University of Basel

Heuristic Planning with Single Action Goal Expansion

Bachelor thesis

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

Examiner: Prof. Dr. Malte Helmert
Supervisor: Cedric Geissman

Remo Christen
remo.christen@stud.unibas.ch
2015-051-469

21.6.2019

Acknowledgments

I want to thank Prof. Dr. Malte Helmert for the chance to write my thesis in the Artificial Intelligence research group.
I am grateful for the guidance and support offered by Cedric Geissman who patiently and diligently helped with my questions and uncertainties.
Calculations were performed at sciCORE (https://scicore.unibas.ch/) scientific computing center at the University of Basel.

Abstract

Heuristic search is a powerful paradigm in classical planning. The information generated by heuristic functions to guide the search towards a goal is a key component of many modern search algorithms. The paper "Using Backwards Generated Goals for Heuristic Planning" by Alcázar et al. [1] proposes a way to make additional use of this information. They take the last actions of a relaxed plan as a basis to generate intermediate goals with a known path to the original goal. A plan is found when the forward search reaches an intermediate goal.

The premise of this thesis is to modify their approach by focusing on a single sequence of intermediate goals. The aim is to improve efficiency while preserving the benefits of backwards goal expansion. We propose different variations of our approach by introducing multiple ways to make decisions concerning the construction of intermediate goals. We evaluate these variations by comparing their performance and illustrate the challenges posed by this approach.

Table of Contents

Acknowledgments **ii**

Abstract **iii**

1 Introduction **1**

2 Background **4**

2.1 State Representation 4
2.2 Planning 5
2.3 Heuristics 5
2.4 Relaxation Heuristics 6
2.4.1 FF Heuristic 6
2.5 Concepts from Regression Search 7
2.6 Backwards Generated Goals 7

3 Single Action Goal Expansion **8**

3.1 Idea 8
3.1.1 Overview 8
3.1.2 Relation to Backwards Generated Goals 9
3.2 Goal Expansion 9
3.3 State Decision Process 11
3.3.1 New Minimum 11
3.3.2 Accuracy 12
3.3.3 Counter 12
3.4 Operator Order 13
3.4.1 Most Satisfied 13
3.4.2 Lowest Layer 14
3.5 Check Legality 15
3.5.1 Deletes Goal Proposition 15
3.5.2 Mutex 16
3.5.3 Dominated by Ex Goal 16
3.6 Safety Features 17
3.6.1 Blocking Operators 17
3.6.2 Original Goal Check 17

3.6.3 Initial State Check 17

4 Evaluation **19**

4.1 Results 19

4.1.1 Forward Expansions in Suited Domains 20

4.1.2 Openstacks 21

5 Conclusion **26**

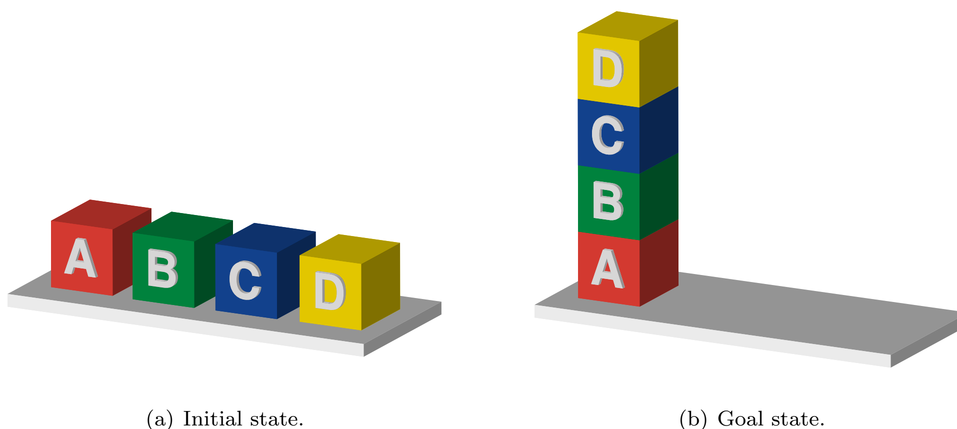
Bibliography **27**

Declaration on Scientific Integrity **28**

1 Introduction

Classical planning is the discipline of finding a sequence of actions to get from the initial state to a goal state. This description is intentionally generic, because the idea of classical planning is to find an algorithm that is capable of solving a variety of different problems. To make this possible, problems have to be formally described in a standardized manner. These descriptions lay out the rules and boundaries of the problem as well as its starting point and goal. Given these prerequisites a single planner can process various tasks of ranging difficulty and size. In the following we will introduce a simple problem from a class of problems called BLOCKSWORLD to illustrate what such a task may look like.

The BLOCKSWORLD domain contains the following three elements: a table, a hand and blocks. The blocks are either on the table or stacked on top of each other. The hand can pick blocks up one at a time, either from the table or from the top of a stack. Once the hand holds a block it can put the block on top of another or put it on the table. Fig. 1.1 shows an example problem of BLOCKSWORLD. We start out with four blocks on the table as seen in Fig. 1.1(a). The goal is to end up with a single tower as shown in Fig. 1.1(b).



(a) Initial state.

(b) Goal state.

Figure 1.1: BLOCKSWORLD example problem.

The most conventional approach to planning is forward search. It starts at the initial state

and explores adjacent states by applying actions. When it reaches a goal state it remembers the actions taken and returns the sequence that leads to the goal.

In our example there are four possible actions that can be taken in the initial state as shown in Fig. 1.1(a): Pick up A, B, C or D. In the next step we can stack the block we picked up on any of the remaining three blocks and so on. This way the search chooses actions until it finds the goal configuration. In this small problem there are only 24 possible paths for the first three moves (without revisiting states). This number increases to 720 for the same problem with ten blocks and to 6840 for twenty. More challenging domains than BLOCKSWORLD can span over millions of states. So while it is possible to blindly try all possible paths in our simple example, most planning tasks are too big to be solved in this way.

Guided search is an approach to solving bigger problems. One way of guiding a search is by means of a *heuristic*. Heuristics estimate the distance from a state to the goal. With this information the search can prefer states that seem to be closer to the goal and thus guide the search towards it. In our example a heuristic would likely let the search know that picking up B is a more promising first move than picking up A. What seems trivial in this example can make a big difference when dealing with big tasks.

In this thesis we expand this approach by supporting the forward search with a backward element. We take the goal and look backwards to find a way to make the goal more accessable from the initial state. In our example there is only one action that leads to the goal: To stack D on C. Thus we know that once the state in Fig. 1.2 is reached, we only have to stack D on C and we have reached the goal. In that sense the state in Fig. 1.2 can act as our new goal because we know how to get to the original goal from there. We call this process *goal expansion*.

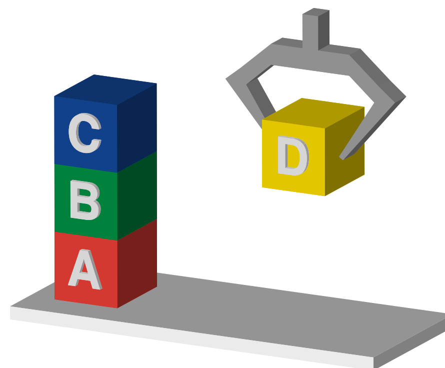


Figure 1.2: State after expanding the goal once.

