Introducing Explicit-Symbolic Planning

MSc Thesis *(Afstudeerscriptie)*

written by

Minke Verweij
(born October 8th, 1997 in Amsterdam, The Netherlands)

under the supervision of **Dr Gregor Behnke**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam*.

Date of the public defense:
June 24, 2025

Members of the Thesis Committee:
Dr. Aybüke Özgün (chair)
Dr. Gregor Behnke (supervisor)
Dr. Malvin Gattinger
Dr. Ronald de Haan

Institute for Logic, Language and Computation

Abstract

This thesis investigates two dominant planning approaches, heuristic based explicit search and symbolic search, and explores their limitations. While heuristics can guide explicit search effectively, they may still lead to exponential expansion in certain problem instances. Conversely, symbolic search can compactly explore large state spaces but may do exponentially worse when heuristics are applied. To address these limitations, we propose a novel hybrid search algorithm that combines the strengths of both methods. We demonstrate that this hybrid approach solves a specific class of planning problems efficiently, where the other described strategies fail. Furthermore, we initiate an analysis of the challenges in hybrid implementation, particularly the difficulty of splitting the variables into a symbolic and an explicit set. Our work strengthens some prior limitation results and extends the frontier of tractable planning problems via the newly proposed hybrid search algorithm.

Contents

1 Introduction

Automated planning is the study of algorithms, systems and representation languages that are able to generate sequences of actions to achieve certain goals. Planning requires intelligence. To be able to plan, one needs to oversee the consequences of performing actions and to use that information to deliberate which steps will be most beneficial to achieve one's goals. Thus, studying planning can give insights into this reasoning side of intelligence.

Moreover, studying planning is necessary to actually build autonomous entities. Automated planning research has been applied in robot control [14], for example in space exploration [9] and autonomous driving systems [10]. It has also been applied production and logistics [21] and construction [19] and many more areas.

Actual implementations of autonomous systems will be rooted in their specific domain. For example, a GPS route planner uses satellite data and its internal algorithms will not immediately translate to other domains. However, in classical planning we look at problems from a domain-independent perspective. That is, we use a generic representation of problems such that we can apply solution strategies to any planning problem that fits our representation. While this general approach does not immediately produce state-of-the-art robotic implementations, it helps to understand the underlying complexity of planning problems. The resulting insights can subsequently be used to improve domain specific implementations.

To be able to reason about planning problems generally, we use standardized formalizations such as the propositional STRIPS introduced by Fikes and Nilsson in 1971 [8] and finite domain reprentation (FDR) introduced by Bäckström and Nebel in 1995 [1]. Both formalisms make restricting assumptions about possible domains. For example that there are only finitely many possible states and that actions are deterministic. Even so, planning is hard, namely, deciding whether a problem instance has a plan is PSPACE complete [4, 1].

1.1 Our Contributions

Given the complexity of planning, there is likely not a single algorithm that will solve all problems efficiently. Two widely used strategies are heuristic based explicit search and symbolic search. In the former we first estimate the cost of reaching goals from a given state and use these estimates to inform our search. So, if we expect it to be costly to reach a goal from a certain state, we defer exploring it. With this strategy the search states are fully and explicitly specified and we always consider a single state at a time. This is in contrast to symbolic search wherein we use concise representations of sets of states to find plans.

Both these strategies have their strengths as well as their drawbacks. Helmert and Röger have shown that there are instances where even very good heuristics may still result in exponentially many expanded explicit

states even when optimal plans are linear in length [12]. Moreover, Speck, Geißer and Mattmüller show that using heuristics in symbolic search can exponentially worsen the search effort in symbolic search [22].

In this thesis we will further investigate the qualities of these known search strategies, and strengthen the aforementioned results. We will use this information to motivate a new hybrid algorithm that uses a partially symbolic and partially explicit heuristic based search approach.

In Figure 1, we visualize this contribution. In the box we have all possible planning problems. Problems in the circle labeled Expl. on the left, can be solved efficiently using explicit search and heuristic methods. For those in Symb., we can find plans efficiently using symbolic search. In their intersection both strategies are good. We want to add a class of problems in the striped hybrid section. That is, to show that the new hybrid algorithm can efficiently solve problems that are difficult for both other strategies. Additionally, we want the hybrid algorithm to adopt all positive instances of both the explicit and symbolic strategies. So, as we can see in Figure 1, the hybrid 'pattern' encompasses the complete area of tractable problems.

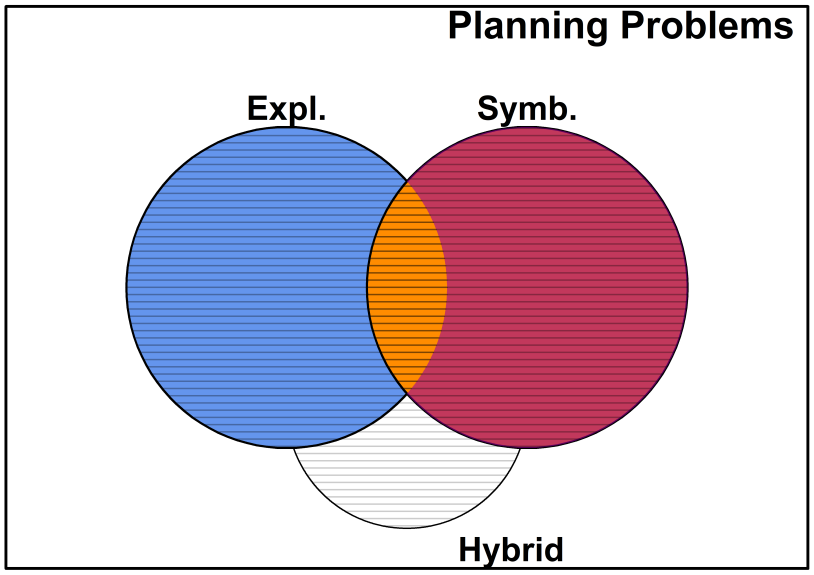

Figure 1: Visualization of Contribution

While we provide a problem where hybrid search is good, we also begin an investigation into some of the difficulties in its implementation.

1.2 Outline of the Thesis

The thesis is structured as follows:

- Chapter 2: Defining planning formalisms and introducing the main search strategies heuristic and symbolic search

- Chapter 3: Introducing the new hybrid algorithm as well as a problem wherein we show the hybrid algorithm is guaranteed to outperform other search strategies.
- Chapter 4: Preliminary investigation into some of the difficulties of using the hybrid algorithm, specifically the allocation of the variables.
- Chapter 5: Conclusion and discussion

2 Preliminaries

2.1 Planning Problems

We will first introduce planning problems formally. We will do so using finite domain representation (FDR) as introduced by Bäckström and Nebel [1].

Definition 2.1. An *FDR planning problem* is a tuple $\Pi = (V, A, c, \mathcal{I}, \mathcal{G})$ where:

- V is a finite set of variables such that each $v \in V$ has a finite domain D_v of values that the variable can take on. We call (partial) functions on V that map each $v \in V$ to an element in D_v *(partial) variable assignments*.
- A is a finite set of actions. Each action $a \in A$ is a tuple $\langle pre_a, eff_a \rangle$ of preconditions and effects which are both partial variable assignments such that there is no $v \in V$ for which we have $pre_a(v) = eff_a(v)$.
- $c : A \to \mathbb{R}_0^+$ is a cost function. We call a problem *unit cost* if for all $a \in A$ we have $c(a) = 1$.
- $\mathcal{I}$ is a complete variable assignment that we call the *initial state*.
- $\mathcal{G}$ is a partial variable assignment that we call the *goal*.

