majority of laying the foundation of the class hierarchy; however, this was hardly ever noticed or obvious to the users. Perhaps the reliance on interfaces and composition is not quite clear enough on first use of the framework and it might need a specific mention in one of the first introductions/guides for the framework. Understanding that most features and components are found in an interface and may have multiple sub-interfaces which further extend it would encourage more thorough looks at the api from users when implementing their own components.

In the implementations of the subjects, their states sometimes included both state and problem level details in the state representation. A generic model for encoding states, problem properties and state mutations might help make the creation of state representations more modular also. In the current case often the state representation and next-function hold all details of the problem context and possible mutations of the state. For the problem context this means that it is difficult to extend or modify the problem context if the code defining it is not under your control. Also for possible mutations a more friendly interface that allows adding mutations later on would fit well with the modular design of the rest of the framework. Perhaps these could even be behaviours in some way. These last suggestions are more dependent on state representation of the user and was out of scope for the project but might be good future work to extend the framework.

5.4.3 Influence of expertise

For programmers with java expertise (not to be confused with programming expertise), the framework was noticeably easier to use than for those without java-specific knowledge. The usage of modern java features (e.g. lamba's and functional interfaces) has led to a higher barrier-to-entry because the users need to understand how these features work. Also, to make maximum use of for example the comparator interface the usage of helper methods such as `Comparator.comparing()` is very useful to easily create comparators, however this may be somewhat hidden for those that do not know about it. The java experts also had a much easier time exploring the structure and hierarchy of the framework.

Regardless of expertise, none of the subjects were able to quickly understand the events system, unlike the behaviour system. This is likely because the behaviour system was introduced to them through sample code in the challenges, each challenge's query included the `LogEventsBehaviour`. There was no code demonstrating the tappable events system to them.

What was also noticeable was that many subjects preferred to modify the query's behaviour by modifying the next-function. The concept and advantage of behaviours is something that needs to be more front-and-center in the introduction of the framework.

5.4.4 Summary

The framework in its current state is sufficiently usable for those with experience with java programming. For those without java programming the framework is usable but the lack of java specific knowledge does put up a barrier and makes the framework API more difficult to use due to its minimalist design.

For new users the sample code provides good examples of how to build queries but lacks in demonstrating the advanced use cases of behaviours and completely lacks to highlight the usage of the tappable events system. Once explained these systems are not incredibly complex to use but understanding them is not easily done through the documentation alone.

Creation of custom behaviours also has a bit of a barrier to entry, since there is no simple base to quickly start with. The usage of `AbstractBehaviour` is too complex and difficult to understand from the current documentation.

The used naming scheme for classes is good and logical for the users but in cases where the name reflects the implementation details rather than the purpose of the class/interface this falls short.

The documentation is sufficient all round, but the `TreeMapFrontier` documentation lacks severely. The `TreeMapFrontier` implementation has hidden requirements when instantiating it.

The API is very modular and generalized for exploration algorithms, its usage case in educational purposes and management of sets of exploration methods is clear.

6 Conclusions

To conclude the work we answer the questions we stated in **Section 1.2**. We follow this up with a reflection on the project's execution. The chapter finishes with pointing out future work for the Abeona project.

Does the general framework support implementation of exploration algorithms through composition?
Yes, the design we produced supports this directly. With the implementation of the pipeline we were able to ensure that the composition API was supported. We have shown with our demo applications that the algorithms evaluated in **Appendix A** can and have been implemented using the compositional method. Furthermore, the demo applications show the application of these algorithms in different ways as was the purpose of using a compositional approach.

Can specific components of algorithms be integrated into other algorithms?
Taken as a direct requirement in the framework design we created a generic set of useful behaviours in Abeona. In the demo applications these get used in different contexts and applications to build different algorithms. This supports our claim that such a framework can support reuse of the (generic) components. During the user studies in our validation, we also saw several of the participants use the built-in behaviours in their query compositions. The modularity shown in **Section 5.2** demonstrates that incremental changes to query instances are possible.

How well does the framework integrate with existing applications of state space exploration?
With our integrations we were able to show that the framework is flexible enough to be integrated into existing solutions as well as to work within other frameworks. For PDDL4J this proved to be possible with little extra code, as can be seen in **Figure 5.16** and **Figure 5.17**. In the case of GROOVE a greater effort was required but this was mostly due to the programming of interface code. In each case the attempt of integration was successful and it led to an expansion of the existing capabilities. For GROOVE, the new search strategy is available directly from the interface, giving the users more control over the particular search strategy to use. The Abeona strategy supports the modular composition by having the users select what pieces of behaviour they want to combine. The implementation made in this thesis has behaviours that are not all new features. One such example of overlap is the filter setting, where the application of a specified rule prevents it from entering the frontier. This is also possible through the grammar settings by specifying a rule as a restriction. The Abeona strategy creates a semantically different feature, where the filtering is now tied to the exploration system rather than to the state space definition itself.

How does the performance of the framework compare to that of existing state space exploration implementations?
From our results it is clear that there is an overhead to using the framework, which we expected as the flexibility of the pipeline is supported by the tappable system at runtime. The comparison with GROOVE (**Figure 5.6**) shows comparable performance, with some overhead for smaller sized problems. Our performance comparison results for PDDL4J (see **Figure 5.8**) show similar results, where Abeona is even able to out-perform PDDL4J in some problems. Our solution was even able to produce results where PDDL4J currently is unable to find solutions using their SSE techniques.

Is such a general framework easy enough to work with while remaining sufficiently general?
From the user studies we conclude that the framework is relatively easy to use, and is sufficiently generalized. However, because of this the required knowledge about the structure of the pipeline and the intended cooperation of behaviours is not obvious up-front. This calls for the creation of supporting materials that will introduce the concepts and best-practices to new users.

Can a general framework for building search algorithms be designed? (main question)
From our findings and the work we produced we can conclude that the design of such a framework is possible. The runtime performance of such a system remains a difficult challenge, especially due to the imposed requirement to stay generalized. The main benefit of such a framework, to manage multiple algorithm compositions and to alter them later on, must outweigh the overhead the framework imposes on the solution. While our efforts do not outperform existing solutions, we have shown to be on-par in several scenarios. This means we were able to provide similar performance compared to an existing solution while offering more flexibility through the compositional framework. On the other hand, there also were cases where the performance scaled badly. It is unclear if this is due to the design of the framework or a fault at the implementation of the problem (state representation) that caused an increased computational complexity to occur. With the feedback from the user studies we are confident in saying we have managed to design and build the framework we proposed and have shown its utility.

6.1 Reflection on the project

Working on the project has been a lot of fun, with a lot of opportunity to learn about varying ways to utilize SSE. Designing and building the framework was a great challenge and I am happy with the results I achieved. The concept of expressing algorithms through composition of components turned out to be a very good way to introduce flexibility onto the base skeleton algorithm structure.

What do I think of the result?
I think that the framework I built is a good starting point for implementing solutions to state space related problems. When writing the knapsack demo I also realized that for problems that require dynamic programming the state space approach is a very intuitive way to represent, encode and solve problems. Abeona also offers a useful testbed to run

explorations from. The query api allows for different algorithms to be implemented with the same framework, this makes managing sets of implementations compact and easier to do than cloning complete code bases to make the necessary changes.

The ability to manipulate the pipeline of Abeona even after an instance is created (by adding more behaviours) turned out to be very useful for adding compatibility with existing solutions. The adapters I built for GROOVE and PDDL4J accepted user-made query instances and then modified it a little with additional behaviours to make it compatible for the GROOVE and PDDL4J environments.

The major downside of Abeona hides in its generality. Because there is no dedication to a particular problem domain, there is no leverage to utilize. A user of Abeona can of course take advantage of their specific problem domain but I feel this kind of shifts the burden of applying the best available domain optimizations back onto the user. Perhaps a future development of Abeona could focus on encapsulating domain specific optimizations by providing additional “domain packages” containing those optimizations.

As for the performance of the framework, I am quite happy with the results. It performs reasonably well, with some overhead to be expected. On large problems it seems to cope less well than state-of-the-art solutions. Again, this is to be somewhat expected since I do not utilize domain-specific optimizations. Given more time I would have looked more into the performance and the overhead of the framework, there are some options to consider for improving the performance at runtime.

What I would have done differently

There are a few things I would have done differently in hindsight. The biggest one being the user studies. For a more formal and statistics based approach I would have tried to team up with a teacher at the UT and try to have an entire class work through a testset as part of a tutorial course on graph algorithms. Something to this extent would have been possible I think and would have made the results of the user study more reliable. The current sample size was tiny and it mostly gave me some good quotes and insights into the API documentation. With a larger sample size I think I could have gotten a more thorough understanding on how the users would approach problem solving creatively and to see if the framework design could provide the tools necessary for those creative solutions. Currently the framework requires a specific approach (composition) and relies on the understanding of the underlying systems (events and the pipeline) to make effective use of the framework. That, to some extent, will always be the case but I could have discovered a neat abstraction that would help new users very well understand the system better.

Something else I would have done differently is to also look into parallelization for SSE. My implementation is single threaded and I have done no work towards supporting multi-core exploration. Including this into the original project could have easily blown the complexity out of proportion. Still, it is an interesting topic for future work.

6.2 Future work

The reflection already mentions some topics for future work, those are included here as well.

In the current state of the framework, more work could be done to reduce the overhead of the framework. Some options to consider are dynamic code generation or using a more low-level language (e.g. C++). With dynamic code generation the event-system could be flattened out into plain function calls (instead of looping over lists). Perhaps the entire pipeline could be compiled into a “handcrafted”-like code blob based on the AST of the behaviours and the event system. But this would make debugging quite a challenge.

Other considerations for optimizations are selecting better data structures and implementations for collections used throughout the framework, as well as reducing allocations made for the event-system during exploration.

The consideration to change the source code language to C++ comes more from the limitation of java. The framework is written in java and thus only JVM applications can use the framework directly. A lower level implementation in C/C++ is interoperable with most existing programming languages, making the framework a more attractive option for those who are tied to a non-java environment. Using an unmanaged runtime would also give greater control over the memory allocations.

Early on in the project, the decision was made not to include multi-core support for the framework. This seems like an interesting area to continue to work in, especially with cloud, super-computer and GPU technologies. For Abeona, the application of SSE in parity games was not considered. While technically a frontier could be made that performs A-B moves this is an interesting area to support with Abeona.

Another way the framework could be expanded is to head (back) into the modelling world. Currently there is no modelling input specified for the framework, any class you program as state representation can be used. This means that for users that already have a model they first need to build a compatible model parser. For some models (such as PDDL) there already exist libraries for parsing such models into the JVM. Creating stand-alone libraries for more model types would benefit Abeona users as they can immediately use their models in Abeona queries. Additionally, making such parsers into standalone libraries would be beneficial to programmers that want to write programs interacting with such models. At a high-level an abstraction of states and actions could be introduced into the framework to provide interfaces to define actions by.

References

[1] E. M. Clarke, “The Birth of Model Checking,” *25 Years of Model Checking*. pp. 1–26, doi: 10.1007/978-3-540-69850-0_1.

[2] S. A. Kripke, “Semantical Considerations on Modal Logic,” *Acta Philosophica Fennica*, vol. 16, pp. 83–94, 1963.

[3] S. Blom, J. van de Pol, and M. Weber, “LTSmin: Distributed and Symbolic Reachability,” *Computer Aided Verification*. pp. 354–359, 2010, doi: 10.1007/978-3-642-14295-6_31.
[4] S. Cranen *et al.*, “An Overview of the mCRL2 Toolset and Its Recent Advances,” *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 199–213, 2013, doi: 10.1007/978-3-642-36742-7_15.
[5] G. Brat, K. Havelund, and W. Visser, “Java PathFinder - Second Generation of a Java Model Checker,” *In Proceedings of the Workshop on Advances in Verification*, Jan. 2000.
[6] A. Rensink, “The GROOVE Simulator: A Tool for State Space Generation,” *Applications of Graph Transformations with Industrial Relevance*. pp. 479–485, 2004, doi: 10.1007/978-3-540-25959-6_40.
[7] M. Fox and D. Long, “PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains,” *Journal of Artificial Intelligence Research*, vol. 20. pp. 61–124, 2003, doi: 10.1613/jair.1129.
[8] J. Hoffmann and B. Nebel, “The FF Planning System: Fast Plan Generation Through Heuristic Search,” *Journal of Artificial Intelligence Research*, vol. 14. pp. 253–302, 2001, doi: 10.1613/jair.855.
[9] M. Helmert, “The Fast Downward Planning System,” *Journal of Artificial Intelligence Research*, vol. 26. pp. 191–246, 2006, doi: 10.1613/jair.1705.
[10] N. Lipovetzky and H. Geffner, “A polynomial planning algorithm that beats LAMA and FF,” in *Twenty-Seventh International Conference on Automated Planning and Scheduling*, 2017, pp. 195–199.
[11] N. Lipovetzky and H. Geffner, “Best-first width search: Exploration and exploitation in classical planning,” in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017, pp. 3590–3596.
[12] A. D. Pimentel, “Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration,” *IEEE Design & Test*, vol. 34, no. 1. pp. 77–90, 2017, doi: 10.1109/mdat.2016.2626445.
[13] S. K. Neema and J. Sztipanovits, *System-level synthesis of adaptive computing systems*. Nashville, Tennessee: Vanderbilt University, 2001, pp. 22–23.
[14] A. H. Land and A. G. Doig, “An Automatic Method of Solving Discrete Programming Problems,” *Econometrica*, vol. 28, no. 3. p. 497, 1960, doi: 10.2307/1910129.
[15] M. Lukasiewycz, M. Glass, C. Haubelt, and J. Teich, “Efficient symbolic multi-objective design space exploration,” *2008 Asia and South Pacific Design Automation Conference*. 2008, doi: 10.1109/aspdac.2008.4484040.
[16] R. Tarjan, “Depth-first search and linear graph algorithms,” *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. 1971, doi: 10.1109/swat.1971.10.
[17] A. Pnueli, “The temporal logic of programs,” *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977, doi: 10.1109/sfcs.1977.32.
[18] L. Lamport, “‘Sometime’ is sometimes ‘not never,’” *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’80*. 1980, doi: 10.1145/567446.567463.
[19] M. Y. Vardi, “Sometimes and not never re-revisited: on branching versus linear time,” *CONCUR’98 Concurrency Theory*. pp. 1–17, 1998, doi: 10.1007/bfb0055612.
[20] P. Wegner, “Concepts and paradigms of object-oriented programming,” *ACM SIGPLAN OOPS Messenger*, vol. 1, no. 1. pp. 7–87, 1990, doi: 10.1145/382192.383004.
[21] A. Snyder, “Encapsulation and inheritance in object-oriented programming languages,” *Conference proceedings on Object-oriented programming systems, languages and applications - OOPLSA ’86*. 1986, doi: 10.1145/28697.28702.
[22] G. Kiczales *et al.*, “Aspect-oriented programming,” *ECOOP’97 — Object-Oriented Programming*. pp. 220–242, 1997, doi: 10.1007/bfb0053381.

[23] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, “NuSMV: A New Symbolic Model Verifier,” *Computer Aided Verification*. pp. 495–499, 1999, doi: 10.1007/3-540-48683-6_44.
[24] A. Cimatti *et al.*, “NuSMV 2: An OpenSource Tool for Symbolic Model Checking,” *Computer Aided Verification*. pp. 359–364, 2002, doi: 10.1007/3-540-45657-0_29.
[25] A. Hegedus, A. Horvath, I. Rath, and D. Varro, “A model-driven framework for guided design space exploration,” *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 2011, doi: 10.1109/ase.2011.6100051.
[26] S. Neema, J. Sztipanovits, G. Karsai, and K. Butts, “Constraint-Based Design-Space Exploration and Model Synthesis,” *Embedded Software*. pp. 290–305, 2003, doi: 10.1007/978-3-540-45212-6_19.
[27] F. Cieslok, H. Esau, and J. Teich, “EXPLORA — Generic Design Space Exploration During Embedded System Synthesis,” *Architecture and Design of Distributed Embedded Systems*. pp. 215–225, 2001, doi: 10.1007/978-0-387-35409-5_21.
[28] T. Saxena and G. Karsai, “MDE-Based Approach for Generalizing Design Space Exploration,” *Model Driven Engineering Languages and Systems*. pp. 46–60, 2010, doi: 10.1007/978-3-642-16145-2_4.
[29] T. Basten *et al.*, “Model-Driven Design-Space Exploration for Embedded Systems: The Octopus Toolset,” *Lecture Notes in Computer Science*. pp. 90–105, 2010, doi: 10.1007/978-3-642-16558-0_10.
[30] M. F. S. Oliveira, E. W. Brião, F. A. Nascimento, and F. R. Wagner, “Model driven engineering for MPSOC design space exploration,” *Proceedings of the 20th annual conference on Integrated circuits and systems design - SBCCI ’07*. 2007, doi: 10.1145/1284480.1284509.
[31] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, “MiniZinc: Towards a Standard CP Modelling Language,” *Principles and Practice of Constraint Programming – CP 2007*. pp. 529–543, doi: 10.1007/978-3-540-74970-7_38.
[32] D. L. Poole and A. K. Mackworth, *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, 2017.
[33] A. Rensink, “GROOVE > Grammars,” *GRaphs for Object-Oriented VErification*, 02-Jul-2018. [Online]. Available: https://groove.ewi.utwente.nl/downloads/grammars. [Accessed: 23-Jan-2020].
[34] D. Pellier, “PDDL4J is an open source library under LGPL license whose purpose of PDDL4J is to facilitate the development of JAVA tools for Automated Planning based on PDDL language (Planning Domain Description Language),” *github.com*, 19-Mar-2019. [Online]. Available: https://github.com/pellierd/pddl4j. [Accessed: 11-Feb-2020].

Appendix A

Contents

Chapter 1

Motivation

In Computer Science the fields of AI planning and model checking share the domain of state space exploration.

The field of planning (commonly named the field of AI planning) focusses on solving problems by finding and optimising a sequence of actions to perform on a subject system to realise a desired solution. In this field state space exploration is one method of solving these problems by modelling the problem into a system for which the state space can be searched for a solution. These problems are on topics like task scheduling, navigation of one or more agents. Puzzles are often used as toy problems as they can encode the complexity of problems without the need for much contextual details and often have a easy to understand visual representation.

Model checking is a field more closely related to verification of system behaviour, when subject to software systems it is also a method for software verification. Model checkers operate on a model of some system and are used to verify properties of the associated state space. For example model checkers can be used to verify that a system (or rather its model) cannot end up in a deadlock or some other error state. In model checking the exploration of the state spaces are usually performed such that they are able to notify the user of error states that are found as well as being able to provide proof. This proof includes a list of operations to perform on the starting state of the model to put it into the found error state.

Each field is of great importance in its own right and hard work is being done to progress these fields. Sometimes the fields intersect or overlap and knowledge are shared. The domains of AI planning and model checking come together on one big topic, in particular, state space exploration. Exploration of state spaces is the subject of many different research groups aiming to build and improve the algorithms, techniques and tools for this area. There are tools such as LTSmin, mCRL2, GROOVE which are large tool-sets with many years of research and development behind them. Beyond this, there are hundreds of works using various exploration algorithms and techniques.

While existing techniques are developed for specific tool-sets and code-bases there is little to no attempt to take a step back from the highly specialised tools and build a reusable model for the existing technology. While the existing tools keep getting improved it is relatively difficult for novel tools within the same domain to leverage those techniques. Reinventing the improvements to a novel tool would take away time and monetary resources unnecessarily.

1.1 Goal

The goal of this thesis is to analyse the algorithms of state space exploration and derive a framework from which such algorithms can be integrated. Beyond the purpose of implementing the algorithms themselves, we also want to make algorithms first-class citizens of the framework such that they can be reused by other algorithms, applications and code. This opens up widespread usage and extension of state space exploration algorithms. External parties from outside the scientific community can then also apply the exploration and modelling techniques through the problem-agnostic framework.

1.2 Who is going to use it?

A framework for state space exploration serves three primary user groups within the computer science discipline. First, are the developers of state space exploration based applications. This would include virtually any application that uses state space exploration in a major capacity. The existing tools and toolsets for state space exploration have been very well developed with lots of years of research and improvement behind them. Some of those frameworks are highly specialised and retrofitting them with a novel framework for exploration algorithms would probably be to the detriment of the work done for those tools already. However, there are (probably) many tools which use state space exploration which do not have their developers focus on improving the state space exploration aspect of the application. Perhaps those tools have integrated some form of state space exploration as an after-thought or to only serve a smaller subset of features. In such cases integrating with a framework would be ideal to bolster the state space exploration capabilities and efficiency. Using a framework promotes re-use of resources over different implementations that use the framework, meaning improvements made to the framework can be utilised by other applications as well. This is not only limited to the efficiency or stability of the framework but also to the components that function within the framework, such as the state space exploration algorithms themselves as they become re-usable between different applications. Using the latest version of algorithms and getting their performance benefits could simply be a case of updating dependencies for the developers of applications. This is in contrast to the current standing, which requires the developers to be experts on state space exploration such that they can understand and implement the newest techniques published by their peers.

Second, the users of the aforementioned applications would also benefit as the framework would likely support a wider range of features than initially implemented for the feature-set the tool serves. The sharing of algorithm implementations that fit within a common framework for state space exploration means that algorithms made for one application could be imported into another application and used by the end-user instead of the (limited) set of initial algorithms available by the developers of the application. Incorporating commonly found concepts in a singular framework and having that framework being used by multiple tools offers those users a relatively comparable experience over different tools. The framework solidifies the terminology across the state space exploration domain and ensures a certain level of customisation and configuration to the end-user.

Third and finally, there are the developers of state space exploration algorithms to consider. A common framework for them would mean that their new algorithms would be supported by a wide range of tools almost immediately. Currently, experts in state space exploration build their algorithms in existing tools. While sometimes it gets implemented in multiple tools this is not guaranteed and doing so is a significant investment for the researcher/developer of such algorithms and improvements. With a reusable framework for state space exploration, the researchers would in an ideal scenario only have to implement the algorithm once and publish it in a format that users of the framework can import. A framework for state space exploration as envisioned here would treat algorithms as first-class components of the framework. This will make the algorithms and the components that comprise them re-usable across different algorithm implementations and re-usable within other algorithms. Algorithms would become a building block that designers of algorithms can re-use in their algorithms. This powerful concept is used throughout computer science. Re-usable components, re-usable solutions and open-source code sharing have become a popular way to share knowledge and solutions within the computer science discipline. This isn't limited to the scientific community, many projects are being hosted, shared and reused throughout the world, commercial, scientific and amateur alike. Billions of lines of code are shared, let us share state space exploration.

1.3 Report objective

Before a general framework can be devised, the existing landscape of state space exploration has to be explored[1]. This will canvas existing algorithms to find what makes them similar and what sets them apart.

> What are the similarities between existing state space exploration algorithms and what differentiates them?

To answer this question first we will define terminology about state space exploration in Chapter 2, after which we will discuss a series of algorithms in Chapter 3. Then in Chapter 4, we go over some features that a framework supporting those algorithms should contain. In Chapter 5 we go over a set of case studies that such a framework could be subjected to and we close off in Chapter 6 with the research questions for building the framework. Chapter 6 also details the methodology for making the project a reality as well as detail how the framework will be validated against the research questions and the claims made.

[1] Pun intended

Chapter 2

Background

First, we define components of a state space and state spaces themselves. Secondly, we will define some useful terminology to use throughout the report. Finally, we define state space exploration and modelling.

2.1 Graphs & state spaces

This chapter uses terms from graph theory to explain terminology of state spaces. Some familiarity with graph theory is required by the reader, at minimum the basic terminology of graph theory[20]. A state space is a directed graph consisting of states (vertices from graph theory) and transitions (edges from graph theory).

When talking about state spaces we may mention a 'section' of a state-space, this complements the term 'subgraph' from graph theory. A section of a state space may also be mentioned in relationship to a particular state; that relationship is reachability (detailed in Section 2.8.2) of states from that mentioned state.