This thesis aims to make bigger tasks solvable by combining heuristic forward search with goal expansion. The goal expansion is informed by the calculation of the heuristic and can thus reuse the knowledge gathered. The ideas behind this are based on a paper by Alcázar et al. [1].

This thesis is structured into three main chapters. In the first part we lay out the theoretical foundation by introducing and defining the concepts used throughout the rest of the work. This is followed by an explanation of the name-giving algorithm, Single Action Goal Expansion, where we give an overview of the idea and subsequently examine its components more indepth. Finally we show what experiments we ran to evaluate the algorithm and analyze their results.

2
Background

This chapter is dedicated to the formalisim and definitions used throughout this thesis. The selection of topics and concepts is given by the requirements of the content, it is by no means meant to be a comprehensive catalogue.

2.1 State Representation

There are multiple ways to formally define planning tasks, we will use the STRIPS representation as described by Fikes and Nilsson [4].

Definition 1 (Planning Task)**.** A planning task is defined by a 4-tuple $\Pi = \langle \mathcal{P}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$:

- $\mathcal{P}$: Finite set of atomic statements which are either `true` or `false`. We will refer to these statements as *propositions*.
- $\mathcal{O}$: Finite set of operators. Every operator $o \in \mathcal{O}$ is defined by the triple $\langle pre, add, del \rangle$:
  - $pre(o)$: Set of propositions that have to be `true` to make o applicable.
  - $add(o)$: Set of propositions that are set to `true` when o as applied.
  - $del(o)$: Set of propositions that are set to `false` when o is applied.

  In case proposition p is found in both $add(o)$ and $del(o)$, add has precedence and p is satisfied upon application of o.
  The elements in the conjunction of $add(o)$ and $del(o)$ are called the *effects* of o.
  Every operator has a cost associated with it so that $cost(o) \rightarrow \mathbb{R}_0^+$.
- $\mathcal{I}$: The initial state with $\mathcal{I} \subseteq \mathcal{P}$.
- $\mathcal{G}$: Set of propositions that form the task's goal with $\mathcal{G} \subseteq \mathcal{P}$. We call the members of this set *goal propositions*.

The following notes define the terminology we will use when talking about planning tasks:
The term *problem* will be used interchangeably with *task*.
We use *action* and *operator* as synonyms.

With *goal set* we mean the set of propositions that define a goal.

As the task definition implies, a *state* s is defined as a set of propositions $p \subseteq \mathcal{P}$ so that all p hold true in s and other propositons are implicitly false. In order to solve planning tasks we must *apply* operators to states and yield their *successors* to find a *path* to the goal. The following definitions are relevant to this process.

Definition 2 (Applicable Operator)**.** An operator o is applicable in state s when all propositions $p \in pre(o)$ are true in s.

Definition 3 (Successor State)**.** A state s' is called successor of state s when there exists an operator o whose application to s' yields s. For this we use the following notation: $s' = s[\![o]\!]$.

Definition 4 (Path)**.** A path π from state s_x to state s_y is a sequence of operators $o_0, \dots, o_n$ whose consecutive application to s_x yields s_y so that $s_y = s[\![o_0]\!] \dots [\![o_n]\!]$.

A path is called a *plan* (or a *solution*) if it starts in the initial state and ends in a goal state. The cost of a path π is the sum of the cost of all operators in π.

2.2 Planning

A planning problem can be solved by exploring the entire state space. That way an algorithm such as breadth-first search can return a plan for any solvable task by traversing the graph implied by the state space. This approach is an example for uninformed search. Uninformed refers to the fact that no information outside the problem definition is used to make decisions on what step to take next, or more specifically, what node to expand next. Breadth-first search for instance visits states in the order they were generated in. While this approach works fine for small problems, difficulties quickly arise when dealing with larger problems. With increasing problem size, state spaces grow quickly. Solving such problems with an uninformed, naive strategy is not feasable due to limitations on the crucial resources: memory and time.

One way to handle large state spaces is by informing the search. That is done by replacing the aimless exploration with an algorithm that actively searches towards the goal. The goal thereby is to distinguish good states, meaning those that will lead us closer to the goal, from bad states. Based on this metric the search can prefer better states and close in on the goal more effectively and efficiently and thus be able to conquer bigger state spaces. Heuristics are a proven method of informed search.

2.3 Heuristics

Definition 5 (Heuristic)**.** A heuristic assigns a number to a state by means of a heuristic function. The heuristic function h maps a state s to a heuristic value h_{val} so that $h : s \to h_{val}$ with $h_{val} \in \mathbb{R}_0^+ \cup \{\infty\}$.

Heuristics try to estimate the distance of a state to the goal. Quantified in the heuristic value this estimate can be used to compare states and their potential of being part of a plan.

States with a lower heuristic value can generally be considered more promising because the heuristic views them as being closer to the goal. A heuristic assigns a heuristic value of ∞ to states from which it believes the goal cannot be reached.

The use of heuristics gives rise to more sophisticated search algorithms such as greedy best-first search which is what we will use in this thesis.

The evaluation function f of greedy best-first search is the heuristic h. Thus for state s the following holds: $f(s) = h(s)$. This means that greedy best-first will always expand the state with the lowest heuristic value.

Various functions can serve as heuristics but they are not created equal. The class of relaxation heuristics is one of the most successfully used paradigms in today's planners.

2.4 Relaxation Heuristics

Relaxation heuristics find a solution for a simplified version of the problem and use it as the basis to calculate a heuristic value.

The most common relaxation heuristics use delete relaxation to construct the relaxed problem. The original problem is transformed to one where all negative effects are removed. In the delete relaxed problem, propositions that are set to true once will remain true until a goal state is reached. This makes the relaxed problem easier to solve than its non-relaxed counterpart because the application of operators always leads closer to the goal, thus the continuous application of available operators will eventually reach a goal state.

In the STRIPS formalism negative effects are all effects in the set of *del* effects. The delete relaxed task can therefore be obtained by removing all *del* effects while keeping the *add* effects. The following definition describes this more formally.

Definition 6 (Delete Relaxed Task)**.** A planning task $\Pi = \langle \mathcal{P}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$ is transformed to the relaxed task $\Pi^+ = \langle \mathcal{P}, \mathcal{O}^+, \mathcal{I}, \mathcal{G} \rangle$ where $\mathcal{O} \rightarrow \mathcal{O}^+$ so that $\{\langle pre, add, del \rangle \rightarrow \langle pre, add, \emptyset \rangle \, \forall \, o \in \mathcal{O}\}$

The solution to a delete relaxed task is called a *relaxed plan.*
There are multiple ways to extract a heuristc value from a relaxed task. We will focus on the one used in this thesis: the FF heuristic.

2.4.1 FF Heuristic

The delete relaxation heuristic used in this thesis is the FF heuristic first introduced by Hoffmann and Nebel [6] which is based on h^{add} from the Heuristic State-Space Planner by Bonet et al. [3]. h^{add} calculates the cost of reaching all goal propositions in the relaxed problem. The heuristic value is the sum of these costs. The assumption made by h^{add} is that all goal propositions have to be reached independently. This is rarely the case so h^{add} generally overestimates the cost of the optimal relaxed plan.

The FF heuristic takes the exploration done by h^{add} as a basis and extracts a more realistic estimation of the optimal relaxed plan. By taking positive interactions between goal propositions into account FF can reduce the number of operators needed to reach all goal propositions and reduce the cost of the resulting relaxed plan. Therefore the FF heuristic

provides a better approximation of the optimal relaxed plan cost than the overestimating h^{add} so that $h^{FF} \leq h^{add}$.