Notation 2.2. We call a variable value pair (v, d) (also write $v = d$) a *fact*, where $v \in V$ and $d \in D_v$. If $D_v = \{\bot, \top\}$ we write v and $\neg v$ instead.

Given a partial assignment p, we can refer to the variables assigned by p as $\mathcal{V}(p) := \{v \in V \mid p(v) \text{ is defined}\}$.

Semantically, we can think of planning problems as state transition systems.

Definition 2.3. Each FDR planning problem $\Pi = (V, A, c, \mathcal{I}, \mathcal{G})$ has an associated *FDR state space* which is a labeled transition system $\Theta_\Pi = (S, L, c, T, \mathcal{I}, S^\mathcal{G})$ where:

- S is the set of complete variable assignments or *states*.
- $L = A$ is the set of transition labels.
- c is the cost function from Π
- T are the transitions where for a pair of states $s, s' \in S$ and an action $a \in A$ with cost $c(a)$ we have $s \to^a s' \in T$ when $a \in A[s]$ and $tr(s, a) = s'$ where:
  - $A[s] = \{a \in A \mid pre_a \subseteq s\}$ is the set of actions *applicable* in s.

– $tr(a, s)$ is the successor state of s after applying a, where for all $a \notin A[s]$ we define $tr(s, a) := \bot$.

Otherwise, when a is applicable in s, we define for every variable $v \in V$ that

$$tr(s,a)(v) := \begin{cases} \mathit{eff}_a(v) & \text{for } v \in \mathcal{V}(\mathit{eff}_a) \\ s(v) & \text{for } v \notin \mathcal{V}(\mathit{eff}_a) \end{cases}$$

- $\mathcal{I}$ is the initial state as in Π.
- $S^{\mathcal{G}} = \{s \in S \mid \mathcal{G} \subseteq s\}$ is the set of *goal states*.

Given the transition function, we can also refer to partial transition functions on a subset of the variables.

Notation 2.4. Given the *transition function* $tr : S \times A \to S \cup \{\bot\}$ where S the set of states and A the set of actions as defined in Def 2.3, we can also define partial transition functions:[1] for $V' \subseteq V$, we have $tr_{V'} : S_{V'} \times A \to S_{V'} \cup \{\bot\}$ where $S_{V'}$ is the set of V'-partial states or the complete assignments of V'. Then $tr_{V'}(s, a) := \bot$ whenever $\mathit{pre}_a \restriction_{V'} \not\subseteq s$. Otherwise $tr_{V'}(s, a) = tr(s, a) \restriction_{V'}$

Finally, we can define a solution to a planning problem as follows:

Definition 2.5. Given a planning problem $\Pi = (V, A, c, \mathcal{I}, \mathcal{G})$ a *plan* π is a finite sequence of actions $\langle a_1, ..., a_n \rangle \in A^*$ such that $\mathcal{I}[\![\langle a_1, ..., a_n \rangle]\!] \in S^{\mathcal{G}}$ where inductively:

- $s[\![\langle\rangle]\!] := s$
- $s[\![\langle a, \vec{a} \rangle]\!] := tr(s, a)[\![\langle \vec{a} \rangle]\!]$ for $a \in A$ and $\vec{a} \in A^*$ and $\bot[\![\langle \vec{a} \rangle]\!] := \bot$ for all $\vec{a} \in A^*$

The cost of a plan $\pi = \langle a_1, ..., a_n \rangle$ is the sum of the costs of all actions in the plan:[2] $\Sigma_{i=1}^{n} c(a_i)$. We call a plan *optimal* if among all possible plans its cost is minimal.

Example 2.6. A very simple logistics planning problem is $\Pi = (V, A, c, I, G)$ with $V = \{t, p\}$ variables representing a truck t and a package p, with $D_t = \{l_1, l_2\}$ and $D_p = \{l_1, l_2, t\}$. For each distinct $x, y \in \{l_1, l_2\}$ we have the following actions in A: $\mathit{drive}(x, y) = \langle \{(t, x)\}, \{(t, y)\} \rangle$ that represents the truck driving from one location to another; $\mathit{load}(x) = \langle \{(t, x), (p, x)\}, \{(p, t)\} \rangle$ and $\mathit{unload}(x) = \langle \{(t, x), (p, t)\}, \{(p, x)\} \rangle$ that represent the respective loading and unloading of the package into or out of the truck. Each action has cost one. The initial state is $I = \{(t, l_1), (p, l_1)\}$ and the goal states $G = \{(p, l_2)\}$.

A plan for this problem is:

$\langle \mathit{drive}(l_1, l_2), \mathit{drive}(l_2, l_1), \mathit{load}(l_1), \mathit{drive}(l_1, l_2), \mathit{unload}(l_2), \mathit{drive}(l_2, l_1) \rangle$.

An optimal plan for this problem is: $\langle \mathit{load}(l_1), \mathit{drive}(l_1, l_2), \mathit{unload}(l_2) \rangle$.

[1] Partial transition functions are also used to compute pattern database heuristics, where the problem is projected onto a subset of the variables [5, 6].

[2] the cost of the empty plan is just 0.

2.2 Planning as Heuristic Search

Generating the entire state space is exponential in the number of variables. So, this is not a feasible strategy to find (optimal) plans. In fact, finding any plan (also called a satisficing plan) and finding an optimal plan are both PSPACE-complete [1]. This also means that there will very likely not be an efficient (polynomial time) procedure that can guarantee finding an (optimal) solution for every possible planning problem. However, some strategies may sometimes be effective at reducing the search effort. One such strategy involves the use of heuristics.

Definition 2.7. For a planning problem Π with states S a *heuristic function* for Π is a function $h : S \to \mathbb{R}_0^+ \cup \{\infty\}$ that estimates for each state the cost of the optimal path to the goal. We refer to the value $h(s)$ as the *heuristic value* or *h value* of s.

Definition 2.8. For a planning problem Π with states S

- The *perfect heuristic* h^* assigns to each $s \in S$ the cost of an optimal plan for s or ∞ if there is no plan for s.
- The *blind heuristic* h^{blind} assigns to each $s \in S$ the value 0.

Given a heuristic h for Π we can perform the A^*-algorithm to find a plan. This algorithm expands states, starting with the initial state, in order of increasing[3] f value $f(s) := g(s) + h(s)$, where $g(s)$ describes the cheapest path that is already explored by A* from the initial state to s.

We call a heuristic function h *admissible* when $h(s) \leq h^*(s)$ for all states $s \in S$. In other words, an admissible heuristic underestimates the true cost. Using heuristics is helpful because A^* with an admissible heuristic is guaranteed to output an optimal plan [18].

In contrast, breadth first search, without any heuristic, will always output a satisficing plan, if it exists, but for non-uniform cost problems this plan may be far from optimal. So, while Dijkstra's algorithm can find a cheapest plan, it may be helpful to use A* with heuristics. An admissible heuristic does not increase the search effort.

Proposition 2.9. For a planning problem Π and an admissible heuristic h, it must be that A^* with a fixed tie-breaking mechanism equipped with h expands at most as many nodes as A^* equipped with the blind heuristic h^{blind}.

Proof. Given any admissible heuristic, A^* will will not explore any state with a higher than optimal cost.[4] For all states $s \in S$ on an optimal path we have by admissibility $f(s) = g(s)+h(s) \leq g(s)+h^*(s) = h^*(\mathcal{I})$ and for a state $s' \in S$ with higher than optimal cost will have $f(s') \geq g(s') > h^*(\mathcal{I})$.