2.2 States

A state is a value from some domain, the state's domain is the domain the state belongs to. The domain contains all possible values for a state.

A state represents a combination of properties and variable assignments for some known system.

The structure of such a state is able to represent that specific system in some condition it can be found in. A state on its own represents that system at a given moment in time. Since the structure of a state is left up to the architect of the system model there is no prescribed data structure for it. Instead these will be treated as elements that can later be extended upon with predicates and mapping functions.

Definition 1. A state is an element in the set S which holds all states the underlying system can occur in.

$$S = (s_0, s_1, ..., s_n)$$

2.3 Transitions

A transition represents a relation between two particular states, the relation is directional having a source state and a target state. We denote a transition t between s_a and s_b labelled as l by $s_a \xrightarrow{l} s_b$ indicating s_a as the source state and s_b as the target state of the transition. It is possible for some pair of states to have multiple transitions in the same direction, it is the function of the label to distinguish them. This label uniquely identifies the transition and like states allow additional properties to be tied to transitions by defining them as functions mapping labels to some value in the domain of that property.

Definition 2. The set of all transitions T holds all transitions

$$T \subseteq S \times L \times S$$

The existence of a transition indicates the relation between two states s_a and s_b within that state space. For a transition t_{ab} in The transition exists if and only if when an instance of the system occurs in state s_a, that system is able to transition immediately to state s_b without occurring in any intermediate state.

Transitions are directional, with the transition pointing from its starting state to its target state. Given transition t, with starting state s_a and target state s_b, we can say that s_a has an outgoing transition t and s_b has an incoming transition t.

We define transitions as tuples of two states and a label: $T \subseteq S \times L \times S$. A transition is written as $s_0 \xrightarrow{l} s_1$ which represents a transition t with s_0 as source, s_1 as target and a label l. The label of a transition allows more data to be associated with it in the future.

Definition 3. The incoming transitions of a state are all transitions which have that particular state as their target state. Similarly, the outgoing transitions of a state are all transitions which have that particular state as their starting state.

We will also use the following notation to denote the neighbours of a state s_x, filtered by the direction of their transition to s_x. To do this we first define two functions T_{in} and T_{out} which represent the sets of incoming and outgoing transitions of a given state s_x (respectively).

$$T_{in}(s_x) = \{t_q \in T | s_y \xrightarrow{l_q} s_x \wedge s_y \in S\}$$

$$T_{out}(s_x) = \{t_q \in T | s_x \xrightarrow{l_q} s_y \wedge s_y \in S\}$$

$$S_{in}(s_x) = \{s_y \in S | s_y \xrightarrow{l_q} s_x \in T_{in}\}$$

$$S_{out}(s_x) = \{s_y \in S | s_x \xrightarrow{l_q} s_y \in T_{out}\}$$

The transitions described by $T_{in}(s_x)$ are those transitions which point have s_x as target state. Similarly, the outgoing transitions set function T_{out} (s_x) reveals all transitions which have s_x as source of the transition. Additionally the functions $S_{in}(s_x)$ and $S_{out}(s_x)$ reveal the corresponding target and source states for those transitions (respectively).

2.4 State space

A state space is a tuple made up of a set of states and a set of transitions. It is represented by a directed graph where states are represented by vertices and transitions by edges.

$$A = \langle S_A, T_A \rangle$$

$$T = (t_0, t_1, ..., t_n)$$

2.4.1 State space explosion problem

The exploration of state spaces is often limited by the size of the state space. A large state space requires more states and transitions to be evaluated, meaning a relatively smaller section of the total state space gets explored in a fixed amount of time. Besides the reach exploration has over time also the memory consumption of the exploration strategy can be a bottleneck for the exploration. The available memory can simply run out and no new states can be saved or generated anymore. In scenarios where every state has a lot of outgoing transitions, the increase of the known state space can be as bad as exponential (relative to the number of components in the system). With petri-nets (see Section 2.9.2) the number of places in the system exponentially increases that state space size for the petri-net. In the model checking community the state space explosion problem is one of the areas where efforts are focused continuously[4][15].

2.5 Known states

Exploration divides the states of a state space into two disjoint subsets, the known and unknown states. Known states are being remembered and are kept track of. The remaining states are the unknown states.

A known state can have an outgoing transition which points to an unknown state. When the exploration algorithm finds a state from the unknown states it is called the discovery of that state. The discovery of a state adds it to the discovered states set and the algorithm may also put the state into the known states set, this is called saving the state.

$$\emptyset = S_{known} \cap S_{unknown}$$

$$S_A = S_{known} \cup S_{unknown}$$

$$S_{discovered} \subseteq S_A$$

The algorithm may also put the state into the known states set upon discovery, this is called saving the state. The condition under which the algorithm puts the state into the known states set will be noted by P_{save}. If the algorithm puts the discovered state into the known states set then the state no longer is part of the unknown states set. Otherwise, if the algorithm does not put the state into the known states set then it remains in the unknown states set, this is called discarding the state.

$$S'_{discovered} = \begin{cases} S_{discovered} \cup \{s_x\} & \rightarrow \quad P_{save}(s_x) \\ S_{discovered} & \rightarrow \quad \neg P_{save}(s_x) \end{cases}$$

$$P_{save} : S \mapsto boolean$$

$$S'_{known} = S_{known} \cup \{s_{discovered}\}$$
$$S'_{unknown} = S_{unknown} \setminus \{s_{discovered}\}$$

A state which has been discovered can, therefore, be discovered again if the state remains in or returns to the unknown states set. The discovered states set is a set which can contain states from both the known and unknown states sets.

Definition 4. A state space is fully known if the set of unknown states is empty.

The set of known states can be divided into another two disjoint subsets, the frontier and the heap.

$$\emptyset = S_{frontier} \cap S_{heap}$$

$$S_{known} = S_{frontier} \cup S_{heap}$$

Moving a state from the known states set to the unknown states set is called purging or forgetting that state. This may happen if the state is deemed to serve no purpose by remaining known. Purging states mostly happens to states within the heap set. This makes the state unknown and it cannot be identified as a previously known state when discovered through an outgoing transition from a known state.

$$S'_{known} = S_{known} \setminus \{s_{purged}\}$$
$$S'_{unknown} = S_{unknown} \cup \{s_{purged}\}$$

2.6 Frontier

The frontier is the set of states for which not all outgoing transitions have been evaluated. When a state is being evaluated, its outgoing transitions are enumerated and each transition is evaluated sequentially. After the evaluation of the state, it is moved from the frontier to the heap. In the area of graph theory and graph traversal algorithms this is also commonly referred to as the set of 'open' states.

2.6.1 Order

Frontiers have need an order in order to be able to deterministically know what state to explore next. Two examples of types of orderings on a frontier are:

Insertion order Using a FIFO or LIFO queue as frontier results in an automatic order of the frontier based on discovery order of the states.

Ordered by property The frontier is sorted based on a function that maps states in the frontier to a comparable unit (e.g. an integer). This function may be an exact value or a heuristic can be used.

2.6.2 State constraints

Discovery of a state adds it to the frontier but there may be constraints placed on the frontier that prevent some states from being added to the frontier upon discovery. These may be state related constraints that filter out an unwanted or costly to explore section (subgraph) of the state space. These constraints can be encoded into a predicate $P_{frontier}$:

$$P_{frontier} : S \mapsto boolean$$

The predicate $P_{frontier}$ identifies states which satisfy the constraints and allow a state to be inserted into the frontier.

2.6.3 Limited size

So far the concept of frontier did not have a sense of maximum capacity. There however may be scenarios where limiting the frontier is required by the algorithm to function property or it might be a beneficial constraint on memory resources. When limiting the size of the frontier the order of the frontier automatically causes the lowest ranked state to be dropped from the frontier. The state that gets dropped may be the state being inserted (for example when using FIFO order) or it may be a state already in the frontier. The size constraint may seem to act like a filter on the frontier but it is notably different in that the outcome of an insertion to a limited size frontier is primarily based on the size and not the intrinsic value of the state to be inserted. A frontier with a fixed size of 1 allows for running simulation style exploration where only a single state is progressed. When limiting the frontier to a single state the application of the heuristic results in the best state of the discovered states during evaluation of a state to survive.

2.6.4 Generator frontier

Instead of using a set of states to store the states in the frontier, one can also use a set of state generators to hold the states in the frontier. A generator is a function (G) that generates the next neighbour state based on the current neighbour. The neighbour states are considered generated once the generator function has returned them. Binding this with the last generated neighbour (s_x) creates a tuple which acts as an iterator over the set of states the generator is able to generate. The iterator (I) provides the current value of iteration over the generator set and a means to progress through the set, albeit in only one direction.

$$G_{next} : S \mapsto boolean$$
$$s'_x = G_{next}(s_x)$$
$$I : (s_x \in S, G_{next})$$
$$(s'_x, G_{next}) = (G_{next}(s_x), G_{next})$$

The iterator tuple can also be extended with a predicate that indicates when the last state the generator can generate is reached to support generators for finite sets of states.

$$P_{last} : S \mapsto boolean$$

$$(s_x \in S, G_{next}, P_{last})$$

$$(s_x, G_{next}, P_{last})' = \begin{cases} (G_{next}(s_x), G_{next}, P_{last}) & \rightarrow \quad \neg P_{last}(s_x) \\ (s_x, G_{next}, P_{last}) & \rightarrow \quad P_{last}(s_x) \end{cases}$$

These generators make it possible to work with infinite sets of states in the frontier. A drawback however is that there is limited support for sorting, only the generated states can be sorted. To address this a cache could be used to increase the number of states peeked ahead from the generator.

2.7 Heap

The other subset of the known states is the heap, this is a set that stores the known states which have been evaluated. In graph traversal algorithms this is also called the set of 'closed' states. The main purpose of the heap is to identify known states while evaluating transitions. The heap together with the

2.7.1 Hashing

Instead of remembering the full representation of states every state in the heap can also be converted into a compressed representation of that state. The value a state gets converted into is called the hash of that state. In computer science hashes are strings of bits, for example a fixed-size 32 bits hash can be represented by an integer. Comparison of bit strings can easily be much faster than testing equality of the original states if their structure is complex enough. Hashes have a finite value range and when mapping arbitrary values to another value range there is the possibility of different values mapping to the same hash value. When two values map to the same hash this is called a hash collision. When using a hashing heap it becomes possible for an unknown state to be falsely identified as a known state when its hash collides with the hash of a known state. Let s_a and s_b be some states in a state space which map to the same hash value and the states are reachable from the initial state. Let s_a be the first state to be discovered, after which s_b will be discovered. However, because the heap contains the hash of s_a already the exploration algorithm will not consider evaluating the transitions of s_b as it is falsely marked as a known state.

2.7.2 External storage

A simple implementation of a heap would keep the states in the computer memory. While RAM memory is fast it is a relatively expensive resource in comparison to persistent memory storage devices such as tapes, hard drives and solid state drives. Using a file system to store the states enables cheaper memory resources to be used and on most computer systems will provide more available storage for the heap. To store a state space to a file system a conversion of states needs to happen to turn the binary representation of a state into a self contained binary representation that can be written to a file. Files on file systems generally consist of two main components: a file name and binary content. The filename introduces the hash collision problem as described in Section 2.7.1, but the problem is not entirely the same as each file on the file system can contain a collection of states. When testing if a state is known the binary representations of states with colliding hashes can be used to identify the states uniquely still.

2.8 Exploration

With the building blocks of state spaces defined we can now define exploration. Exploration is thought of as the execution of an algorithm which attempts to find states in the state space and put them in the known states set. State spaces can be very big and as such, each exploration

is done with a specific goal in mind. At the beginning of exploration, all states from the set of initial states are in the frontier. To increase the set of states which are known, the known states can be explored. Exploring a state means following the outgoing transitions to attempt to discover states. Exploration only ever follows outgoing transitions, following incoming transitions is not possible as the source state is not guaranteed to be reachable by the system. State space exploration itself is simply repeatedly exploring known states to increase the set of known states. Exploration of the state space uses states from the frontier to discover unknown states. Exploration terminates when there are no more known open states, this is called the exhaustion of the state space. Instead of exhausting the state space, the exploration may also be searching for a state which matches a given predicate. When using such a predicate, the states for which the predicate holds are called accepting states.

2.8.1 Initial states

Exploration of a state space always starts from one or more initial states. These are states known before any exploration has been performed. When exploration starts, the frontier contains all the initial states. In this report we may refer the the set of initial states by using $S_{initial}$ in formulae.

2.8.2 Reachable states

Exploration of a state space occurs through evaluation of the outgoing transitions of a state. This leaves the possibility for states to exist that may not be reached during exploration. Following the definition of reachability from graph theory, this divides the states in a state space by the initial states into two sets: states that are reachable from the initial states and that are not. States that are not reachable from the initial states may be called unreachable. We will use the terminology to mean that reachability is always implicitly in relation to the initial states of the exploration, when those initial states are not mentioned.

2.8.3 Goal states

There are several ways exploration can terminate, one of those ways is when exploration is focused on discovery of a specific state. When searching for a specific state or for states that match some requirements exploration may also be terminated early when those states have been encountered. A goal state is a state that matches a set of requirements and for which the existence and/or identify of those states is the primary goal of exploration. To identify goal states a predicate P_{goal} exists which represents the requirements and identifies states that match them.

$$P_{goal} : s \mapsto boolean$$

Such a predicate can be used to terminate exploration when a state satisfying the predicate is encountered. This method of terminating the exploration can have many variations, for example collecting a number of goal states or a set of requirements on the set of goal states prior to terminating the exploration before frontier exhaustion.

2.8.4 Traces

At the end of exploration a desired goal state may have been found but the state itself is not the only result possible for exploration. It often is very useful to be able to identify the path from the initial states to goal states. Paths from an initial state to some other state in the state space are called a trace.

A technique of generating traces is to store the source state of an outgoing transition when it leads to discovery of a state upon evaluation. The transition is stored on the target state which allows for reconstruction of the trace to the initial state at a later point in time. This can be used to provide proof that a state is reachable from an initial state and also is used in model checking as part of counter-example generation.

2.8.5 Depth

The depth of a state is its distance to the initial state measured over the trace to the state. The depth of a state s_n can be defined as being one higher than the lowest depth of its neighbours that have an outgoing transition to the state. Also the depth of initial states is defined as zero.

$$f_{depth}(s_x) = \begin{cases} 0 & \rightarrow \quad s_x \in S_{initial} \\ 1 + \min\limits_{s_n \in S_{in}(s_x)} f_{depth}(s_n) & \rightarrow \quad s_x \notin S_{initial} \end{cases}$$

Depth can be used to limit the exploration of states by only allowing states below a predefined depth to enter the frontier. This limits the size of the state space that will be explored and can limit algorithms that may unfavourably prefer exploring states at greater depths (e.g. DFS, see Section 3.4).

It is possible for an exploration to find multiple paths to the same state but result in different depths when finding the same state. For example, let there be a state s_a and two neighbouring states s_b and s_c connected by transitions t_{ab} and t_{ac} respectively. Let there also be a transition t_{bc} going from s_b to s_c. It is possible that an exploration which starts at s_a will evaluate t_{bc} before t_{ac} if s_b is discovered before t_{ac} has been evaluated. In that case the depth of s_b upon discovery is determined as 2 while the lowest possible depth for the state is 1 (when discovered through t_{ac}). The definition of the depth function given previously is complete in the sense that it is aware of all neighbours of s_x that have outgoing transitions to s_x. It is up to the exploration algorithm to define the behaviour when a known state is encountered at a lower depth when evaluating transitions of another state.

2.8.6 Workflow

We want to establish a term to describe the context from which a state space exploration algorithm may be used, this is the workflow. The workflow may be an algorithm or some system that uses state space exploration to solve a specific problem within that context.

A workflow likely includes pre- and post-processing of the state space. For example, the iterative deepening (depth first) search algorithm is a state space exploration algorithm which repeatedly uses a depth-limited depth first search algorithm to explore the state space to explore the state space. The iterative deepening search algorithm is a workflow from which another state space exploration algorithm is used. In this case the workflow is an algorithm that is also a state space exploration algorithm itself but that doesn't have to be the case. Section 3.8 details the application of genetic algorithms to state space exploration. There the genetic algorithm is the workflow which uses a specific state space exploration algorithm to evaluate the score of a chromosome.

2.8.7 Simulation

A special form of exploration is where the exploration focuses on picking from every state in the frontier only a single neighbouring state to progress the exploration by. This is called simulation as the exploration by following a single outgoing transition essentially transformed the state into another one, simulating the execution of that system. The frontier is not required to be limited to a single state but the key characteristic of the search is that the exploration discovers at most a single state upon evaluation of a state. Due to states not having all their neighbouring states discovered upon evaluation there is no guarantee that the exploration will exhaust the state space or that a goal state will be found. Algorithms that fit this category may not require a heap as searches are not intended to be exhaustive and the frontier will not grow beyond an initial fixed size (based on the initial states set).

2.8.8 Query

The workflow defines what exploration strategy gets used and how it gets used. To encapsulate that information we denote the exploration query. The information we want to encapsulate includes all the information to start exploring a state space. The algorithm used for exploration

is included, with parameter assignments to any configurable parts of the algorithm and the set of initial states to be able to start exploring. The inclusion of 'parameter assignments to any configurable parts' is rather vague and depends somewhat on the implementation of the algorithm as to which options are there. But it also has some common parts that many algorithms use such as the goal predicate (P_{goal}, see Section 2.8.3).

2.8.9 Snapshot

The state of an algorithm during exploration is considered the set of known states, discovered states, frontier, heap and all additional data used and created specifically by the algorithm. To make it less confusing to talk about the state of exploration we will call it the exploration snapshot. The set of unknown states is not part of the snapshot, as otherwise the entire state space would be known already. The contents of a snapshot are decided by the algorithm used for exploration. Depending on the algorithm used some of these sets will be used and also additional sets and functions may be part of the snapshot. An example of such an extension is a parent function, which returns for a given state the state from which it was discovered. This extra context from exploration could be used for example to produce a trace through the state space as a result of the algorithm.

2.9 Modelling

A common practice for creating exploration queries is to define a system model and initial state for the system. The model consists of predicates that define properties for states and a set of mutations. A mutation is a tuple of a precondition predicate and a post-condition predicate. The mutations are used to generate outgoing transitions for a given state. A mutation is applicable if the precondition holds for the given state. The outgoing transitions are then created for all states which can be created for the states for which the post-condition predicate holds. Essentially the mutations are templates for transitions in the state space.

States in a state space that is created from a model can be either an extentional state or an intentional state. An extentional state is described by the model directly (also called a literal definition). An intentional state, however, is not defined by the model but it exists as its properties do not break the rules of the model. The existence of an intentional state does not guarantee that it can be reached from any other state, including the initial states.

2.9.1 Planning domain definition language (PDDL)

The planning domain definition language (PDDL) is used in the field of AI planning to encode systems into a model. It was originally developed by D. McDermott, et al[10] as part of organising the first International Planning Competition possible. The language emerged to unify problem descriptions such that planning algorithms could be tested and compared more reliably. This has led to many problems being encoded in this language and many planning algorithms are implemented with support for PDDL.

There are two parts to the PDDL language, domain definitions and problem definitions. The domain definition describes the domain in which a problem exists, consisting of predicates and actions. The predicates are symbols which do not have some implementation, rather the problem definition assigns the value of the predicates for the initial state in order to formulate what that state is. The actions are a bit more complex, they describe how states can be mutated. The mutation of a state creates a new state (that has the mutation applied) that is automatically the neighbour of the original state. Actions consist of a set of predicates describing the pre-condition under which the action may be applied and a set of predicates describing the post-condition of the state.

The problem definition defines a system by a collection of objects and defines the initial state by assigning for the objects which predicates do and do not hold.

The goal is described by the problem definition also, by a boolean expression that defines what combination of predicates should hold for which objects. Similar to our definition of states

in Section 2.2 the PDDL language does not have an explicit definition of a state other than a collection of predicates which define the properties it has.

2.9.2 Petri nets

Another example of a modelling language for systems is petri-nets[6]. A petri net model consists of places, transitions (not to be confused with our terminology) and directed edges between them. Every edge is between a pair nodes consisting of one place and one transition. This makes the graph a bipartite graph and one of the disjoint sets of nodes contains all places and the other contains all transitions (again not our transitions).

In petri net modelling there is also the concept of tokens. A token is an entity that is assigned to one of the places in the petri net. When a token is assigned to a place, that place is considered to 'hold' or have ownership over that token. A place may hold multiple tokens. Extensions of the petri net theory allow for those tokens to be labels which can hold arbitrary data. A state in the state space of a petri net model consists of assignments of tokens to the model's places. Petri nets are used for modelling of asynchronous and parallel systems. This is because petri nets model supports actions that can be applied simultaneously but not together to result in multiple neighbouring states. This essentially models all possible occurrences of events and actions whenever multiple actions are possible that interfere with each other.

2.10 Heuristics

Heuristics have already been mentioned in Section 2.6.1 where a heuristic could be used to sort a frontier. A heuristic is a class of algorithms (term algorithm does not relate to exploration here) that solve problems by providing approximate answers rather than exact answers. The two main applications for this is faster computation of such answers or loosening the requirements for a solution. Heuristic algorithms are usually derived from some original algorithm that finds the exact answer to the problem. In state space exploration heuristics can be used to approximate properties of states where the property relates to other states in the state space. Since finding the exact value of those properties could require exploration of the entire state space or large sections of it there is good cause to use approximations during the execution of an exploration algorithm.

2.11 Informed algorithms

Exploration algorithms can be categorised as either uninformed or informed. An informed algorithm uses knowledge about the positioning of states within the state space, in particular related to nearby goal states. There informed algorithms use heuristics to derive these properties (as concrete knowledge about states as they relate to goal states would reveal the goal state).

Chapter 3

Algorithms

3.1 Example problems

When describing the algorithms some examples will be used to play out the execution of the algorithm over an example problem. This section introduces examples and explains some of the terminologies to come.

3.1.1 Maze

Mazes are well represented in graphs, the mazes used in the examples of this chapter use a specific representation. The mazes are 2d orthogonal grids of square cells. Two cells are connected unless a wall is present on the grid-line separating them. The maze together with the player forms the game being played. A state of the game consists of the state of the maze (layout and content) and the player's position. Since the layout of the maze does not change we can say that the state of the game simply is the position of the player. When talking about positions and cells in the maze, each cell has a distinct state in the state space where the player is visiting that cell. In other words, a cell has a one-to-one relationship with a state in the state space. The player can move through the maze by changing their position to that of a neighbour cell. The maze includes a goal cell, the goal of the player is to reach this goal cell. There exists a class of mazes called 'perfect' mazes, these are mazes which have only one solution to the maze. This means there are no loops in the maze which would allow for multiple routes to the same goal or multiple goals to reach. When concerning the length of the solution this will be referred to as the true length of the solution or path through the maze.

3.1.2 Dining philosophers

A popular problem to solve using model checking and state space exploration is the dining philosophers problem. The problem itself[2] asks for a solution in the form of an algorithm that guarantees no deadlock occurs. State space exploration is used to verify that there is no deadlock state which is reachable from the initial state. In the dining philosophers problem, there is a set number of philosophers dining at a round table. In between every two philosophers sitting next to each other is a spoon (the number of spoons equals the number of philosophers). Each philosopher has a simple program:

while *forever* **do**
    pickup the left or right spoon;
    pickup the other spoon;
    eat for a while;
    put both spoons down;
end

They slowly pick up and drop spoons and there exist only two possible deadlock states: all the philosophers are holding the left spoon at the same time or all philosophers are holding the right spoon at the same time. The large state space in combination with the very specific

deadlock condition makes it difficult to find the deadlock at any point a philosopher may pick up a second spoon or the wrong spoon first and the path to the deadlock becomes a few steps longer. Using random or blind walks through the state space will make it very unlikely the deadlock will be found in a reasonable amount of time.

