University of Basel

Implementing and Evaluating Successor Generators in the Fast Downward Planning System

Bachelor Thesis

Faculty of Science at the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence

Examiner: Prof. Dr. Malte Helmert
Supervisor: Silvan Sievers

Yannick Zutter
yannick.zutter@stud.unibas.ch
2015-052-723

September 30, 2020

Acknowledgements

I would like to thank Malte Helmert for giving me the opportunity to work in such an interesting field of computer science. Additionally, I would like to thank my supervisor Silvan Sievers for spending countless hours discussing details and helping with all the problems that have occured during this time. I also want to thank my wife Florence and my son Jarek for giving me the strength to stay cool even in the hottest phases of the thesis.

Abstract

Fast Downward is a classical planning system based on heuristic search. Its successor generator is an efficient and intelligent tool to process state spaces and generate their successor states. In this thesis we implement different successor generators in the Fast Downward planning system and compare them against each other. Apart from the given fast downward successor generator we implement four other successor generators: a naive successor generator, one based on the marking of delete relaxed heuristics, one based on the PSVN planning system and one based on watched literals as used in modern SAT solvers. These successor generators are tested in a variety of different planning benchmarks to see how well they compete against each other. We verified that there is a trade-off between precomputation and faster successor generation and showed that all of the implemented successor generators have a use case and it is advisable to switch to a successor generator that fits the style of the planning task.

Table of Contents

1 Introduction

Automated Planning and Acting as described by Ghallab et al. [6] is one of the oldest studies in artificial intelligence. This Planning is all about finding operator sequences to solve a given problem. We start with a given initial state and apply operators to generate its successor states. From these states we carry on finding new states until we finally reach one of the desired goal states. For example, we have the given problem of a simplified Sudoku puzzle in Figure 1.1. The initial state is an empty field without any numbers in it. Then we generate the successor state of the initial state by applying the following operator: we set the upper left (empty) field to 1.

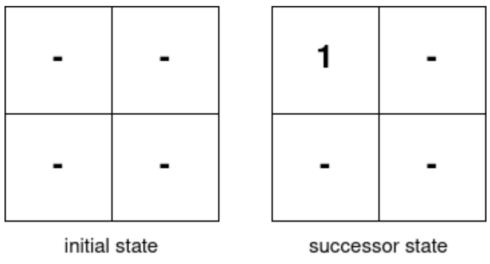

Figure 1.1: simplified version of an 2x2 Sudoku puzzle

But we can not just apply any operator we want. All applications of operators are limited by conditions. For example in Sudoku each number may only appear once in each row and column. Beforehand the whole problem needs to be modeled in a way that all these conditions are predefined. Each operator has preconditions that need to be fulfilled before it can be applied. The operator also has effects on the state when it is applied. In this example, the effect is to write a “1” in the upper left box. We finish our search if the current state satisfies the given goal conditions. In this example, this would be a state where all boxes are filled with numbers and each number only appears once in each row and each column. On our way to the goal state, we remember all operators we applied. In the end we can return a sequence of operators that leads from the initial state to the goal and shows how the problem is solved.

Planning standardizes given problems, so we can not only solve a small set of specific problems but all problems that can be described in the given standardization. Planners that solve problems in this way are called general planners, based on the General Problem solver of Newell et al. [12]. In order to efficiently solve planning problems, a planner needs to have an efficient successor generator. A successor generator is a function which generates all the possible successors of a given state. One problem solver with a unique successor generator, which has already been proven very successful, is the Fast Downward planning system introduced by Helmert [8]. Its successor generator generates a tree with a variable as node and a vertex for each value in the domain of this variable. While searching for operators to apply we can traverse this tree with the given state space and collect all operators that may be applied. This is only one possible method to generate successor states. In this thesis we want to implement four other ideas for successor generation and evaluate them against each other. These successor generators are the Fast Downward successor generator, a naive generator, a generator based on the idea of delete relaxation [1], one generator based on the PSVN planning system introduced by Holte et al. [10] and one on the idea of Watched Literals from SAT solving described by Moskewicz et al. [11]. We start by introducing the theory that is needed to understand the concepts of the successor generators and shtplanning in general and talk about the implementation of the successor generators. Followed by an evaluation of our experiments, we compare the successor generators to each other. Lastly we give a conclusion of what we accomplished and discuss some potential starting points for further work.

2 Background

In planning, a planning task is needed to define the properties of a given problem. Starting from the initial state we want to expand new states with the given operators until we finally satisfy a goal condition. Each operator has preconditions that need to be satisfied and effects that transform the given state. Each operator has a cost, which indicates how expensive it is to apply this operation. By traversing the state space we generate a sequence of operators. This sequence of operators is also called a plan. A plan is created if we reach a goal, to show how we get from the initial state to a goal state. We can differentiate between optimal and satisficing planning. In optimal planning we guarantee that the plan we found is an optimal plan, which means that it has minimal cost. Satisficing planning means that the plan satisfies our needs, which means that the cost is beyond a certain threshold or that a solution has been found.
To define planning tasks, we use a Finite Domain Representation (FDR) as defined by Helmert [9].

Definition 2.0.1 (State Variable)**.** *A State variable in FDR is a variable with a non-empty finite domain D_v.*

If we support the use of axioms there would be additional state variables so called derived variables, which are computed by axioms. Because we do not use axioms within our successor generators we will not further define them.