Essential for this thesis is the fact that FF provides a plan for the relaxed problem. We will make use of the information it provides for backwards goal expansion.

2.5 Concepts from Regression Search

Regression search is the idea to find a plan by starting at the goal and searching backwards until the initial state is reached. It's an old idea and has been discussed for example by Alcázar et al. [2]. The main concepts from regression search that we make use of in this thesis are the process of checking the *legality* of an action and *mutual exclusion*.

Definition 7 (Legal Operator)**.** An operator is called legal if it can act as the last operator o_n in a path $o_0, \ldots, o_n$.

Definition 8 (Mutual Exclusion)**.** Two propositions p_1 and p_2 are mutually exclusive if there exists no reachable state s where both p_1 and p_2 are true.

Definition 9 (Domination)**.** The set of propositions s_x dominates the set of propositions s_y iff $s_x \subseteq s_y$.

These concepts are useful when expanding nodes in the backwards direction, starting from the goal and progressing towards the initial state. We will encounter them when discussing the goal expansion process.

2.6 Backwards Generated Goals

The paper "Using Backwards Generated Goals for Heuristic Planning" by Alcázar et al. [1] forms the basis of the ideas we pursue in this thesis. Their proposal involves using information gained during the computation of a relaxed plan to construct intermediate goals.

Definition 10 (Intermediate Goal)**.** An intermediate goal g is a set of propositions from where a known sequence of operators $o_0, \ldots, o_n$ exists, whose application to g leads to the goal.

Alcázar et al. construct multiple intermediate goals while their heuristic guides them to the closest one. This way they are able to reduce the depth of the relaxed plan computation as well as the forward search. The consideration of delete effects when constructing the intermediate goals can also help to detect and overcome difficulties close to the problem goal that may mislead traditional forward search. After providing an overview of our approach we will touch on the differences between SAGE and Backwards Generated Goals.

3

Single Action Goal Expansion

In this section we will present the functioning of our search algorithm, Single Action Goal Expansion (SAGE), by first introducing the idea behind it and then examining its individual components in detail.

3.1 Idea

Forward search informed by delete relaxation heuristics is one of the most successful approaches to classical planning today. We use this proven paradigm as a starting point and combine it with ideas proposed by Alcázar et al. [1] in an attempt to solve big planning tasks.

The backbone of our approach consists of an eager greedy best-first search using the FF heuristic. On top of that we expand the goal backwards using information provided by FF's relaxed plan. The goal is thereby expanded in a sequential fashion, one action at a time, while the tip of the expansion acts as the goal. The expansion is done in an attempt to both make the goal more reachable for the forward search as well as to detect and circumvent constraints close to the goal.

3.1.1 Overview

The first step of the Single Action Goal Expansion is to decide when the goal should be expanded. The decision of wheter to expand or not is made on every call to the FF heuristic, we examine three strategies to make this decision: NewMinimum, Accuracy and Counter.

Once we have concluded that it's time to expand we collect operators that lead to the goal. This is done by taking all operators that satisfy a goal proposition in the relaxed plan constructed in the current FF computation (Counter does this differently as will be explained in Section 3.3.3).

From these operators we have to choose one to expand the goal with. We present two criteria to sort the operators by: PropositionalLayer and MostSatisfied. The aim here is to get a metric for how useful an expansion with each operator would be and sort

them according to this metric.

With a sorted set of operators in place we check if the operators can legally expand the goal. This check involves three conditions the operator has to fulfill which refer to keeping the goal reachable and avoiding backtracking. We start with the operator we deemed most helpful in the previous step and check the legality of operators until one satisfies all three constraints. The first legal operator found is then used to actually expand the goal.

3.1.2 Relation to Backwards Generated Goals

Our algorithm is based on the approach by Alcázar et al. [1] described in Section 2.6. The key feature we modify is the generation and handling of intermediate goals. We always expand along a single sequence of actions, meaning that new expansions always happen at the goal set constructed during the previous expansion. In contrast Alcázar et al. may generate multiple goal sets leading to the same intermediate goal. This ties in with the fact that their heuristic takes into account multiple intermediate goals and is thus guided towards the closest one. Instead of dealing with multiple goal sets we always calculate the heuristic for the most recent goal set.

With our heuristic leading the search towards a single goal we lose the generality of maintaining multiple goal sets. One factor that can aleviate this drawback is the fact that we generate fewer intermediate goals. Alcázar et al. describe in their paper that intermediate goals are generated on every call of the heuristic and later added to the set of goals when the corresponding state is expanded in the forward search. We avoid the construction of unused intermediate goals by not doing so on every computation of the heuristic. Instead we only generate the intermediate goals we actually expand with. This forces us to make a decision when to expand which is explained in Section 3.3.

The aim of these modifications are to increase efficiency while preserving the benefits of expanding the goal backwards.

3.2 Goal Expansion

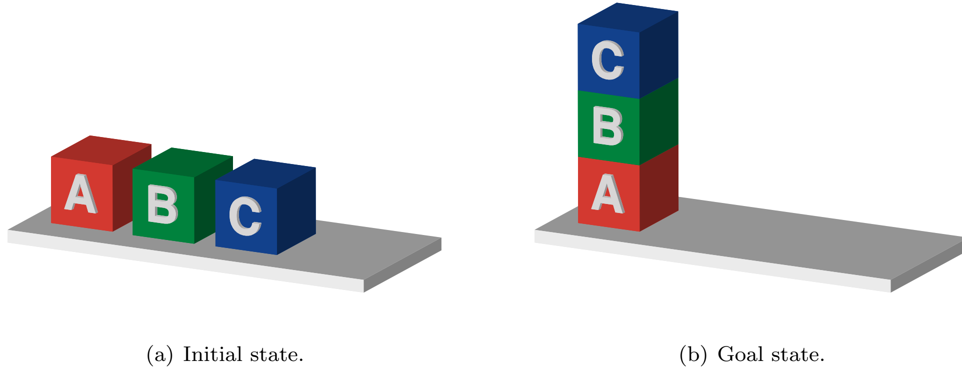
Before we explain the decisions that lead to a goal expansion, we lay out how the goal expansion itself works. In order to illustrate the process we use the Blocksworld problem pictured in Fig. 3.1.

We start out with the goal propositions of the original goal. In our case this is the state shown in Fig. 3.1(b) which is described by the following set of propositions:

$$original_goal = \{(\texttt{on C B}), (\texttt{on B A})\}.$$

Next we need the operator op to expand with. For us that is (`stack C B`) with the following properties:

$$\begin{aligned} pre(op) &= \{(\texttt{clear B}), (\texttt{holding C})\} \\ add(op) &= \{(\texttt{on C B}), (\texttt{clear C}), (\texttt{handempty})\} \\ del(op) &= \{(\texttt{clear B}), (\texttt{holding C})\} \end{aligned}$$



(a) Initial state. (b) Goal state.

Figure 3.1: BLOCKSWORLD example problem with three blocks.