The dining philosophers problem suffers from the state space explosion problem because there is a large number of states in the state space. The size of the state space can be computed by taking into account the possible states of any spoon. A spoon can either be on the table, in the hand of the left-hand side philosopher or in the hand of the right-hand side philosopher. This gives three possible states per spoon and a state of the table can be represented by the state of its spoons (rather than by its philosophers). This makes the size of the state space equal to 3_s^n where n_s equals the number of spoons on the table. The dining philosophers problem has a large state space size that exponentially increases in size as the problem is scaled up in size.

This problem will be used as example for algorithms to show how they may deal with the state space explosion problem and how they might try to find the deadlock states faster.

3.2 Travelling salesman problem

A well known planning / optimisation problem is the travelling salesman problem. It was originally described by Hassler Whitney in 1934 at a conference. In the problem there is a salesman who wants to visit all cities from a given set of cities. To visit each city the salesman has to travel between them. The distance between every possible pair of cities is known. The travelling salesman start in some city, visit each other city once and finally end up at the initial starting city again. The goal is to find the shortest possible route for such a path. This problem is NP-hard and dynamic programming is a suitable method to solve it.

To encode the problem into a state space a state can represent a possible solution, with its neighbours being mutations or variations of that solution. A goal state will be defined as any valid solution that meets the criteria of the problem (a visit to each city once).

In the context of the travelling salesman problem we will use the symbol c to denote a city and c_n to be some city labelled n. Next we define a function $f_d(c_x, c_y) : \mathbb{I}^+$ which maps a pair of cities to the distance between them.

A state in the state space will be an array of cities with the order reflecting the order in which the salesman visits them. For simplicity we can imply that the first element in the array is the city the salesman starts in and implicitly the salesman will return from the last city in the array to the first city in the array to complete the route travelled.

$$s : \{c_1, c_2, ..., c_n\}$$

The distance travelled for a given state can be computed by looping over the array and sum the distance between each city with the next city in the array. The city element will have its distance calculated between it and the first city in the array to complete the tour of cities. We denote a function $f_{ds}(s_x) : \mathbb{I}^+$ as the function that computes the distance travelled for some state s_x.

A state is not required to contain each city once, there may exist partial solution states which contain a subset of the cities that exist. A partial solution state cannot be a goal state as it violates the requirements for a solution.

The state space of the problem includes every possible solution, with no inherent way to determine the best solution that exists or way to identify which state is the optimal solution.

3.3 Breadth first search

The breadth first search (BFS) algorithm, originally described by Moore, E.F. [12], is a simple exploration algorithm. It is a very generic algorithm as it depends very little on the problem context and thus can be used in a very wide range of problem contexts. Using the terminology from chapter 2 we can describe the working of breath first search as well as identify its components: The algorithm explores states in a state space from an initial state. The exploration

uses an ordered frontier (see Section 2.6.1), specifically a FIFO queue. A key property of BFS is that the sequence in which states are discovered is such that when a state is discovered its depth is never lower than any state discovered before it. The side effect of the discovery order is that when a goal state is found that the path between the initial state to the goal state that can be identified by the backtrace is the shortest (in number of transitions) possible. The algorithm also guarantees that all reachable states from the initial state will be evaluated, simply by virtue of not having any state constraints on the frontier (see Section 2.6.2). These two properties make it a useful exploration strategy / algorithm in common scenarios where it is desirable to find the shortest trace to goal states. BFS uses a heap to keep track of discovered states to prevent duplicate exploration. BFS can terminate exploration early when finding a goal state, for this a goal state predicate needs to be supplied to the algorithm by the user in order to identify such goal states.

3.3.1 Pseudocode

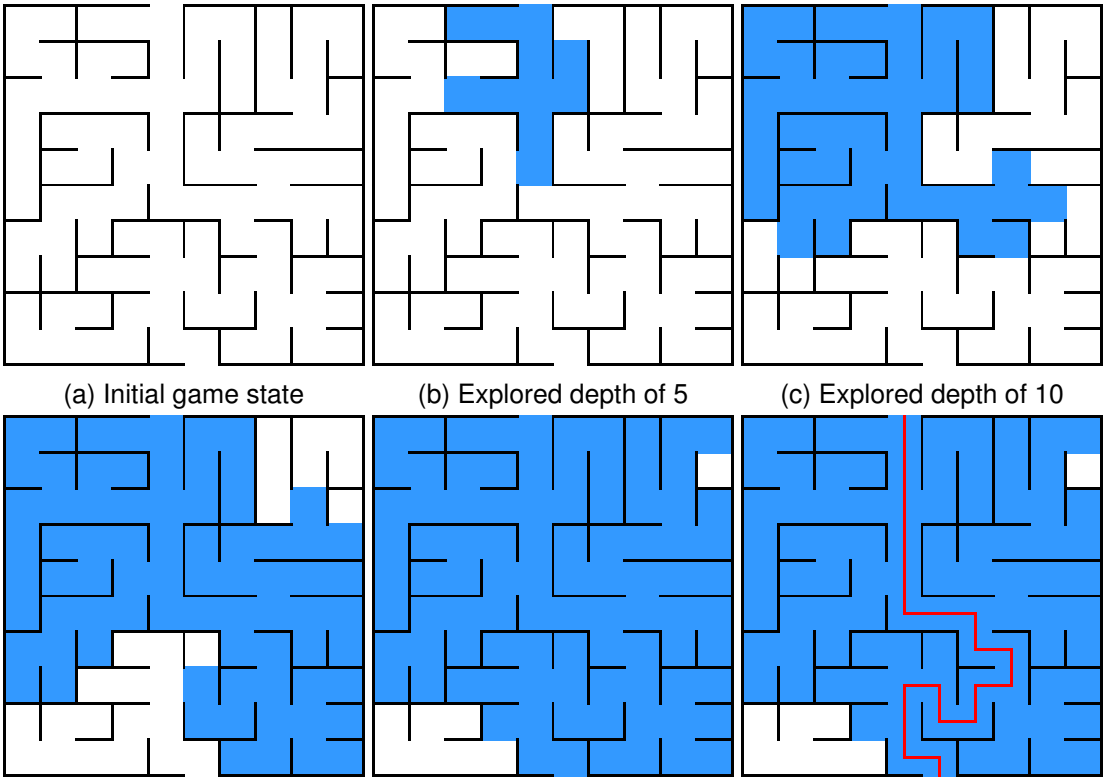
function `bfs`($S_{initial}$, P_{goal})
    $S_{frontier} = S_{initial}$;
    $S_{known} = S_{initial}$;
    while $S_{frontier} \neq \emptyset$ **do**
        pick s_c from the frontier, such that s_c is the state which had been in the frontier longest;
        $S_{frontier} = S_{frontier} \setminus \{s_c\}$;
        if $P_{goal}(s_c)$ **then**
            return s_c;
        end
        for $s_{neighbour} \in s_{out}(s_c)$ **do**
            if $s_{neighbour} \notin S_{known}$ **then**
                $S_{known} = S_{known} \cup \{s_{neighbour}\}$;
                $S_{frontier} = S_{frontier} \cup \{s_{neighbour}\}$;
            end
        end
    end
end

Algorithm 1: BFS algorithm implemented with a queue

The pseudocode for an implementation of the BFS algorithm is shown in alg. 1. In the pseudocode the BFS algorithm receives a set of states $S_{initial}$, this are the initial states from which exploration will start, and a predicate P_{goal} which identifies goal states. Before the main exploration loop starts the frontier gets filled with the initial state. The pseudocode iterates over the frontier until it is empty or until a goal state has been found. The algorithm iterates over the frontier and each iteration it evaluates a state from the frontier, noted as s_c. The selection of the state to be evaluated is deterministic as the order of the frontier is used to determine the state to be evaluated. After the state is picked it is removed from the frontier and then evaluated. Evaluation of a state consists of testing the state against the goal predicate. If the state is a goal state then the algorithm terminates (in this case by simply returning the identified goal state). Depending on the application of BFS the behaviour upon finding a goal state may differ from the pseudocode. If the state is not the goal state however, its neighbours are enumerated. Every neighbour ($s_{neighbour}$) which is unknown gets added to the frontier.

3.3.2 Example - Maze

When it comes to solving maze puzzles BFS is an algorithm that is often used when there is little to no additional information available. It guarantees to find the solution if one exists (given sufficient resources). An example trace is shown in Figure 3.1 with the explored cells (known states set) marked in blue at various intervals. The maze is being solved from the opening at the top and the goal cell is set to the opening at the bottom. In the problem context of solving a maze there is more information that could be utilised and other algorithms are better suited



(a) Initial game state (b) Explored depth of 5 (c) Explored depth of 10

(d) Explored depth of 15 (e) Solution at depth 19 (f) Solution

Figure 3.1: Progression of maze exploration using BFS

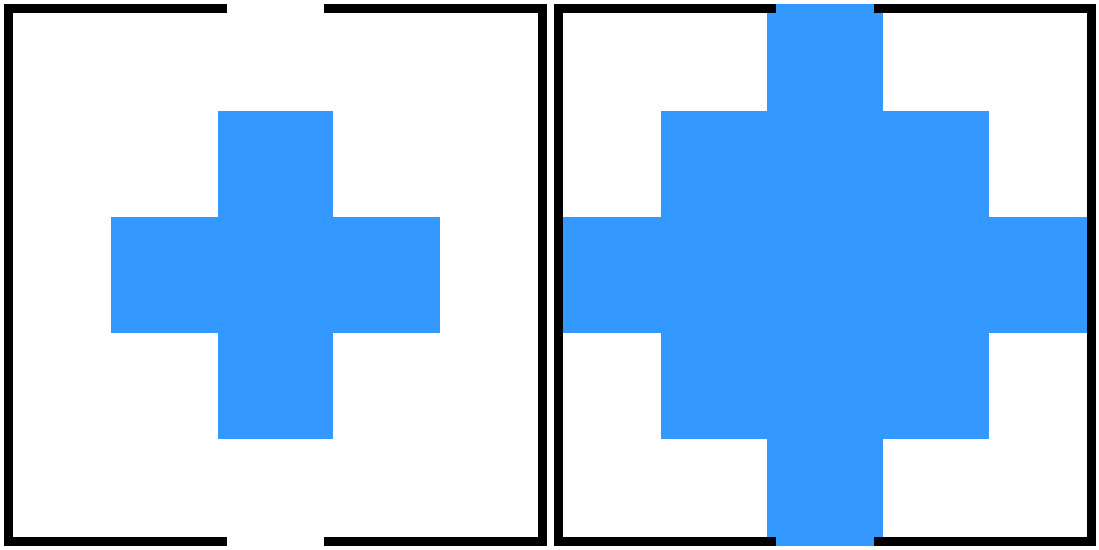
than BFS when they are able to use this information successfully.

It is possible to derive a worst-case complexity formula relative to the true distance to the goal. We can make assumptions about what the size of the explored state space is at a given depth because DFS explores the least deep states before exploring parts of the state space at greater depths. A worst-case scenario for BFS would be an infinitely large maze where the player starts in the center cell of the maze and there is no exit (so the algorithm searches infinitely). This will reveal the upper bound for the number of states discovered at some depth d. The first cell will be evaluated and 4 new states will be discovered, one for each open cell in each orthogonal direction the player can travel in. After evaluating those 4 neighbours all states of depth 1 will have been evaluated and the frontier will contain 8 states (see Figure 3.2). The same pattern continues for every depth following, with the number of states discovered per layer increasing by 4 after exploring all states at the same depth. This makes the number of states discovered at depth d equal to $4d$. To summarise the number of cells we can use the formula for the sum of first n natural numbers, multiplied by the growth rate of the number of states: $n = 4(d/2)(d+1) = 2d(d+1)$. This is then also the worst-case complexity for BFS in such mazes, the big-O notation results in being $O(d^2)$ with d being the depth at which the exit exists.

3.3.3 Example - Dining philosophers problem

When applying BFS on the dining philosophers problem it will always find a solution to the problem. But it is rather slow since there is no optimisation to find a deadlock state faster or with less memory usage.

The encoding of the problem requires a minimum depth to be explored before being able to encounter a deadlock state. For BFS this means that it has to discover all states at lower depths and evaluate them before being able to encounter a goal state. This makes BFS not very well suited because the memory needed to hold those states grows exponentially(Section 3.1.2) relative to the number of philosophers. The lowest depth the solutions exist at can be computed for n philosophers to be n when each philosopher picks up the correct spoon to trigger the deadlock immediately. This results in a memory footprint of 3^n states before the solution can be found. However, some of the states may be detected as equivalent when considering



(a) Known states after exploring depth 0 (b) Known states after exploring depth 1

Figure 3.2: Expansion of known states for maze with no walls

shifting the philosophers from positions as well as mirroring the hands with which philosophers have picked up their spoons. With this optimisation 3^n is the upper bound but it still grows exponentially since the optimisation does not address the relationship between philosophers and the spoons which cause the state space to grow exponentially. For this problem, given that there are enough resources to execute the exploration, BFS is a suitable solution because it reliably produces the desired result as well as a minimal trace to the goal state(s).

3.3.4 Example - Travelling salesman

This example problem cannot be solved using BFS, as BFS does not have any property that allows for optimising the solution it produces. The only optimisation BFS is able to do is in terms of the lengths of traces. A better suited algorithm would be to use Dijkstra's algorithm (see Section 3.5.5).

It however is possible to emulate Dijkstra's algorithm by shaping the state space such that there are intermediate states which exist between two states in order to increase the depth of those states artificially. Using that technique you could represent the total length travelled by the salesman by the depth of nodes in the graph. Using that technique comes with the drawback that it significantly increases the number of states in the state space.

Ignoring the fact that Dijkstra's algorithm is already better equipped to tackle the travelling salesman problem than BFS would be, there still is the underlying issue that the representation of the problem results in a enormous state space. The encoding of the problem would require unique states for every path that can be taken by the salesman through the graph. This would result in a minimum state space size of $n!$ states for a graph with n nodes. Simply having a starting node which connects to each of these states would make the exploration nothing more than an enumeration of the solutions, thus a brute force. However any other layout that groups some of the states representing full tours of the graph (albeit not the optimal tour) would introduce additional states. As mentioned in the description of BFS, there is no guidance or optimisation to the discovery order for states at equal depths and exploring such gigantic state spaces would ultimately be bound by resource constraints (specifically time and memory footprint). This all combined makes BFS unsuitable to solve such a problem.

3.4 Depth first search

The depth first search (DFS) algorithm searches through a state space in a manner that reaches larger depths faster than other algorithms do. This algorithm can be described using terminology from chapter 2 as follows:

Exploration starts from an initial state for which the neighbours are evaluated in a recursive manner. DFS evaluates states as they are discovered and does not require a separate frontier to track states that have not yet been evaluated. Although it does need some method of remembering what states are being evaluated. When the algorithm evaluates a state it needs to be able to return back to that state when a neighbours is discovered and evaluated immediately upon discovery.

DFS has some properties in common with BFS, mainly that it ensures a goal state will be found (if reachable) and that all reachable states will be explored. Also a goal predicate can be used to terminate exploration early when discovering a goal state.

3.4.1 Heaps

DFS uses the heap in order to prevent getting trapped in (directed) cycles. But it is possible in some scenarios to not need the heap during a DFS exploration: if the state space does not any cycles in the reachable section of the graph (based on the initial state). However a cycle-less graph may still contain multiple edges that point to the same state and as a result the DFS algorithm can explore sections of the state space multiple times. Take for example a state space which contains three states: s_a, s_b and s_c. States s_a and s_b both have an outgoing edge to s_c and both states are reachable from the initial state. Also lets say that during some execution of the DFS algorithm s_a is reached first. When this happens it will discover s_c and all states reachable from it, this can be an expensive operation. When not using a heap the algorithm will have to perform the evaluation of s_c a second time when reaching s_b.

3.4.2 Backtracking

DFS uses backtracking to return to states which have been partially evaluated. When DFS reaches a state that is known it backtracks to the previous state that was being evaluated. This also happens once DFS finishes evaluating all neighbours of a state. When the algorithm has evaluated all neighbours of the initial state and needs to backtrack from the initial state this is equivalent to exhausting the frontier in BFS and the exploration terminates.

In computer science the DFS algorithm is often implemented with a program that uses recursion when evaluating a state. This leverages the program return stack as the queue needed to be able to backtrack to partially evaluated states. There however is also a implementation possible that is much closer to the pseudocode of BFS (see alg. 1). The backtracking mechanism is replaced with an ordinary frontier with an ordering. Using an ordered frontier like we defined in Section 2.6.1 the backtracking mechanism can be replaced with a frontier. This results in an algorithm which is build up exactly as BFS with the exception of a differently ordered frontier. A small difference between the recursive implementation and an ordered frontier is that the usage of the frontier requires the neighbours of any state to be fully enumerated in order for each to be added to the frontier. This is remarkably different from the recursive implementation because the recursion upon discovery pauses enumeration of some state to go explore one of the enumerated neighbours. For states with an infinite amount of neighbours this creates a problem when using a frontier. For such state spaces it is still possible to explore them using DFS with a frontier based implementation but the frontier needs to be a generator frontier (see Section 2.6.4) to be able to hold an infinite amount of neighbours.

3.4.3 Pseudocode

```
function dfs(S_initial, P_goal)
    S_frontier = S_initial;
    S_known = S_initial;
    while S_frontier ≠ ∅ do
        pick s_c from the frontier, such that s_c is the state which had been in the frontier
         shortest;
        S_frontier = S_frontier \ {s_c};
        if P_goal(s_c) then
            return s_c;
        end
        for s_neighbour ∈ S_out(s_c) do
            if s_neighbour ∉ S_known then
                S_known = S_known ∪ {s_neighbour};
                S_frontier = S_frontier ∪ {s_neighbour};
            end
        end
    end
end
```

Algorithm 2: DFS algorithm implemented with a loop

The pseudocode for a DFS algorithm is given in alg. 2, and when comparing it to alg. 1 it is almost identical bar one line of pseudocode. The one difference between DFS and BFS in the pseudocodes is the picking of states from the frontier. This is because the selection from the frontier integrates the sort order of the frontier into the algorithm and is what sets BFS and DFS apart. For purposes of this report we do not describe the recursive version of the algorithm but rather a queue based implementation. The pseudocode can be seen in alg. 2, if you compare it to the pseudocode for BFS (see alg. 1) you will notice that it is nearly identical bar the selection of states from the frontier.

3.4.4 Improving traces

Terminating exploration with a DFS algorithm when a goal state is found does not make any guarantees about the trace from the initial state to the goal state. This is a remarkable difference from BFS which is why DFS is less used when the trace to a goal state is important as it often is preferred to be the shortest trace possible.

A variation of DFS can be made that guarantees the shortest trace for goal states. This variation upholds the ability to explore a state space without the need of heap if the state space is suitable for heap-less exploration for regular DFS (acyclic state space).

When executing DFS it is possible that a state is discovered at a depth which is not the lowest discovery depth possible for that state. This is why to guarantee the shortest path to a goal state, the exploration cannot be terminated immediately when finding a goal state but it can update the exploration query to speed up exhaustion of the frontier. Upon finding the first goal state the algorithm will no longer terminate, instead a state constraint will be applied to the frontier. Also, the known goal state with the lowest depth will be kept track of, we will name this state the 'tracked goal'. The state constraint has to block states at depths equal to or higher than the tracked goal. When the constraint is set or updated it eliminates violating states from the frontier. The exploration is also altered in the evaluation of transitions of states. The known states are no longer ignored but checked if they are goal states and at lower depths than the tracked goal. This allows the tracked goal to be updated and for the shortest trace to the tracked goal to be remembered. This guarantees that DFS will not explore at depths greater than the tracked goal and that the shortest trace to the tracked goal is the shortest known up to the point of exploration. When the exploration terminates these properties guarantee that the tracked goal is the closes goal state to the initial state and that the shortest trace to the goal state can be deduced from the exploration behaviour (for example by updating the backtrace when updating the tracked goal).

Figure 3.3: Maze with solution which challenges worst case complexity

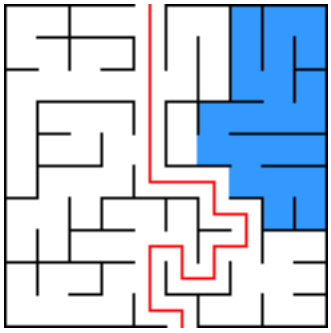


Figure 3.4: Maze with solution (red), area of interest highlighted in blue

This variation on DFS is strictly speaking not DFS anymore but a new yet unnamed algorithm. Perhaps 'shortest trace DFS' or ST-DFS would be suitable.

3.4.5 Example - Maze

DFS can be used to solve mazes, but only guarantees results for finite-sized mazes. For finite-sized mazes which are prefect mazes (see Section 3.1.1) the solution is guaranteed to be optimal. An infinite maze may result in DFS exploring a corridor that is infinitely long. In that case the DFS algorithm would never reach the end of the corridor (the end being a junction or a dead end), and thus also never backtrack to a previous state being explored.

This problem also translates to finite-mazes as DFS may happen to head into a section of the maze which does not lead to the exit of the maze but is costly to fully explore. An example of this can be seen in Figure 3.3 where the first junction in the maze divides the maze into a very expensive to explore section and a direct path to the exit of the maze.

DFS does not have any guidance with regards to the problem context (the maze) and whether the expensive section of the maze is explored or not all depends on order that the neighbours of the junction cell are enumerated in.

Using DFS in a way that takes advantage of its properties would be to use heapless exploration. In Figure 3.4 the exploration may explore the entirety of the blue section of the graph but once DFS backtracks all the way to the top and enumerates the next neighbour of the red line then the blue section does not need to be remembered. The maze in Figure 3.4 is however a special maze as it is perfect, a maze with no cycles. A maze with cycles can still be explored without a heap if the implementation of the algorithm uses the recursive implementation and tracks partially explored states through some sort of stack. The stack then functions as a very selective heap that only prevents the algorithm to get stuck exploring a cycle.

DFS has the same worst-case complexity as BFS, and it is also prone to diving into large sections of the state space that do not contain a solution. Those sections have to be fully explored before backtracking to the state from which the section was entered. In Figure 3.4 the area marked in blue would be a pitfall which pushes the resource usage towards the worst-case upper bound. If during execution DFS chooses to explore the first blue cell (located top left of the area) then DFS will explore the entire blue area before exploring the next cell on the solution path (red line). The worst-case complexity works out to be $O(m)$ with m being the number of cells in the maze. DFS has no limitation on the depth it explores before backtracking which causes this worst-case complexity. The worst-case scenario would be a maze as shown in Figure 3.3 where the goal cell could be the last cell discovered by DFS should it take unfortunate direction early on.

3.4.6 Example - Dining philosophers

The dining philosophers problem results in dense state spaces with a lot of transitions between the states because each philosopher is able to act on its own in parallel to all other philosophers.

Using DFS to explore the state space would be a very expensive operation both with and without using a heap. Not using a heap to explore the state space of a dining philosophers system reduces memory usage immensely but since the state space is so densely connected the algorithm is able to visit almost every state before reaching a state where all neighbours are already on the partially explored states stack (when using the recursive implementation). Alternatively for the frontier based implementation where a heap is also used the lack of guidance in the algorithm makes the worst case more likely to occur for any given exploration.

3.4.7 Example - Traveling salesman

Regular DFS is not able to solve the traveling salesman problem because there is no guarantee made on the trace when finding a goal state. This plus the absence of another usable property to order quality of goal states or exploration order it is not possible to solve the problem with DFS. Using the variation described in Section 3.4.4 (ST-DFS) which introduces the guarantee of shortest distance. The workaround from Section 3.3.4 where Dijkstra's algorithm is simulated by manipulating the state space becomes possible. Taking into consideration the properties of the ST-DFS variation we can reason about the computational complexity of the solution.