However, A^* with the blind heuristic the f values are simply the g values and so all states with cheapest paths from the initial state that

[3] In case of equivalent f values, some tie-breaking mechanism has to be in place. To expand fewer states, in case of ties states with higher g values get priority.

[4] This also implies that A* with an admissible heuristic finds an optimal plan.

are cheaper than an optimal path must be expanded. For a non-blind heuristic we do not have to explore all these states. For any state $s' \in S$ not on an optimal path we have $g(s') + h^*(s') > h^*(\mathcal{I})$. So, whenever $h(s') > h^*(\mathcal{I}) - g(s')$, A* will not explore state s'. □

However, computing informative heuristics is difficult. If we could compute the perfect heuristic in polynomial time, we could solve any planning problem in constant time, since if $h^*(\mathcal{I}) = \infty$ the problem is unsolvable. If the problem has a solution, we can also find optimal plans efficiently. Given the perfect heuristic, using A^* we can find a plan in time linear to the length of an optimal plan provided that we break ties in favor of lower h values and we do not have zero-cost actions.[5] In that case, we will only expand the nodes along the path of some optimal plan.

We have shown that admissible heuristics will not increase the search effort of A^*. However, even when using very good heuristics we cannot guarantee that we find a plan efficiently. For example, using so-called *almost* perfect heuristics that are only at most a constant off the perfect heuristic, Helmert and Röger show that for some problems the number of expanded nodes is still exponential in the number of problem variables [12]. This is the case because all states with an f value strictly smaller than $h^*(\mathcal{I})$ must be expanded by A*, and we can construct problem instances where with an almost perfect heuristic there are exponentially many states with an f value smaller than $h^*(\mathcal{I})$.

Definition 2.10. Given a planning problem Π and a heuristic h, we can define $\mathcal{N}(\Pi)$ as a lower bound on the *nodes expanded by* A^*, where $\mathcal{N}(\Pi) := |\{s \in S \mid f(s) < h^*(\mathcal{I})\}|$. So, $\mathcal{N}(\Pi)$ is the number of states that have a $g + h$ value that is smaller than the cost of an optimal plan.

Observe that $\mathcal{N}(\Pi)$ is indeed a lower bound on the number of expanded states as the A^*-algorithm terminates once a goal state has been expanded and the f value of any goal state is at least $h^*(\mathcal{I})$. However, A* might expand more nodes than $\mathcal{N}(\Pi)$ as non-goal states with an f value equal $h^*(\mathcal{I})$ may also be expanded depending on how ties are broken.

Moreover, any other algorithm outputting an optimal plan must also expand all nodes s with $f(s) < h^*(I)$, since otherwise optimal solutions can be missed. So, the lower bound of expanded states holds for all optimal search algorithms.

Definition 2.11. Given a planning problem Π, the family of *almost perfect heuristics* is characterized by, for any positive constant c, $(h^* - c)(s) := \max(h^* - c, 0)$.

Notation 2.12. For an almost perfect heuristic that is off by c, we denote $\mathcal{N}^c(\Pi)$ as the lower bound on the number of states expanded by A^*. That is, the number of states s such that $g(s) + (h^* - c)(s) < h^*(\mathcal{I})$

[5] In the case of zero-cost actions we can have several states with the same h and g values. Then a last in-first out queue policy can still ensure A^* yields an optimal plan in linear time.

Problems where almost perfect heuristics still need to explore exponentially many states will have a lot of optimal paths that are distinct because of different action applications. For example, consider the Gripper task. In this taks you have a robot with two grippers that can both grab and

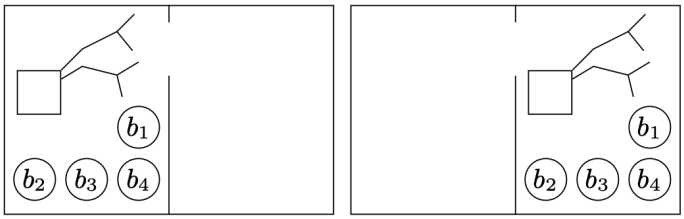

Figure 2: The gripper task with 4 balls. Initial state (left) and goal state (right).

drop balls when the robot is in the same room as the balls. The robot can also move between the rooms. In Figure 2 the initial and goal state of the task with 4 balls is depicted. For this task with n balls there are a total of $|S| = 2 \cdot (2^n + 2n2^{n-1} + n(n-1)2^{n-2})$ states. The outer factor of 2 is for the two possible robot locations, each of the terms respectively 0, 1 and 2 balls being carried by the distinct grippers and the possible locations of the other balls.

Theorem 2.13. *[12] For the Gripper task with n balls, A* with the almost perfect heuristic off by 1 has to explore $\mathcal{O}(2^n)$ states.*

Proof. The full proof can be found in Helmert and Röger's paper [12]. The idea is that for an odd number of balls any state except the ones where all the balls are in one room and the robot is in the other are part of some optimal plan. So, as the goal state has an f value equal to the length of an optimal plan, we get $\mathcal{N}^1(\mathcal{I}_n) = |S| - 3 \in \mathcal{O}(2^n)$.

The even case is similar except that some states where the number of balls in the rooms is odd are not part of an optimal plan. However, all the states with an even number of balls in the room (except the two described above) are part of some optimal plan and so we still get $\mathcal{N}^1(\mathcal{I}_n) \in \mathcal{O}(2^n)$. □

2.3 Binary Decision Diagrams

Another way to find (optimal) plans is to represent large sets of states concisely so that instead of expanding states individually, we can expand sets. One way to represent sets of states is via binary decision diagrams.

Definition 2.14. A *binary decision diagram (BDD) B* over a set of binary[6] variables V_{bin} is a directed acyclic graph with a single root node, sinks that are labeled 0 or 1, and inner nodes labeled with a variable $v \in V_{bin}$ that have two outgoing arcs labeled 0 and 1. The BDD represents those

[6] As D_v need not be binary, we can use $\lceil \log_2(|D_v|) \rceil$ binary variables to represent v.

assignments of V_{bin} such that when traversing the BDD according to the assignment, taking arc 1 when the decision variable is true and 0 otherwise, ends in a 1-sink. The *size of a BDD* B is denoted $\|B\|$ and has as value the number of non-sink nodes in B.

While there are many BDDs representing the same set of states S, given a fixed variable ordering, there exists a unique reduced ordered BDD representing S. We define an ordered BDD, then a reduced BDD.

Definition 2.15. A BDD B is *ordered* with respect to a total variable ordering $\prec$ on V if whenever there is an arc from a node with decision variable v to a node with decision variable u, then $v \prec u$.

Definition 2.16. A BDD B is *reduced* if it does not allow isomorphism reduction or Shannon reduction. *Isomorphism reduction* can be done when B has two nodes n and n' such that the BDDs rooted at n and n' are isomorphic. For the reduction all incoming arcs of n' are redirected to n and the parts that are no longer reachable from B's root are removed. A *Shannon reduction* can be performed if there is a non-sink node n where both outgoing arcs go to the same node m. Then node n can be removed with its incoming arcs redirected to m.

Given a variable ordering on V, there exists a unique reduced ordered BDD for every formula with the variables in V as propositions [3]. From here on out, when we refer to a BDD we mean its reduced ordered counterpart.

Example 2.17. In the logistics example 2.6, we have two variables t, p where we can use the binary variables x_t, x_{p_1}, x_{p_2} and the conversion as described in Table 1.