Definition 2.0.2 (Fact)**.** *a fact f is tuple $\langle v, d\rangle$ where v is a variable from the set of state variables $v \in V$ and d a value from the variable's domain $d \in D_v$.*

Definition 2.0.3 (Partial State)**.** *A partial state is a partial variable assignment over V which is a function s over a subset of V so that $s(v) \in D_v$. A partial state is called an extended state if it is defined for all variables $v \in V$. A partial state is called a reduced state if it is defined for all variables in V.*

Definition 2.0.4 (FDR Planning Task). *A Finite Domain Representation is given by a 4-tuple* $\Pi = \langle V, s_0, s_*, O \rangle$ *with following components:*

- V *a set of state variables.*
- $s_0 \subseteq V$ *the initial state*
- s_* *a partial state over* V *as the goal*
- O *is a finite set of operators over* V *with:*
  - *pre(o) the preconditions of o as a set of facts*
  - *eff(o) the effect of o consisting of:*
    - *cond the effect condition as a partial variable assignment*
    - v *the affected variable*
    - $d \in D_V$ *the new value for* v
  - *cost(o) the cost of the operator*

Definition 2.0.5 (Applicable Operator). *An operator is applicable in state s, if all preconditions pre(o) are true in s. This means for all facts in* $pre(o)\langle v, d\rangle$ *holds that* $d = s(v)$.

Definition 2.0.6 (Successor State). *A state s' is a successor state of s, if there is an applicable operator o so that o applied to s leads to state s' (notated as* $s \xrightarrow{o} s'$*). For each fact in* s' *holds that* $s'(v) = d$ *for all effects in o,* $v := d \in eff$ *and* $s'(v) = s(v)$ *for all other variables.*

Definition 2.0.7 (FDR State Space). *An FDR State Space s is a set of facts where* $\langle v, d\rangle$ *for* $v \in V$ *and* $d \in D_v$.
The state space of an FDR $\Pi = \langle V, s_0, s_*, O \rangle$*, denoted as* $S(\Pi)$*, is a directed graph. Its vertex set is the set of states of* V*, and it contains an arc* (s, s') *if there exists some operator* $\langle pre, eff\rangle \in O$ *such that* $s \xrightarrow{o} s'$.

Definition 2.0.8 (Plan). *A plan P of a planning task* Π *is a solution in form of a sequence of operators* $\langle o_1, \ldots, o_n\rangle$ *where the operators lead from the initial state* s_0 *to a goal state* $s_n \in s_*$.

Definition 2.0.9 (Cost Of A Plan). *The cost c of a plan is the sum of all costs in the given plan.* $c = \sum_{k=0}^{n} cost(o_k)$ *where* $o_k \in P$.

3 The Successor Generators

A successor generator is a module inside the planning system to determine the successor states from a given state s and the given operators. In the Fast Downward planning system, a successor generator does not calculate the successor states, but only returns a list of all applicable operators. This is due to the fact that generating the applicable operators is the hard part of the successor generation and getting to a successor state from an operator is trivial. So while describing the different successor generators we do not consider the successor states, but only the applicable operators which are generated and lead to these successor states.
We implemented the successor generators described in this chapter, apart from the already existing Fast Downward successor generator in the Fast Downward planning system. The implementation of these successor generators can be found on this GitHub repository[1].

3.1 Naive Successor Generator

As defined above, an operator is only applicable if all of its preconditions are satisfied. The easiest method to generate the applicable operators in a successor generator is to check for each operator if all given preconditions are satisfied under the given state. If yes, the operator is applicable. If no, the operator is not applicable. This method is shown in Figure 3.1.

3.2 Fast Downward Successor Generator

Fast Downward is a classical planning system based on heuristic search introduced by Helmert [8]. It uses any PDDL planning task as input and converts it to a Finite Domain Representation. The successor generator is a tree consisting of two different nodes: selector nodes and generator nodes. A selector node has a variable $v \in V$ called the selection variable and $|D_V + 1|$ outgoing

[1] https://github.com/YannickZutter/fast-downward

Algorithm 1: generate_applicable_operators

```
Input: State s
for each Operator o do
    for each Precondition p in o do
        if $p.val \neq s[p.var].val$ then
            break
    report o as applicable
```

Figure 3.1: Naive generate applicable operators

edges. These edges have a label d for each of the values $d \in D_v$ and one additional edge labeled $\top$, the "don't care" edge.
A generator node is a leaf node which stores a set of applicable operators O at this leaf. Each operator $o \in O$ must occur in exactly one generator node, and the set of edge labels leading from the root to this node (excluding don't care edges) must equal the preconditions of o.

We build the tree by first choosing a variable order $v_1 \prec v_2 \cdots \prec v_n$. Then following this order, we set the selection variable of the root node as v_1, then we classify the set of operators according to all operators having a precondition on v_1. We then split these operators according to the children, e.g. operators with a precondition $v_1 = d$ will be represented in the child node associated with d. Operators without a precondition on v_1 follow the don't care edge. We then carry on with the child nodes and variable v_2 in the same way as in the root node, with one exception. To avoid the unnecessary creation of selection nodes, nodes where no operator in a certain branch has a condition on v_i, then v_i is not considered as a selection variable in this branch. The construction stops if all variables have been considered and we create a generator node as a leaf node with all the associated operators.
For a given tree and a given state, we can compute the applicable ops by traversing tree as follows:

- At a selector node with variable v, follow the edge $v = s(v)$ and the don't care edge.
- At a generator node, report the generated operators as applicable.

As mentioned above, this is the default successor generator as described by Helmert [8] and is already implemented.

3.3 Marking Successor Generator

The Marking successor generator is based on the idea of delete relaxation and the relaxation heuristics as introduced by Bonet and Geffner [1].