In order for ST-DFS to terminate either the entire state space needs to be explored or the frontier needs to be exhausted. We will name the goal state with the lowest depth the true goal state. In order to exhaust the frontier at least all states with depths lower than the true goal state have to be explored. The number of states at the depth of the goal state is upper bound by the total number of solutions in the state space and lower bound by the number of combinations possible with all the roads between the cities whose combined length totals below the depth of the goal state.

This is only the lower bound of the number of states to evaluate before exhausting the frontier. Because this best case scenario assumes that the goal state is found when first reaching the depth at which the solution exists. It however is much more likely that the algorithm first finds a less optimal solution at a greater depth. The algorithm then has to explore all states at depths below that algorithm before being able to find the true goal state at the lowest reachable depth.

The lack of guidance makes the exploration of the state space essentially a brute force of the solution as every solution is generated and gradually only solutions with shorter lengths get generated but all partial solutions with lengths shorter than the true solution do get generated.

3.5 Dijkstra's Algorithm

Dijkstra's algorithm considers weighted graphs and finds the shortest path between any two nodes in the graph (if a path exists). The algorithm uses the weights present on the transitions to order the frontier. Dijkstra's algorithm is sometimes also described as lowest-cost-first search [17]. The weights in the graph represent a certain cost associated with the system performing that particular transition. This creates a measure of performance that can go beyond the number of steps from the initial state. The measure of performance can be fine-tuned to enforce solutions to optimise for a particular metric.

The order is not based on the state itself but its placement within the graph. The algorithm introduces an associated cost to each node which represents the sum of costs for each transition between the node and the starting point. The nodes in the frontier are sorted such that a node with the lowest cost is explored first. If all edges in the graph have the same weight the algorithm behaves as the BFS algorithm.

In Dijkstra's algorithm it is dangerous for state spaces to contain negative costs on edges as it becomes possible for a cycle to have the sum of its transition become negative. A cycle

with a negative total cost represents a repeatable action that can be indefinitely executed to lower the total cost of a solution. This traps exploration as any solution that is able to execute such a cycle at least once can be reached from an infinitely long trace which loops through the cycle infinitely to lower the cost of the solution trace. A similar problem exists for sums of cycles equal to zero that can be looped indefinitely instead of exiting the cycle to continue exploration the remaining state space.

3.5.1 Transition cost

The cost of transitions can be bound to transitions through the label of transitions:

$$f_{tc} : L \mapsto \mathbb{R}$$

3.5.2 Visit cost

Frontiers hold states not transitions so the cost associated with a state can differ based on the transitions used to end up at a state from the initial states. States will be augmented with a property 'visit-cost' which identifies the cost associated with that state to reach it from the initial states. We will denote this association as a function f_{vc} that identifies the associated visit-cost for a given state:

$$f_{vc} : S \mapsto \mathbb{R}$$

When discovering a state the visit-cost of that state becomes the sum of the transition being evaluated combined with the visit-cost of the source of that transition. Since there is no transition needed to reach the initial states their visit cost is automatically 0. In order to cover the scenario where multiple transitions point to some state s_x, with those transitions possibly originating from different states, it is necessary to update the visit-cost of a state when a lower trace to a known state is found. It should be noted that this is only relevant for states in the frontier as states in the heap will not be evaluated again by definition. The visit-cost should be updated during the evaluation of transitions, this is when a lower visit-cost for a state in the frontier can be found. If a lower visit-cost can be determined for a state in the heap then this indicates that there is a negative cost sum transition cycle in the state space.

3.5.3 Pseudocode

function `dijkstra`($S_{initial}$, P_{goal}, f_{tc})
  $S_{frontier} = S_{initial}$;
  $S_{known} = S_{initial}$;
  update f_{vc} to associate 0 with all initial states;
  while $S_{frontier} \neq \emptyset$ **do**
    pick s_c from the frontier, such that f_{vc} (s_c) is the lowest for all states in the frontier;
    $S_{frontier} = S_{frontier} \setminus \{s_c\}$;
    if $P_{goal}(s_c)$ **then**
      return s_c;
    end
    for $t_n \in T_{out}(s_c)$ **do**
      $s_n = t_{target}(t_n)$;
      $n_{vc} = f_{vc}(s_c) + f_{tc}(t_n)$;
      if $s_n \notin S_{known}$ **then**
        $S_{known} = S_{known} \cup \{s_n\}$;
        $S_{frontier} = S_{frontier} \cup \{s_n\}$;
        update f_{vc} to associate n_{vc} with s_n;
      end
      else if $s_n \in S_{frontier}$ **then**
        if $n_{vc} < f_{vc}(s_n)$ **then**
          update f_{vc} to associate n_{vc} with s_n;
        end
      end
    end
  end
end

Algorithm 3: Dijkstra's algorithm implemented with a frontier

Dijkstra's algorithm has been implemented in pseudocode in alg. 3, here the exploration algorithm is implemented in the 'dijkstra' function.

Dijkstra's algorithm receives the initial states set and goal state predicate from the user, this is common for an exploration query (Section 2.8.8). Additionally the algorithm receives a function that associates a (user defined) cost value to transitions in the state space (f_{tc}).

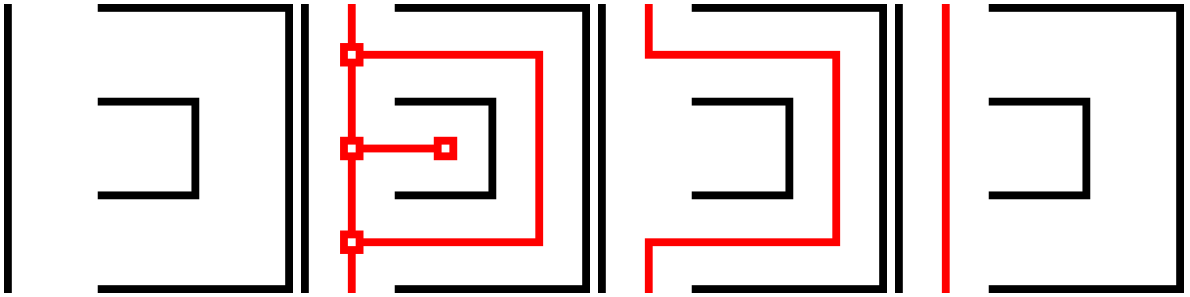
Looking at the pseudocode in alg. 3 shows similarities in structure to BFS (again). The ordered frontier is clear to see by the instruction of picking s_c according to the order of the frontier. The behaviour of evaluating a state is slightly different, beyond additional behaviour. Here the transitions are enumerated, the code for BFS used a notational shorthand here where the neighbours were directly enumerated. When enumerating a transition, the discovery of unknown states still leads to insertion into the frontier as usual, with the added behaviour of associating the visit-cost to the state. Additionally if the state is known then for known states in the frontier the associated visit-cost (n_{vc}) is updated for the neighbour state s_n if the cost of the current transition leads to a lower visit-cost.

3.5.4 Example - Maze

For mazes with the encoding provided in the introduction (Section 3.1.1) the performance of Dijkstra's algorithm is on-par with BFS. Since the discovery order for these algorithms is rather similar. However Dijkstra's algorithm can handle different representations of the same maze while still providing the same (accurate) solutions.

The grid based encoding of a maze can be optimised by reducing corridors of multiple cells into a single edge in a graph. This reduces a maze from a grid of cells to a skeleton of its junctions and dead ends. The edges of the graph can have the number of cells they represent encoded as their cost. The cost of a transition then is the distance travelled in the original maze. An example of such a conversion is shown in Figure 3.5.

BFS is not able to give correct solutions for the skeleton graph because it does not take into account the length of paths between junctions in the maze. This can be seen in Figure 3.5c,



(a) Original maze cell grid (b) Maze with skeleton graph (c) Skeleton maze solved by BFS (d) Skeleton maze solved by Dijkstra

Figure 3.5: Conversion of a grid maze to a skeleton graph

where the found solution walks the loop around the outside border of the maze as the shortest path because those 5 cells are represented with a single transition (of length 6).

Dijkstra's algorithm will deal with this accordingly because the visit cost of the junctions vertically below the entrance are lower than the visit cost of the bottom cell (by the long transition around the edge of the maze). When Dijkstra's algorithm evaluates the cell to the left of the dead end in the middle then it will find that the transition leading from that cell to the cell below it costs less than the visit cost established for that cell by the long transition around the edge of the maze.

Dijkstra's algorithm still suffers from the same shortcomings that BFS has, which is the large memory usage. Also, while Dijkstra's algorithm has the transition cost to help it optimise the solution it has no guidance that helps it distinguish the future quality of a state, for example whether a transition leads to a state that is in some sense closer or further away from a goal state.

3.5.5 Example - Dining philosophers

In the dining philosophers you the search for deadlock states can be sped up by marking transitions that let philosophers release their resources (utensils) as having a cost and for those where a philosopher just picks up a utensil as having no cost. This results in all states where utensils to be picked up to be found before the first state where a utensil has been dropped to be evaluated. This greatly speeds up the state space search in order to find the deadlock.

This usage of the cost function demonstrates that the cost function can be used to favour a particular set of operations on a system before evaluating states where less favourable operations have taken place.

3.5.6 Example - Travelling salesman

In the travelling salesman example for BFS we already described a convoluted way of encoding the problem into a state space that is suitable for BFS. We noted there that Dijkstra's algorithm would be more suitable there. Using the encoding described in the example introduction (Section 3.2) we can perform a Dijkstra exploration by defining the transition cost of two states as the difference in the sum of travelled distance in the route between the source and target states of a transition. For example two states s_a and s_b with a transition $s_a \xrightarrow{l_n} s_b$ will have a cost of $s_a - s_b$. The resulting search will perform a branch and bound solution to the travelling salesman problem. This solution has a performance speed with a lower bound of the generation of all partial solutions with shorter total length than the optimal solution. The upper bound is the time needed to enumerate all partial solutions with a length of the longest path between all the cities, which can be simplified to the enumeration of all solutions since there are no solutions that are longer that can be deducted from the set of all partial solutions. The frontier will need to hold all those partial solutions and so there is both a time and memory resource usage in the order of $n!$ for n cities.

3.6 A*

The A* (a star) algorithm is an exploration algorithm that allows for exploration to be guided by a heuristic cost function. A* can be seen as an extension of Dijkstra's algorithm to make it an informed search algorithm by taking into account an estimate of the remaining path cost for a node. Dijkstra's algorithm only considers the cost up-to a given node, A* uses a heuristic function to estimate the remaining cost to the goal state. This estimate prioritises the frontier not only by the cost of visiting the state but also by how likely that state is to lead to a goal state (in the near future).

For state spaces where the problem context is able to provide such an estimate the A* is a good choice as it tends to head towards goals faster and evaluates less states in order to reach the same goals as for example Dijkstra's algorithm of BFS would find.

Compared to Dijkstra's algorithm, the A* algorithm can be characterised as mostly the same algorithm except for (again) a different ordering of the frontier.

3.6.1 Cost functions

The cost function A* uses to order the frontier (F) is composed of two cost functions (G and H).

$$F(s) = G(s) + H(s)$$

$$F : s \mapsto \mathbb{R}$$

$$G : s \mapsto \mathbb{R}$$

$$H : s \mapsto \mathbb{R}$$

G defines the lowest cost of some trace from an initial state to the given state. This can be encoded into the search algorithm by using the visit-cost property from the Dijkstra search algorithm. To define this for a state space the transitions in the state space can be labelled with some cost and A* will use these weights to track the cheapest path to all known states. From Section 3.5.1 this would be the same as f_{vc}. The function H is a heuristic function which approximates the cost of the cheapest trace starting from the given state to a goal state.

As with Dijkstra's algorithm, while executing an exploration using the A* search strategy the order of discovery may be such that the cost of the traces leading up to states may be altered as transitions are evaluated which have their target states in the frontier.

3.6.2 Admissible heuristic

The heuristic is provided by the user and is required to be admissible. Admissible means that the heuristic function never overestimates the true cost of the remaining path. An admissible heuristic guarantees that A* produces the same result as Dijkstra's algorithm with the time complexity having an upper-bound of Dijkstra's algorithm. If the heuristic is not admissible these guarantees do not hold, not even that the result is optimal. However, A* will find a solution if Dijkstra's algorithm regardless of the heuristic's admissibility.

3.6.3 Pseudocode

function `astar`($S_{initial}$, P_{goal}, f_{tc}, H)
- $S_{frontier} = S_{initial}$;
- $S_{known} = S_{initial}$;
- update f_{vc} to associate 0 with all initial states;
- let $G(s) = f_{vc}(s) + H(s)$;
- **while** $S_{frontier} \neq \emptyset$ **do**
  - pick s_c from the frontier, such that $G(s_c)$ is the lowest for all states in the frontier;
  - $S_{frontier} = S_{frontier} \setminus \{s_c\}$;
  - **if** $P_{goal}(s_c)$ **then**
    - **return** s_c;
  - **end**
  - **for** $t_n \in T_{out}(s_c)$ **do**
    - $s_n = t_{target}(t_n)$;
    - $n_{vc} = f_{vc}(s_c) + f_{tc}(t_n)$;
    - **if** $s_n \notin S_{known}$ **then**
      - $S_{known} = S_{known} \cup \{s_n\}$;
      - $S_{frontier} = S_{frontier} \cup \{s_n\}$;
      - update f_{vc} to associate n_{vc} with s_n;
    - **end**
    - **else if** $s_n \in S_{frontier}$ **then**
      - **if** $n_{vc} < f_{vc}(s_n)$ **then**
        - update f_{vc} to associate n_{vc} with s_n;
      - **end**
    - **end**
  - **end**
- **end**
- **end**

Algorithm 4: A* algorithm implemented with a frontier

The pseudocode for A* can be found in alg. 4 which is implemented as a modified version of the implementation for Dijkstra's algorithm (see alg. 3). The only difference in the implementation is the addition of the H to the order of the frontier, the existing behaviours to set and maintain the f_{vc} function has remained the same.

The 'astar' function implements the A* algorithm and receives the common exploration query elements: initial states and a goal predicate. Since the implementation is based on the pseudocode for Dijkstra's algorithm it also receives an argument specific for that algorithm (f_{tc}). For A* itself a user-defined function H is also required. In the setup of the algorithm the state cost function that sorts the frontier (F) gets defined as the result of f_{vc} combined with H. By defining G (component of F) as f_{vc} the algorithm is able to reuse most logic from Dijkstra's algorithm.

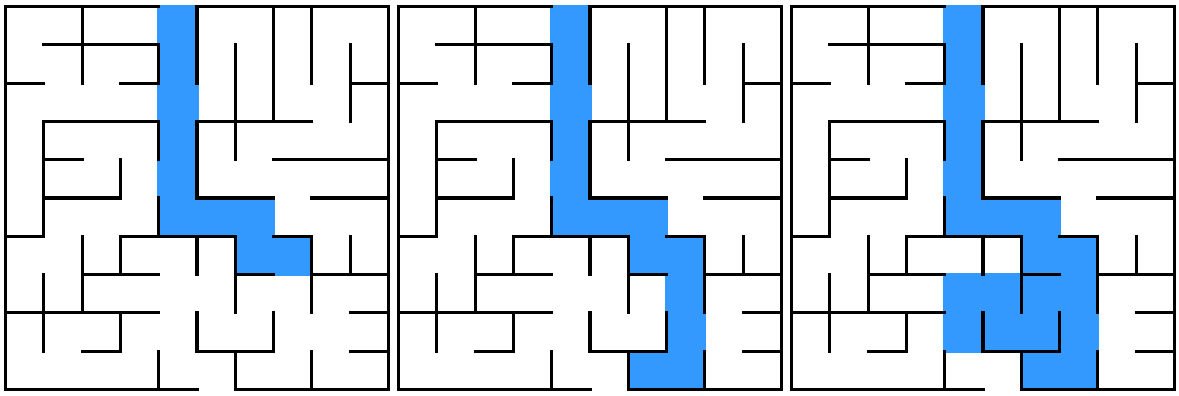
3.6.4 Example - Maze

Using A* to solve mazes gives much better performance than previous algorithms when comparing the number of evaluated states as well as the memory usage by the algorithm.

In Figure 3.7 a maze without any inner walls is shown with its solutions given by BFS (left) and A* (right). The areas marked in blue are the states where that cell was the current position of the player. These areas show what part of the maze has actually been discovered by the algorithms and the difference is clear: A* heads straight for the exit while BFS spreads out sideways to check every possible position.

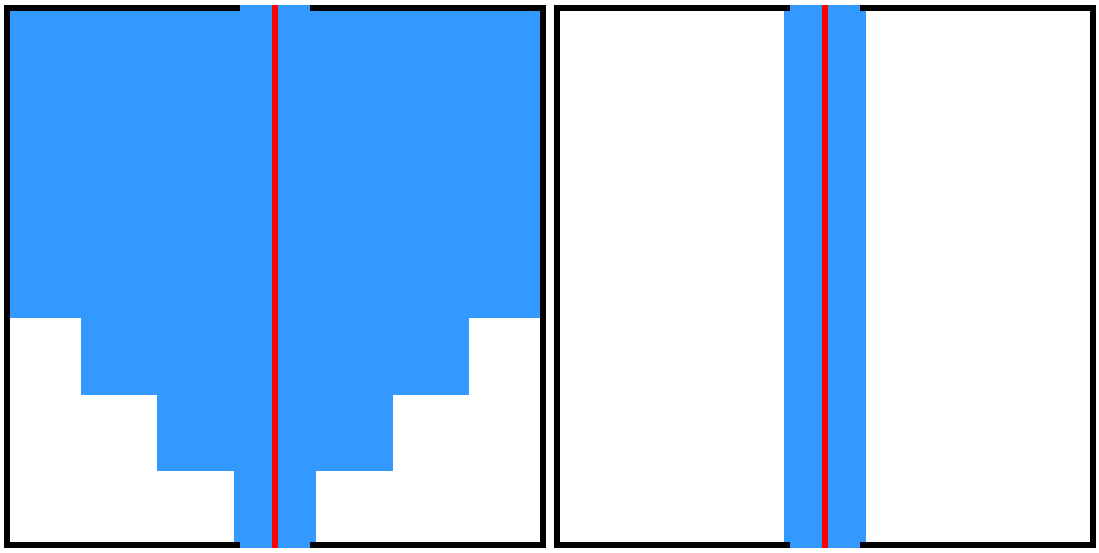
The number of state evaluations also is an indication on the time taken to produce the solution. BFS evaluated (up to)

With the running example of a maze, the difference the heuristic has on the number of visited cells is notable. The set of discovered cells includes the visited cells and their neighbours, which in the example also includes some additional white cells. In Figure 3.6 the results of using A* are visualised, where A* used the Manhattan distance between a position in the maze



(a) After 10 state evaluations (b) After 14 state evaluations (c) After 20 state evaluations

Figure 3.6: Progression of A* exploration through a maze



(a) Evaluated states for exploration with BFS (b) Evaluated states for exploration with A*

Figure 3.7: A 7x7 sized wall-less 'maze' explored by two algorithms, evaluated states (cells) marked in blue

and the exit (bottom opening) as the remaining cost. The heuristic provided to A* computes the Manhattan distance between a cell and the goal cell. The visited cells are marked in grey and when comparing them to the markings of the Dijkstra's algorithm it becomes clear that A* is capable of performing much more targeted searches. Notable is Figure 3.6b which is where the heuristic misled the search as moving down instead of moving left at the previous T-junction. Once the dead-end is hit at the 14th state evaluation the next state to be evaluated is the state where the player moved left on the T-junction and the algorithm will rapidly find the solution after.

3.6.5 Example - Dining philosophers

Exploring the state space of the dining philosophers problem will result in the same performance as Dijkstra's algorithm unless a good admissible heuristic is used. For the dining philosophers there is not much information about a state in relationship to its nearest goal state. The only information present is the difference in what philosophers are holding which spoons. A possible heuristic would be to use the technique from Section 3.5.5 to penalise states that are known to be further away from a goal state to then estimate the number of steps needed to deadlock the system. For A* this heuristic marginally improves the performance because the number of known conflicts between philosophers that have to be resolved before a solution is possible can be correctly estimated. With a conflict we mean that a philosopher is holding only one spoon and is holding it in the opposite hand relative to the larger group of philosophers where they are all holding one spoon in the same hand. So if three philosophers are holding one spoon in their left hand and two philosophers are holding spoons in their right hand (this means there are at least 6 philosophers), then the penalty for that state is 2 because the philosophers that are holding spoons in their right hand need to pickup the other spoon and drop it before being able to get into a state where they are holding only a spoon in their left hand. Similarly philosophers which are holding both spoons can be penalised because a philosopher which is holding both spoons is always able to drop them and thus always introduce at least one possible neighbouring state in the state space. The described heuristic is admissible because the logic used guarantees the penalty never exceeds the number of 'moves' needed to transition the state into a goal state because the estimate is not towards the goal state but towards an intermediate state in which there are no more conflicts and the deadlock can be reached. Using an inadmissible heuristic, for example by penalising conflict-less states would obviously have the opposite effect and ensure that any states with conflicts that get discovered are evaluated prior to conflict-less states, prolonging the exploration unnecessarily. Picking a admissible heuristic is key to using A* effectively, otherwise the performance can be worse than Dijkstra's algorithm.

3.6.6 Example - Travelling salesman

The guarantees of A* ensure that the exploration will find a goal state if reachable and thus it will also find one for the travelling salesman problem. However the usage of a heuristic function to determine the frontier prioritisation introduced the possibility that a non-optimal solution is found, if the used heuristic is not admissible. Otherwise, if the heuristic is admissible then the solution will be optimal. For the travelling salesman problem specifically the information available for a heuristic to use is limited to the remaining cities to be added to the partial solution and the distances between them. A possible heuristic function could be to use the last visited city of the partial solution and compute the length of the path should the next 2 closest by cities be visited next. Such a heuristic is not based on any formal method of improving the search strategy but merely a demonstration of an application of a heuristic.

The travelling salesman problem is an optimisation problem and for this kind of problem A* may not always be suitable. The encoding of the problem (as laid out in Section 3.2) causes the state space to be filled with partial and valid solutions to the problem. Those valid solutions are identified as goal states but they may not be the most optimal solution to the problem there is. This puts the responsibility of finding the optimal solution (state) to the A* algorithm and by extension to the heuristic function. To correctly answer the problem with the optimal solution it is required that the heuristic cost function used by A* is admissible. In problem contexts where an

optimisation problem is encoded such that the state space contains at least one non-optimal solution it is required that a admissible heuristic can be defined. It is also assumed that in those problem contexts the transitions have associated with them the correct costs, otherwise the optimisation based on the leading cost up to a state is wrong also. Otherwise, there is no guarantee that A* will find the optimal solution to the problem. It is worth mentioning that a heuristic cost function which for every input state resolves to a constant value the algorithm behaves like Dijkstra's algorithm and the optimal solution will be found, but at a penalty to performance in increased execution time and increased memory footprint.

3.7 Sweep-line

The sweep-line method, first described by [3], is a method of using a sense of progression to be able to reduce memory usage during exploration. The initial solution placed a strict requirement on the state space which was later lifted in [9]. The method requires that the method of the system has a measurable progression, for example, phases of a protocol. The assumption the method operates on is that a system that has progression does not return to states from earlier progression stages.

Sweep-line introduces a function which measures the 'progress' of a state. This is a number indicating how far the state is along a defined measure of progress. The function has to be defined by the user and is usually closely tied to the state representation. Now that states have a sense of progression there are also terms for how states connected by a transition are related to each other.

Definition 5. $s_x \xrightarrow{l} s_y$ is progressive iff $F_p(s_x) \leq F_p(s_y)$.

Definition 6. $s_x \xrightarrow{l} s_y$ is regressive iff $F_p(s_x) > F_p(s_y)$.

The notion of progression places an assumption on the shape of the state space; the state space transitions are mostly progressive with little to no regressive transitions. The sweep-line algorithm needs the user to define a function to get the progression value of a state. The definition of progression in the context of the state space is left up to the user. The function receives a single state as input and outputs a real number. We will call groups of states with the same progression value a layer.