$s(t)$	x_t
l_1	1
l_2	0

$s(p)$	x_{p_1}	x_{p_2}
l_1	0	0
l_2	0	1
t	1	0

Table 1: FDR to binary variable conversion

The intial state where both the truck and package are at l_1 then corresponds to the formula $x_t \wedge \neg x_{p_1} \wedge \neg x_{p_2}$. The BDDs depicted in Figure 3 also represent this formula. Dotted lines signify that the variable is false. The BDD on the right is the reduced ordered BDD given the $x_t \prec x_{p_1} \prec x_{p_2}$

Variable Orderings BDDs can represent sets of states in a concise manner. However, a poorly chosen variable ordering can have exponential consequences on the size of a BDD. For example, consider the formula $\varphi_n := (x_1 \wedge x_1') \vee (x_2 \wedge x_2') \vee ... \vee (x_n \wedge x_n')$. If we use the variable ordering $x_1 \prec x_1' \prec x_2 \prec x_2' \prec ... \prec x_n \prec x_n'$ we need $2n$ non-sink nodes. We only need to check the truth value of a primed variable when its unprimed counterpart is true. When we find a primed-unprimed pair true, we immediately know the formula is satisfied.

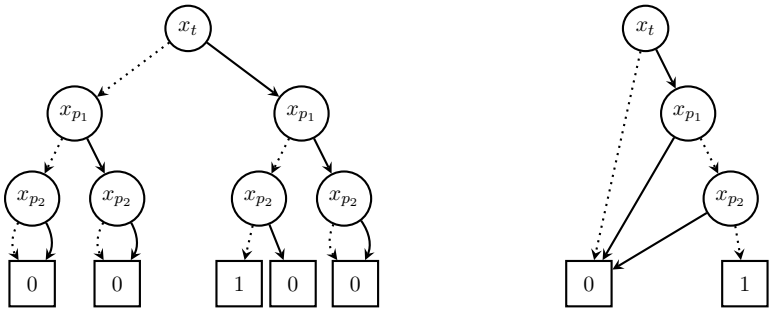

Figure 3: BDDs of initial state; Reduced on the right

If we choose ordering $x_1 \prec x_2 \prec ... \prec x_n \prec x'_1 \prec x'_2 \prec ... \prec x'_n$ instead, we need $2^{n+1} - 2$ nodes. Every combination of truth values of the unprimed variables except where they are all false can still yield a satisficing assignment so the BDD needs to have a full binary tree for these variables yielding $2^n - 1$ nodes. Then we need to check each primed node if its unprimed counterpart was true. Unprimed x_1 is true in 2^{n-1} cases the BDD distinguishes so far. As we can immediately go to the sink when we find a primed-unprimed both set to 1, we have only 2^{n-i} cases to check for each subsequent x'_i.

This is an example of the importance of choosing an optimal variable ordering. Unfortunately, given a BDD and an ordering, deciding whether there is an ordering that results in a smaller BDD is NP-Complete [2]. So, this also means we cannot efficiently decide on an optimal variable ordering. However, there are strategies that provide reasonable orderings efficiently [16, 17].

2.3.1 Operations for Binary Decision Diagrams

We will now briefly describe some operations on BDDs and their run times. This will help explain the decision to use BDDs over some other state set representations in planning.

Equivalence Testing Given the unique reduced ordered BDD representation, some operations are very efficient on BDDs. For one, testing whether two sets of states are the same is practically a constant time operation.[7] In particular, testing whether a BDD represents any state is checking whether it is not just the 0 sink.

Complement, Union and Intersection To compute the complement, we can switch the 0 and 1 sinks. For union and intersection, we can do a recursive procedure that visits each node in the two BDDs that we are

[7]Theoretically, equivalence testing is in linear in the size of the smaller BDD as we need to traverse both BDDs until a difference is found or find they are equivalent. In practice, a hash function is used to map all relevant BDDs to a unique value such that equivalence can be checked in constant time.

combining once. Reducing the resulting BDD can be done in a time linear to the number of nodes [20]. So, for BDDs B_1 and B_2 computing $B_1 \circ B_2$ can be done in time $\mathcal{O}(\|B_1\| \cdot \|B_2\|)$ for $\circ \in \{\cap, \cup\}$. For algorithms that are about the inner workings of computing BDDs we will use $bdd\_intersect(B_1, B_2)$ and $bdd\_union(B_1, B_2)$, $bdd\_complement(B)$. In more general cases we use the logical symbols for each operation.

Conditioning, Forgetting and Renaming Conditioning a variable to a certain Boolean value can be done by going through the nodes from the root and at the relevant variable, we discard the decision node and all branches of this variable except the path that follows from the determined truth value. Finally, we reduce the BDD to its canonical form. So, for a BDD B if we want to set v to $\top$ to get the BDD $B[\top/v]$, we need time in $\mathcal{O}(\|B\|)$. Semantically, this means that for B we only keep the states (assignments) with the variable set to its chosen condition.

To 'forget' a variable v in BDD B, we want the BDD that represents $B[\top/v] \vee B[\bot/v]$. So that we capture all the states fo the original BDD regardless of the truth value of v. This will have a runtime of $\mathcal{O}(\|B\|^2)$. In a BDD specific algorithm this will be performed by a function $bdd\_forget(B, v)$.

Lastly, it will be useful to rename variables in a BDD. To compute the renaming of a BDD B in which we want to relabel the variable v' as v, we will have to go through the BDD and simply rename the v' decision nodes v, which can be done in $\mathcal{O}(\|B\|)$. This will be performed by a function $bdd\_rename(B, v', v)$.

2.4 Symbolic Planning

In this section, we will describe how to use BDDs in planning. Since BDDs have the potential to represent sets of states in a compact manner, we can generate a sequence of BDDs that represent successively reachable states. We will first describe the most elementary algorithm that does so.

2.4.1 Blind Symbolic Search

For a uniform cost problem, Algorithm 1 describes a forward search progression algorithm to find whether there is a plan. This algorithm is a version of breadth first search using state sets rather than individual states.

In this algorithm the $models(p)$ finds the set of states represented by the partial variable assignment p. Each *reached* layer is a set of states reachable at that layer or cost. For this algorithm these state sets need not be represented by BDDs, but we do get some motivatation why we choose BDDs rather than Boolean formulas to represent state sets. In the algorithm we need to repeatedly check whether a state set is non-empty. To do so using the (reduced ordered) BDD representation is a simple constant time opertation. For a formula on the other hand this is equivalent to SAT which is NP-complete.

Algorithm 1: Forward Symbolic Search

Input: V, A, $\mathcal{I}$, $\mathcal{G}$
Output: the (non-)existence of a solution

```
1  goals ← models(G)
2  i ← 0
3  reached[i] ← {I}
4  all_reached = {I}
5  while true do
6      if reached[i] ∩ goals ≠ ∅ then
7          return Solution Found
8      reached[i + 1] ← image(reached, A)
9      if reached[i + 1] \ all_reached = ∅ then
10         return No Solution Exists
11     else
12         all_reached ← all_reached ∪ reached[i + 1]
13     i ← i + 1
```

Now, in Alg. 1 we use an image function to find the subsequent layer of reachable states. This *image* function takes as input a set of states and a set of actions and outputs the set of states that can be reached by applying some action in any of the input states. We will first define the set that the *image* function generates given an input state set and a set of actions and then describe how we can implement this function for BDDs.

Definition 2.18. The *image* of a set of states S' under a set of actions A' is computed by a function $image : 2^S \times 2^A \to 2^S$ such that:

$$image(S', A') := \{tr(s, a) \mid \text{for some } s \in S' \text{ and some } a \in A' \text{ s.t. } tr(s, a) \neq \bot\}$$

where $tr(s, a)$ is the transition function from Defintion 2.4.

So, in Algorithm 1 we keep simultaneously applying all actions to each successive layer of reached states until we reach a goals state or until we get a layer that does not add any states. In that case, all reachable states have been explored and since we have not found a goal, the problem is unsolvable.