3.3.1 Delete Relaxation

If we consider a special case of our Finite Domain Representation where we only have binary variables ($D = \{True, False\}$), we get a propositional planning task instead of a FDR planning task. In this propositional planning task, operators can be written with add and delete effects. An operator o consists of the triple $\langle$*pre(o), add(o), del(o)*$\rangle$ where $pre(o)$ are the preconditions for o, $add(o)$ the set of add effects and $del(o)$ the set of delete effects for o. All of these three sets are represented as a set of variables. $pre(o)$ is the set of preconditions that shows which variables need to be True in order for the operator to be applicable. $add(o)$ is the set which represents which variables are set to True. $del(o)$ is the set which represents which variables are set to False. The delete relaxation o^+ of a propositional operator o is the operator with $pre(o^+) = pre(o)$, $add(o^+) = add(o)$, $cost(o^+) = cost(o)$ and $del(o^+) = \varnothing$. A planning task without any delete effects is called a delete-free planning task or a relaxed planning task. We can use the solution of a relaxed planning task as a heuristic for solution costs for the original planning task or as we will see to create a successor generator which uses a modified idea of the computation of the delete relaxation and its heuristics.

3.3.2 Relaxed Planning Graph

A relaxed planning graph represents which variables in Π^+ can be reached and how. We build a graph with variable layers V^i and action layers A^i.

- variable layer V^0 contains the variable vertex v^0 for all $v \in I$.
- action layer A^{i+1} contains the action vertex a^{i+1} for action a if V^i contains the vertex v^i for all $v \in pre(a)$.
- variable layer V^{i+1} contains the variable vertex v^{i+1} if previous variable layer contains v^i, or previous action layer contains a^{i+1} with $v \in add(a)$.
- goal vertices G^i if $v^i \in V^i$ for all $v \in G$.
- graph can be constructed for arbitrary many layers but stabilizes after a bounded number of layers ($V^{i+1} = V^i$ and $A^{i+1} = A^i$).
- directed edges:
  - from v^i to a^{i+1} if $v \in pre(a)$ (precondition edges)
  - from a^i to v^i if $v \in add(a)$ (effect edges)
  - from v^i to G^i if $v \in G$ (goal edges)
  - from v^i to v^{i+1} (no-op edges)

Using the notation introduced in the "Foundations of Artificial Intelligence" lecture held by Prof. Dr. Helmert and Dr. Keller at the University of Basel we depicted a relaxed planning graph for

following propositional planning task.
$\Pi = \langle V, s_0, s_*, O \rangle$ with $V = \{a, b, c, d, e, f\}, s_0 = \{a\}, s_* = \{g, f\}$ and $O = \{a_1, a_2, a_3, a_4, a_5\}$ with $cost = \{a_1 \to 2, a_2 \to 4, a_3 \to 1, a_4 \to 1, a_5 \to 5\}$ and

$$\begin{array}{ccc} pre(a_1) = \{a\} & add(a_1) = \{b, c\} & del(a_1) = \{\} \\ pre(a_2) = \{a\} & add(a_2) = \{c, d\} & del(a_2) = \{\} \\ pre(a_3) = \{b, c\} & add(a_3) = \{d\} & del(a_3) = \{c\} \\ pre(a_4) = \{c\} & add(a_4) = \{e\} & del(a_4) = \{c\} \\ pre(a_5) = \{e\} & add(a_5) = \{f\} & del(a_5) = \{a, b\} \end{array}$$

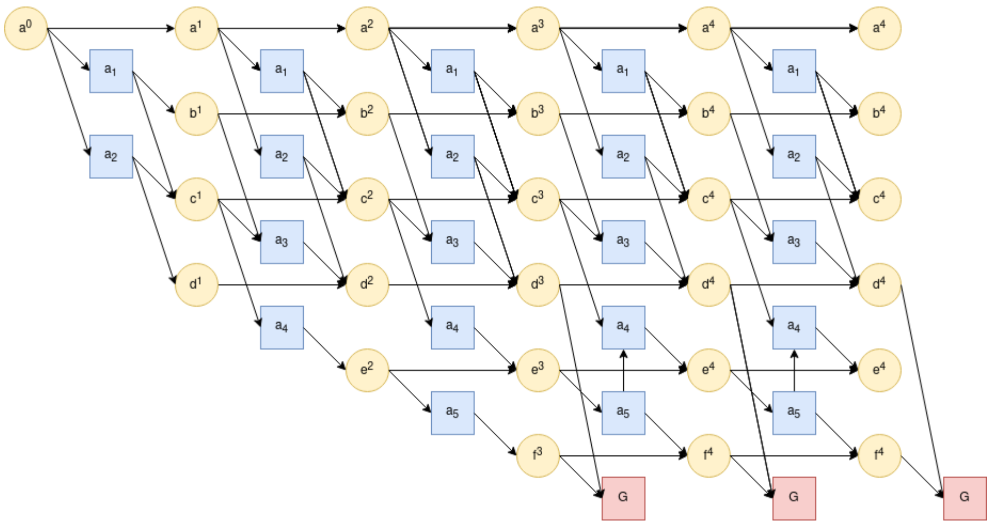

Figure 3.2: relaxed planning graph

3.3.3 Converting To the Successor Generator

To generate our successors we do not need to calculate the whole relaxed graph. Because we only need the next step and not the solution of the problem, it suffices if we only expand one layer of the graph. For this we copy the idea of the delete relaxation heuristic implementations from Geißler et al. [5]. For each operator we count the number of preconditions and store them as a precondition counter. Then we check all the variables in the layer before. Each time a precondition is satisfied, we decrease the precondition counter by one. If one of the counters reaches zero, all the preconditions for the corresponding operator are satisfied and therefore the operator is applicable.

3.3.4 Implementation Notes

To make the successor generator more efficient we do not want to reset the counter list in each function call. Instead we only want to reset all the entries we actually want to look up. For this we have two different ideas.