In order to expand the goal, the expanding operator must satisfy at least one goal proposition in the current goal. We confirm that (`on C B`) is indeed element in both *original_goal* and *add*(*op*). With this information we can start building the set of propositions that will constitute the goal after this expansion. We take the propositions of *original_goal* and remove the propositions satisfied by *op*. In our example we get:

$$new_goal = \{\sout{(\texttt{on C B})}, (\texttt{on B A})\}$$

The final step is to add *pre*(*op*) to *new_goal*. Resulting in:

$$new_goal = \{(\texttt{on B A}), (\texttt{clear B}), (\texttt{holding C})\}$$

This set of propositions now describes the state seen in Fig. 3.2. The expansion was successful, (`stack C B`) is applicable in *new_goal* and its application satisfies all propositions in *original_goal*. Indeed the resulting set of propositions is more descriptive than the original goal set with the two added propositions (`handempty`) and (`clear C`), this does not concern us because it still describes a valid goal state. In general we ignore the effects of the expanding operator that don't interfere with propositions from *original_goal*, or any goal set that is expanded upon.

We expand the goal one more time in order to reach a state that we can use to illustrate upcoming concepts. This time we expand with (`pick-up C`):

$$pre(op) = \{(\texttt{clear C}), (\texttt{ontable C}), (\texttt{handempty})\}$$
$$add(op) = \{(\texttt{holding C})\}$$
$$del(op) = \{(\texttt{clear C}), (\texttt{ontable C}), (\texttt{handempty}))\}$$

The *new_goal* from the last expansion is now the *current_goal*:

$$current_goal = \{(\texttt{on B A}), (\texttt{clear B}), (\texttt{holding C})\}$$

After expansion with (`pick-up C`) we get the following goal which is illustrated in Fig. 3.3:

$$new_goal = \{(\texttt{on B A}), (\texttt{clear B}), (\texttt{clear C}), (\texttt{ontable C}), (\texttt{handempty})\}$$

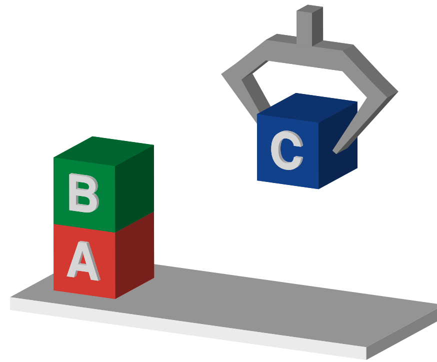


Figure 3.2: Goal after expanding with (`stack C B`).

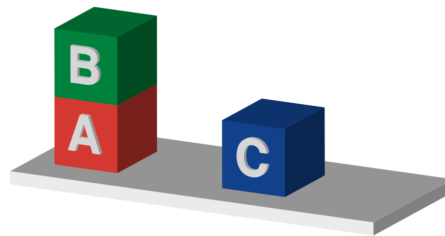


Figure 3.3: Goal after expanding with (`stack C B`) and (`pick-up C`).

3.3 State Decision Process

During a search the number of evaluated states is typically very large. Considering that we are using greedy best-first search, we can expect multiple states to be evaluated for every expanded state. Unlike Alcázar et al. we do not generate an intermediate goal on every evaluation so we have to find a way to choose during the evaluation of which states to expand the goal.

In order to get helpful goal expansions we try to avoid choosing states with high heuristic values as these states are more likely to be further from a solution path. The relaxed plan calculated during the evaluation of these states therefore mark a relaxed path that is less likely to contain operators that are part of a solution. Expanding the goal based on such a state can mislead the search because the resulting expansion is less likely to lie on a possible solution path. The aim of this decision process is thus to detect states that lead us in the direction of the goal in order to improve our chances that the resulting expansion will be beneficial to the forward search.
In the following we explain three strategies that try to detect such states.

3.3.1 New Minimum

The NewMinimum strategy is the simplest of the three. We keep track of the lowest heuristic value we have encountered. We expand the goal whenever we encounter a state

whose heuristic value is lower than any previously seen. The new minimum is stored for future comparisons.

The intuition behind this strategy is that a state with a new minimal heuristic value is taking a step towards the goal. As touched on in the beginning of this section we assume that such a state leads to a relaxed plan that is closer to an actual solution plan and will therefore contain operators that lead to helpful goal expansions.

3.3.2 Accuracy

Accuracy tries to limit backwards expansions to states with an accurate heuristic value. The accuracy is estimated based on the difference between the calculated heuristic value and the heuristic value of the initial state s_0. The difference *diff* for state s with path p from s_0 to s is calculated as follows:

$$diff = abs(\, h(s_0) - h(s) - cost(p)\,)$$

The assumption is therefore that a heuristic value is accurate if the decrease in heuristic value is similar to the cost of reaching s from s_0. To make the decision we have to provide one parameter, the maximum difference $diff_{max} \in \mathbb{N}_0$ with which the goal is expanded so that we expand whenever $diff \leq diff_{max}$.

Accuracy is less greedy than NewMinimum. Instead of naivly looking for small heuristic values it looks for states where the improvement in heuristic value is in line with the generated cost. This means that states which trigger a goal expansion with NewMinimum may not do so with Accuracy because the drop in heuristic value is not justified by the added cost. Yet ultimately both approaches are constrained by the quality of the heuristic.

3.3.3 Counter

The Counter strategy is different than the two previous strategies in that it does not depend on heuristic values at all. Instead it seeks to combine the information of multiple calls to the heuristic function and thus eliminate the risk that comes with having to choose a single state to base the expansion on. This is done by storing all candidate operators (explained in Section 3.4) provided by the relaxed plan. The number of occurances of every operator are added up over the following calculations of the heuristic. Candidate operators that come up in many of the relaxed plans will be considered for goal expansion.

We must specify two parameters to define the behavior of Counter. Firstly the number of evaluations that have to occur before expansion is considered at all, $eval_{min}$, and secondly the percentage of heuristic calls the operator has to be present in to be eligible for expansion, *percentage*. The number of past evaluations is given by $eval_{total}$.

Once $eval_{total} \geq eval_{min}$ we run the following test for all encountered operators o where $count(o)$ is the number of occurances of o in the past evaluations $eval_{total}$:

$$count(o) \geq eval_{total} * percentage$$

The operators that pass this check now form the new candidate operators. Whenever this happens $eval_{total}$ as well as $count(o)$ for every operator o are reset to 0.

3.4 Operator Order

We have reached the point where it is decided that the goal is expanded in the current evaluation. First all operators that will be considered for expansion are collected. In this step we take advantage of information the relaxed plan provides, using it to narrow down our choice of operators.

The basis is the relaxed plan calculated by the FF heuristic. Every goal proposition is reached by a specific action, all of which are stored as candidate operators. In the following we refer to the set of operators that are considered for expansion as *candidates*. Knowing that only one operator of the set will be used to expand, we have to find a way to decide in what order we want to check their legality. We suggest two ways of ordering the candidates that are explained in the following.

To show the difference in practice we refer to the example goal from Fig. 3.3. We assume the given state is the current goal with the propositions:

$$current_goal = \{(\texttt{on B A}), (\texttt{clear B}), (\texttt{clear C}), (\texttt{ontable C}), (\texttt{handempty})\}$$

We assume that we are given two possible operators to expand the goal further. Either (`stack B A`):

$$pre(op) = \{(\texttt{clear A}), (\texttt{holding B})\}$$
$$add(op) = \{(\texttt{on B A}), (\texttt{clear B}), (\texttt{handempty})\}$$
$$del(op) = \{(\texttt{clear A}), (\texttt{holding B})\}$$

or (`put-down C`):

$$pre(op) = \{(\texttt{holding C})\}$$
$$add(op) = \{(\texttt{ontable C}), (\texttt{clear C}), (\texttt{handempty})\}$$
$$del(op) = \{(\texttt{holding C})\}$$

We can easily see that (`stack B A`) would be the better choice than (`put-down C`) in this example.