To perform Algorithm 1 using BDDs we need to define an implementation of an image function using BDDs. To do so, we first define the transition function of a single action.

Definition 2.19. For the *transition BDD* $\tau_V(a)$ of action $a = \langle pre_a, \mathit{eff}_a \rangle$ we need the BDD variables of the current state set for which we use the variables[8] V. The variables in $V' = \{v' \mid v \in V\}$ are for the successor state. The BDD is constructed from the formula:

$$pre_a(V) \wedge \mathit{eff}_a(V') \wedge frame_a(V, V')$$

[8] possibly converted to ensure we only have binary variables

where $pre_a(V)$ is the conjunction of all the precondition facts, $\mathit{eff}_a(V')$ is the conjunction of the effects and $frame_a(V, V')$ ensures that all the variables that do not have an effect on them stay the same, that is:

$$frame_a(V, V') := \bigwedge_{v \in V \setminus \mathcal{V}(\mathit{eff}_a)} v \leftrightarrow v'$$

Definition 2.20. The *global transition BDD* $TR_V(A)$ is the BDD that is constructed from the following formula:

$$\bigvee_{a \in A} \tau_V(a)$$

Using the global transition BDD we can define the image procedure described in Algorithm 2.

Algorithm 2: Image Function BDD

Input: *reached* (set of states), V, A
Output: BDD representing the image of *reached* under A

```
1 B ← TR_V(A)
2 B ← bdd_intersection(B, reached)
3 for v ∈ V do
4 |   B ← bdd_forget(B, v)
5 |   B ← bdd_rename(B, v', v)
6 return B
```

In this algorithm, we construct the global transition BDD $TR_V(A)$ and take its conjunction with the set of reached states so that we get successor states only if the preconditions are met in the current reachable state set. Then we forget all the variables relating to the current state set and rename the primed variables to obtain the new set of states that can be reached.

Example 2.21. For the logistics example we computed the initial state BDD in 2.17. The transition function of $drive(l_1, l_2)$ would look as follows, where the variable for the location of the truck is the binary x_t variable that is true when the truck is at l_1 and false at l_2 and the variable of the location of the package has been transformed into two binary variables x_{p_1}, x_{p_2}. Then we get

$$x_t \wedge \neg x'_t \wedge x_{p_1} \leftrightarrow x'_{p_1} \wedge x_{p_2} \leftrightarrow x'_{p_2}$$

In the transition BDD variable ordering we interleave the unprimed and corresponding primed variables resulting in the BDD on the left of Figure 4. On the right, the BDD that includes both drive actions.

Plan Reconstruction Algorithm 1 only tells us whether a solution exists. In Algorithm 3 we describe how we can use the successive *reached* BDDs that the search algorithm produced to construct a plan. We put this list of BDDs reached at each step in the input as *layers*.

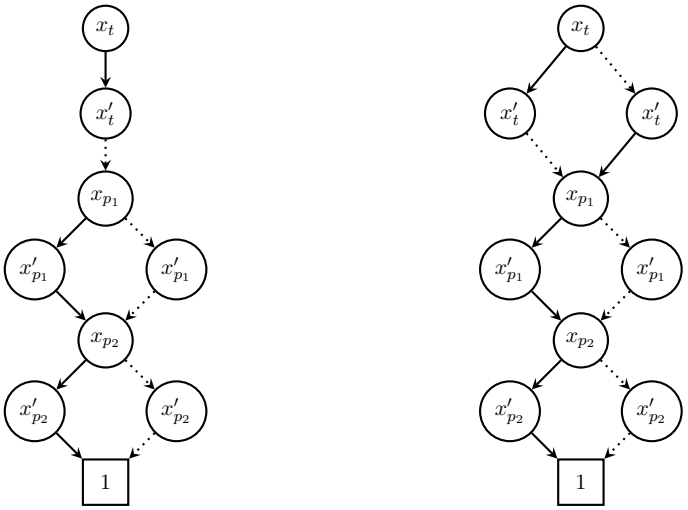

Figure 4: Transition BDD for $drive(l_1, l_2)$ (left) and $\{drive(l_1, l_2), drive(l_2, l_1)\}$ (right)

Algorithm 3: BDD Plan Reconstruction

Input: *layers* (list of BDDs for each search layer), A, *goals*
Output: Plan as a sequence of actions
`// using Python syntax for list indexing`

```
1  reached_goals ← bdd_intersection(layers[−1], goals)
2  plan ← [ ]
3  i ← 2
4  while i < #(layers) do
5      for a ∈ A do
6          pre_im_a ← pre_image(reached_goals, a)
7          reached_from_a ← bdd_intersection(pre_im_a, layers[−i])
8          if reached_from_a ≠ ∅ then
9              plan ← [a] ++ plan
10             i ← i + 1
11             reached_goals ← reached_from_a
12             break // go to start of while loop
13 return plan
```

The pre-image computation is analogous to the image computation except that for the transition BDD we have the preconditions as primed variables, the effects unprimed and the frame ensures that all variables without a precondition on them retain their value. The algorithm must terminate because we know that for every state in a non-initial layer there must be a state in the preceding layer from which we got to the state by application of some action.

2.4.2 Blind Symbolic Search with Non-Uniform Cost

For non-uniform cost problems we can also perform symbolic search. If we want to find optimal plans, we need to take into account the action costs.

Provided we have rational action costs, we can transform the costs into integer values such that $costs = \{0, 1, ..., C_{max}\}$. For all of the possible costs with which we can reach some state, we maintain a 'bucket' of the set of states reachable at that cost. We use A^c as shorthand for all actions with cost c.

Algorithm 4: General Forward Symbolic Search

Input: V, A, $\mathcal{I}$, $\mathcal{G}$
Output: A plan or an indication that no solution exists

```
1  goals ← models(𝒢)
2  reached[0] ← {ℐ}
3  queue.push(0)
4  while queue ≠ ∅ do
5      g ← queue.pop
6      if reached[g] ∩ goal ≠ ∅ then
7          return reconstruct_plan(reached)
8      for c ∈ costs do
9          image_c = image(reached[g], A^c)
10         nextCost = g + c
11         reached_already = ⋃_{i≤nextCost} reached[i]
12         new_reached = image_c \ reached_already if
            new_reached ≠ ∅ then
13             reached[nextCost] = reached[nextCost] ∪ new_reached
14             queue.push(nextCost)
15 return No Solution Found
```

If a plan exists, this algorithm will return an optimal plan because from any g value, starting from 0 at the initial state, it will first generate all possible successors and put them in the appropriate $g + c$ bucket. If this results in new states, then that bucket is pushed to the queue where we keep unique g values in ascending order.

This means we will always explore the lowest possible g value that still has unexplored states. If there is a cheapest plan of cost n, then we will have explored all states reachable at cost less than n and added their successors to the appropriate reached layer. If a penultimate state on a shortest path is reachable at cost less than n, we will have added a goal state to $reached[n]$ and reconstruct a plan. Otherwise, a goal state must be reachable by 0 cost actions within $reached[n]$. Since we keep re-adding n to the queue as long as new states are generated, this goal state must also appear.

This plan can be constructed in a similar manner to Algorithm 3. However, we do need to take into account the specific cost indices for the reached layers as well as the different action costs when computing the pre-images.

When a problem is unsolvable, at some point we will have generated all reachable states with the last unseen reachable state added at path

cost n. This implies that there is no value greater than n in the queue, otherwise there are also new reachable states at cost greater than n. When we dequeue n after adding this last unseen state, we will not add anything to the queue as we will have reached all states at a cost up to and including n. So, the queue is emptied and we exit the loop and notify the user of the absence of a plan.