The initialize method looks similar for both algorithms and is shown in Figure 3.3.

```
Algorithm 2: Initialize
list counter = new list(size_of(operators))
int var_size = size_of(all variable domains)
preconditions_of = new list(new list(var_size))
list first_visit = new boolean list(size_of(operators)) (for boolean algorithm)
int current_timestamp = 0 (for timestamps algorithm)
list timestamps = new list(size_of(operators)) (for timestamps algorithm)
for each Operator o do
    counter[o] = size_of(o.preconditions)
    for each Fact f in o.preconditions do
        preconditions_of[p].push_back(o)
    end
end
```

Figure 3.3: Initialize Algorithm for the marking successor generator

To generate the applicable ops we differentiate between the two methods. The boolean method uses a list of booleans to indicate if we visit the counter for the first time. If yes, we reset it to the initial value. This algorithm is shown in Figure 3.4.

```
Algorithm 3: generate applicable operators, boolean approach
Input: State s
list first_visit = new boolean list(size_of(operators, True))
for each Fact f in s do
    for each Operator o in preconditions_of(f) do
        if first_visit[o] = True then
            counter[o].reset_to_initial_value
        counter[o] = counter[o]-1
for each operator where counter = 0 do
    report o as applicable
```

Figure 3.4: Algorithm for the boolean approach

The second approach uses timestamps to determine whether to update the counter or not. If the counter is smaller than the current timestamp, we need to update the counter. This algorithm is shown in Figure 3.5.

3.4 PSVN Successor Generator

The PSVN successor generator is based on the planning formalism introduced by Holte et al. [10]. We first discuss the PSVN planning formalism and then switch to the implementation of the successor generator.

Algorithm 4: generate applicable operators, timestamp approach

```
Input: state s
current_timestamp = current_timestamp +1
for each Fact f in s do
    for each Operator o in preconditions_of(f) do
        if timestamps[o] < current_timestamp then
            counter[o].reset_to_initial_value
        counter[o] = counter[o]-1
for each Operator where counter = 0 do
    report o as applicable
```

Figure 3.5: Algorithm for the timestamp approach

3.4.1 PSVN

PSVN is a planning formalism introduced by Holte et al. [10] for describing state space problems using variables with a finite domain "that lends itself to efficient calculation of a state's successor and predecessor without having to fully ground the operators" [10]. A state in PSVN is a vector of fixed length n. Each entry i is drawn from the domain D_i. Transitions between states are specified by operators. Operators have a left-hand side (LHS) specifying the preconditions of the operator and a right-hand side (RHS) specifying the effects of the operator. LHS and RHS are both a vector of size n and each entry in the vector is either a constant from the domain D or a variable symbol. A state $s = \langle s_1, \dots, s_n \rangle$ matches the LHS $= \langle L_1, \dots, L_n \rangle$ if $s_i = L_i$ for every L that is constant and $s_i = s_j$ for every i and j where L_i and L_j are the same variable symbol. An operator is deterministic if every variable symbol in its RHS is also in its LHS. An operator is non-deterministic if one or more of the variable symbols in its RHS do not occur in its LHS. We call such variable symbols unbound. The effect of a deterministic operator applied to a state $s = \langle s_1, \dots, s_n \rangle$ with a matching LHS is to create a state $s' = \langle s'_1, \dots, s'_n \rangle$ such that:

- if position j of the RHS is a constant $c \in D_j$ then $s'_j = c$
- if position j of the RHS is a variable symbol that occurs in position i of the LHS then $s'_j = s_i$

The effect of applying non-deterministic operators on a state $s = \langle s_1, \dots, s_n \rangle$ is to create a set of successor states. We create one successor for every possible combination of values of the unbound variable. For example if $n = 4$ and all positions have domain {1,2} then the rule $\langle 1, A, B, C \rangle \rightarrow \langle E, 1, D, E \rangle$ would create four successors when applied to state $\langle 1, 2, 1, 2 \rangle$: $\langle 1, 1, 1, 1 \rangle, \langle 2, 1, 1, 2 \rangle, \langle 1, 1, 2, 1 \rangle$ and $\langle 2, 1, 2, 2 \rangle$. For the sake of readability, PSVN allows one additional symbol, the dash("-") on the LHS and RHS. A dash in position i on the LHS means, there is no precondition on the value in position i. A dash in position i on the RHS means that the value in position i does not change, when the operator is applied. Each operator can be in one of three categories:

- An operator is unsatisfiable, if one of the preconditions is unsatisfiable given a state. If we know $v_1 = 2$ and have the rule $\langle 1, X\rangle \to \langle X, 0\rangle$, then this operator is unsatisfiable.
- An operator is satisfiable if all preconditions are satisfied given a state. If we have the same example from above with $v_1 = 1$, then the operator is satisfiable.
- An operator is plausible, if it is neither satisfiable nor unsatisfiable. For the same example if we have $v_2 = 3$, then the operator is plausible.

Before applying an operator o we check if the LHS of the operator matches the given state s. If yes, we can start building the successor state s' with the RHS of o. Each variable in s is transformed according to its corresponding value in the RHS of o and changes its value to the value in the RHS which produces s'.

Definition 3.4.1 (Goals)**.** *goals are defined by special goal conditions. These goal condition are operators where the condition takes the same form as the LHS of a normal operator. Each goal state must be defined in the planning task and each goal condition is interpreted on its own.*

Definition 3.4.2 (Directed Acyclic Graph)**.** *A Directed Acyclic Graph is a graph consisting of vertices which are connected by directed edges. If none of the edges form a cycle inside the graph, we call it acyclic. Vertices with outgoing edges are called inner vertices, vertices without any outgoing edges are called leaf vertices.*

3.4.2 Successor Generation In PSVN

In PSVN we precompute a Directed Acyclic Graph similarly to the causal graph in Fast Downward. A vertex in the PSVN DAG contains a set of plausible operators, a set of all variable assignments, a set of all satisfied operators and a choice for the next variable to test. An edge contains a set of variable assignments and a set of satisfied operators. We build the DAG starting from the root recursively in the following way:

1. We first set aside any operators which are satisfied without any knowledge (operators without preconditions) that can immediately be applied.
2. At the beginning, all operators are plausible because no state variable has been assigned any value yet. We create the graph with a single entry containing a set of plausible operators containing all operators, an empty set of satisfied rules and the choice for the first variable we want to test.
3. We chose a variable to test that has not yet been assigned. For the root this means that we can choose any variable, because none has been assigned yet.
4. We create $|D_v|$ many outgoing edges from the vertex where $|D_v|$ is the domain of the chosen variable in the outgoing vertex. Each edge represents an assignment d of v where $d \in D_v$.