3.4.1 Most Satisfied

MostSatisfied determines the operator order according to the number of goal propositions each operator satisfies. All operators in *candidates* satisfy at least one, but there may be operators that satisfy multiple goal propositons. The operator that satisfies the most is prefered while ties are broken by the order the operators were added to *candidates*.

The intuition here is that it should be easier to satisfy the preconditions of one operator that in turn satisfies multiple goal propositions than having multiple operators that satisfy the same number of goal propositions. Having to expand several times means having more preconditions from the multiple operators that have to be met in order to reach the expanded goal.

Our example decision would end in a tie for MostSatisfied. The three effects of both (`stack B A`) and (`put-down C`) would be newly satisfied in *current_goal*. In this case the operator that was first added to the candidates by the FF heuristic would be given priority. This means that MostSatisfied could make either the right or wrong decision here.

3.4.2 Lowest Layer

LowestLayer does not only look at the number of satisfied goal propositions but primarily at the expansion layer at which the propositions were added to the goal.

The original goal propositions are defined as having been added on layer 0. After one goal expansion the newly satisfied propositions are removed and the preconditions of the operator used are added to the new goal (as seen in Section 3.2). The added propositions are considered to be on layer 1. Going forth the newly added propositions on the n-th expansion are on the n-th layer.

LowestLayer prefers the operator that satisfies the goal proposition that was added on the lowest layer. Ties are broken by choosing the operator that satisfies more propositions at this lowest layer, if still equivalent the operator that satisfies more propositions in the next higher level is chosen and so on. When the number of satisfied propositions are equal on all layers the operator that was first added to *candidates* prevails.

The intuition behind this strategy is that propositions that were added on lower levels should be more difficult to satisfy, thus operators that do so are considered first.

With LowestLayer the decision is clearer for this example. The three propositions satisfied with (`put-down C`) have all been added during the previous expansion, they are all on the highest layer. In contrast (`on B A`) which is satisfied by (`stack B A`) has already been a goal proposition in the original goal, meaning it is on layer 0. Therefore LowestLayer would prefer (`stack B A`) over (`put-down C`) which is the smarter choice in this example.

Algorithm 1: Check Goal Expansion

```
Data: Considered operators candidates
Result: true if goal was expanded, false otherwise.
1 candidates_sorted ⟵ sort_operators(candidates)
  foreach op ∈ candidates_sorted do
2     if is_legal(op) then
          expand_with(op)
          return true
  return false
```

Algorithm 1 shows the procedure after initiating a goal expansion. The collected candidates are ordered on Line 1 by either MostSatisfied or LowestLayer. After that we check the

legality of every candidate on Line 2, starting with the most preferred one. When a legal operator is found we expand the goal and return `true`, if none of the candidates are legal we return `false`. The Process of checking the legality of an operator is described in the following section.

3.5 Check Legality

In this step we are given a candidate operator which has to be checked for legality. This is done in an attempt to ensure that the goal resulting from expanding it with this operator is reachable. A second task of this function is to determine whether the expansion would actually advance the backwards propagation and not loop back to a previous intermediate goal. Algorithm 2 gives an overview of the steps taken. In the following we elaborate on the three necessary checks while referring to the associated lines in Algorithm 2.

Algorithm 2: Legal Operator

Data: Operator to be checked *op*
Set of current goal propositions g_{curr}
List of previous goal sets G_{ex}
Result: `true` if the operator is legal, `false` otherwise.
1 **if** $del(op) \cap g_{curr} \neq \emptyset$ **then**
  return `false`
2 *remaining_propositions* ⟵ Set of propositions in g_{curr} **not** supported by *op*
3 **if** `is_mutex`(*pre*(*op*), *remaining_propositions*) **then**
  return `false`
4 $g_{new} \longleftarrow \{ g_{curr} \setminus supported_propositions \} \cup pre(op)$
5 **foreach** $g_{ex} \in G_{ex}$ **do**
  if `dominates`(g_{ex}, g_{new}) **then**
    return `false`
return `true`

3.5.1 Deletes Goal Proposition

The first constraint, seen on Line 1, is that the delete effects of operator *op* may not contain any propositions present in the current goal g_{curr}. Having an existing goal proposition deleted would make the final plan invalid in one of two ways. Either we reach the end of the plan with a goal proposition deleted, or an operator from an expansion we have to backtrack through is not applicable anymore when trying to execute the plan. Therefore it is essential that no expanding operator deletes goal propositions from the current goal set.

3.5.2 Mutex

The second constraint deals with the relation between the newly to be added propositions and the propositions that remain unsatisfied from the current goal. As shown on Line 2 the remaining propositions have to be collected, meaning the propositions present in the current goal that are not satisfied by *op* and would thus remain part of the new goal if we were to expand with *op*. What is tested is whether any combination of one proprosition of $pre(op)$ and one proposition of $remaining_propositions$ are mutually exclusive (see Definition 8). This happens on Line 3. If that is the case, the goal set resulting from the expansion with *op* would be unreachable, because per definition the two propositions that are mutually exclusive could never be satisfied in the same state.

3.5.3 Dominated by Ex Goal

This last constraint ensures that we do not loop back to a previously encountered goal set. Such an expansion would not benefit us for two reasons. The fact that the same goal set has been encountered in an earlier goal expansion suggests that it is closer to the original goal than our current goal set. Seeing that our aim is to expand the goal towards a more reachable state for the forward search by expanding in its direction, this step would not make sense as it would bring us closer to the original goal again. The second and more critical behaviour this check avoids is expansion in a loop. If we were to allow expansions to previous goal sets it is possible that the expansions could cycle through a number of previously encountered goal sets and thus render the expansions useless.

Although a check on whether the new goal set is equal to an ex goal set is already beneficial we can make the constraint stronger by testing for domination (see Definition 9) instead. The new goal set is not allowed to be dominated by any ex goal set. To demonstrate the use of this we consider the state shown in Fig. 3.2 with the following propositions:

$$current_goal = \{(\texttt{on B A}), (\texttt{clear B}), (\texttt{holding C})\}$$

We are considering expansion with (`unstack C B`):

$$pre(op) = \{(\texttt{on C B}), (\texttt{clear C}), (\texttt{handempty})\}$$
$$add(op) = \{(\texttt{clear B}), (\texttt{holding C}), (\texttt{handempty})\}$$
$$del(op) = \{(\texttt{on C B}), (\texttt{clear C}), (\texttt{handempty})\}$$

This would lead us to the new goal set:

$$new_goal = \{(\texttt{on C B}), (\texttt{on B A}), (\texttt{clear C}), (\texttt{handempty})\}$$

This goal set is now dominated by the original goal of the problem. We are essentially in an equivalent state to the original goal, and have merely added (`clear C`) and (`handempty`) to the propositions. These could not have had any other value when the goal is reached but were not explicitly specified in the original goal. Therefore it makes sense to not allow this expansion because it would only lead back to the original goal.

When an operator passes these three constraints, we consider it as legal and it will be used to expand the goal.

