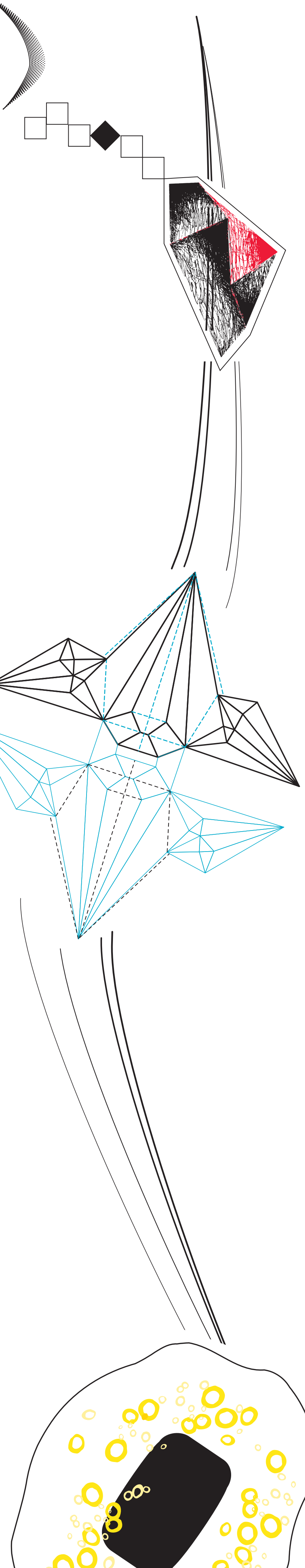
October 15, 2019



RESEARCH TOPICS

GENERAL FRAMEWORK FOR BUILDING STATE SPACE EXPLORATION STRATEGIES IN GRAPH TRANSFORMATIONS

Bram Kamies (s1811045) <b.kamies@student.utwente.nl>

UNIVERSITY OF TWENTE.

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).