5. We then add the newly chosen variable assignment to the set of variable assignments from the outgoing vertex and update the set of plausible operators. Any operator that is now satisfied is removed from the set of plausible operators and added to the set of satisfied operators. Any operator that is now unsatisfiable is discarded. All other operators stay in the set of plausible operators.

6. We simplify the two new lists in the vertex. We remove all preconditions from the set of plausible operators that have been satisfied. Also we discard any variable assignments that are no longer referenced by any of the plausible operators (e.g. we keep only the variable assignments that are used by any of the preconditions).

7. We then check if a vertex with these sets of plausible operators, satisfied operators and variable assignments already exists.

   - If there is a vertex with the given features, the edge leads to this vertex and the recursion stops.
   - If not, we create a new vertex with these features and let the edge lead to this new vertex.

8. If we were to create a new vertex in the step above and it still has any plausible rules left, we continue processing the vertex beginning again from step 3. If there are no plausible rules left we stop the recursion.

To decrease the size of DAG, it is advised to split the DAG if it gets too big. In this thesis we begin to split the operators in two groups and build a separate DAG for each of them if the number of vertices in the DAG succeeded 500'000. This can be done recursively if splitting in half still does not reduce the DAG enough.
When we finish building the DAG or the multiple splitted DAGs, we create a function for every vertex, every operator in an edge and every operator with empty precondition we set aside in the beginning. A vertex function performs the test specified for the chosen variable at the vertex. For each result, it performs some action which depends on the DAG structure. If there are any satisfied operators in the edge which is labeled with the test result, it returns the result of calling the first edge rule function of that edge. Otherwise, if the child vertex has any plausible rules it returns the result of calling the child vertex function. Lastly, if there are no rules in the edge and the child vertex has no rules the function returns failure to indicate that there are no more children left to generate.

The edge rule function returns the rule which is to be applied and the next edge rule function if there are more rules on the edge on the vertex function for the child vertex otherwise.
The root rule functions return the rule and the next root rule function if there are more rules with empty precondition or the vertex function for the root of the DAG otherwise.
We can iterate through the children by calling the first root rule (or vertex function if there are

no rules without preconditions) applying the returned rule and continue with the function which is returned.

3.4.3 Implementation Notes

We abandoned the original PSVN notation with the LHS and RHS and use the notation introduced by Helmert [8] because we are implementing the successor generator in Fast Downward and want to use the existing structures without translating back and forth. Because of this change of notation, we do not have any operators without preconditions anymore and can start creating the DAG at the root directly. Also the original PSVN successor generator follows a compiled approach, which produces programming code promptly. We did not follow this approach and took a different route instead. The DAG is created as a list of vertices with its root as the first entry. Each newly created vertex is appended to this vertex list. The edges are represented inside the vertices as a pointer to the children's position inside the vertex list. To efficiently determine if we have already created a vertex, we store all vertices in a hash map and can check for existence in constant time where we map from the vertex' hash value to its position in the vertex list. We traverse the DAG in the following manner: For each vertex we store all of its children in a list. We recall a vertex has $|D_v|$ many edges, one for each possible variable assignment in v. If $D_v = \{d_1, \ldots, d_n\}$ then the first stored children is d_1, the second d_2 and so on. We can now easily follow variable assignment d_i by following child number i. In each vertex we visit we can collect all the operators that are applicable at that point. Each applicable operator appears once on the way through the DAG. An implementation can be seen in Figure 3.6 for the initialization and in Figure 3.7 for the generation of applicable operators.

3.5 Watched Literals Successor Generator

The idea of this successor generator is based on the boolean satisfiability problem (SAT). The boolean satisfiability problem tries to solve the question: *"given a boolean formula, is it satisfiable?"*, which was the first problem to be proven NP-complete by Cook [2]. We solve the satisfiability question with an backtracking algorithm that works in the following way:
We chose an unassigned literal and assign one of its values (True or False) to it. We then simplify the formula by applying the assignment to it and recursively check if the formula has been satisfied. If it has not yet been satisfied we swap the literal to the other value and check again. With this we create two subproblems, one for each assignment of True and False. Each of the sub problems can then again be divided in smaller sub problems by assigning other literals a value. Typically we work on a standardized input in conjunctive normal form (CNF) to exploit properties of it. The CNF is a conjunction of one or more clauses, where each clause is a disjunction of literals. A boolean formula in CNF might look like this: $(a) \wedge (\neg b \vee a) \wedge (\neg a \vee c \vee d)$. We can exploit the fact that we need to satisfy all clauses and a clause is only satisfied if one of its literals is satisfied. Additionally, the formula is unsatisfiable if one of its clauses is unsatisfiable, which is

the case when all literals in the clause are unsatisfiable. For solving problems in CNF, we can use a more efficient backtracking algorithm introduced by Davis and Putnam [3], and later redefined by Davis et al. [4], the DPLL algorithm. DPLL works similarly to the backtracking algorithm described above with a few additions:
If a clause only contains one unassigned variable, we can apply unit propagation to it. This means we can trivially assign the unassigned variable to satisfy the clause. The second addition is that if one variable in a clause is satisfied, the clause is satisfied no matter what we assign to the other variables in the clause. So while checking the formula for satisfiability we can discard any clauses that have been satisfied which will give us a smaller sub problem to compute. If a variable is unsatisfied in a clause, we can remove this variable as it will stay unsatisfied. If this would lead to an empty clause (a clause without any variables in it), the clause is unsatisfiable and therefore the whole formula is unsatisfiable.
The Two Watched Literals technique as described by Moskewicz et al. [11] tries to exploit the fact that we only have four different cases in a clause:

- A literal in the clause has been satisfied and we can delete the clause from the formula.
- All literals in the clause are unsatisfiable and we can report the formula as unsatisfiable.
- Only one unassigned literal is left in the clause and we can apply unit propagation.
- More than one unassigned literal are left in the clause and we need to further apply backtracking to the formula.

For each clause we only want to know if we can apply unit propagation or if all literals are unsatisfiable. For this we want to know for each clause when it changes from *“more than one literal unassigned”* to *“only one literal unassigned”*. We do this by watching two literals for each clause. If one of the two watched literals becomes satisfied, we can remove the clause because it has been satisfied now. If one of the two literals becomes unsatisfiable we search for a new unassigned literal. If we can not find another unassigned literal we have only one unassigned literal left and can apply unit propagation to the clause. With this technique we do not need to check each literal in a clause again and against. We only need to check two literals for each clause which makes the backtracking algorithm a lot faster.

3.5.1 Converting To The Successor Generator

We want to apply the idea of the Two Watched Literals to the operator preconditions, so we do not need to check all preconditions over and over again. Because SAT works differently than generating applicable operators, we first need to define how we can exploit Two Watched Literals for successor generation. Because Fast Downward lives in a finite domain world, we do not have boolean variables anymore. Variables can have an arbitrary big domain. Also if we look at the operators, we do not have CNF anymore, but its counterpart the Disjunctive Normal Form (DNF). We have a disjunction of operators and each operator is a conjunction of preconditions. This means each operator needs to be satisfied for itself and an operator is only applicable if all of its preconditions are satisfied. This means we want to know for each operator if all preconditions have been satisfied or not. Additionally, an operator is not applicable if one of its preconditions is unsatisfiable. We only need to watch one precondition for each operator while generating the applicable operators. Accordingly, we individually check each fact in the given state individually. For all operators that are watching this fact we check if they have an unsatisfied precondition. If they do not, we can report the operator as applicable. If they have an unsatisfied fact we make the fact of this precondition the new watcher for this operator.

3.6 Implementation Notes

In order to access the watched operators in constant time, we need efficient data structures that support this kind of access. Either we implement the watcher list as a hash map or we use the same data structure as used in the Marking successor generator where we stored a list of operators for each possible variable assignment.
In order to not switch watching between the same two variables back and forth because we always look at them in the same order, it is important to watch them in cycles. We do not always watch the preconditions from start to beginning but remember which precondition we checked last and start with this one. If we reach the end of the preconditions list we start again from the beginning until we checked all preconditions of the operator.

Algorithm 5: Vertex Object

```
list plausible_operators
list variable_assignments
list satisfied_operators
list children
int choice
int hash
```

Algorithm 6: PSVN Initialization

```
Vertex v
v.plausible_operators = all_operators
v.satisfied_operators = new list()
v.variable_assignments = new list(size_of(variables), -1)
vertex_list.push_back(v)
hashmap.insert(v.hash, vertex_list.end)
create_DAG_recursive(vertex_list.end)
```

Algorithm 7: create_DAG_recursive

```
Input: int pos
if !vertex_list[pos].plausible_operators.empty() then
    vertex_list[pos].choose_unassigned_variable()
    for each value v in vertex_list[pos].choice do
        plaus_ops = list(empty)
        sat_ops = list(empty)
        vars = vertex_list[pos].variable_assignments
        vars[vertex_list[pos].choice] = v
        split_and_simplify(vertex_list[pos], plaus_ops, vertex_list[pos].choice, v, sat_ops)
        Vertex v = new Vertex(plaus_ops, sat_ops, vars)
        if v.hash not in hashmap then
            vertex_list.push_back(v)
            hashmap.insert(v.hash, vertex_list.end)
            vertex_list[pos].children.push_back(vertex_list.end)
            create_dag_recursive(vertex_list.end)
        else
            vertex_list[pos].children.push_back(hashmap.find(v.hash))
```

Algorithm 8: split_and_simplify

```
Input: Vertex v, list plausible_ops, int var, int val, list satisfied_ops
for each operator o in v.plausible_ops do
    list temp_preconditions
    bool unsatisfied = false
    for each Precondition p in o do
        if var = p.var then
            if val ≠ p.val then
                unsatisfied = true
                break
        else
            temp_preconditions.push_back(p)
        if !unsatisfied then
            if p.empty then
                sat.push_back(o)
            else
                plaus.push_back(new Operator(o.id, temp_preconditions))
```

Figure 3.6: PSVN Initialization

Algorithm 9: generate_applicable_operators

```
Input: State s
traverse_DAG(vertex_list[0], s)
```

Algorithm 10: traverse_DAG

```
Input: Vertex v, State s
for each Operator o in v.satisfied_operators do
    report o as applicable
if !v.children.empty() then
    var = v.choice
    val = s[var]
    Vertex next_vertex = vertex_list(v.children[val])
    traverse_DAG(next_vertex, s)
```

Figure 3.7: PSVN generate applicable operators

Algorithm 11: Watched Literals Successor Generator

```
Input: state s
for each Fact f in s do
    for each Operator o watching f do
        if if any o.precondition = unsatisfied then
            watcher[o].remove()
            watcher[o] = unsatisfied precondition
        else
            report o as applicable
```