3.6 Safety Features

The approach we examine in this thesis comes with its difficulties. These mostly stem from the fact that, in an attempt to gain efficiency, we trade in the generality given when using Backwards Generated Goals. The forward search is always guided towards the single current goal instead of the multiple intermediate goals as in the approach of Alcázar et al. Being so specific leads to problems because we cannot guarantee that the current goal set is reachable from the initial state, even if the original goal was. While the legality check tries to ensure that goal expansions are reachable it cannot do so with certainty. An example that illustrates this is given in Section 4.1.2.

The following mechanisms have been added in an attempt to avoid rendering tasks unsolvable.

3.6.1 Blocking Operators

The idea of blocking operators is to try to detect when we have expanded to an unreachable goal set. We cannot do this directly but we make the assumption that the current goal is unreachable when we try to expand it further but none of the candidate operators are legal.

When this occurs the last goal expansion is rolled back, the goal that we deemed unreachable is replaced by the goal set preceeding it and the operator used in this expansion is marked as blocked. The blocking of the operator prevents it from being used for the same expansion again. Yet the block only applies to the layer the expansion occured in, meaning that the operator will be considered again in case the goal is either expanded using a different operator or rolled back another layer.

In case we undo all goal expansions with no unblocked and legal operators available we stop trying to expand the goal.

This optimization is not used when SAGE is ran with the COUNTER option. That is because the selection of possible operators per expansion is already limited by COUNTER while NEWMINIMUM and ACCURACY both use all operators that satisfy a goal proposition in the relaxed plan of FF. We do not block operators and undo expansions based on the limited operator selection provided by COUNTER.

3.6.2 Original Goal Check

We check whether our forward search finds the original goal despite being guided towards an expanded goal. This is an attempt to find a solution even when the search is mislead by an unreachable goal set. Small problems can be more prone to such a situation. When the goal expansion doesn't effectively expand towards the goal, the forward search may reach the original goal first.

3.6.3 Initial State Check

In the same spirit as the previous check we test whether the expanded goal is satisfied in the initial state. This can happen when a new goal set crosses into the already explored part of a forward search. Such a scenario may be salvageable in small problems if the backwards

expansion reaches the initial state before the complete state space has been explored, in that case this check would fire and return a plan.

4
Evaluation

The implementation of the presented algorithms was done in the Fast Downward planning system by Helmert [5]. The goal of the experiments was to compare the performance of the SAGE algorithm with its different settings.

The experiments were run on a benchmark set of 1827 tasks. The domains are a collection from past International Planning Competitions (IPC). The time limit was set to 30 minutes and the memory limit to 3.5 GB.

For Accuracy we set the maximum difference to be expanded to 0. This decision was made based on smaller scale, preliminary experiments that suggested that 0 performs better than a higher threshold. This setting puts the strongest constraint on the number of goal expansions, any parameter higher than 0 would lead to more backwards expansions.

The parameters used for counter are 25 for the minimum number of evaluations and 90% for the minimum ratio of occurances. The percentage was set high to ensure that only frequent operators are considered and the minimum number of evaluations was chosen to constrain the selection of operators while still allowing goal expansions.

4.1 Results

In Table 4.1 we can see a comparison of coverage between the different combinations of SAGE settings. The coverage of eager greedy using the FF heuristic is shown as a baseline.

While the coverage of NewMinimum and Accuracy are comparable, Counter performs significantly worse.

We think that the number of goal expansions shown in Table 4.2 may be an indicator for where the difference in performance comes from. Counter expanded the goal more than 10 times as much as NewMinimum and Accuracy. We have already seen in Section 3.6 that our approach to goal expansion has a chance of rendering the task unsolvable by backwards expanding into goal sets that are not reachable for the forward search. We assume that the high number of goal expansions in Counter aggravate this issue by giving the algorithm more chances to expand to an unreachable goal.

Coverage (Total 1827)	NewMin	Accuracy	Counter	Eager Greedy (FF)
MostSatisfied	1051	1165	727	1503
LowestLayer	1027	1184	726	

Table 4.1: Total coverage of different setting combinations.

Goal Expansions Total	NewMin	Accuracy	Counter	Eager Greedy (FF)
MostSatisfied	11699	7338	108636	0
LowestLayer	11986	6488	189694	

Table 4.2: Total number of goal expansions accross problems solved by all approaches.

While the number of goal expansions is likely to be connected to the the performance of Counter it can also be viewed as a symptom of poorly chosen parameters. Enforcing a higher percentage as well as more minimal evaluations should lead to a decreased number of goal expansions and may thus improve Counter's coverage. The comparison between the different settings would have certainly been more meaningful with more comparable goal expansion numbers.

Another possible explanation is the fact that we chose to disable the operator blocking mechanism for Counter. Despite our initial doubt it may make sense for Counter also. This and/or a different set of parameters could definitely have the potential to yield a version of Counter that can close the gap to the other two state decision strategies.

A second point of view on the coverage numbers is to compare MostSatisfied with LowestLayer accross the different state decision options. We can see that there is very little variance between the two operator orderings. This invariance persists when comparing the numbers for individual domains in Table 4.4. Within the three state decision pairs the difference in coverage hardly exceeds one or two problems for any given domain. In the context of the experiments we ran the similarities are so consistant that we can say that MostSatisfied and LowestLayer perform essentially the same.

4.1.1 Forward Expansions in Suited Domains

In this section we compare the number of forward expansions for domains SAGE performs well in with eager greedy. Table 4.3 shows the geometric mean of forward expansions over a hand-picked selection of domains. We selected domains where at least one of the SAGE algorithms can match or in a few cases exceed the coverage of eager greedy. From this collection we can see that we can solve suitable problems with significantly fewer forward expansions than eager greedy. Especially NewMinimum performs rather well in this regard.

We believe that NewMinimum can achieve better results here than Accuracy because it performs more goal expansions in general (as seen on Table 4.2). The unusually good coverage of SAGE in these domains suggests that they are more suitable for our approach than other domains. For such domains it seems to be beneficial to expand more than Accuracy does.

Forward Expansions	NewMin	Accuracy	Counter	Eager Greedy (FF)
blocks (32)	**112.70**	444.76	525.74	464.76
driverlog (14)	207.41	63.29	**40.21**	115.88
elevator-opt08-strips (19)	**1379.44**	2691.59	2154.87	3467.17
elevator-opt11-strips (12)	1462.71	2691.59	**925.79**	3516.75
gripper (17)	134.88	192.99	**125.87**	380.28
logistics00 (26)	153.92	34.04	**25.35**	42.20
miconic (130)	56.14	50.79	**28.32**	68.54
rovers (16)	111.78	**104.46**	153.03	225.05
scanalyzer-08-strips (23)	65.04	84.06	**29.25**	89.42
scanalyzer-opt11-strips (15)	71.71	45.73	**19.00**	48.49

Table 4.3: Geometric mean of forward expansions for problems solved by all variants per domain. The operator order is LowestLayer.

We suspect that Counter on the other hand performs too many goal expansions while suffering from the problems pointed out in the beginning of Section 4.1 and thus does not perform as well as NewMinimum. Interestingly Counter seems to perform exceptionally well in our example domain Blocksworld, using significantly less forward exansions than any other configuration.

Overall Table 4.3 serves to show that our approach does manage to preserve some benefit from the goal expansion when used on suitable domains. We can therefore say that our goal of reducing the depth of the forward search by expanding the goal can work on suitable domains. The next interesting question to ask would be what the factors are that make a domain suitable. Unfortunately we were not able to find a specific property that unifies the empirically determined “suitable” domains.