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

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

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

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

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

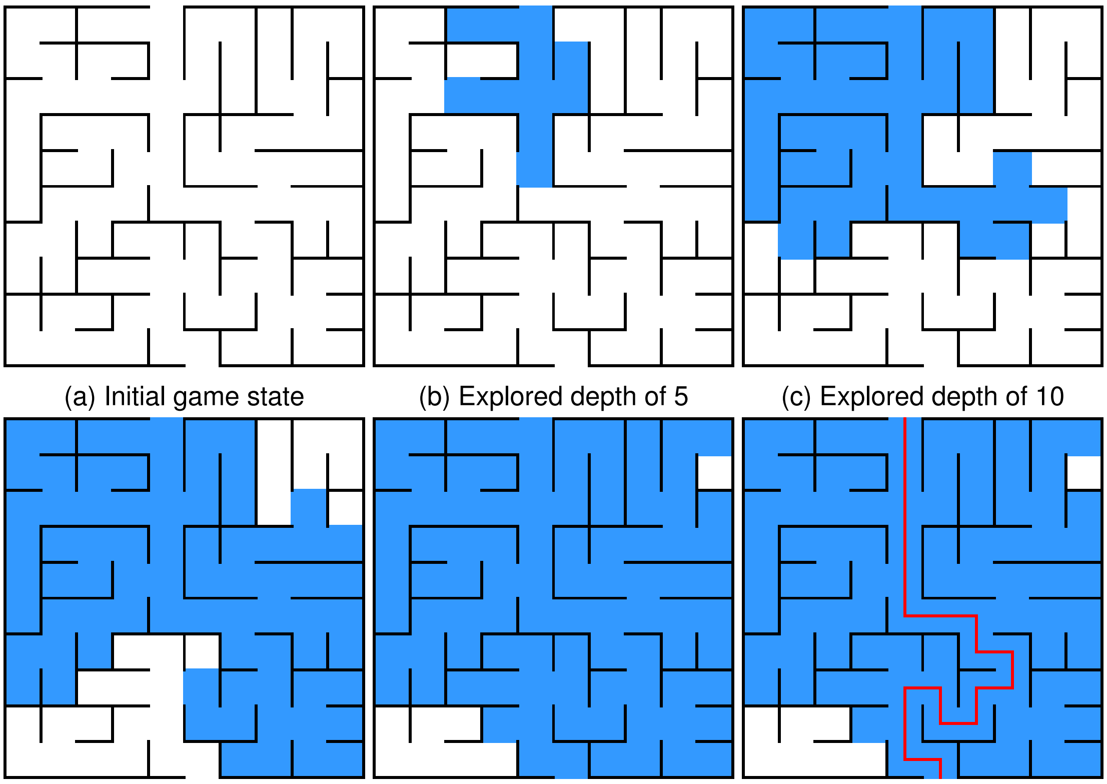
```
function bfs($S_{initial}$, $P_{goal}$)
    $S_{frontier} = S_{initial}$;
    $S_{known} = S_{initial}$;
    while $S_{frontier} \neq \emptyset$ do
        pick $s_c$ from the frontier, such that $s_c$ is the state which had been in the frontier
          longest;
        $S_{frontier} = S_{frontier} \setminus \{s_c\}$;
        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

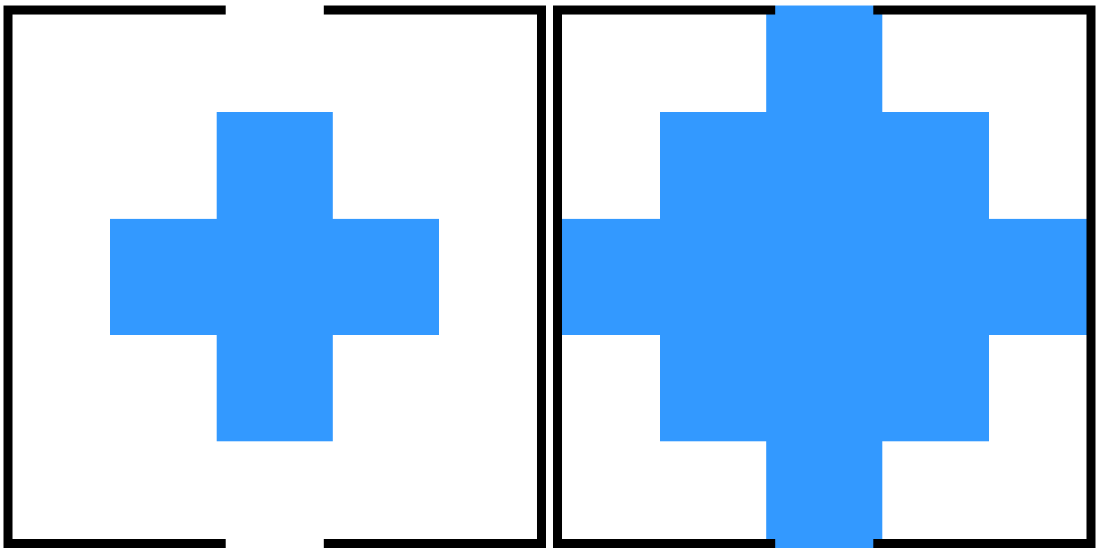

(a) Known states after exploring depth 0 (b) Known states after exploring depth 1

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 $S_{frontier} \neq \emptyset$ do
        pick $s_c$ from the frontier, such that $s_c$ is the state which had been in the frontier
          shortest;
        $S_{frontier} = S_{frontier} \setminus \{s_c\}$;
        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 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).