We also observe that this implementation is very inefficient because we redo a lot of computations. For one, we keep regenerating the *reached_already* up to a certain cost. Also, when there are zero-cost actions, we keep adding the same g value to the queue until convergence (or until a plan is found) and each time also generate the successor states at higher cost. In the next section on heuristic symbolic search we will treat zero cost actions seperately, a similar addition would already improve the performance of this algorithm too.

Additionally, for plan reconstruction, we need to also keep layers of the zero cost actions, this means the $reached[g]$ sets will themselves be indexed by the number of zero-cost actions needed to reach a state in $reached[g]$. For these zero cost actions plan reconstruction is exactly as described in the plan reconstruction algorithm (Alg. 3). For the other actions we need to take action cost into account in the plan reconstruction too.

Backwards and Bidirectional Search Both Algorithm 1 and 4 are forward search algorithms. This means that they start at the initial state and progress until a goal state is found. Of course you can also do backwards search and start at the goal states and regress until you find the initial state. Moreover, one could do bidirectional search wherein forward and backward search are alternated until they have some state in common.

2.4.3 Heuristic Symbolic Planning

We can also combine symbolic search with heuristics. To do so, we also need BDDs of states with the same heuristic value, which we can precompute. Unlike the general symbolic search where we kept track of the states reachable at a certain cost, we now keep similar buckets but for states with the same pair of g, h values, where g is the path cost and h the heuristic value.

In BDDA*, analogously to A*, we first expand the state sets that have the lowest f values, that is, the sum of g and h. Also, like before, we assume that the problem is such that both the costs and the heuristic values are integers. An implementation of BDDA* can be found in Algorithm 5. For readability, we assume that all relevant $reached[g][h]$ are initialized to be empty.

The structure of the algorithm is similar to that of Alg. 4, the general forward symbolic search. Like that algorithm, the implementation is not particularly efficient. In BDDA* we sort the successors in g, h buckets. To ensure termination for general symbolic search, we checked whether newly computed states had already been reached at a lower cost. To do so in

Algorithm 5: Symbolic Heuristic search: BDDA*

Input: V, A, $\mathcal{I}$, $\mathcal{G}$, *heur* (list of BDDs for each heuristic value), max(*heur*) (the maximal finite value of the input heuristic)

Output: A plan or an indication that no solution exists

```
1  goals ← models(G)
2  f ← 0
3  while {I} ∩ heur[f] = ∅ do
4      f ← f + 1
5  reached[0][f] ← {I}
6  queue.add((0, f)) //ordered by incr. val_1 + val_2, and val_1 if tied.//
7  while queue ≠ ∅ do
8      g, h ← queue.pop
9      if reached[g][h] ∩ goal ≠ ∅ then
10         return reconstruct_planBDDA(reached)
11     if A has zero-cost actions then
12         // See routine below //
13     for c ∈ costs^{>0} do
14         image_c = image(reached[g][h], A^c)
15         nextCost = g + c
16         for h' ∈ [0, ..., max(heur)] do
17             succ_{nextCost,h'} ← image_c ∩ heur[h']
18             reached_already ←
                ⋃_{i≤nextCost} ⋃_{j≤nextCost+h'−i} reached[i][j]
19             new_succ_{nextCost,h'} ← succ_{nextCost,h'} \ reached_already
20             if new_succ_{nextCost,h'} ≠ ∅ then
21                 reached[nextCost][h'] ←
                    reached[nextCost][h'] ∪ new_succ_{nextCost,h'}
22                 queue.push((nextCost, h'))
23 return No Solution Found
```

BDDA* we check that a state is not already in a bucket with a lower f value or one with the same f value but a lower g value.

Algorithm 6: BDDA*: zero-cost subroutine

Output: Updates the relevant *reached* buckets, and iterates until $reached[g][h]$ converges

$i \leftarrow 0$
$reached[g][h][i] \leftarrow reached[g][h]$
while $reached[g][h][i] \neq \emptyset$ **do**
  if $reached[g][h][i] \cap goals \neq \emptyset$ **then**
    return $reconstruct\_planBDDA(reached)$
  $image_i \leftarrow image(reached[g][h][i], A^0)$
  for $h' \in [0, ...h-1, h+1, ..., \max(heur)]$ **do**
    $succ_{g,h'} \leftarrow image_i \cap heur[h']$
    $reached\_already \leftarrow \bigcup_{i \leq g} \bigcup_{j \leq g+h'-i} reached[i][j]$
    $new\_succ_{g,h'} \leftarrow succ_{g,h'} \setminus already\_reached$
    if $new\_succ_{g,h'} \neq \emptyset$ **then**
      $reached[g][h'] \leftarrow reached[g][h'] \cup succ_{g,h'}$
      $queue.add((g, h'))$
  $succ_{g,h} \leftarrow image_i \cap heur[h]$
  $i \leftarrow i + 1$
  $reached[g][h][i] \leftarrow succ_{g,h} \setminus reached[g][h]$
  $reached[g][h] \leftarrow reached[g][h] \cup reached[g][h][i]$

Unlike in Alg. 4 where we do not treat the zero cost actions differently, we do so in BDDA* (Alg 5). Though this is not necessary for the correctness of the algorithm, it is an improvement in efficiency and it will help illustrate the point about uniqueness of each $reached[g][h]$ BDD once we expand it, provided we have a consistent heuristic.

We also note that in the BDDA* implementation, we break f value ties in favor of lower g values. This is different than A* where we prefer to expand states with lower h values in case of a tie to avoid expanding many non-goal states with equivalent f values. However, in BDDA*, by expanding state sets with lower g values first we avoid re-expansion, also under the condition that we have a consistent heuristic.

A heuristic is called consistent when for all transitions $s \rightarrow^a s' \in T$, we have $h(s) \leq h(s') + c(a)$.

Lemma 2.22. Given a consistent heuristic, the f value of expanded state sets by BDDA* never decreases.

Proof. In BDDA* we expand the states that currently have the lowest g, h value pair or in case of ties the lowest g value. So, in order to expand a state set with lower f value, it needs to be enqueued as a successor of the g, h state set.

If a lower f value successor is enqueued, there needs to be a state s' with heuristic value h' that is reachable via an action costing c from a g, h

state s such that $g + c + h' < g + h$. This violates consistency. □

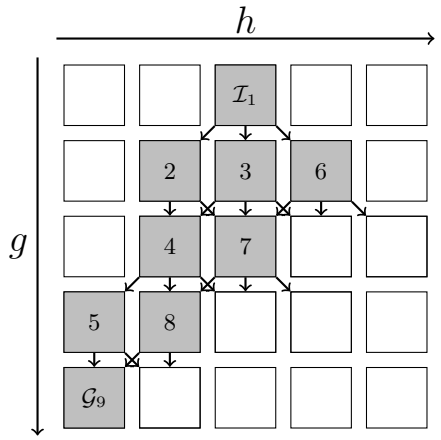

Figure 5: BDDA* expansion with a consistent heuristic

This means that given a consistent heuristic, we always expand the states along an f diagonal and then jump to the next f diagonal. In Figure 5 there is a visualization of these expansions along successive f-diagonals. Each shaded block depicts an expanded BDD numbered according to expansion order. When expanded all successors are generated and sorted into the respective g, h buckets as indicated by the arrows.[9] From the visualization we also observe that with a consistent heuristic it is sensible to break f value ties in favor of lower g values. Otherwise, after expanding the initial state and the BDD labeled 2, we would expand the BDD labeled 4 with a lower h value before 3. We might then find successors in 5 and expand this too. However, then we still need to expand 3 before proceeding to the next f value. Unfortunately, 3 might have successors in 4, meaning we need to re-expand this BDD, and possibly 5 again after it.