3.7.1 Purging the heap

For now, we will talk about sweep-line as acting on state spaces with no regressive transitions, this will later be addressed. Using the assumption on the shape of the state space allows sweep-line to determine when states can be safely removed from the known states set. The states in the known states set have already been evaluated and their neighbours have been found, only states in the frontier are known states which have yet to be evaluated. Sweep-line tries to eliminate as many states from the known states set as possible by removing entire layers which have a progression value lower than that of any state in the frontier. The assumption on the state space shape indicates that states from those layers are unlikely to show up during evaluation again, and therefore do not need to take up space in the known states set. The moment a layer is lower in progression than any in the frontier we call the layer completed and it can be removed from memory. The lowest progression value of the frontier is called the sweep-line threshold. The sweep-line algorithm continually removes all completed layers from the known states set, we name this act purging.

3.7.2 Regressions

As mentioned before, there is the issue of regressive transitions. If during exploration a state gets purged it can get re-discovered by a regressive transition. Using the sweep-line method as described thus far could lead to an infinite loop if the state is reachable from itself and purged each time before re-discovery. To prevent this sweep-line marks all states reached through a regressive transition as persistent. Persistent states cannot be purged from the known states

set. When marking a state as persistent it also gets put into the frontier (again) because there is no indication that the state was found before. It is possible that the state was not reachable from the initial states but only through the regressive transition. The frontier of sweep-line is ordered by the progression measure such that the sweep-line threshold increases as quickly as possible.

3.7.3 Pseudocode

function `sweepline`($S_{initial}$, P_{goal}, F_p)
    $S_{frontier} = S_{initial}$;
    $S_{persistent} = \{\}$;
    while $S_{frontier} \neq \emptyset$ **do**
        pick s_c from the frontier, such that $F_p(s_c)$ is the lowest for all states in the frontier;
        $S_{frontier} = S_{frontier} \setminus \{s_c\}$;
        if $P_{goal}(s_c)$ **then**
            return s_c;
        end
        `/* Purge-list of states that are lower in progress than the lowest progressed and not persistent */`
        $S_{pl} = \{s_x \in S_{known} | F_p(s_x) < F_p(s_{cur}) \wedge F_p \notin S_{persistent}\}$;
        $S_{known} = S_{known} \setminus S_{pl}$;
        for $s_n \in S_{out}(s_c)$ **do**
            if F_p ($s_{neighbour}$) < F_p (s_{cur}) **then**
                $S_{persistent} = S_{persistent} \cup \{s_{neighbour}\}$;
            end
            if $s_n \notin S_{known}$ **then**
                $S_{known} = S_{known} \cup \{s_n\}$;
                $S_{frontier} = S_{frontier} \cup \{s_n\}$;
            end
        end
    end
end

Algorithm 5: Pseudocode for sweep-line implemented with a frontier, supports regression handling

The pseudocode reflects the algorithm explanation prior and follows the similar pseudocode pattern as used in the pseudocodes for the algorithms in the previous sections. The pattern is centred on iterating over the items in the frontier and using these states to discover new states to add to the frontier and iterate over in the future. Sweep-line requires a progress function from the users and this is shown in the code by the additional parameter F_p which represents this progress function. The frontier is sorted by the progress measure, when the a state is picked for evaluation this is known to be the state with the lowest progress measure and at that moment the heap can be purged. In the pseudocode the purging of the heap is split into two lines of code for sake of readability (the S_{pl} could be in-lined). The variable S_{pl} holds the 'purge-list' which is the subset of states in the heap which are not marked as permanent and have a progress function lower than that of the state being evaluated.

The evaluation of outgoing transitions has additional behaviour (in comparison to the most basic form of pseudocode shown so far; BFS) which marks the target state as persistent if it is a regressive transition. As the heap is being purged each iteration step of the frontier a regression edge is not possible to target a known state. As such the condition that follows in the pseudocode will always evaluate to true and a regressive edge will always result in the target state being inserted into the frontier, unless the state has already been marked as persistent. Once a state is marked as persistent it is not possible for it to be removed from the known states set.

3.7.4 Example - Maze

Mazes (and many other puzzle games) have the property where almost any action can be undone from the resulting state. For example, in a maze, the player can navigate into another cell and then just move back. This means that whatever progression measure is used for maze games, it should not increase on undo-able actions. If for example the Manhattan distance[1] would be used for the progression measure, making it such that a shorter distance causes a higher progression score, then all states will become persistent states (because of the ability to move back) and the advantage of sweep-line is completely lost. Any states that do get removes will be small in numbers and will not account for the extra overhead introduced by sweep-line logic.

There however are cases where sweep-line would be useful for puzzles, including mazes. The point is to base progression on actions that cannot be undone or results from actions that cannot be reverted. Altering the maze puzzle by adding a key to the maze which has to be picked up before reaching the goal is an example of this. The state of the game consisted only of the position of the player, now it also includes whether a flag indicating the player has picked up the key or not. Making it impossible to drop the key causes the puzzle to be divided into two steps, finding a path the key and then finding a path to the exit. The key adds a sense of progression to the puzzle which is much easier to measure than the distance to the exit of the maze. Once the player has picked up the key the entire state space of states where the player did not have the key can drop away. However, it will take a while before this happens because first, all states of the lower progression measure (no key picked up) will have to be explored. This will cause the maze to be fully explored before continuing from the key position. This is a drawback of sweep-line as the only way to mark a layer as completed is by exhausting the states within it.

3.7.5 Example - Dining philosophers

The dining philosophers problem has a few properties that can be used to derive progress. There are the philosophers, the spoons that they are holding and what state they are in. However, when basing the progression measure on the number of spoons being picked up or the number of philosophers being in a blocked state (where they want to pick up another spoon but cannot do so) there is a lot of regression bound to happen. Any philosopher that is holding two spoons will soon be dropping them, also when philosophers that can pick up a second spoon there is a neighbouring state where they pick it up and then eventually drop both again. It is only a very specific sequence of events that can cause a deadlock state in the dining philosophers problem, which is what makes it such a challenge to verify. Sweep-line is not in any particular way suitable for the dining philosophers problem to improve the solution over algorithms discussed previously. The purging of memory cannot be effectively applied because there are a lot of regressive transitions in the state space.

3.7.6 Example - Travelling salesman

The effectiveness of the sweep line algorithm on the travelling salesman problem is determined mainly by the definition of the progress function, this could for example be the summed length of the path produced by the (partial) solution state. The performance the progress function influences is both memory consumption and exploration time, this is due to the function influencing both the purging of states as well as the (evaluation) order of the frontier. A very fortunate property of the state space that results from the encoding of the problem is that states do not have regressive edges because partial solutions are only ever extended upon. This however assumes that the distance between two cities is not negative (although this seems impossible unless deliberately circumvented). Taking for example the proposed solution from Section 3.5.6 then the algorithm would perform at the same time complexity, although it make take a little longer on account of performing the heap purges. The result of using the solution's travelled distance causes partial solutions to be removed from memory once all neighbours

[1] This measures the distance between two points in a 2d coordinate system as distance along each axis of the coordinate system summed together

of that (partial solution) state have been discovered. There is a special case where a pair of cities has a distance of zero between them, in this case partial solutions may hang around until the all cities in that same spot have been been enumerated and all permutations of visitations to those cities have been evaluated. The resulting purge will still wipe out all partial solutions holding the permutations, resulting in a higher peak memory usage while the heap contains those partial solutions.

Lets consider a group of cities for which every pair of those cities has a distance of 0 a 'zero-distance cluster' of cities. When sweep-line adds a city to a partial solution then the heap usually gains a single state which will be purged as soon as all partial solutions with the same length have been solved. If that city is the first city of a zero-distance cluster then instead of a single state being added, a total of $n!$ partial solutions will result that all have the same distance travelled. This causes the peak which in memory usage and the memory usage will persist until all partial solutions with that particular length (or shorter) have been evaluated.

Arguably for the travelling salesman problem a cluster of cities with no progression in adding them to the solution seems to make little sense but it does show a pitfall for sweep-line. For a system which is modelled by a group of actions which may or may not be applied to any given state to mutate it into a different state, if there are permutations possible over multiple actions which result different states with the progress values then this causes peaks in heap memory usage. This can also be applied more generally to the progress layers, the coarser the categorisation of states into progress layers, the bigger the layers will be and thus the larger the peak heap memory usage will be.

3.7.7 Sweep-line as a workflow

With the knowledge from the maze example (Section 3.7.4) the sweep-line purge technique could be expanded upon by considering the algorithm as a way to reduce memory after solving a layer of progression. In this sense any exploration algorithm could be used to explore a progression layer, and not all problems require the layer to be completely explored/known before moving on to the next layer. The exploration of a progression layer can be seen as a regular state space exploration, with the initial states set to all known states in the layer and the goal predicate set to match any state that has a higher progression than the current layer. Exploration algorithms so far have tended to terminate exploration upon finding the first goal state but for solving a progression layer this may not be suitable so the termination needs to be defined as some function that identifies for an exploration snapshot whether the layer has been explored.

This describes the application of sweep-line as a workflow that uses some inner exploration strategy to solve progression layers.

The sweep-line workflow variant needs to tackle regression as well since fully exploring the state space from a state in a lower progression layer can increase the exploration of a given layer to the complexity of all prior layers combined.

3.8 Genetic algorithms

There is a group of algorithms which are solution optimising algorithms called genetic algorithms. They build solutions and evolve them into better versions over many iterations and many attempts to find a solution to a problem.

A genetic algorithm consists of a few pieces that work together. The main data structure genetic algorithms use to evolve solutions is a chromosome. A chromosome is a string of genes which represent a single piece of data, usually a bit or a number. Chromosomes are grouped into a population. Genetic algorithms use populations and evolve them into better populations. Evolution is the process of evaluating the entire population, removing bad chromosomes from the pool and filling up the pool with new chromosomes based on the remaining 'good' chromosomes.

First the evaluation of the population: this step uses a user-defined function called the fitness function. It receives a chromosome and returns a score (called their fitness) based on the performance of that chromosome. The chromosomes are ranked based on their fitness

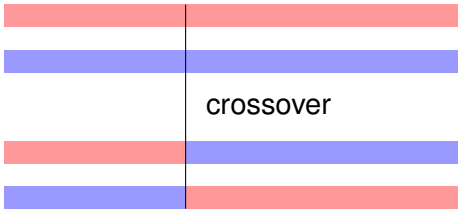


Figure 3.8: Chromosome crossover diagram

and in the second step, a portion of the population is removed. There are different ways the population can be cut down, usually by either removing the bottom ranking chromosomes or probabilistically while favouring the chromosomes with high fitness scores to remain in the chromosome pool. Finally, the pool is replenished using the remaining chromosomes. The new chromosomes are created not at random but by combining pairs of the remaining chromosomes into new chromosomes (crossover). There are many different ways two chromosomes can be combined into one or more new chromosomes. In the crossover process, two chromosomes are put in parallel and at the same point in the chromosomes they are cut in two and the tails are swapped and stitched to the other chromosome.

A diagram displaying how the crossover affects a pair of chromosomes is shown in Figure 3.8.

After creating new chromosomes they are also subject to mutation, this randomly alters the chromosomes to introduce more diversity into the population and allows the genetic algorithm to avoid getting stuck in local maximum. Again there are many options as to how this step is executed. For genes made up of binary numbers, this often involves flipping the values of genes in the chromosomes randomly. Increasing the probability that a genes is mutated increases the average diversity in the population but when set too high it will cause the population to destabilise as there is too much randomness for dominant patterns to hold over multiple evolutions (this depends on other factors also such as the number of chromosomes that survive elimination from the fitness function). Usage of genetic algorithms involves executing this loop until either a resource budget is depleted or a suitable result is found in the population. Genetic algorithms can run for a long time and will make most of their progress early on. Meaning that the biggest improvements are made early on and in later evaluations, the fitness will increase comparatively less.

The application of a genetic algorithm to state space exploration was done by P. Godefroid and S. Khurshid[5] where the chromosomes are made up out of a string of bits and a fitness function that executes state space exploration based on the chromosome. The exploration is run in simulation fashion, with at most a single state in the frontier. The next state is picked by consulting the chromosome, the exact usage of a chromosome to pick the transition to follow is dependant on the problem encoding. The technique used by the mentioned research used a chromosome which held a string of bits from which integers were decoded. This technique has a few caveats also addressed by the researchers. First, for states which have several outgoing transitions that are not a power of two. To solve this they pick a random number when the number read from the bit string exceeds the transition count. Second is the observation that changes to a bit early in the chromosome can have large differences in the trace later on, as the path can change early on and this, in turn, changes the meaning of the bits in later transition picks.

The application of state space exploration, in this case, is mostly as a tool for a workflow, which itself is an algorithm. While the exploration is heavily controlled by the genetic algorithm and representation that chromosomes have, it is important to note how exploration can also be used as a building block for larger algorithms.

3.8.1 Pseudocode

```
function explore(s_initial, c_gen, P_cont, P_next, F_score)
    i = 0;
    S_frontier = {s_initial};
    while S_frontier ≠ ∅ ∧ P_cont(i, c_gen) do
        pick s_c from the frontier, as the only state in the frontier set;
        S_frontier = ∅;
        i = i + 1;
        for t_n ∈ T_out(s_c) do
            if P_next (c_gen, i, t_n then
                S_frontier = {s_n};
                break;
            end
        end
    end
    return F_score(s_c);
end
```

Algorithm 6: Genetic algorithm implemented with singleton frontier

A pseudocode implementation of a genetic algorithm evaluation function is shown in alg. 6. Note that this is not the full implementation of a genetic algorithm, but rather just the evaluation function for which state space exploration is used to compute the score of a chromosome. This implementation while somewhat similar to the implementation of previous algorithms, has some key differences that are a result of the purpose of the function. For the function exploration of the state space is simply a means to an end, the actual exploration is not a goal but rather the execution itself is the result for which quality is measured. The implementation of this evaluation function differs from previous pseudocodes in that the state evaluation loop is controlled by a secondary (user defined) criteria. Also, not all transitions are evaluated or lead to discoveries, even when the target states of those transitions are unknown. This is done to fulfil the simulation behaviour and 'progress' the 'current' state of simulation to one particular 'next state'. In chapter 2 terminology this can be seen as a filter (Section 2.6.2) on the frontier, however it closely relates to the size of the frontier as well since the filter also blocks states once the frontier contains a state.

To this end the function receives a few parameters to facilitate an exploration guided by a chromosome: the chromosome c_{gen}, a continuation predicate P_{cont}, a predicate P_{next} and a scoring function F_{score}. The main control over exploration is through the P_{next} predicate. This is a predicate which identifies the transition that should be evaluated for the 'current' state of the simulation. It receives the chromosome, a counter i which identifies the iteration for which the exploration is going on and the transition. The counter and the chromosome also control the continuation of the simulation through the predicate P_{cont}. This predicate is defined by the user to allow for early termination should for example the counter be used to point to an element of the chromosome and have that counter become out of bounds. When evaluating a state, the counter i and chromosome can be used to read a value from the chromosome which determines what transition should be followed. Each outgoing transition of the state is tested against this predicate and the first to match is marked as discovered and put in the frontier. Since it is a simulation the act of putting it in the frontier stops evaluation of other transitions of the state. The state in the frontier is progressed until no suitable outgoing transition is present or until the counter points to an element outside of the chromosome. Ultimately the state evaluation loop exits because the frontier is empty or the user defined P_{next} has marked that evaluation should no longer occur. The last used value s_c is then used as argument to the scoring function F_{score} to output the evaluation score of the simulation led by the chromosome c_{gen}.

3.8.2 Example - Maze

The encoding of the maze consists of the position the player is at in the maze. A gene has to mutate this state, the 4 directions of movement can be encoded into the bits stream easily

by taking two bits, decode them as one of the 4 possible directions to move in. We can say that the chromosome is a string of movement commands that the exploration algorithm uses to trace through the state space. These commands form a limited set which each may or may not be applicable to a given state.

There also needs to be some measure of score for a performed trace through the state space. The final state in the trace of exploration can be compared to the goal state to see what the distance between the player and the goal cell is. A chromosome's performance would then be entirely tied to the final position reached.

The maze problem is not very well for the genetic algorithm because the possible commands to a state can differ wildly. A small mutation to a chromosome influences some of the commands and this in turn likely has side-effects to commands thereafter. For example changing a single command halfway the command string may put the player in a different section of the maze which the following commands may or may not be suitable for. The genetic algorithm is better suited for problems where the same set of commands are possible for any possible state, this removes the need to handle invalid commands/mutations.

The encoding of the gene is easy enough, however not every move is possible at a given position in the maze. A wall could be blocking the path of an attempted move, two simple ways to deal with such a violation is to either terminate the exploration or ignore the command. Ignoring the command gives the chromosome the opportunity to increase its score after the mistake, as terminating immediately can render a small mutation on a chromosome into a chromosome with a drastically lower score. Since the genetic algorithm prefers to make gradual changes over time the termination upon the first illegal command goes against this because those small mutations are less likely to propagate into larger changes needed to mutate a chromosome into a better one (local maximum problem).

The application of the genetic algorithm to solve a maze by exploring the state space of moves through the maze is not effective for a single trace. The strength of the genetic algorithm is to perform cheap to execute traces in favour of expensive explorations which explore multiple paths. The genetic algorithm doesn't require a heap which is a property it shared with DFS (under some conditions), but the chromosomes are used to guide the repeated exploration instead of exploring all states.

3.8.3 Example - Dining philosophers

To apply a genetic algorithm to the dining philosophers problem we pick a different representation than presented prior. Our representation will be problem-specific instead of a generic representation used before, this allows us to overcome the caveats by taking advantage of the problem context. To overcome the caveats we take the dining philosophers problem and try to encode the possible actions into the chromosome such that each genes has the same number of bits. The genes will contain the following: a binary encoded number indicating a philosopher and a bit indicating the left or right spoon. For this, we introduce a restriction: the number of philosophers must be a power of 2. The purpose of genes is to represent an action on the current state. This involves a philosopher which will perform the action and the left or right spoon which will be picked up or dropped. It is, however, possible that an action is invalid: for example, a philosopher may be holding the left spoon but not the right and the action indicates to use the left spoon. In this case, the only thing the philosopher can do is pick up the right spoon and to solve this we simply void the action, yielding no changes. This means that sometimes a genes has no effect on the state but this allows a much easier way to encode the actions possible on the state at any given time. For the fitness of the chromosome, we execute each genes on the initial state and after each genes we record how many philosophers are deadlocked. We consider a philosopher deadlocked when it cannot pick up any spoons while the state of the philosopher is to pickup spoons.

When effect of this encoding is similar to what was experienced with the maze exploration solution in that ignoring illegal moves means that modifications to good genes have cascading effects throughout the execution of the exploration. The set of deadlock states always contains two states, where the philosophers are either all holding the left or right spoon. The solution is a very exact sequence of actions and deviating from this sequence (by having a philosopher pick up a utensil at the wrong side) is a setback which has to be recovered from. Genetic algorithms

perform best when small changes to the chromosome correspond to small changes in outcome of the evaluation function. For the dining philosophers problem, as with the maze example, this is not the case because a small change to the chromosome could represent changing what section of the state space the exploration heads in. When the exploration first deviates from the path that (one of) the parent chromosomes took does not mean that the resulting state and the immediate states neighbouring it are relatively similar. When upon deviation the resulting neighbourhood is not sufficiently similar the following actions from the chromosome will change in meaning and these effects cascade throughout the remaining actions. State spaces where different neighbours of the same state have very similar shapes of following states are better suited for genetic algorithms because small changes to the chromosome do not cascade throughout the evaluation and radically change the output. The principal of genetic algorithms to increment solution quality by combining chromosomes and applying small mutations does not uphold for this problem and so the performance suffers as the algorithm is not able to leverage its properties.

3.8.4 Example - Travelling salesman

To approach the travelling salesman problem with the encoding setup in the problem description (Section 3.2), the bit-stream needs to be interpreted. An alternative for the travelling salesman problem is to use a non binary stream but for example a stream of integers. There exist genetic algorithms which deal with different types of chromosomes, including integers. For integer chromosomes (where each gene is an integer) there are different types of mutations that can be applied to pairs of chromosomes. The class of genetic algorithms where the chromosomes are encoded out of integer elements is called integer-coded genetic algorithm (ICGA). Those ICGA's function mostly the same as binary-encoded genetic algorithms, but the mutation operation is no longer possible (since there are no bits). Alternative mutators exist such as created by Michalewicz's solution[11] for real-coded genetic algorithms.

Given that there is some mechanism in place to a series of integers from a chromosome, we can look at following transitions from those chromosomes. The exploration of the state space uses a single state as subject to continue exploration and the integers 'read' from the chromosome determine the transitions that will be evaluated to find the next state to continue from. For the travelling salesman encoding used here, we have always one partial solution and a set of cities that the path in the partial solution does not visit yet. From this we can interpret the integers from the chromosome as some instruction for which city to add onto the solution path. Of course once the solution is no longer partial the exploration can terminate early. Actually mapping an integer at some stage of exploration to one of the remaining cities is no trivial task. One possible solution is to assign every city an index and let the numbers from the chromosome identify the city to be added. This creates the possibility of attempts to create invalid solutions, a problem similar as with the maze example previously. In the maze example the two options presented were to either ignore invalid commands or terminate early, the same consequences hold for this problem as well. Another solution might be to consider the cities as some sort of list, containing only cities not yet present in the solution. Now let an integer i from the chromosome for a list of cities containing n cities indicate the $(i \mod n)$-th city in the list. This method of picking the next state to continue with has the drawback that it suffers from side-effects where changes in the beginning of the chromosome can change the behaviour of later segments of the chromosome. For example take a list of cities $[c_1, c_2, c_3]$ and let the integers from a chromosome with length 3 be $[0, 0, 0]$. This original chromosome will pick the first element every time from the list, resulting in the selection order to be: c_1, c_2 and finally c_3. Changing the first integer to any other value (that is not a multiple of 3) also affects every other city being picked later on in the chromosome. This decreases the likely hood that small mutations to chromosomes will survive long enough to be able to evade local maximum in the chromosome pool.

The score of a chromosome can be measured as the total length of the produced (possibly partial) solution. A problem for the genetic algorithm workflow is that not every state at the end of a trace using a certain chromosome will produce a non-partial solution. Because of this the pool of chromosomes may contain chromosomes that produce either a partial or non-partial solution, there is no guarantee that a given chromosome actually produces a non-partial solu-

tion. The problem this poses is that the scoring function which measures the quality of a given chromosome needs to divide the non-partial solutions from the partial solutions. Otherwise the pool may be overrun by non-partial solutions that happen to be very short. One way to address this is to take the longest distance between the two most far apart cities and penalise a partial solution with this distance multiplied by the number of unvisited cities. This creates a cut-off point in the range of possible scores where all partial solutions are guaranteed to have at best the same score as the optimal solution, but likely worse. Also the chromosomes producing non-partial solutions can still be ranked on their length, at least between those that visit the same number of cities the comparison is purely between the total distance of the solution.

3.9 Simulated annealing

In metallurgy, the annealing process is used on materials to increase its ductility (making it easier to bend) among other things. The metal gets heated up to a high temperature to loosens up the internal structure (crystallisation) allowing for a new structure to emerge when cooling down. After heating the metal is cooled down slowly to let the metal recrystallise, forming a new structure internally. The slow cooling of the material allows for better alignment of the ions to appear than would happen with rapid cooling. The number of defects in crystals inside the metal are decreased and the size of the crystals that are formed are larger than before. This process allows brittle metal to be transformed into harder and more workable metals. Rapidly cooling the material is likely to create defects in the material, making it more brittle. When the material has fully cooled down it has reached a so-called ground state, which is where a state where the crystallisation process has finished and the new internal structure of the metal is set. In annealing there is a measure for how close a metal is to a ground state, this measure is called the energy of the material. A low energy level indicates the state of the material is close to a ground state.