4.1.2 Openstacks

The Openstacks domains pose an interesting anomaly in the results. As can be seen in Table 4.4, SAGE performs exceptionally badly in these domains. So much so that most configurations solve no problems at all. We will look at an example problem from these domains to understand and illustrate a main weakness of our approach and why Openstacks highlights it well. Specifically we will work with the notation given in `openstacks-opt08-strips`.

An Openstacks task consists of a stack, orders and products. The goal is to ship all orders. This is done by placing orders on the stack (starting the order), making the products the order contains and then shipping the completed order. A product can only be made when all orders this product is part of are on the stack and an order can only be shipped when all its products have been made. The stack starts out with a capacity of 0. This capacity can be extended by opening a new stack and persists once increased, meaning that starting and shipping orders only affects the stack capacity temporarily.

In our example problem we have two orders `o1` and `o2` which both require a single product `p1`. The available stack is given by `n0`, `n1` or `n2` so that the number indicates how

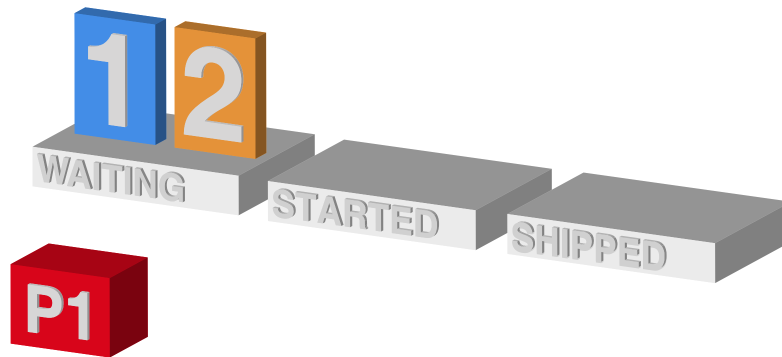


Figure 4.1: The initial state of our example OPENSTACKS problem. The product `p1` is red to indicate that it is currently not made.

many spaces are available on the stack. The initial state is shown in Fig. 4.1 and defined by the following propositions:

{(`stacks-avail n0`), (`waiting o1`), (`waiting o2`), (`not-made p1`)}.

To get an understanding for the problem and the available operators we present a solution. This is a possible plan:

```
(open-new-stack n0 n1)
(open-new-stack n1 n2)
(start-order o1 n2 n1)
(start-order o2 n1 n0)
(make-product p1)
(ship-order o2 n0 n1)
(ship-order o1 n1 n2)
```

The (`nx ny`) at the end of operators refers to the change of available space on the stack from `x` to `y`. For example (`ship-order o1 n1 n2`) in our solution increases the number of available spaces on the stack from 1 to 2 because one space was already available after `o2` was shipped and now the second space is available as well.

In order to illustrate the problems that arise when expanding the goal we must first examine the original goal set. The goal set pictured in Fig. 4.2 is given by the two propositions (`shipped o1`) and (`shipped o2`). The questionmarks in the figure refer to the fact that the original goal does not make a statement about how many spaces are available on the stack or whether `p1` is made or not. Now we try to expand the goal. There are four operators that satisfy a goal proposition: (`ship-order o1 n0 n1`), (`ship-order o1 n1 n2`), (`ship-order o2 n0 n1`) or (`ship-order o2 n1 n2`). All these are considered legal by our legality check. For our example it doesn't matter which of the two orders is chosen, so we assume `o1`. Having narrowed down the choices to two operators we further assume

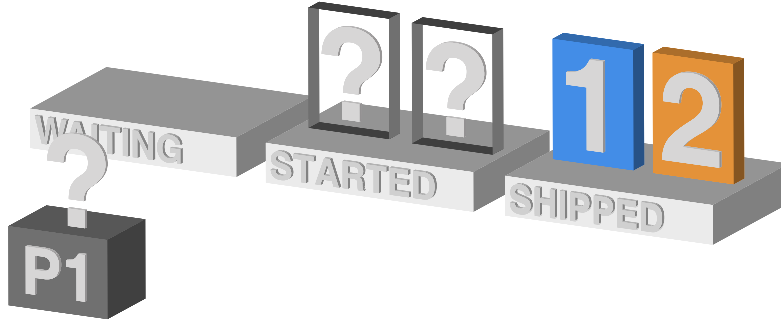


Figure 4.2: The goal of our OPENSTACKS example problem. The questionmark and gray box for `p1` indicate that the goal does not make any statement about `p1`. The questionmarks and dark frames in "started" signify that no statement is made about the available spaces on the stack.

that the FF heuristic returns $op =$ (`ship-order o1 n0 n1`) as the operator that satisfies (`shipped o1`):

$$pre(op) = \{(\texttt{started o1}), (\texttt{made p1}), (\texttt{stacks-avail n0})\}$$
$$add(op) = \{(\texttt{shipped o1}), (\texttt{stacks-avail n1})\}$$
$$del(op) = \{(\texttt{started o1}), (\texttt{stacks-avail n0})\}$$

Let's expand the goal with *op*. The following is our new goal set (depicted in Fig. 4.3):

$$new_goal = \{(\texttt{shipped o2}), (\texttt{started o1}), (\texttt{made p1}), (\texttt{stacks-avail n0})\}$$

The critical thing that happens here is the statement (`ship-order o1 n0 n1`) makes about the available space on the stack. In *new_goal* the `o2` has to be shipped *and* there have to be 0 spaces available on the stack. The only possible way to solve this problem is to ship one order after the other as the two last steps because they must both have been started in order to make *p*1 and thus shipping both are the only remaining steps to be taken. It is now impossible that we reach *new_goal* while using an operator to ship `o2`. The two possible operators are (`ship-order o2 n0 n1`) or (`ship-order o2 n1 n2`) the *add* effects of these contain (`stacks-avail n1`) or (`stacks-avail n2`) respectively. Thus the application of either of those cannot lead to *new_goal* where (`stacks-avail n0`) holds true.

What we learn from this example is that an expansion with a operator we deemed legal can still lead to an unsolvable goal. Our assumption that FF would give us (`ship-order o1 n0 n1`) or (`ship-order o2 n0 n1`) as the operator to expand with is based on small scale investigative tests where this was the most frequent outcome. In this domain the problem of entering an unsolvable goal on the first goal expansion tends to persist over many further expansions. This is because there are many legal expansions that can be done with the goal sets remaining unsolvable. While blocking operators would eventually find its way out of this cycle, the state space in exhausted before enough goal expansion attempts have been made to explore every possible expansion of further unreachable goals.

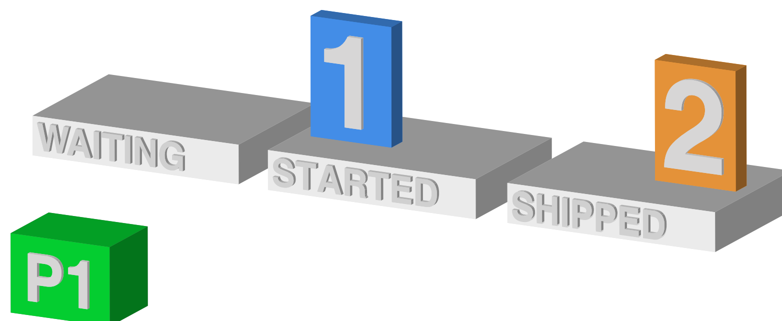


Figure 4.3: Goal after expanding with (`ship-order o1 n0 n1`). The undefined aspects from the original goal are now explicit, product $p1$ has to be made and there is 0 stack space available.