So, via the implemented tie-breaking mechanism, given a consistent heuristic, we only need to expand each g, h valued BDD once and, after applying zero cost actions until convergence, we can speak of the unique g, h-BDD.

2.4.4 Limitations of BDDs in Planning

We argued that BDDs were a good choice to perform symbolic search algorithms with because we can check for satisfiability in constant time. However, since SAT is itself NP-Complete, constructing the BDD from any formula must also be NP-hard.

Also, computing unions and intersections is still quadratic in the size of the BDDs which may compound when the BDDs get large. Additionally, all these operations may still result in exponential BDDs as a good variable ordering for one BDD might be very poor for another. The described

[9]For simplification the visualization appears to be of a unit cost problem, in general there can also be arrows across multiple g rows that do not cross the f diagonal.

symbolic algorithms do these operations frequently, which is potentially problematic. In practice, techniques like dynamic variable reordering are used to mitigate this issue [15].

Moreover, we already discussed how certain variable orderings can have exponential effects on the BDD size. Adding to this there are formulas (and so also planning problems) such that, regardless of variable ordering, there is an exponential lower bound on the number of BDD nodes [13]. The planning problems that suffer from this have a permutational character such as the sliding tiles or (n^2-1)-puzzle [7].

Finally, whereas for explicit heuristic search a perfect heuristic could guarantee an optimal solution in time linear to the solution length, Speck, Geißer and Mattmüller show that using the perfect heuristic in symbolic search can actually lead to exponential blowups compared to when using the blind heuristic [22].

3 Introducing Hybrid Planning

3.1 The Symbolic-Explicit Model

We have described two main ways to approach planning problems with or without heuristics: explicit or symbolic state search. We have also described for each of these strategies some strengths and drawbacks. Some of these strengths and drawbacks for different solution strategies are orthogonal to one another. For example, any explicit heuristic search algorithm (without access to the perfect heuristic) needs to expand many nodes if there are many different shortest paths to the goal, a blind symbolic approach also needs to expand those states but potentially can do so in a more concise manner. So, to exploit their strengths and avoid some of the drawbacks we introduce a combined explicit-symbolic approach.

Definition 3.1. Given a planning problem $\Pi = (V, A, c, \mathcal{I}, \mathcal{G})$, and a split of the variables into explicit (V_E) and symbolic (V_S) variables where $V = V_E \cup V_S$ and $V_E \cap V_S = \emptyset$, we can perform *partially symbolic search*. For this search we consider tuples (s_E, S_S) to represent search states, where s_E is a complete assignment of V_E and S_S is a non-empty set of assignments of V_S such that (s_E, S_S) encodes the set of states $\{s_E \cup s_S \mid s_S \in S_S\}$.

Example 3.2. Consider the simple logistics example described in Example 2.6. It has only two variables, one for the package (p) and one for the truck (t). A possible variable split is $V_E = \{p\}$ and $V_S = \{t\}$. An example of a hybrid search state provided this split $(\{(p, truck_1)\}, \mathcal{S}_S)$, where $\mathcal{S}_S$ represents the set of all complete assignments of the variables in V_S. In this case, that means the truck can be at any location.

Not only does symbolic search offer the potential to represent large sets of states concisely, it also has the ability to apply several actions at once. In the hybrid model, when we consider two actions with a different precondition on some explicit variable, for all hybrid states (s_E, S_S) at least one of these actions must not be applicable. Similarly, when two actions have different effects on the explicit variables, we cannot get to a unique succesive hybrid state if we apply both actions. However, actions that are indistinguishable for the explicit variables can be applied at the same time.

Definition 3.3. Given a planning problem $\Pi = (V, A, c, \mathcal{I}, \mathcal{G})$, a partition of its variables into explicit V_E and symbolic V_S ones and a hybrid state (s_E, S_S), an action $a \in A$ is *explicitly applicable* in (s_E, S_S) if $pre_a \restriction_{V_E} \subseteq s_E$.

We can partition the explicitly applicable actions into explicit equivalence classes. The actions $a, a' \in A$ are *explicitly equivalent* for s_E when $c(a) = c(a')$ and $tr_{V_E}(s_E, a) = tr_{V_E}(s_E, a')$. Finally, for each set of applicable explicitly equivalent actions A_i there is a *hybrid succesor state* of (s_E, S_S) defined as (s'_E, S'_S) where:

- $s'_E = tr_{V_E}(s_E, a)$ for any $a \in A_i$
- $S'_S = image_{V_S}(S_S, A_i)$

using the image function from Def 2.18 restricted to the symbolic variables.

By construction, we know that all actions a, a' in some explicit equivalence class A_i, with respect to the explicit state s_E, have the same explicit successor.

We can now use the notion of explicitly equivalent actions to define the hybrid state space. This equivalence is useful because it avoids regenerating the same (original problem) states in distinct hybrid successor states. Since we can now apply sets of actions and we know that when we have two sets of actions $A' \subsetneq A''$, all successors of A', given some predecessor state, must also be successors of A''. We do not want to apply both A' and A'' to the predecessor because it will make the search space unnecessarily large.

Definition 3.4. Given a planning problem $\Pi = (V, A, c, \mathcal{I}, \mathcal{G})$, a partition of its variables into explicit V_E and symbolic V_S we can define the hybrid transition system $\Theta_{\Pi,V_E} = (\mathcal{S}, L, c, T, \mathcal{I}, S^{\mathcal{G}})$ where:

- $\mathcal{S}$ is the set of hybrid states (s_E, S_S).
- $L = 2^A$ is the set of transition labels which can now be any subset of actions of A.
- c is the cost function from Π
- T are the transitions where for a pair of states $(s_E, S_S), (s'_E, S'_S) \in \mathcal{S}$ and a set of actions $A_i \subseteq A$ we have $(s_E, S_S) \rightarrow^{A_i} (s'_E, S'_S) \in T$ when:
  - All actions in A_i are explicitly applicable in (s_E, S_S).
  - All actions in A_i are explicitly equivalent for s_E and there are no applicable actions in $A \setminus A_i$ that belong to the same explicitly equivalent class.
  - The costs of the actions in A_i are equal.
  - (s'_E, S'_S) is a hybrid successor state of (s_E, S_S).
- $\mathcal{I} = (\mathcal{I} \restriction_{V_E}, \{\mathcal{I} \restriction_{V_S}\})$ is the hybrid version of initial state.
- $S^{\mathcal{G}} = \{(s_E, S_S) \in \mathcal{S} \mid \mathcal{G} \restriction_{V_E} \subseteq s_E \text{ and } S_S \cap models_{V_S}(\mathcal{G}) \neq \emptyset\}$ is the set of *hybrid goal states.*

We now define a search algorithm along the lines of this transition system. This algorithm (Alg. 7) is a combination of explicit A* and symbolic blind search. In blind symbolic search, we group together the sets of states that can be reached at the same path cost value (see Alg. 4), in hybrid search we group together symbolic state sets that have the same corresponding explicit state as well as the same path cost.

Given a variable split V_E, V_S and a heuristic on the explicit part of the states h, we explore hybrid states (s_E, S_S) in order of ascending f values, which is now defined as the path cost to (s_E, S_S) plus the explicit heuristic $h(S_E)$. We generate successor states by applying all explicitly equivalent actions simultaneously.