This process is simulated in computer systems as a strategy to solve optimisation problems[7]. In the simulation, the system under test is allowed to change its state based on random probabilities that guide it towards a ground state. The change made to the state of the system is dictated by the energy levels of the current state and the new state of the system, as well as the 'temperature' of the system. For the temperature variable we denote T which may be any positive number ($\mathbb{R}_+$).

The temperature is lowered slowly (like in metallurgy) in order to allow for the optimal configuration of the system to emerge. The initial value for T allows for a starting grace period during which basically any transition is allowed to be followed. Near the end of the simulation when T approaches zero rarely any transition that leads to a higher energy state is followed and the algorithm dives into the nearest local minimal for state energy. This bounds the duration of the algorithm and allows for a configurable computation budget.

3.9.1 Energy

A state has a certain level of energy, the function that maps a state to its energy value we will denote with E.

$$E : S \mapsto \mathbb{R}_+$$

The function which determines the probability that a transition is allowed to be followed (and the current state to update to the transition target) is named D_p. The probability function is defined to resolve to a positive number whenever $E > E'$ and otherwise to approach zero as T approaches zero. For transitions that lead to states with lower energy values this enables the transition almost always while preventing enabling transitions to higher energy states increasingly as the system cools down.

$$D_p : (E, E', T) \mapsto \mathbb{R}_+$$

$$D_p(E(s_x), E(s_y), T) = \begin{cases} 1 & \rightarrow \quad E(s_c) > E(s_n) \\ 0 \leq n \leq T & \rightarrow \quad E(s_c) \leq E(s_n) \end{cases}$$

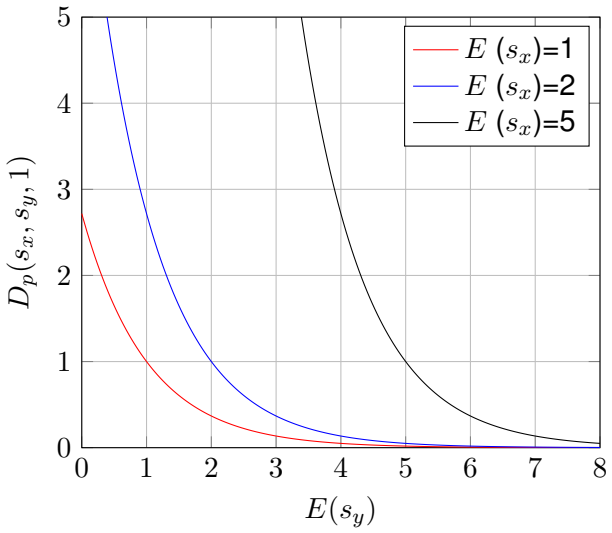


Figure 3.9: Graph of threshold values between various current states plotted for a next state candidate

A possible probability function is plotted in Figure 3.9 where D_p is defined as $e^{\frac{E(s_x)-E(s_y)}{T}}$. The graph shows for some different energy states the probabilities for each state plotted against the energy level of another state. The temperature is set to 1, this indicates that temperature has no dampening effect on the probabilities (as per the probability definition above).

3.9.2 Convergence

The simulated annealing algorithm is designed to 'converge' on the global optimal solution to the problem. The probability of convergence on the global optimal solution approaches 1% as the time budget for the cooling process increases. The algorithm is not guaranteed to provide the optimal solution under any circumstance, but the algorithm is guaranteed to be more likely to return the optimal solution as the provided time budget increases. Bertsimas D. and Tsitsiklis J. showed in their work [1] that the simulated annealing algorithm converges on the optimal solution. The application of simulated annealing is practical under the assumption that the time budget set for the cooling of the solution is sufficient to give reasonable odds to converge on the global minima while using less resources than a exhaustive search would take.

3.9.3 Annealing frontier

The simulated annealing algorithm is a simulation exploration algorithm (Section 2.8.7) because the exploration progresses through a single state. To implement a frontier which holds only a single state, it can simply be a restriction on the frontier. The process of selecting one neighbour as the next state then simply becomes the first successful insertion into the frontier. Normally this would insert the first neighbour but the energy-based random chance selection can be implemented by the frontier. Adding a state restriction on the frontier is sufficient, the restriction would perform the energy based comparison and if necessary the probability test.

This leaves open the chance where no neighbour is found fit enough to be inserted into the frontier. In case after evaluating the neighbours of some state, the frontier is still empty then that state needs to be entered into the frontier again. For states without neighbours (deadlocks) the state does not need to be added to the frontier and instead the exploration can terminate immediately.

3.9.4 Pseudocode

```
function annealing(s_initial, P_goal, E, D_p, T)
    S_frontier = S_initial;
    i = 0;
    while S_frontier ≠ ∅ do
        pick s_c as any state from the frontier;
        S_frontier = ∅;
        i = i + 1;
        S_frontier = {};
        if P_goal(s_c) ∨ S_out(s_c) = ∅ then
            return s_c;
        end
        for s_n ∈ S_out(s_c) do
            if D_p(s_c, s_n, T(i)) > R() then
                /* The frontier is replaced by a new singleton                */
                S_frontier = { s_n };
                break;
            end
        end
        if S_frontier = ∅ then
            S_frontier = { s_c };
        end
    end
end
```

Algorithm 7: Simulated annealing algorithm implemented with a frontier

A possible pseudocode implementation for simulated annealing is shown in alg. 7. Given to the function is an initial state $s_{initial}$, a goal state predicate P_{goal}, a state energy function E, an annealing probability function D_p and a function T. The probability function behaves as described prior in Section 3.9.1. The temperature function works slightly different than described before, here it is a function which provides the temperature of the system after a given number of iterations. Also used in the pseudocode is a function $R()$ this indicates the use of a random number generator in order to determine the outcome of a probability. In practice a common practice is to evaluate the probability against a uniformly distributed random number between 0 and 1. Although this does depend on the implementation of D_p.

The implementation uses a set as frontier merely to make the pseudocode more similar to those of previously discussed algorithms. A small but notable difference for this algorithm in the usage of the frontier is that after evaluating all transitions, if the frontier is empty then the current state is re-inserted into the frontier. This ensures that if no neighbour was picked (and there were neighbours to pick from) that the frontier does not exhaust but rather the temperature is lowered as usual and the state is evaluated again.

There is no usage of a heap because there is no need to test for known states against neighbours.

3.9.5 Example - Maze

Applying simulated annealing to a maze heavily relies on being able to define a very good energy function. Maze problems are limited in information pertaining to the solution of itself, for a given cell you have the local area around the cell, the global position within the maze and the positions of known areas of interest (e.g. exits and keys). This on its own is unlikely to provide a usable energy function for sufficiently complex mazes. Simulated annealing is prone to getting stuck in long corridors leading to dead ends, especially if moving forward through the corridor leads to lower system energies. The corridor is a local minimal (in terms of state energy) and very long corridors can trap the exploration within the corridor as multiple moves to higher energy states need to be performed in order to return to the junction where a wrong move was made. And this problem is faced at every junction of the maze. Mazes contain

a large amount of local minimal, some can be multiple transitions deep before reaching the maximal.

The simulated annealing algorithm is a simulation algorithm which as explained in Section 2.8.7 means that there is a limited set of states from which exploration occurs and at most a single discovery occurs during the evaluation of that state. This means that state spaces where states have low numbers of outgoing transitions, as with the corridor maze where cells in the corridor only connect to two other cells do not lend themselves well to simulated annealing because it is likely that in 'corridor' scenarios the local minimum can only be overcome by a series of discoveries to higher energy states. Which makes escaping the corridor statistically much more unlikely than say a corridor that is less long.

This follows from the convergence analysis done by Bertsimas D. and Tsitsiklis J.[1] where they described the probability for simulated annealing to escape a local minima to be exponential in relation ship to the depth of the local minima.

3.9.6 Example - Dining philosophers

Simulated annealing works best for problems where most states in the state space have many outgoing transitions to neighbouring states. This gives the algorithm more options to jump to states with higher energy and thus the likely hood for those states to lead to better maxims is also larger. The dining philosophers problem is such a problem where the state space is rather densely connected.

To represent the problem such that it can be approached by the simulated annealing we have to define the energy function. This function will produce a low value when the state is near (in the number of transitions) a desired state (deadlock state). And higher values when the state is further away from a desired state.

This is an ideal function because there is no local maximal the algorithm can get trapped in. However, it is not a realistic definition because it requires the entire state space to be known beforehand. A realistic energy function would be one where the energy is based on the number of waiting philosophers, as there is little other information that can be used for a useful energy function.

The result of this is that it becomes very likely the algorithm will follow down paths where many philosophers are waiting. However, there are a few pitfalls to this. First, when there are multiple philosophers waiting but they are part of waiting chains that flow in different directions then the algorithm must let the philosophers release their utensils. This is because one of the deadlock scenarios requires all philosophers to be waiting for the philosopher next to them. And two philosophers cannot be waiting for the same philosopher because that would mean the 3rd philosopher is in the middle and holding both utensils. This would lead to a possible action of dropping the utensils and looping those actions indefinitely or another philosopher can pick up their second utensil and eat.

The energy function does not take this into account and thus there could be multiple chains in different directions. Removing one of the wait chains would require a decrease in the number of waiting philosophers. This is where simulated annealing seems promising because while the temperature of the system is high it is possible for simulated annealing to break up those chains and may find a system deadlock ultimately.

The fixed budget simulated annealing can work under means that the algorithm can be run many times with a smaller budget to have multiple attempts at finding a deadlock solution (the desired state to be found). This would be a workflow algorithm that uses simulated annealing repeatedly. In the end, simulated annealing gives no guarantee on finding the desired solution or finding the best solution meaning that it is not a good algorithm to verify the dining philosophers problem with but it has very little memory consumption (in exchange for a larger need on computational resources for more repeated attempts).

3.9.7 Example - Travelling salesman problem

A classic problem for simulated annealing to solve in the context of state space exploration is the travelling salesman problem. The problem is a popular benchmark for planning algorithms

and scheduling systems. It is a problem that is NP-hard and suffers greatly from the state space explosion problem.

> Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?

An example implementation of the simulated annealing algorithm to solve this problem is to define the states as the list of cities in order and score the state by the total length of the path. The transitions between states are mutations on the order of the cities, a mutation is a swap between two adjacent cities in the list (first and last are the same city and don't need to be swapped). When executing the algorithm, its navigation through the state space mutates the current path of the salesman and will find paths of shorter and shorter lengths between each iteration. In the beginning, when the temperature of the system is high the system will make, among mutations that improve the state, a bunch of mutations that do not directly improve a better result but this allows the algorithm to escape an early local maximal. Because the chance of navigation is influenced also by the score difference between the states it will most likely navigate to any state that has a better score. As the temperature drops the algorithm starts to only make small improvements as the ability for the system to go to a lower scoring state lowers and it can almost only make improvements on the current state.

3.10 Conclusion

The algorithms shown in this chapter have shown to be able to be implemented with pseudocodes that are very similar in structure and parts of reusable components and behaviours. This means that a framework can be build that supports all the algorithms from this chapter as well as allow for variations for those algorithms to be made by changing the parameters of the exploration query.

3.10.1 General form pseudocode

The general form of a search algorithm is laid out like so:

function `search`($S_{initial}$, P_{goal})
  $S_{frontier} = S_{initial}$;
  $S_{known} = S_{initial}$;
  while $S_{frontier} \neq \emptyset$ **do**
    pick s_c from the frontier, based on the order imposed on the frontier;
    $S_{frontier} = S_{frontier} \setminus \{s_c\}$;
    if $P_{goal}(s_c)$ **then**
      return s_c;
    end
    for $s_{neighbour} \in s_{out}(s_c)$ **do**
      if $s_{neighbour} \notin S_{known}$ **then**
        $S_{known} = S_{known} \cup \{s_{neighbour}\}$;
        $S_{frontier} = S_{frontier} \cup \{s_{neighbour}\}$;
      end
    end
  end
end

Algorithm 8: The general form for exploration algorithms

The formulated algorithm is not immediately compatible with all the algorithms discussed so far, especially the loop which evaluates the outgoing transitions of the picked state. To account for this a framework which would support the range of algorithms discussed and more, has to also support deviation from the formulated generic exploration logic. This can be accomplished by identifying the sections of logic that have to be modifyable and allow for those parts of logic to be changed by the implementor of an algorithm. With regards to creating a framework where algorithms are implemented by a collection of components rather than crafting the algorithm

Algorithm	F_o	F_c	F_s	Heap	User	Aspects
BFS	FIFO	-	$\inf$	req.	-	-
DFS	LIFO	-	$\inf$	opt.	-	-
Dijkstra	f_{tc}	-	$\inf$	req.	f_{tc}	eval-trans
A*	$f_{tc} + H$	-	$\inf$	req.	f_{tc}, H	eval-trans
Sweep-line	F_p	-	$\inf$	req.	F_p	state-picked
Genetic algorithm	-	X	1	-	c_{gen}, P_{cont}, P_{next}, F_{score}	state-picked
Sim. annealing	-	X	1	-	E	state-picked, post-eval

Table 3.1: Composition of algorithms as an exploration query

code from the ground up this would mean that components must be able to modify specific parts of the algorithm logic by injecting and replacing the logic present. This has close ties with aspect oriented programming and event-driven programming.

3.10.2 Compositions

Using the general form and the pseudocodes of the algorithms we can deduce what settings each algorithm uses for the concepts defined in (Chapter 2). The final overview summarising the setting for the exploration query for each algorithm is shown in Table 3.1.

In the table, the column names have the following meaning: frontier order (F_o), frontier constraint (F_c), frontier size (F_s), type of heap (Heap), user defined functions (User), modifying behaviours (Aspects). In the table a (-) denotes that there is no value specified for that setting. Settings that are marked with an (X) indicate that a setting exist but it is not generic but specific to that algorithm alone. The first three columns are related to the frontier concepts explained in Section 2.6. The heap column shows whether the algorithm requires (req) a heap to be present, is optional (opt.) or not used (-). The user column shows the user defined functions that need to be provided by the end user as required by the algorithm, for their meaning consult the relevant algorithm section from this chapter. Finally the column aspects defines what parts of the algorithm have custom behaviour injected. In programming this is comparable to aspect oriented programming as the original algorithm code is augmented with the specific behaviours. The values in this column have the following meaning:

state-picked Behaviour after a state has been picked from the frontier

For the sweep-line algorithm this is the moment where the heap has its states purged. The simulation algorithms (genetic algorithms and simulated annealing) use it to update control variables that help identify each state evaluation.

eval-trans Behaviour for evaluation of a transition

In Dijkstra's algorithm and A* the evaluation of transitions is modified to allow for states in the frontier to have their cost updated (if a lower one is detected).

post-eval Behaviour after all transitions of a state have been evaluated

In simulated annealing this is used to ensure that the frontier contains the evaluated state if no neighbour was accepted due to the energy levels.

Chapter 4

Framework

4.1 Features

This section details the features of an algorithm. The goal is to specify commonalities and variabilities of the algorithms so the framework has a targeted feature-set to implement.

4.1.1 Next state selection

Picking the next state is a very distinct feature of any algorithm the framework wants to support. Each algorithm needs its method of picking the next state from the frontier for the algorithm to function. The implementation of this feature is tightly coupled with the algorithm it serves and as such, this feature does not have a fixed set of options but rather is something each algorithm extends.

4.1.2 Frontier size

A simple frontier that can be used to exhaustively explore a state space is one that has an infinite size and accepts all discovered states such that all states have their outgoing transitions evaluated. While complete in exploration this costs greatly in terms of time spent exploring and also some memory. State spaces with many states and transitions simply cause a lot of states to be inserted into the frontier as well. Limiting the size of the frontier helps speed up exploration by skipping some states and not exploring the state space fully. The feature can also be used to run simulations when using a frontier of size 1. Some algorithms may require a specific frontier size.

- Fixed frontier size
- Unlimited frontier size

4.1.3 Heap

The heap stores the known states of the state space. However, we found some algorithms don't use the heap at all. As such the framework should accommodate for that by not spending memory on remembering states. One of the algorithms (NDFS) we covered, uses multiple heaps. In that case, there is a main heap and a nested heap. But there is no indication that there will always be a hierarchy to the heaps.

- No heap
- One heap
- Multiple heaps

Most algorithms only push states to the heap and never remove them from it. Sweep-line, however, does remove states from the heap. Dividing the two types of the heap can allow efficient implementations (in terms of for example lookup or insertion speed) for the permanent

heap to be made as this implementation has the additional assumption of never having to remove a state from the set.

- Permanent heap (states cannot be removed)
- Weak heap (states may be removed)

4.1.4 State comparison

In computer science the testing of equivalence between two entities can be a very expensive operation, for example should two elements be graphs then the equivalence testing may be a graph isomorphism test. This expensive computation may also be substituted for a heuristic (for example Section 2.7.1). Comparing two states is highly dependent on the model of a state. The comparison is left up to implement by the person that implements the state representation. The system simply requires an equality testing function to be set.

- Equality test on all properties
- Graph isomorphism (GROOVE specific)
- Hash comparison (requires hashing function)
- Custom comparison

4.1.5 Result type

Each algorithm serves a specific purpose. The algorithms explore the state space in their uniquely designed way to find states or patterns of states. The primary goal of the algorithm is to explore the state space and find unknown states. The framework will support methods for reacting to the discovery of states and expose those states that match the goal criteria as results from the exploration. As states are reported as goal states, the framework can also support responding with traces for each goal state. Besides these general response types, an algorithm may be dedicated to producing specific results and the framework has to allow the algorithms to produce custom results as well.

- Goal state
- Trace to goal state (optional)
- Custom result (by algorithm)

Some algorithms may not support some of the response types such as traces. A user may not even be interested in traces to goal states, therefore we should allow the user to disable producing traces in case they only want to find states.

4.2 Exploration constraints

The following sections contain possible constraints that should be built into the framework to allow the user to limit exploration by the algorithm. The settings are part of the exploration query.

4.2.1 Frontier filter

The previous feature detailed the possibility to save on exploration time by limiting the frontier. While limiting the size is effective it takes little note of the context of the search. This is where the frontier filters feature steps in. In explorations for a specific type of state in the state space, problem knowledge can be used by the user of the framework to specify a filter that prevents states from entering the frontier. Leaving the feature open for implementation allows the user to use this feature in many different ways. For example, large portions of the state space can

be guarded of and not be explored to save time. Also in situations where the user cannot guard of specific areas the filter can be used in combination with a heuristic to create a state quality filter of sorts. A third example is a limit on the depth of exploration. This, however, requires depth to be a concept used by the algorithm and for the algorithm to instrument the states to contain such information.

4.2.2 Result count

The number of results an algorithm can produce may be more than just a single state or trace. Some algorithms may not be capable of returning more than one result. For those algorithms that can return multiple results, their mode of operation may differ between returning single or multiple results.

- Single result mode (0 or 1)
- Multi result mode (0 or more)

4.2.3 Early termination

A user may not need to explore the entire state space to find the result they are looking for. Previously this has included goal states and a goal predicate, but more complex results may require collection of multiple goal states or states in particular locations in the state space. This feature is optional as exploration also terminates when the frontier becomes empty. Cases, where early termination is desired, is to accumulate a specific number of results or when working under time constraints. Additionally, the user may define a problem specific termination condition based on results and the snapshot. This is the most general extension point for additional types of early termination. The different options of this feature can be combined to create multiple early termination conditions.

- Terminate on result count limit (1 or more)
- Terminate on result conditionally (includes snapshot data)
- Terminate after a timeout from exploration start
- Terminate after a timeout from the last result

4.3 Functions

Functions used throughout the algorithms and framework have been categorised such that there is the possibility of reuse. Specifying the different types of functions documents their purpose and general behaviour and allows them to be reused by new algorithm implementations and features in the future.

4.3.1 State equivalence

The most basic comparison the framework needs to be able to make on states is a test for equivalence. Equivalence functions are predicates that receive two states as input and resolves to true when those two states are considered equivalent. There are different ways states can be considered equivalent. The choice on which function is used is something that is closely tied to the context of the equivalence test. If the context demands a specific type of equivalence test then there must be a specific function that fulfils the specific type of equivalence test. This would be the case when wanting to exploit a specific property of the equivalence relationship that can be established. There also are general cases where equivalence needs to be tested without having some exploitable property in mind. In such cases, the equivalence is based on the intrinsic model of the states and thus relies on the (meta)model of the states to define one. Those equivalence tests will be mostly used in the framework implementation as well, since using states without knowledge of their internals is part of the foundation of the framework.

The user of the framework, those that decide what model is used for states has to define the equivalence test.

4.3.2 Goal function

In the description of algorithms, we incorporated the termination of the search in most pseudocode. This is because exploration is usually targeted and done with some goal in mind. This common use case of searching with a goal in mind requires a standardised way for algorithms to determine when the exit criteria are met. There are many ways to define the exit criteria, but when it comes to states there is some filter the user wants to use to distinguish non-goal states from goal states. This function is a predicate which can receive any state and indicates which states are goal states. Other features such as the number of goal states before triggering early termination are not of concern to the predicate. However, it is possible that a goal state may be identified by some pattern of neighbouring states. An example of this would be to identify states without outgoing transitions, in some models these are called deadlock states as they block execution of any further action.

4.3.3 Heuristic

With guided search algorithms there is a level of guidance the user of the algorithm or framework specifies through a heuristic function. A heuristic function takes in the context of the progress made in exploration (that is to say the 'state' of exploration, not to be confused with some state from the state space). The context of the state space exploration is required because a heuristic is used to create an ordering of interesting states, such that the most interesting state can be explored next. In other words, an array of states can be sorted using a heuristic function which identifies for each state a score which the states can be ordered against. A heuristic is a tuple of a heuristic function along with a direction in which elements should be sorted. The primary use of heuristic functions is to create order between multiple states. The direction indicates whether interesting states are located at higher or at lower scores. Sorting is not limited to use of a single heuristic, multiple can be used simultaneously. When sorting according to multiple heuristics an order is needed on the heuristics, which determines their priority. Sorting with multiple heuristics works as follows: the array of states is sorted by the most important heuristic. Next, for each group of states which have the same heuristic score, the inner ordering of their group is determined by the next most important heuristic. And so on until there is no heuristic to sub-sort the group by. In cases where some states score equal for the first heuristic dimension then the subset of states which have the same score can be sorted again using a second heuristic. This can be done with as many heuristics as you could need. When performing the sort based on heuristic values, there needs to be a consistency in the direction of the sort result. To do this we can simply say that sorting will always result in descending order according to the heuristic values. Meaning, more interesting states should receive higher values, regardless of the scale of the numbers. The implementation of multi-dimensional sort can be done through a modified version of quicksort where the comparison is based on heuristic scores and in the case of equal heuristic scores the next sort dimension (heuristic) is used until no more heuristic functions are left or a distinguishing order exists between the two states by one of the heuristics.

4.3.4 Progress measure

While progress is only used by sweep-line, it may seem comparable to a heuristic but it is a separate type of function since it is used differently and has different effects than a heuristic does.

A progress measure identifies for a given state a score (similarly to a heuristic). But the meaning of that score differs from a heuristic in that it does not indicate an order of interest among other states but rather indicates the shape of the state space. The value of progress increases the further a system has progressed from the initial state. It is implied that there are little to no transitions leading from a state to one with a lower progress measure.

When designing the progress measure for a particular model, system or state space the shape of the state space has to be known somewhat as the sweep-line is best used when taking advantage of the sense of progress a particular model may make.

The meaning of progress is that whenever one or more states have a lower progress measure than any state in the frontier it means that those states will likely not be found in outgoing transitions in future evaluations of states. As such the states will be removed from the heap. There can be as many states with the same progress measure as needed and the value can be any integer. The choice to use integers over real numbers is to prevent infinite progress layers as the number of layers between any two layers represented by integers is finite (or at least countably infinite).