Coverage accross domains	Counter, LowestLayer	Counter, MostSatisfied	Accuracy, LowestLayer	Accuracy, MostSatisfied	NewMin, LowestLayer	NewMin, MostSatisfied	Eager Greedy
agricola-opt18-strips (20)	0	0	0	0	0	0	**20**
airport (50)	6	6	14	16	12	13	**36**
barman-opt11-strips (20)	3	3	17	17	0	0	**20**
barman-opt14-strips (14)	3	4	10	10	0	0	**14**
blocks (35)	32	32	**35**	**35**	**35**	**35**	**35**
childsnack-opt14-strips (20)	2	1	3	2	6	6	**7**
data-network-opt18-strips (20)	7	9	14	12	11	11	**16**
depot (22)	6	7	12	14	9	12	**16**
driverlog (20)	15	15	19	18	**20**	**20**	18
elevators-opt08-strips (30)	25	21	**30**	**30**	27	29	**30**
elevators-opt11-strips (20)	15	13	**20**	**20**	18	**20**	**20**
floortile-opt11-strips (20)	**11**	9	9	9	9	9	9
floortile-opt14-strips (20)	**14**	9	12	11	12	11	11
freecell (80)	5	4	5	3	4	8	**79**
ged-opt14-strips (20)	10	10	**20**	**20**	**20**	**20**	**20**
grid (5)	**5**	4	4	4	4	4	4
gripper (20)	19	17	**20**	**20**	**20**	**20**	**20**
hiking-opt14-strips (20)	15	15	**20**	**20**	**20**	**20**	**20**
logistics00 (28)	26	**28**	**28**	**28**	**28**	**28**	**28**
logistics98 (35)	7	9	**30**	**30**	29	**30**	29
miconic (150)	130	130	**150**	**150**	**150**	**150**	**150**
movie (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**
mprime (35)	8	10	23	24	13	15	**31**
mystery (30)	8	7	13	12	7	6	**17**
nomystery-opt11-strips (20)	2	0	12	11	0	0	**15**

openstacks-opt08-strips (30)	0	0	0	0	0	0	**8**
openstacks-opt11-strips (20)	0	0	0	0	0	0	**3**
openstacks-opt14-strips (20)	**0**	**0**	**0**	**0**	**0**	**0**	**0**
openstacks-strips (30)	0	0	0	0	26	26	**28**
organic-synthesis-opt18-strips (20)	**7**	**7**	**7**	**7**	**7**	**7**	**7**
organic-synthesis-split-opt18-strips (20)	8	8	**20**	**20**	5	7	**20**
parcprinter-08-strips (30)	3	3	7	7	0	0	**23**
parcprinter-opt11-strips (20)	0	0	5	5	0	0	**17**
parking-opt11-strips (20)	0	0	15	16	15	14	**20**
parking-opt14-strips (20)	0	0	15	16	8	10	**20**
pathways-noneg (30)	6	5	5	4	2	1	**9**
pegsol-08-strips (30)	4	4	29	29	24	25	**30**
pegsol-opt11-strips (20)	1	1	19	19	18	18	**20**
petri-net-alignment-opt18-strips (20)	6	5	12	11	10	14	**20**
pipesworld-notankage (50)	11	13	16	14	14	14	**31**
pipesworld-tankage (50)	8	11	17	16	15	14	**23**
psr-small (50)	47	48	**50**	**50**	48	**50**	**50**
rovers (40)	24	21	31	27	**35**	32	26
satellite (36)	8	12	19	21	24	25	**27**
scanalyzer-08-strips (30)	27	25	28	29	29	**30**	28
scanalyzer-opt11-strips (20)	17	16	**20**	**20**	19	**20**	**20**
snake-opt18-strips (20)	2	2	**15**	**15**	14	**15**	**15**
sokoban-opt08-strips (30)	12	14	26	26	24	24	**28**
sokoban-opt11-strips (20)	8	9	18	18	15	15	**20**
spider-opt18-strips (20)	0	0	3	4	0	0	**18**
storage (30)	14	16	18	17	19	**21**	19
termes-opt18-strips (20)	14	14	**19**	**19**	**19**	**19**	**19**
tetris-opt14-strips (17)	5	2	14	14	**15**	**15**	**15**
tidybot-opt11-strips (20)	2	2	13	10	1	1	**18**
tidybot-opt14-strips (20)	0	0	12	7	1	0	**19**
tpp (30)	13	12	17	17	17	19	**23**
transport-opt08-strips (30)	12	13	**23**	**23**	15	15	22
transport-opt11-strips (20)	7	7	**20**	**20**	16	15	19
transport-opt14-strips (20)	4	6	**20**	**20**	9	10	**20**
trucks-strips (30)	0	0	6	6	0	0	**14**
visitall-opt11-strips (20)	19	19	**20**	**20**	**20**	**20**	**20**
visitall-opt14-strips (20)	13	13	**19**	**19**	17	17	**19**
woodworking-opt08-strips (30)	6	7	21	20	15	14	**30**
woodworking-opt11-strips (20)	2	3	15	15	9	9	**20**
zenotravel (20)	12	16	**20**	18	18	18	**20**
Sum (1827)	726	727	1184	1165	1027	1051	**1503**

Table 4.4: The coverage of all setting combinations per domain.

5 Conclusion

Although the evaluation section hasn't ended on a optimistic note, we were able to learn something about the chances and challenges of the SAGE algorithm. We found a small selection of domains where we could document minor improvements over eager greedy as well as show that the aim of reducing depth for the forward search is possible under the right conditions.

Our experiments give an overview on the diffirences between the various decision strategies which were in general smaller than we expected. It was surprising to see that all four combinations of NewMinimum and Accuracy showed such similar results in total coverage. More extensive experimentation with a wider range of parameters could certainly shed more light on the potential of the different approaches.

Instead of experimenting on the concepts touched on in this thesis it may be more interesting to explore more radical ways of dealing with the problems of our approach. One possible solution to handle the biggest problem, the one of unsolvable goal sets, is to restart the forward search upon goal expansion. We have made attempts to test such a process but were unable to build a working version due to time constraints. If we could invest more time into this approach, we would further pursue this line of thought to examine whether the added overhead of multiple forward searches could be amortized by the improved safety of the goal expansion.

Besides the conceptual improvements, the algorithm could also benefit from a cleaner, more efficient implementation.

In conclusion the SAGE algorithm in its current form poses many unanswered questions. While its performance is currently no match against an algorithm such as eager greedy with FF, there are many areas where improvements could be made. We believe that a smart strategy to deal with unreachable goal sets in conjunction with a more in depth analysis of the different decision options could significantly increase performance. It would be interesting to see this idea develop further.

Bibliography

[1] Vidal Alcázar, Daniel Borrajo, and Carlos Linares López. Using backwards generated goals for heuristic planning. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, 2010.

[2] Vidal Alcázar, Daniel Borrajo, Susana Fernández, and Raquel Fuentetaja. Revisiting regression in planning. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

[3] Blai Bonet, Gábor Loerincs, and Héctor Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of the Fourteenth National Conference of the American Association for Artificial Intelligence (AAAI-97)*, pages 714–719, 1997.

[4] Richard Fikes and Nils Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

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

[6] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.