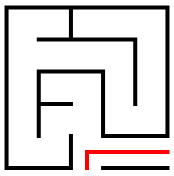


Figure 3.3: Maze with solution which challenges worst case complexity

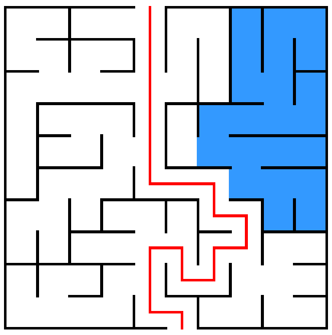


Figure 3.4: Maze with solution (red), area of interest highlighted in blue

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}).

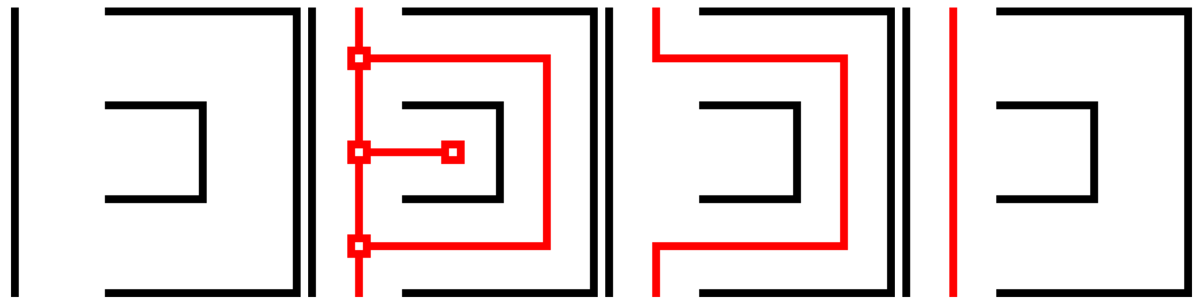
Looking at the pseudocode in alg. 3 shows similarities in structure to BFS (again). The ordered frontier is clear to see by the instruction of picking s_c according to the order of the frontier. The behaviour of evaluating a state is slightly different, beyond additional behaviour. Here the transitions 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

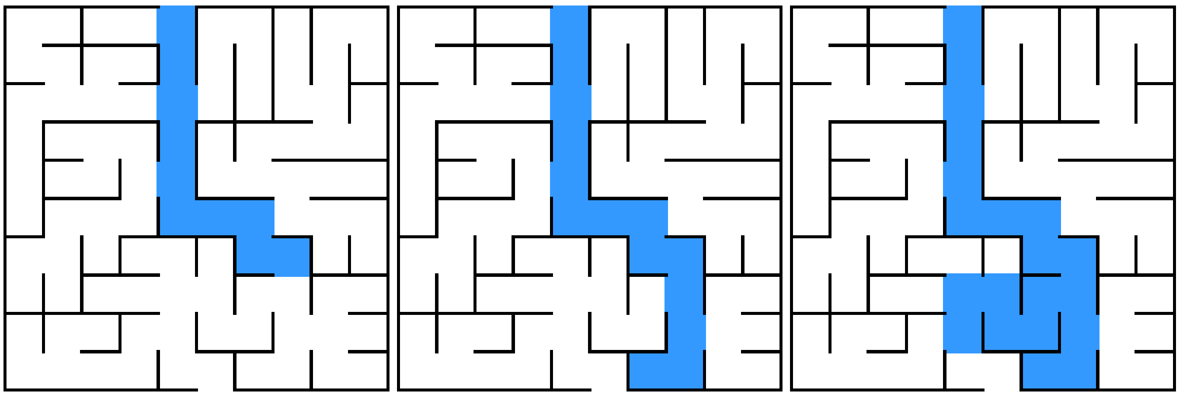

(a) After 10 state evaluations (b) After 14 state evaluations (c) After 20 state evaluations

Figure 3.6: Progression of A* exploration through a maze

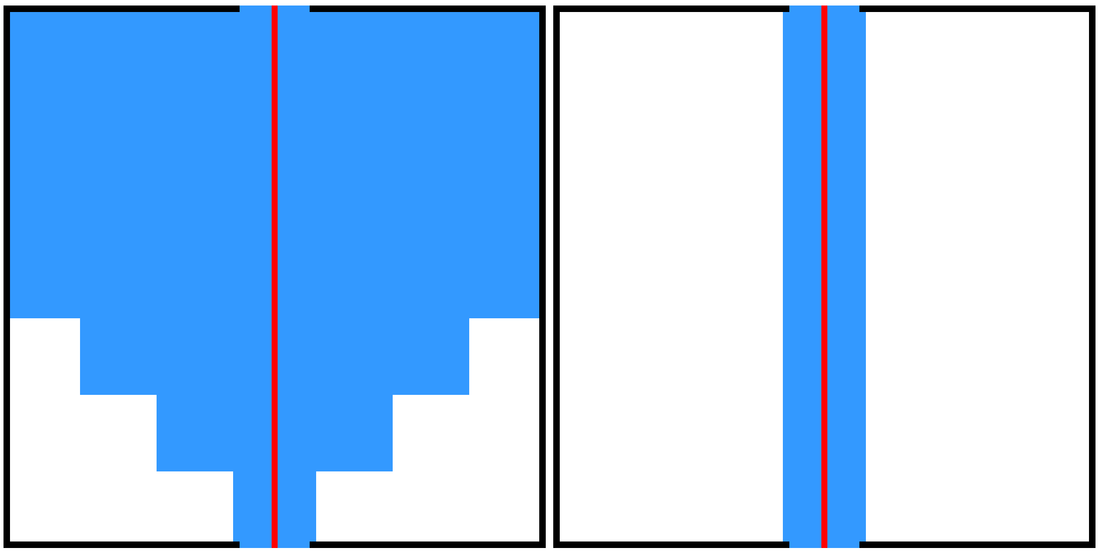

(a) Evaluated states for exploration with BFS (b) Evaluated states for exploration with A*

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

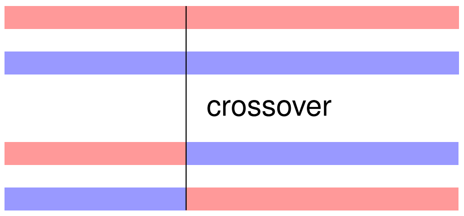


Figure 3.8: Chromosome crossover diagram