Similar to heuristics, multiple progress measures can be used simultaneously to create a multi-dimensional progress measure. The application and workings of multi-dimensional sweep-line were developed by [8]. The details of how the sweep-line implementation differs to one-dimensional sweep-line are out of scope for this report, but the addition of multiple dimensions allows for sweeping and purging of states according to multiple properties simultaneously. Important to note is that if not careful, the number of persistent states will dramatically increase as a regression any single progress measure marks are at permanent.

4.3.5 System energy

The system energy function is a function like heuristics and progress measure that operate on a single state as input and return some numerical score. System energy functions are a subset of heuristic functions tuples, those with a specific sort direction. The interpretation for system energy functions is that their use in simulated annealing assumes lower scores are more preferred than higher scores. The dedicated sort direction in sorting states according to a system energy function is to align the lowest valued states with the highest interest region of the sort.

Functions that represent system energy will likely base their result on the inner data and representation of the state. This places the burden of implementing such functions on the user of the framework (those who decide the model of states) because the framework design does not give any indication on how to otherwise build a generic system energy function.

4.3.6 Transition cost function

Dijkstra's algorithm uses transition cost and is so far the only algorithm to use information about transitions. But it is not unlikely that users would want to integrate transition information into their explorations.

The transition cost function relates a numerical cost to a given transition. Transition cost functions can use information from the transition such as source and destination of the edge, but also user set information such as weights and other properties. The function results in a number indicating the cost to travel/explore the transition. The higher the number, the larger the cost and therefore intuitively the less likely the transition will be evaluated next. Transition functions can be used to order the frontier (like with Dijkstra's algorithm) by the transition through which they were discovered.

4.3.7 Transition function

Functions related to transitions are used by algorithms to prioritise the frontier based on the transitions leading to new states. These functions produce a number ($\mathbb{R}$) given a transition as input. The interpretation of the number produced by the function can be one of two categories. Either the function is a cost function or a score function. A cost function indicates the transition should be avoided if the result is high compared to other transitions. The score functions are the opposite and indicate transitions are favourable when the result is higher than that of other transitions. Dijkstra's algorithm uses a transition cost function as the frontier is sorted where transitions with a lower cost are explored sooner than those with higher costs.

Chapter 5

Case studies

The research questions will be answered by building a framework in which state space exploration algorithms can be built. To verify the validity of the framework, a set of case studies will validate the claims made and whether it solves the problem at hand.

- Ease of application by users
- Requirement validation: simulation
- Requirement validation: listed algorithms
- Demonstrate building block architecture
- Explore sweep-line variation

5.1 Anthill simulation challenge

A form of state space exploration is simulation; where the frontier has only a single state and from the outgoing transitions a single one is picked to progress with. Typically simulation does not keep track of known states as the goal is to progress the state into the frontier, of course, can be variations that do track explored states. A framework surrounding state space exploration should be capable of having simulation implemented with little difficulty. There exist some simulations from challenges and competitions we can use to verify compatibility with simulation as well as benchmark the performance.

One such simulation is the AntWorld simulation challenge from the 2008 GraBats transformation tool competition [19]. The challenge defines an anthill sitting in the middle of the field where ants go out into the field searching for sugar to bring back to the anthill. There are various details about the challenge that make it increasingly difficult to simulate. Besides doing simulation on the AntWorld problem, we could also switch the settings over to an exploration algorithm to see how much of the state space can be explored with limited resources.

Interesting for the 2008 GraBats competition is that they have archived the solutions of contestants to the AntWorld problem in the form of virtual machine archives (.VDI files). These are available at [14], where solutions for other competitions are available as well. We hope to be able to gain access to those images which provide solutions to the AntWorld problem.

Alternatively, we can explore solution papers from competitors and attempt to reproduce a system with the same resources as their benchmarks had. By under-clocking the CPU and setting memory constraints appropriately the hardware limitations can be approximated. This approach is much less reliable however since we can't ensure we have the same environment for our solution.

In the problem there is an event which expands the world border, leading to the world to grow indefinitely over time. Considering that the size of the world is part of the system state this means that it is impossible to return to a previous state once the world border has expanded. This is a form of progression that sweep-line could exploit for purging large groups of states. We expect that when using sweep-line on the problem we will be able to explore greater depths than other algorithms which do not purge states. But as the world size increases the number

of states per layer increases drastically and we instead will run out of resources once a layer is too large to keep within the memory resources available.

5.2 User application

For frameworks and code libraries it is important to build a framework that is easy and intuitive to use. To validate claims of usability and ease-of-use, a case study can be executed where programmers try to build a solution to a problem where state space exploration or graph exploration is applicable. From their solution, the implementation time is recorded and an interview is had where the programmer is asked how they experienced the use of the framework. Also, the problems they had and how they overcame them are useful for further improvements to the framework.

The study is set up to have a problem which is related to state space exploration or graph exploration such that application of the framework provides a good and logical solution. There are also requirements on the programmer(s) which participate in the study. They should know the problem domain or be reasonably expected to understand the domain and the terminology of the framework. This doesn't mean that the programmer has to be an expert beforehand but should be able to understand the problem when explained. Both the problem and the framework has to be described to the programmer. The problem should be explained to the programmer and the programmer should be able to ask questions about the problem. Being able to get a complete understanding of the problem is important for the programmer to have so the difficulty of the task is not in figuring out the solution to the problem but mostly to implementing it correctly. The explanation on the framework should be provided before-hand (like regular documentation on an open-source project). Questions that arise about the framework reflect on a short-coming in the documentation or ease of finding the information. The questions asked by the programmer about the framework are an important part of the output of the study.

5.3 Extension to GROOVE

One of the case studies ought to be a practical application of the framework to existing software to demonstrate compatibility and points of improvement. The application of choice is GROOVE. GROOVE is a graph transformation system modelling program which explores state spaces of graph transformation systems. A single state in GROOVE is a graph which represents the state of the system at a given moment. Noteworthy is that transformation rules are also expressed as graphs which makes modelling and exploration a very easy to understand visual experience for the user. The program itself has a few build-in exploration settings and allows customisation of the exploration with its control language. The control language has flow controls such as conditional statement execution and also non-deterministic statement execution through its *choice* operator.

The application of the framework to GROOVE would entail extending or replacing the exploration engine of GROOVE with one that uses the framework to explore the state space. In the case of replacement, the existing exploration settings have to be reproduced using the framework. Also, the user interface would need to be updated to let it control the settings of the exploration query. To evaluate the changes to GROOVE a set of models can be explored with both the altered and current version of GROOVE to check for the resulting state space to be correct. The set of models will include the samples provided with GROOVE during installation and a set of models for known problems that others have also explored.

5.4 Sweep-line workflow

The definition of sweep-line has a drawback that layers have to be fully explored before it can be purged. There may very well be problem context where some states may not be of interest to finish a layer and these states would be weighing back the algorithm. In the case where

those states are truly useless re-drawing the layer boundaries would only allow you to purge a layer quicker but those states would then need to be explored in the next layer. There is no way for sweep-line to purge a layer early to save time. Sweep-line is a great base algorithm to use as a building block for other algorithms once you can split the state space into progression layers.

In a sense, sweep-line is a workflow for specific kinds of state spaces where some inner algorithm is exploring progression layers. The frontier as presented for sweep-line is a single frontier but when used as a work-flow, the discovered states would be reported to sweep-line and either inserted into the frontier of the current layer or (when it is larger in progression) saved for the exploration query of the next layer.

An alteration of sweep line (turning it into a workflow) would allow it to run any algorithm to solve progression layers while keeping the memory benefits of sweep-line. To do this the nested algorithm would have a separate frontier from that of sweep-line. The nested algorithm has a frontier that only receives states that are in the current progression layer being explored. When a state gets discovered that is further in progression than the current layer it is put into a frontier managed by sweep-line to start the exploration of the next layer. The nested algorithm has its frontier and controls when exploration of the layer is done.

To show the correct working of the strategy and the memory usage of the technique the nesting of algorithms will be tested by solving a maze with a key. The maze will contain intermediate checkpoints which have to be visited before the exit can be reached. The results from using the regular sweep-line algorithm will be shown together with those of sweep-line as a workflow in combination with other exploration algorithms such as BFS and A*.

A major draw-back of sweep-line is that the purging of the layers destroys the ability to produce traces to goal states at the end of exploration. A general solution mentioned in [9] uses disk storage to save the full state space for later examination (and reconstructing back-traces). For this problem context, the number of states in any layer is much larger than the number of initial states for the next layer because there is a limited number of checkpoints and goal states. This means that traces can be saved onto the states before purging the previous state. Each purge extends the existing traces and this ensures full traces are available at the end of exploration. This increases the overhead sweep-line has compared to using only A*.

Expected is to see nested sweep-line combine the memory advantages of sweep-line with the exploration speed of A*. The required time to complete the puzzle is expected to be roughly the same (bar from the overhead of sweep-line) but to see memory consumption to be lower due to the purges which occur after a checkpoint has been reached. This would lead to believe that the nested sweep-line form is capable to handle maze puzzles at a larger scale than A* itself could do with the same resources.

Chapter 6

Research questions

Chapter 3 has shown that we can define and implement algorithms with code patterns that are very similar and that these algorithms can be generalised into a set of components combined on top of a general base search algorithm. In Chapter 4 we specified some of the features that those algorithms have in common and the components that such a framework would need to support. Our main question now is whether this can be successfully implemented in a manner that allows for highly performing algorithms to share the same interface. We want to build a framework for exploration algorithms that can support the basic algorithms discussed and proof that other algorithms can be implemented as well. The framework also needs to satisfy our goal of promoting reusable components. This should manifest itself as allowing for tweaking in parameters of exploration queries as well as being able to replace small components of the exploration query with customised behaviour to modify the exploration algorithm behaviour.

6.1 Research question

Research Question 1. Can a general framework for building search algorithms be designed?

In the design of the general framework there will be a few challenges and concerns that need to be taken into account to ensure our goals of creating a framework that promotes code reuse is satisfied while remaining easy to use and generic enough to apply to sufficient problem contexts. For this we formulate the following research sub-questions:

6.1.1 Does the general framework support implementation of exploration algorithms through composition of behaviours?

With this we mean to ensure that algorithms can be implemented by combining existing features, components and behaviours from the framework and possibly other programmers. We ensure this by implementing the algorithms from chapter 3 in such a manner that they are not programmed in a fashion similar to their pseudocode but rather by using the same general base form and injecting into that the behaviours that make the exploration behave as the algorithm defines. Verification of this requirement on the framework can be done by implementing demo's and test code that uses the framework in a manner just described. Additionally we can also verify the support for this by making it part of the usability tests to program an algorithm in a compositional manner. The method for this will be described below.

6.1.2 Can specific behaviours of algorithms be integrated into other algorithms?

This sub-question is meant to extend the previous sub-question to verify that the framework supports reuse of components between different algorithms. Given the previous sub-question and requirement it introduced an algorithm may be implemented by composition of some set of components but does not yet proof that those components are generic enough to reuse amongst different algorithms. Not every component needs to be interoperable with each other but at least within sets of components that implement different concepts there should be some

reusablility to the components. For example for two sets of components: one which contains implementations of frontiers and one with implementation of heaps, there should be implementations of frontiers and heaps that can be combined freely unless the definition for the behaviour combines those concepts and tightly couples them together. To answer this question a feature diagram can be made of the components. The feature diagram shows the elements an exploration query (see Section 2.8.8) is made up. This will show what components place constraints on each other. To verify that the behaviours can be combined sufficiently the diagram has to reveal that there only exists constraints between components for which the underlying concept (e.g. components related to sweep-line heap purging) tightly couples those components together. For example for sweep-line the behaviour that purges the heap places a requirement on the frontier to be sorted according to a specific metric. This constraint is allowed because the concept of sweep-line heap purging requires it. But a generic component, for example a queue-like frontier, should not place a requirement on another component because it is generic enough to not warrant it therefore its implementation should not have such requirements or incompatibilities either. The feature-diagram justifies incompatibilities between reusable components.

6.1.3 How well does the framework integrate with existing applications of state space exploration?

Our goal of reuseability starts with creating a framework that can be applied to unknown state representations so the framework is usable for existing state encodings. To proof that the framework is a useful innovation it will be integrated with two existing state space exploration applications. The applications that will be integrated with are GROOVE [18] and pddl4j [16].

GROOVE is a visual editor for defining graph transformation models and explore their state spaces. It already contains various exploration strategies and options as well as an integrated exploration programming language. Besides the application the project also includes the exploration and modelling core of the application as a library and commandline tools to export state spaces and traces for a given GROOVE model. Integrating the framework with this application will aim to replace the existing exploration code with the framework and refactor the existing exploration options to be supported by the framework. This will show that the framework can be integrated with existing software as well as possibly provide a wider featureset to the end-user immediately after.

The second integration will be with pddl4j, which is a java library for parsing and model checking problems encoded as pddl models. The project already includes support for exploration strategies including an interface to deliver your own exploration strategy. The goal of integration with pddl4j will be twofold: first the integration will provide support for exploration on pddl models to the framework and secondly the integration will provide new explortion strategies to the framework through the mentioned interface. Integrating with pddl4 shows compatibility of the framework with externally defined state representations as well as highlight usability of the framework in existing exploration solutions. Finally given that the feature-set of the framework exceeds the feature-set of exploration strategies present in pddl4j the integration is also a contribution to expanding the exploration capabilities of pddl4j.

6.1.4 How does the performance of the framework compare to that of existing state space exploration implementations?

Once the framework has been implemented it can be compared to existing implementations of state space exploration. The comparison is expected to reveal that the computation complexity is similar but to there might be specific optimisations in existing solutions that the framework is not able to take advantage of. Should these come up during testing of the performance it will also be relevant to analyse the implementation being compared against to see if there are optimisation strategies the framework could make use of. Especially optimisations which are not possible to implement in the framework are of interest.

The performance will be measured in two areas: memory usage and execution time. The memory usage will be measured by the program memory (RAM) used during the program

execution. It is expected that the complexity of memory usage and execution time should be the same where the same exploration query (algorithm and user functions) are used. Besides executing the same exploration query within the created framework there will also be a set of variations on the algorithms created. These are made to show the versitility of the framework but also allows us to compare the improvements the variations make on their original algorithm. Given that the algorithm is able to be integrated into existing applications (see previous sub-question), this will proof that improvements the variations are able to achieve would translate to improvements in the existing applications as well. When running the performance tests the following conditions for testing will be used to make the test results as relevant as possible:

- Executions occurs on the same machine
- Executions are granted the same time and memory resources
- Executions are repeated over multiple iterations to create reliable mean results
- Executions use the same input problems
- Executions are only compared for solutions of equal quality, or results are penalised for inferior solutions

These conditions ensure that the algorithms have the same resources to perform the task as well as make sure that in scenarios where the answers differ (this is expected in algorithms employing randomness and heuristics) is accounted for. The problems that can be tested include the planning problems that are part of the pddl4j[16] project and models made in GROOVE[18] explored with original GROOVE exploration algorithms and with the integrated framework. For GROOVE there is also historical data for its participation in the GRABATS tournament which can be usedSection 5.1. The final results of the performance tests will result in a table where algorithms have their performances laid out against the set of problems they have been employed on. This table will reveal the performance of the framework compared against existing solutions and proof that the performance is on-par.

6.1.5 Is such a general framework easy enough to work with while remaining sufficiently general?

Besides the goal of building a performant framework which is well suited for existing problems and solutions it also needs some sense of usability. A framework may support a very wide range of features and capabilities but this may lead to a highly complex framework that is difficult to use especially for those who are not familiar with it. This places a barrier to entry on new users of the code framework where they need to have a deep understanding of the framework and its architecture before being able to effectively use it. To mitigate this the framework should be designed such that the architecture as percieved by the user from the API and documentation abstracts away mechanisms and code patterns used within the framework. This can be verified with usability tests in which a group of programmers are introduced to the framework and are asked to use it. The usability test will simulate a newcomer to the framework who has a problem they want to solve with state space exploration. The test will comprise of validating not only the ease-of-use sub-question but contain a series of tasks which also touch on the other sub-questions and requirements in them. The usability test will be concluded with a small survey where the test subjects are asked a series of questions as well as able to provide free feedback on the test experience.

The method of testing will aim to minimise the complexity of the problem to solve such that the application of the framework becomes the main problem to solve for the test subject. To this extend the test subjects will receive guidance on the problem to solve but not when in relation to usage of the framework. For this they receive the same resources (documentation, demo, source code) as a regular newcomer would have available. It is expected that the demo code would be the main resource the test subjects will consult after which comes the documentation and source code.

The methodology for the usability test will be based on cognitive walkthroughs, feature inspection and standards inspection (from a summary by J. Nielsen [13]) to ensure a rigorous testing method will be used.

6.2 Methodology

The following plan has been devised to maximise the output of the project and try to answer the research question to the best of our abilities.

6.2.1 Requirements

The first step is to take the framework description and (while keeping the explained algorithms in mind) derive a set of requirements for the framework.

After creating the requirements the software architecture can be decided. The first and foremost choice will be to decide on the interface that will be used to let others use the framework. Keeping universality in mind a possible solution is to build the framework out as a network-based service (e.g. HTTP/REST or SOAP) to let virtually any programming language and application be able to interface with the framework. This would immensely increase the technical complexity of the solution but would achieve great compatibility with any existing programming language as nearly all have the capability of network messaging. Otherwise if opting for a more traditional framework in the form of a code library there is a considerable amount of options in terms of programming languages.

6.2.2 Implementation phase 1

After the framework has been designed it can be implemented and tested. Testing involves first running simple (toy) problems to verify a basic working of the system and demonstrate its feature-set. After this has been done a set of difficult problems will be attempted to solve, this will highlight the scalability and bottlenecks of the solution.

After these tests, there is a small integration task to complete before iterating over the framework. Integration with GROOVE will be build to prove the framework can be used from the java language and can extend existing applications.

After these tests and challenges improvements can be made to the framework, this may also be done along the way for critical problems and bottlenecks. It can be expected that after the tests and integration with GROOVE some new requirements may be found or found to have been under-specified. These changes will make the framework more complete and more ready to be used by developers upon release. Before the usability tests begin, the documentation for the framework will have to be created as this is part of the preparation of the usability tests.

6.2.3 Implementation phase 2

At the point, the framework approaches its release candidate state the usability test(s) can be run. This will involve a willing participant programmer(s) to try and solve some problems (as described by the case studies section). The usability test serves to point out lacking documentation and insufficient ease of use. They cover three topics of use: ease of use by developers, reusability of components and extendability of the framework.

6.2.4 Project wrap-up

After the second round of testing the final changes will be made to the documentation and bug-fixes for any bugs that may have presented itself to the participating programmer(s).

The answer to the research questions can then be found by analysing the test data and the framework will be presentable as a product to developers and users of state space exploration.

Bibliography

[1] Dimitris Bertsimas, John Tsitsiklis, et al. Simulated annealing. *Statistical science*, 8(1):10–15, 1993.

[2] K. Mani Chandy and Jayadev Misra. The drinking philosopher's problem. *ACM transactions on programming languages and systems*, 6(4):632–646, 1984.

[3] Søren Christensen, Lars Michael Kristensen, and Thomas Mailund. A sweep-line method for state space exploration. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 450–464. Springer, 2001.

[4] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics*, pages 176–194. Springer, 2001.

[5] Patrice Godefroid and Sarfraz Khurshid. Exploring very large state spaces using genetic algorithms. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–280. Springer, 2002.

[6] Michel Hack. Petri net language, 1976.

[7] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[8] Lars Michael Kristensen and Thomas Mailund. A compositional sweep-line state space exploration method. In *International Conference on Formal Techniques for Networked and Distributed Systems*, pages 327–343. Springer, 2002.

[9] Lars Michael Kristensen and Thomas Mailund. A generalised sweep-line method for safety properties. In *International Symposium of Formal Methods Europe*, pages 549–567. Springer, 2002.

[10] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl-the planning domain definition language, 1998.

[11] Z Michalewicz and Zbigniew Michalewicz. *Genetic Algorithms+ Data Structures= Evolution Programs*. Springer Science & Business Media, 1996.

[12] Edward F Moore. The shortest path through a maze. In *Proc. Int. Symp. Switching Theory, 1959*, pages 285–292, 1959.

[13] Jakob Nielsen, Robert L Mack, et al. *Usability inspection methods*, volume 1. Wiley New York, 1994.

[14] FMT University of Twente. Share - home, 2019.

[15] Radek Pelánek. Fighting state space explosion: Review and evaluation. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 37–52. Springer, 2008.

[16] Damien Pellier and Humbert Fiorino. Pddl4j: a planning domain description library for java. *Journal of Experimental & Theoretical Artificial Intelligence*, 30(1):143–176, 2018.

[17] David L Poole and Alan K Mackworth. *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2010.

[18] Arend Rensink. The groove simulator: A tool for state space generation. In *International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 479–485. Springer, 2003.

[19] Arend Rensink and Pieter Van Gorp. Graph transformation tool contest 2008, 2010.

[20] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, NJ, 1996.

Appendix B

Test Results: S1

Performed tasks 1 and 4, skipped 2 and 3 due to time constraints.
Programming skills: java expertise

Introduction

During the explanation of SSE he was largely familiar with the concepts of graphs and those terms, the concepts of SSE were quickly understood.

Challenge 1

Task 1 - Run goat game and draw the state space

When drawing the state space diagram the occurrence of transitions to a known state was sometimes confusing because then no new state needed to be drawn. In the end without much help he was able to draw the correct state space diagram.

He did not run the exploration to termination, but stopped once the ‘solution’ state was drawn, because of this the final few states do not have all their outgoing transitions drawn. The transition leading from the solution to the state prior to it is missing.

He was able to deduce that the algorithm executed was BFS.

Task 2 - Change the algorithm to DFS

He searched through the code to understand the framework.
He used the inheritance of the classes and interfaces to figure out how the framework was built up, and did not pay much attention to javadoc. The naming scheme of classes was the most important method of discovery.

He ended up finding the correct frontier and static method.

Task 3 - Implement early goal termination

He first inspected the state representation class (GameState) and found the GameState::isGoal method. Now he wanted to know how he could use that to end exploration.
Solution 1: Change the next-function to not provide states if the given state is the goal state. This did not work since the exploration still had states left in the frontier once the goal was found.

Solution 2: Use the heap in some way
He wanted to test the heap for presence of a goal state and use that to figure out if the goal state had been found. This attempt was not fully implemented and did not work

Solution 3: Wipe the frontier once goal is found
(I had hinted at this point that the frontier would still contain states once the goal has been identified)
The next-function would test states to be the goal state and once found then the frontier would be cleared and the function would not return any new states.

(I hinted now that he should have used a behaviour for this)
Solution 4: Using a behaviour
He looked at the query::addBehaviour method and started to implement an anonymous implementation of the interface (ExplorationBehaviour) to do this.
I had to explain to him that he needed to use the tappable system and helped him setup the tap. He understood taps (the interceptors) as decorators from python.

Behaviours were difficult to grasp: the link between query events and behaviours was not obvious. The purpose of the AbstractBehaviour class also wasn't clear and only the base interface was used.

Challenge 2

Skipped

Challenge 3

Skipped

Challenge 4

First attempt was a manually crafted next function that optimizes the exploration. His next-function would only return the stepping-back option when near the top of the staircase. When the additional requirements were introduced his approach changed.
He looked at the implementation of the goatgame state and then implemented a similar next-function where all options are generated.

When wanting to optimize the order of the frontier he implemented his own frontier. He did not register TreeMapFrontier as a suitable frontier type. I think he may have read over it because the words did not trigger relevant to him.
His custom frontier implemented the base Frontier interface and only tracked a single state.

Unfortunately we ran out of time at this point.

Test Results: S2

Background: non computer science background
Performed challenges 1, 2 and 3. Challenge 4 was skipped due to being too difficult.

Programming skills: programming basics, java unfamiliar.

Introduction

Challenge 1

Task 1 - Run goat game and draw the state space

He was able to draw the state space without much difficulty.
He had drawn the transitions with lines instead of arrows.
He numbered the state discovery order.
He was able to figure out that the exploration was BFS.