Figure 3.8: Watched Literals Successor Generator

4 Evaluation

4.1 Overview

To evaluate the successor generators we used a benchmark collection from the International Planning Competitions, years 1998 to 2018. The benchmark collection can be found on GitHub[2]. We used all optimal benchmark instances to evaluate, which is a collection of 65 different domains with a total of 1827 different planning tasks. The search algorithm we use is a blind A* algorithm Hart et al. [7]. All instances have a time limit of 30 minutes and a memory limit of 3.5GiB. When exceeding this limit, the search will stop.
We run two different big experiments with all successor generators, once without any limitation on state expansion to see how they perform solving the whole planning task and a second one where we limited state expansion to 10'000. Additionally, we performed two smaller experiments, one on the two implementations of the Marking successor generator and one on the PSVN successor generator with and without the DAG splitting. All tests were performed at sciCORE[3], the scientific computing center at the University of Basel. The experiments were run on an Intel Xeon E5-2660 2.2 GHz processor with 16 cores using the Downward Lab testing environment created by Seipp et al. [13].

4.2 Results

In this section we will discuss all the results from Tables 4.1 and 4.2. These tables are a summary of the experiments we described above. The complete experiment reports are two documents with over 700 pages which would be too much to fit in this thesis. Still they are available on the GitHub repository.[4] We start analyzing the experiments by first discussing some general observations and then comparing the successor generators against each other for each of the

[2] https://github.com/aibasel/lab
[3] https://scicore.unibas.ch/
[4] https://github.com/YannickZutter/fast-downward/tree/yannick02/lab_test_results

attributes of the tables. Let us shortly introduce these attributes:

- **Coverage:** Number of solved problems.
- **Out Of Memory:** Number of problems which exceeded the memory limit.
- **Out Of Time:** Number of problems which exceeded the time limit.
- **SG Init Time:** Time to initialize the successor generators in seconds summed up.
- **GAO Time:** Total duration of the generate applicable ops function calls summed up in milliseconds.
- **GAO Mean:** Total duration of the generate applicable ops function calls as geometric mean.
- **Total time** Total run time in seconds as geometric mean.

Before we start discussing the experiments, let us first have a look at the Marking successor generator and the comparison of its two implementations in Table 4.1. As we can see it does not make a big difference if we run the algorithm with a boolean list or with timestamps. The differences are rather small and for the overall evaluation we only consider the boolean implementation as the Marking successor generator because it performed a little bit better than its counterpart. We also tested the PSVN successor generator with and without splitting the DAG. We will talk about this later in this chapter.

Summary	Boolean	Timestamps
Coverage	**680**	679
Out Of Memory	1'007	**997**
Out Of Time	**93**	104
SG Init Time	1'227.23	**1'182.48**
GAO Time	**31'872.11**	35'313.44
GAO Mean	**0.0103**	0.0144
Total Time	**2.15**	2.20

Figure 4.1: Marking successor generator comparison

As a general observation we can say that both experiments showed similar results, which shows that the implemented successor generators are consistent and work in the same way on smaller test runs which are limited to fewer state expansions as they run on planning tasks with up to millions of state expansions.

Let us start evaluating the different successor generators by looking at the coverage. All successor generators solved around the same number of problems, which shows that there is not a successor generator that can solve planning tasks with a better consistency. If we look at the Out Of Memory attribute, we can see that the Fast Downward, the Marking and the PSVN successor generator reached the memory limit far more often than the Naive or the Watched Literals successor generator. This is due to the fact that these successor generators need to precompute a lot more than the Naive or the Watched Literals successor generator. This can also be supported if we look at the initialization time of these algorithms which is significantly higher than the one of the Naive or the Watched Literals successor generator which do not have to initialize a lot of data or structures. Additionally, we can see that the Naive and the Watched Literals successor generator reached far more often the time limit than the other three successor generators. This is also shown in the GAO time and GAO mean. While having almost no time to initialize, the Naive and Watched Literals take significantly longer to generate the applicable operators. In the total time we can see that overall the Fast Downward successor generator works the fastest in all of the experiments.

In figure 4.2 we run an experiment with two versions of the PSVN algorithm. The normal version with only one DAG and an implementation where we split the DAG if it has more than 500'000 vertices as discussed in Chapter 3.4.2. We can see that while having a slightly slower generation time for the applicable operators, the split version had overall a way better performance than the default version. We reduced the memory usage for the algorithm by a lot. This is shown in the fact that we brought down the instances that reached the memory limit from over 1'500 to a bit more than 800. Also the initialization time has been cut in half with the addition of the split tree. On one hand, the split version has a longer to generate the applicable operators because it has to traverse multiple DAG to collect the applicable operators. On the other hand, the DAGs get a lot smaller if we can split them. Instances where the DAG has over 3 million vertices can be cut down to two DAGs with a total of more or less 100'000 vertices.