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 $S_{frontier} \neq \emptyset \wedge P_{cont}(i, c_{gen})$ do
        pick $s_c$ from the frontier, as the only state in the frontier set;
        $S_{frontier} = \emptyset$;
        $i = i + 1$;
        for $t_n \in T_{out}(s_c)$ 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 solution. 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.

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

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

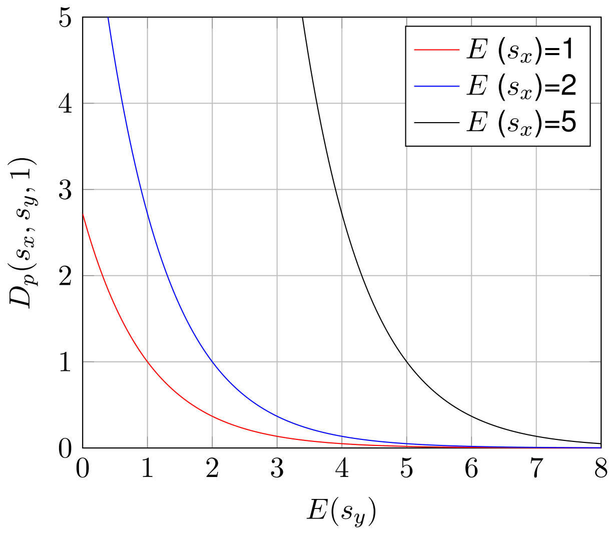


Figure 3.9: Graph of threshold values between various current states plotted for a next state candidate

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 $S_{frontier} \neq \emptyset$ do
        pick $s_c$ as any state from the frontier;
        $S_{frontier} = \emptyset$;
        $i = i + 1$;
        $S_{frontier} = \{\}$;
        if $P_{goal}(s_c) \vee S_{out}(s_c) = \emptyset$ then
            return $s_c$;
        end
        for $s_n \in S_{out}(s_c)$ 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 $S_{frontier} = \emptyset$ 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.