Task 2 - Change the algorithm to DFS

He used autocomplete on the QueueFrontier to find the lifo frontier fairly quickly.
He knew that the frontier would have to be a lifo frontier because I hinted to this during the introduction.

Task 3 - Implement early goal termination

I hinted that he would need to use a behaviour to implement this.
I hinted that the framework had a set of standard behaviours but he did not end up looking through them.
He looked through the LogEventsBehaviour (already added to the query) source code to figure out what a behaviour could do.

Solution 1: Loop through the frontier to find the goal state
He wanted to loop through the frontier before or after the query execution. I clarified that this would not work because the frontier would not contain any states yet.

Solution 2: Use some query events to do something with states.
I started providing guidance with step-by-step building of a solution. Breaking the solution down into some steps:

- Perform some logic when we find new states
- Identify a goal state
- Stop the query somehow

He looked through the query class but skipped past the tappable events because he did not understand them. Once I explained that the tappable system was an event system that he could use to interact with the pipeline he picked the beforeStateEvaluation.
While he was trying to write the callback for the evaluation event he was looking through the documentation of the class, he did not see the link to the base class in the source-code and because of this missed the method to terminate exploration manually.
Once I hinted to this he was able to find the correct method and finish the custom behaviour.

Challenge 2

He understood the link between positions in the maze and states in the state space.

Task 1 - Determine the exploration method

He deduced that the method of exploration was BFS

Task 2 - Terminate the exploration when reaching the goal state

He did not spot the TerminateOnGoalBehaviour again when looking through the list of framework provided behaviours. I think directed him to the correct behaviour to see how usage of the behaviour would go.
He had difficulty building the predicate, he did not fully understand that it was a callback and how it was supposed to function.
Although the lack of documentation on the PlayerState class did not help in this case.
It became clear that java was a major obstacle for this participant in terms of completing the tasks.

Task 3 - Optimize the query

He came up with the idea to explore states nearest to the goal state instead of using the BFS order.
I hinted that there is a frontier that is able to do that and while he was looking through the frontiers he found the TreeMapFrontier.
This class was lacking in documentation on its function and how to use it.
He first tried to create an instance of it with the constructor, he ignored the error thinking it was because the invoke was not correct in arguments yet. Once he had fully setup the comparator then he realized that the constructor was private and could not be used.
He looked at the constructor and noticed the static methods below it and then tried to use the first one of them (withExactOrdering).
When executing the query the flaw with the exact ordering came up where cells that are transposed from each other in the grid would be considered equivalent.
He thought this was because he used the wrong frontier.
I had to explain this quirk of the exact ordering frontier and helped him correct it with the hashcode fix.
After that his solution worked fine.

Challenge 3

Task 2

The task was already completed in the default setup of the testset

Task 4

Implementing the sweep-line behaviour was very difficult.
The concept itself was not fully understood.
The documentation was used by the subject and explained what he had to do to get the SweepLineBehaviour to work to some extent.
When displaying the effect of the saw-tooth the subject noticed that the effect was not very notable in this use-case.
In hind-sight this test should have been re-designed to make it easier to implement SL, and highlight its benefits more.

Challenge 4

Skipped due to complexity

Test Results: S3

Studies master in computer science (ST track)
Had sufficient programming experience, not specifically java but was well capable.

Introduction

No notes

Challenge 1

Task 1 - Run goat game and draw the state space

He drew the state space correctly with directional arrows in both directions.
At the second step he started drawing the initial state again but realized that it was simply a transition back instead.
He was able to deduce that the algorithm was BFS.

Task 2 - Change the algorithm to DFS

He changed the lettering on the fifo function call and had then immediately implemented the required change.
I asked him to verify the change by running the simulation again and he ran it again. When running again the discovery of both states at the main branch in the state space threw him off for a second but then realized that the exploration only actually continued on one path.

Task 3 - Implement early goal termination

When looking through the standard behaviours he found the termination behaviour and was also able to implement the lambda fairly easily.

He verified that the solution worked, he figured out that manual termination meant that the exploration was over.

Challenge 2

Task 1 - Determine the exploration method

During the explanation of the interface he already noted that the exploration occurs as BFS.

Task 2 - Terminate the exploration when reaching the goal state

He applied the same behaviour again and was able to have the query terminate at 2219 steps.

Task 3 - Optimize the query with an ordered frontier

The purpose of TreeMapFrontier was not clear at first, he noted that the naming scheme only clarified the implementation details of the frontier and not its actual behaviour.
The private constructor of the TreeMapFrontier confused the subject, he eventually found one of the static factory methods.
He was able to implement a comparator easily enough.
His solution also suffered the equivalence-bug, once I explained why this was the case he was able to implement a solution to this.

Task 4 - Freely optimize the query further

He wanted to only let the actual solution trace be part of the frontier.
Because the maze was the same every time the program was run he wanted to try and only let the true solution enter the frontier.

- He identified the FrontierFilterBehaviour as a behaviour to do this in.
- Unclear how Query::addBehaviour and ExplorationBehaviour::attach() cooperate, the AbstractBehaviour was also not clear in purpose and utility.

I explained that the TerminateOnGoalBehaviour had an event that you could use to retrieve the solution of the maze after a run completed.

Feedback: Would like to be able to extend a generic behaviour for which you can override some of the methods. He got this idea because he wanted to do the same as the LogEventsBehaviour has with a bunch of methods for the events.

The event system was not clear when and how to use it, the terminology was not understood and getting the info needed about events was not clear to the subject from the documentation alone.

Challenge 3

Skipped

Challenge 4

He implemented

Task 3 - Log number of evaluations

No idea how to create their own behaviour.
Was able to use the events of the query directly to implement the counter.

Solution 1: Count the number of states in the heap
He wanted to get the heap from the query with Query::getHeap but was disappointed he only got a Heap instance back which did not implement the functionality he desired.
He did not figure out that casting it to ManagedHeap would let him perform the solution anyway.

Solution 2: Increment a counter
He used a tappable event to increment a global counter and after exploration he prints the counter value.

Task 4 - Avoid broken steps

From the previous challenge he wanted to use the FrontierFilterBehaviour again and implement a predicate that identified the broken steps based on a list.

(task interrupted as we ran out of time)

Afterwards

After wrapping up the tests he noted that he saw a potential to use the framework to easily manage a larger number of sets of tests and implementations, that also switching out queries was pretty easy for a given problem.

This inspired me to later ask more participants about this.

Test Results: S4

A Computer science master student
Programming experience: java expertise

Introduction

We went very quickly through the introduction.
The explanation about the pipeline was a lot of new information, perhaps not all details were fully absorbed.

At this point I started to explain explicitly the functions of frontier and heaps more clearly, as in previous explanations their purpose may have been left blank. This means that now the subject also knew about the next-selection responsibility the frontier has.

Challenge 1

Task 1 - Run goat game and draw the state space

He only drew transitions for discoveries, not for undo-transitions.
He established that the algorithm was BFS.
He did not draw the full state space.

Task 2 - Change the algorithm to DFS

He looked through the documentation of functions and classes
He noted that he would have liked to have had the documentation separate from the source-code.
He identified that the exploration order should change and found the lifo function on the QueueFrontier without difficulty.
He verified that the frontier had the correct behaviour.

Task 3 - Implement early goal termination

He looked through the list of default behaviours and identified the TerminateOnGoalBehaviour as a suitable candidate to implement this.
He copied the example of applying the LogEventsBehaviour and adapted it to the termination behaviour.
Implementing the prediate was easy due to the autocompletion that intelliJ provided.

Summary

He found the documentation sufficient to figure out what to do

Challenge 2

Task 1 - Determine the exploration method

BFS was obvious to the subject, he noticed it already during the introduction of the UI.

Task 2 - Terminate the exploration when reaching the goal state

He implemented the termination behaviour again.
Exploration terminates after 2219 steps.

Task 3 - Optimize the query with an ordered frontier

The TreeMapFrontier was found but the documentation made it difficult to use.
I had to explain the usage of the api here for the subject to be able to use it.

After using the new frontier the termination occurs after 669 steps.

Task 4 - Freely optimize the query further

Idea: add direction to the position class to track where you came from.
The subject thinks of the states as mutable entities, perhaps it should be clearer that they are supposed to be immutable.

Challenge 3

While it was thought to be broken we did perform the challenge.
After pointing out that he had to use the SweepLineBehaviour to implement the required changes at task 3 the subject was able to use the documentation on how to implement the required pieces and components to make the algorithm function correctly.

Challenge 4

For the state representation the subject decided to make the class hold two integer variables: one for the current position of the robot and one for the height of the staircase. What was not clear during the programming was that the subject had to create a hashcode and equals implementation also, however intelliJ is able to generate them based on the class fields so it was not much of a problem.

The subject copied the static method to build a query object from the previous examples and adjusted the generic types to his needs.

For the next function, he created one using the NextFunction.wrap() utility together with a lambda.
He implemented two methods, climb and descend which each created a new state with the proper modifications.
He first wanted to do bounds checking himself but after a short while decided to use a filter on the output stream instead. He used the same technique that the other challenges used where a filter removes all illegal states from the next function output.

Task ? - Avoid broken steps

He implemented this quickly as the time was almost over, but managed to do it in time by using the FrontierFilterBehaviour.

Afterwards

I asked him what he thought of the experience afterwards and he gave the following feedback:

Found the filter feature very useful
Also sees that variations on queries are easy to manage.

Test Results: S5

Technical Computer Science student
Programming experience: Mostly with python, no java expertise

Introduction

I had explained the project to Joanne prior, so she may have been more comfortable with the materials.

Challenge 1

Task 1 - Run goat game and draw the state space

She drew the state space without any flaws
She drew them with directional arrows
She started drawing the initial state a second time before realizing that it was a backlink.
She deduced the exploration was BFS.

Task 2 - Change the algorithm to DFS

What do you want to do first?

- “Make the frontier order different”
- She inspected the auto complete of the QueueFrontier and found the lifoFrontier method
- She was looking for “filo” and lifo was identified as a suitable candidate

Task 3 - Implement early goal termination

I hinted that there are standard behaviours.
She looked through them and found the right behaviour from the names alone.
She saw from the constructor that she needed a predicate.
She did not know what a predicate was, I had to explain this and how it interacts with java lambda expressions.
Unclear what the predicate needed or how to implement it.
I helped her implement the lambda expression, the difficulties may have been lack of knowledge on how java lambda’s function. Once she understood that a state was input it was pretty clear.
Understood the manual termination signal in the simulation.

Summary

When asked how she would have compared the implementation to writing the entire algorithm herself she noted that she prefers using lifo-fifo frontiers over implementing DFS

with recursive implementation, but the comparison was not very realistic since she did not know how to implement the algorithm herself in the first place.

Challenge 2

Task 1 - Determine the exploration method

Identified BFS by playing with the interface

Task 2 - Terminate the exploration when reaching the goal state

Started adding the behaviour, was able to figure out how to use the Maze state api, being able to apply this displays an understanding of the effect the predicate has on execution of the algorithm.
She tested the solution with the simulator and verified it worked.

Task 3 - Optimize the query with an ordered frontier

Looking through the frontiers, the TreeMapFrontier did not stick out as a suitable candidate.
She noticed OrderedFrontier and used the IDE to figure out what classes implement it.
She followed through the DynamicallyOrderedFrontier interface to TreeMapFrontier.
She wanted to instantiate it directly with the constructor. The constructor was difficult to use as she did not understand how to use it, the documentation was lacking for the TreeMapFrontier class.

The comparator was relatively easy to build, some maze-specific details she needed help with to implement the comparator but otherwise the purpose was clear.
I preemptively fixed the withExactOrdering bug, as she could not have known of its existence and the implemented solution ended up working.

Challenge 3

Task 1 - Add the sweep-line behaviour
This task was difficult to complete, and needed hints to use the Helpers functions.

She noted that this style of building algorithms is like “drag-and-drop” programming.
“This is useful for problems where using different algorithms is key.”
“It is easy to try different approaches on a problem.”

Challenge 4

Build the query and execute it

When building the main function, she wanted to pass the initial states to the explore method and did not know what to try otherwise.
When met with the empty console termination, she thought that the next-function was wrong.

It was not clear that an initial state had to be provided.

(although up until that point that was not required anyway)

Count the number of states

Solution: To count the states she wants the heap to count the number of states.
When counting for a staircase of 1000, then 997 states were in the heap.

Afterwards

- Java heavy
- Educational purpose is good(/clear) and also very generalized
- The framework is very generalized and not geared to any specific exploration algorithm
- She thinks that once you understand the framework it can save time.

Analysis - How did things go

Changing the frontier from BFS to DFS

In C1T2 the subjects needed to change the algorithm from BFS to DFS and received hints that led them to conclude they needed a different frontier.
What led them to find the correct frontier?

S1

He had a lot of prior experience with java programming and was familiar with the IDE (intellij) this led him to explore the framework as soon as possible, looking at the class hierarchy and interface implementations. Once he knew what he was looking for he managed to find the method that creates the correct frontier in the source-code of the QueueFrontier class.

S2

He received the hint that the frontier order would need to be different and checked for alternate methods on the QueueFrontier class and found the method quickly.

S3

He figured out that the exploration order would have to be “filo” and used the autocomplete function to list the alternate methods on the QueueFrontier.

S4

He looked into the source code of the QueueFrontier and found the method defined there

S5

She received the hint that the frontier would need to be ordered differently, from there she looked into alternate methods on the QueueFrontier class using auto completion and found the method there.

Subject	Looked at source-code	Used javadoc	Main method
S1	Yes	Yes	Source code
S2	No	No	Autocompletion
S3	No	No	Autocompletion
S4	Yes	No	Source code
S5	No	No	Autocompletion

This leads to believe that for java-experts the framework is fairly self-explanatory and from the initial example they are able to make a basic modification.

Implement on goal early termination

In each challenge one of the first tasks was to change the query to make it terminate the exploration when a goal state would be found. Here we analyse how that was achieved

S1

They received no guidance and made several attempts before achieving a solution. They first focussed on the next-function, perhaps in an understanding it was the most controllable source of data. They did look through the list of default behaviours before this attempt but did not register the default termination behaviour in the list. Their successful attempt consisted of clearing the frontier using the afterStateEvaluation event. They needed guidance on how to implement an abstract-behaviour.

S2

When looking through the list of default behaviours they did not register the suitable default behaviour. He only saw the list of filenames in the folder view of the source code, he had not taken the time to inspect the documentation thoroughly. When tasked with using the behaviour system or event system he could not figure out how to use them.

S3, S4 and S5

They all had a similar experience in applying the termination behaviour: They found the suitable default behaviour by the class name in the project explorer. They had varying

difficulty using the predicate, mostly due to it being a relatively new feature they had not used before. But the purpose was clear from context and/or the documentation.

Observations & conclusions

The experience of S1 and S2 points out that for newcomers the concepts of the events, tappables and behaviours is left unexplained and that the documentation of the functions which expose and consume them are not sufficient by themselves. A possible solution to this is to provide small guides/tutorials on these systems which explain their usages, how to implement them and how to find/recognize them in the code base.

The experiences of S3, S4 and S5 have shown that for the TerminateOnGoalBehaviour is relatively easy to use without much help.

A specific experience S3 had with this behaviour was in Challenge 2, Task 4 where he received the advice that the behaviour exposed events that signal when a goal state is found. By that description alone he could not figure out how to use the behaviour in such a way. The major obstacles there were again lacking knowledge in the events system. So one of the guides may have to be how to use events exposed by non-query systems (such as another behaviour).

Using a ordered frontier

In Challenge 2, Task 3 the subjects were asked to change the frontier to one which is sorted by a user-specified metric.

S1

He had difficulty finding the correct frontier, he used the project explorer to find out what code resides in the “abeona.frontiers” folder but did not think (at first) that TreeMapFrontier would be the required class for the solution. After some exploration of the class/interface hierarchy of the Frontier interface it became clear for him that he had to use that class.

S2

When applying the TreeMapFrontier he looked at the source code and documentation to figure out what it exposed. The first view he had was of the top of the class and this included the constructor. As such he immediately attempted to call it, even though it was private. This sequence of events also occurred with several other subjects.

S3

Similarly to S1 there was some difficulty spotting TreeMapFrontier as the correct solution, he specifically remarked that the naming of this class was more a hint towards the implementation details than it was a descriptive name of its purpose. First attempts to use it focussed on the constructor instead of the factory methods.

S4

Once the TreeMapFrontier was found the usage was difficult to deduce because of lacking documentation.

S5

She did not notice the TreeMapFrontier by name as a suitable solution but found it ultimately by checking the class hierarchy for the Frontier interface. The implementation details causing problems for comparators which report states as equivalent caused the subject to think the frontier was wrong.

Observations & conclusions

Based on S1, S3 and S5 we can see that the name of the TreeMapFrontier is not very eye-catching and threw off the subjects early on. The OrderedFrontier was more often the focus of the subjects, and usually lead to them back to the TreeMapFrontier. Perhaps a naming scheme where TreeMapFrontier contains a word closely related to “ordered” or “sorted” would be better suited.

Besides that, the usage of a Comparator was also not entirely clear to users immediately. The name comparator is not java-specific itself but it application in java and the framework is. The documentation for the TreeMapFrontier did not adequately explain what was required of the comparator and only java-experts knew to use the Comparator factory functions. Those factory functions were kept in mind when designing the TreeMapFrontier API and as has become clear the documentation may need to hint towards those functions to help new users implement comparators easily.

The TreeMapFrontier instantiation was confusing for several subjects, when they first viewed the class they saw the documentation and source code of the top of the class. Since this included the constructor they immediately tried to call the constructor. The documentation of the class may need to specifically mention that it can only be constructed with one of the two factory methods.

Finally, none of the subjects had any information about the comparator edge-case/bug where some states would be considered equivalent based on the equivalence. Perhaps the framework should implement a workaround so they are never exposed to it. Alternatively, the subjects each picked the first occurring factory method they saw from the source code view, simply putting the factory method which implements the workaround first could also nip this problem in the bud.

Executing their own query

Not every subject worked on challenge 4, where they had to build their own state representation as well as use the query object themselves.

S1

He looked through the Query class to find the methods available and noticed the Query::explore method. I had to remind him that the frontier needs an initial state when on the first run the query terminated immediately.

When trying to optimize the query to use a greedy algorithm he tried implementing his own frontier. Because he did not do Challenge 2 the existence of the TreeMapFrontier was not known to him. From the documentation alone he was able to figure out how to implement the Frontier interface. The fact that there were other more specialized interfaces of Frontier (e.g. ManagedFrontier) was lost on him, perhaps the documentation on the Frontier interface could further clarify this, however he did not carefully read the documentation in the first place so perhaps exploring sub-interfaces of Frontier might come naturally.

S2

Did not perform this challenge

S3

He copied the structure of a “createQuery” method from the previous examples to built the query in a similar fashion as in prior challenges. When it came to executing the query he was confused why it terminated immediately, he expected the framework to use some initial state, though he did not specify one. He used the source code of one of the examples to figure out how the query object built by him was being used. There he discovered that the initial state has to be inserted into the frontier before exploration.

S4

He first wrote his state representation and then built the query code.
His state representation contained methods to create alternate versions of the current state with some mutation applied, such as moving forward or backward.
When building the query code he also copied the createQuery method from a previous example and adjusted it to his needs. The next-function was made with the NextFunction.wrap helper. When explaining the next-function interface he wondered what the wrap function was for and if he needed to implement it. The signature of the function was too confusing (due to the large generics declarations) for him to figure out how the wrap function relates to the next-function immediately.

S5

She did not have enough java experience to complete this task on her own, I had to assist in writing the code in a timely manner. I also hinted at reusing code from previous exercises to speed up building the query. Her implementation of the state representation used a in-class next-function which was something the other subjects had not done. She decided to do this after I explained how the next-function works and could be implemented with either a lamda or on a class directly.

In her approach she used if-statements in the next-function to determine whether some actions are possible or not.

Observations & conclusions

On every first run of the query the subjects did not specify an initial state. This is because none of the queries prior were built with an initial state specified. They all assumed that the framework somehow built an instance automatically. S5 specifically asked whether the Query::explore method could take a state as input and this might be an easy to implement solution to provide a simple API that solves this. Additionally, an error-signal or special termination type may be created for starting queries without an initial state.

A small observation I made is that in the framework interfaces used for the majority of laying the foundation of the class hierarchy, however this was hardly ever noticed or obvious to the users. Perhaps the reliance on interfaces and composition is not quite clear enough on first use of the framework and it might need a specific mention in one of the first introductions/guides for the framework. Understanding that most features and components are found in an interface and may have multiple sub-interfaces which further extend it would encourage more thorough looks at the api from users when implementing their own components.

In the implementations of the subjects, their states sometimes included both state and problem level details in the state representation. A generic model for encoding states, problem properties and state mutations might help make the creation of state representations more modular also. In the current case often the state representation and next-function held all details of the problem context and possible mutations of the state. For the problem context this means that it is difficult to extend or modify the problem context if the code defining it is not under your control. Also for possible mutations a friendlier interface that allows adding mutations later on would fit well with the modular design of the rest of the framework. Perhaps these could even be behaviours in some way. These last suggestions are more dependent on state representation of the user and was out of scope for the project but might be good future work to extend the framework.

Influence of expertise

For programmers with java expertise (not to be confused with programming expertise), the framework was noticeably easier to use than for those without java-specific knowledge. The usage of modern java features (e.g. lamba’s and functional interfaces) has led to a higher barrier-to-entry because the users need to understand how these features work. Also, to make maximum use of for example the comparator interface the usage of helper methods such as Comparator.comparing() is very useful to easily create comparators, however this may be somewhat hidden for those that do not know about it.

The java experts also had a much easier time exploring the structure and hierarchy of the framework.

Regardless of expertise, none of the subjects were able to quickly understand the events system. This was different from the behaviour system and I think this is because the behaviour system was introduced to them through sample code in the challenges, each challenge's query included the LogEventsBehaviour. There was no code demonstrating the tappable events system to them.

What was also noticeable was that many subjects preferred to modify the query's behaviour by modifying the next-function. The concept and advantage of behaviours is something that needs to be more front-and-center in the introduction of the framework.

Conclusions

The framework in its current state is sufficiently usable for those with experience with java programming. For those without java programming the framework is usable but the lack of java specific knowledge does put up a barrier and makes the framework API more difficult to use due to its minimalist design.

For new users the sample code provides good examples of how to build queries but lacks in demonstrating the advanced use cases of behaviours and completely lacks to highlight the usage of the tappable events system. Once explained these systems are not incredibly complex to use but understanding them is not easily done through the documentation alone.

Creation of custom behaviours also has a bit of a barrier to entry, since there is no simple base to quickly start with. The usage of AbstractBehaviour is too complex and difficult to understand from the current documentation.

The used naming scheme for classes is good and logical for the users but in cases where the name reflects the implementation details rather than the purpose of the class/interface this falls short.

The documentation is sufficient all round, but the TreeMapFrontier documentation lacks severely. The TreeMapFrontier implementation has hidden requirements when instantiating it.

The API is very modular and generalized for exploration algorithms, its usage case in educational purposes and management of sets of exploration methods is clear.

Suggestions

- Create a guides/tutorials on how to use the framework:
    - Creating your own state class
    - Creating a next-function
    - Creating and using a query
    - Using query events

    - Using query behaviours
    - Creating custom behaviours
    - Using behaviour events
- Create a generic abstract behaviour which hooks all events and exposes these as overridable methods
- Rename the TreeMapFrontier to make it more obvious for its use-case
- Improve the TreeMapFrontier documentation
- Provide better out-of-the-box workaround for the TreeMap Comparator equivalence edge-case
- Make it clearer that the TreeMapFrontier needs to be instantiated through the factory methods instead the constructor
- Allow the Query::explore API to ingest initial states
- Warn the user of starting explorations without initial states
- Create a modular system for defining/composing next-functions
- Improve documentation on what interfaces and implementations exist for the base interfaces of components (frontier, heap, behaviour)