Summary	PSVN	Split
Coverage	253	**405**
Out Of Memory	1'522	**818**
Out Of Time	**0**	278
SG Init Time	335.54	**167.82**
GAO Time	**873.75**	1'242.73
GAO Mean	**0.0014**	0.0020
Total Time	0.51	**0.49**

Figure 4.2: Comparison PSVN and Split DAG

Summary	Fast Downward	PSVN	Marking	Watched Literals	Naive
Coverage	**712**	253	680	658	689
Out Of Memory	1'098	1'522	1'006	866	**773**
Out Of Time	**0**	**0**	94	256	348
SG Init Time	0.08	335.54	0.59	0.02	**0.01**
GAO Time	**841.53**	873.75	1'592.82	3'079.83	3'735.03
GAO Mean	**0.0014**	**0.0014**	0.0026	0.0050	0.0061
Total Time - Mean	**0.09**	0.51	0.10	0.11	0.12

Table 4.1: Results without any bound

Summary	Fast Downward	PSVN	Marking	Watched Literals	Naive
Coverage	254	161	225	227	**255**
Out Of Memory	**0**	1’393	**0**	**0**	**0**
SG Init Time	0.08	325.98	0.58	0.02	**0.01**
GAO Time	**3.27**	3.67	10.77	35.86	30.55
GAO Mean	**0.0014**	0.0015	0.0045	0.0148	0.0126
Total Time - Mean	**0.03**	0.34	0.04	0.04	0.04

Table 4.2: Results with bounds at 10’000 expanded states

4.3 Final Evaluation Of The Successor Generators

As shown in the results, we can see a trend for the successor generator implementations. There is a clear trade-off between precomputing and applicable operator generation. The more precomputation we perform, the faster the generation of applicable operators. On one side we have the implementations with a lot of precomputation like the PSVN successor generator with its DAG or the Fast Downward with the tree. In the middle we have The Marking successor generator which does some precomputation and has a decent run time in the successor generator. On the other side, we have the Watched Literals successor generation which only has a little precomputation but needs to calculate much more during the successor generation. Lastly, the Naive successor generator which does not precompute anything at all and does all the work in the generation of the applicable operator. If we set up a list from fastest applicable operators generation to fastest initialization we would get following:

1. PSVN/ Fast Downward
2. Marking
3. Watched Literals
4. Naive

All successor generators can be used in a specific scenario. If we have huge planning tasks where we probably need millions of steps to find a goal, we are better off choosing a successor generator with a higher precomputation time like PSVN or Fast Downward. Besides, if we have a lot of small planning tasks that will be solved quickly we do not need a huge precomputation and can accept that we do not have a fast generation of applicable operators because we only need a few steps to find the goal.

5 Conclusion & Future Work

5.1 Conclusion

As we have seen in the evaluation of the successor generators we have a use case for all of them. Depending on what kind of planning task we want to solve, it is advisable to switch to a successor generator that is suited best for the experiment. Although the implemented successor generators give an estimate on how well they perform, we also need to take into account that all of these successor generators were implemented in rather short time except for the already existing Fast Downward successor generator. These successor generators have been optimized in the time given, but there might still be a lot of potential to optimize so that their performance could be much better. For example the PSVN successor generator was made twice as fast with one tweak on how the operators and variables to test were stored and compared. As seen in the comparison of the PSVN successor generator it is also advisably for successor generators that precompute a tree or graph to split up the operators on different trees.

5.2 Future Work

Due to the limited time for this thesis, there are still some topics left that have not been worked on. Having tested the DAG splitting in the PSVN successor generator we found that it is advisable to split the operators and generate multiple DAGs for bigger planning tasks that have a lot of operators. This approach can also be implemented and tested for the Fast Downward successor generator whose precomputed tree can be split in the same manner as with the PSVN DAG. It would also be interesting to see how the trade off between splitting the DAG and having to traverse multiple DAGs to generate the applicable operators would look like and at which point it would be optimal to split a DAG before it is too big.
As we said above, all of the successor generators can be optimized. Especially for the PSVN successor generator, there might be a lot of potential to improve the performance of the algorithm. If we consider the A* algorithm we might be in a situation where we are currently at a promising

path while expanding states. We expand one state after another without switching to a different state in the algorithm's open list because expanding "here" still gives us a more optimal path than jumping to another state in the open list. We assume that expanding states in the same "area", they do not differ that much from another. Therefore we always store the state and the applicable operators in each function call to generate the applicable operators. We can then check the previous state and copy all operators that have preconditions on variables that have not been changed. This has the potential that we do not have to calculate all the operators but we could copy some of them. This would only have a meaningful impact if we test it with a heuristic search instead of a blind one, because in blind search we only expand layer after layer without following any promising paths.

Bibliography

[1] Blai Bonet and Hector Geffner. Planning as heuristic search. *Artificial Intelligence*, 129: 5–33, 2001.

[2] Stephen A. Cook. The complexity of theorem-proving procedures. *3rd Annual ACM Symposium on Theory of Computing.*

[3] Martin Davis and Hilary Putnam. A computing procedure for quantification theory share on. *Journal of the ACM*, 7:201–215, 1960.

[4] Martin Davis, George Logemann, and Donald Loveland. A machine programm for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[5] Florian Geißler, Thomas Keller, and Robert Mattmüller. Delete relaxations for planning with state-dependent action costs. *International Conference on Artificial Intelligence*, pages 1573–1579, 2015.

[6] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning and Acting.* Cambridge University Press, 2016. ISBN 978–1–107–03727–1.

[7] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

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

[9] Malte Helmert. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173:503–535, 2009.

[10] Robert C. Holte, Broderick Arneson, and Neil Burch. *PSVN Manual (June 20, 2014)*, 2014.

[11] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. *38th annual Design Automation Conference*, pages 530–535, 2001.

[12] Allen Newell, John Clifford Shaw, and Herbert Alexander Simon. Report on a general problem-solving program, 1959.

[13] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. https://doi.org/10.5281/zenodo.790461, 2017. URL https://doi.org/10.5281/zenodo.790461.

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud
beinhaltet Erklärung zu Plagiat und Betrug

Author — Autor
Yannick Zutter

Matriculation number — Matrikelnummer
2015-052-723

Title of work — Titel der Arbeit
Implementing and Evaluating Successor Generators in the Fast Downward Planning System

Type of work — Typ der Arbeit
Bachelor Thesis

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

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, September 30, 2020

[illegible]

Signature — Unterschrift