Algorithm 7: Hybrid Search

Input: V, A, $\mathcal{I}$, $\mathcal{G}$, h (a heuristic for the explicit states)
Output: A plan or an indication that no solution exists

$init \leftarrow (\mathcal{I}_E, \{\mathcal{I}_S\})$
$exp\_goals = \mathcal{G}_E$
$sym\_goals = models_S(\mathcal{G})$
$g \leftarrow 0$
$reached[0][\mathcal{I}_E] \leftarrow \{\mathcal{I}_S\}$
$queue \leftarrow [(g, h[\mathcal{I}_E], \mathcal{I}_E)]$
while $queue \neq \emptyset$ **do**
    $g_{val}, h_{val}, s_E \leftarrow queue.pop()$
    $S_S \leftarrow reached[g_{val}][s_E]$
    if $exp\_goals \subseteq s_E \wedge S_S \cap sym\_goals \neq \emptyset$ **then**
        return $reconstruct\_hybrid\_plan(reached)$
    $exp\_equiv\_actions \leftarrow dict()$
    for $a \in A$ **do**
        $s'_E \leftarrow tr_E(s_E, a)$
        if $s'_E \neq \perp$ **then**
            if $(s'_E, a.cost) \in exp\_equiv\_actions$ **then**
                $exp\_equiv\_actions[(s'_E, a.cost)].add(a)$
            else
                $exp\_equiv\_actions[(s'_E, a.cost)] \leftarrow \{a\}$
    for $key \in exp\_equiv\_actions$ **do**
        $exp\_succ, cost \leftarrow key$
        $ops \leftarrow exp\_equiv\_actions[key]$
        $sym\_succ \leftarrow image_S(S_S, ops)$
        $already\_reached \leftarrow \bigcup_{i \leq g_{val}+cost} reached[i][exp\_succ]$
        $sym\_succ \leftarrow sym\_succ \setminus already\_reached$
        if $sym\_succ \neq \emptyset$ **then**
            if $cost = 0$ **then**
                $i \leftarrow 0$
                while $reached[g_{val}][exp\_succ][i] \neq \emptyset$ **do**
                    $i \leftarrow i + 1$
                $reached[g_{val}][exp\_succ][i] \leftarrow sym\_succ$
                $i_{max}(g_{val}, exp\_succ) \leftarrow i$
            $reached[g_{val} + cost][exp\_succ] \leftarrow$
             $reached[g_{val} + cost][exp\_succ] \cup sym\_succ$
            //insert g, h and exp. state based on lowest f value//
            $queue.insert((g + cost, h[exp\_succ], exp\_succ))$
return *No plan found.*

We first note that in case of zero-cost actions, we need to do some extra bookkeeping. We do so to be able to do plan reconstruction. If we are unable to distinguish the 'layers' of zero cost actions applied, we may indefinitely add a zero cost action to the plan where both the predecessor and successor are in the same state set. This type of bookkeeping is also necessary for symbolic blind search (Alg. 4) even though that implementation does not show it for simplicity. As we introduce the hybrid algorithm in this thesis, we have also added the path reconstruction algorithm with some motivation to Appendix A.

Secondly, we note that the hybrid search algorithm (Alg. 7) indeed behaves like forward symbolic search (Alg. 4) when the variables are all symbolic. In that case we do need a dummy explicit variable that never changes value to do the explicit transition computation. Then, all actions with the same cost will be explicitly equivalent. For all these costs, the successor sets are computed. We check that we actually generated a new successor not reached at lower cost and if so we merge the successor set with other states reachable at the same path cost and add this path cost to the queue.

Moreover, the hybrid search algorithm (Alg. 7) behaves like A* when all states are explicit. To make the algorithm functional, we need a dummy symbolic variable because otherwise we will never add any state to the queue. Then, when we have already added a state to the queue, we find that this dummy variable and value have already been reached and never re-add it to the queue unless we can do so at a lower path cost then previously seen.[10] Like in A* we can also tie-break on higher g value in case of f value ties.

However, as we need these dummy variables in both edge cases and perform some unnecessary computations, from now on, we assume that given an extreme split, i.e. all variables symbolic or all explicit, the hybrid search algorithm just defers to actual blind symbolic search or A*.

We will now show correctness of the hybrid search algorithm (Alg. 7):

Claim 3.5. The hybrid search algorithm (Alg. 7) is sound and complete.

Proof. First, whenever the algorithm returns a plan, this must actually be one. The algorithm returns a plan once it reaches, for some $s_E \supseteq \mathcal{G}_E$, a symbolic state set $reached[g][s_E]$ that contains a symbolic goal state. As we only add to each reached set by applying actions consecutively, for each symbolic state $s_S \in reached[g][s_E]$ there must be a sequence of actions $a_1, ..., a_k$ with a total cost of g and a sequence of explicit states $s_{E_1}, ..., s_{E_k}$ such that $s_{E_1} = \mathcal{I}_E$, $s_{E_k} = s_E$, $s_{S_k} = s_S$ and for all $i < k$ there exists $s_{S_i}, s_{S_{i+1}}$ such that $tr((s_{E_i}, s_{S_i}), a_i) = (s_{E_{i+1}}, s_{S_{i+1}})$.

Secondly, the algorithm will always terminate. If there is a plan, it will eventually find it. The algorithm will only stop expanding states once a

[10] In this implementation we do keep a states in the queue that we added when we had not found a cheaper path to it. If we find the cheaper path before expanding the state, and we get to the same state in the queue at higher cost it will be expanded again but then we find no new successors.

plan is found or all reachable states have been expanded. Suppose there is no plan and let s_E, s_S be an arbitrary reachable state and $a_1, ..., a_k$ a sequence of actions from the initial state that results in s_E, s_S. If s_E, s_S is not expanded, then its a_k predecessor must also not be expanded. Otherwise the set of explicitly equivalent actions that contains a_k would generate s_E, s_S would have resulted in a successor containing the unseen s_E, s_S which would be added to the queue. Continuing down this chain, this means the initial state is not expanded which is a contradiction.

Now, once all states have been expanded at their lowest path cost, we will not add any new states to the queue. So, in that case we will exit the while loop and return that no plan has been found.

□

3.2 Heuristics in the Hybrid Model

In the hybrid algorithm, we only use the heuristic for the explicit part of the state. We can still define admissibility and consistency.

Definition 3.6. Given a planning problem Π, a variable split V_E, V_S and a heuristic on the entire explicit state space $h : \mathcal{S} \to \mathbb{R}_0^+ \cup \{\infty\}$, we can define the deduced heuristic h_E as follows: $h_E(s_E) := \min_{s_S \in \mathcal{S}_S} h(s_E, s_S)$.

The *perfect explicit heuristic* $h_E^* : \mathcal{S}_E \to \mathbb{R}_0^+ \cup \{\infty\}$ is deduced[11] from the perfect heuristic h^*.

An explicit heuristic h_E is *admissible* when for all explicit states $h_E(s_E) \leq h_E^*(s_E)$.

An explicit heuristic h_E is *consistent* when for any explicit states $s_E \in \mathcal{S}_E$ and any action $a \in A$ whenever $tr_E(s_E, a) = s'_E$, then $h_E(s_E) \leq h_E(s'_E) + c(a)$.

Given an explicit heuristic h_E, we can also define the *associated complete heuristic* h by $h(s_E, s_S) := h(s_E)$ for all symbolic states $s_S \in \mathcal{S}_S$.

Note that the definition of the deduced heuristic is different than the abstraction heuristic based on pattern databases (PDB). There we first project the state space onto a subset of the variables (the pattern) and then compute the perfect heuristic on this abstracted state space. These PDB heuristics will always be admissible and consistent when we project onto the explicit variables. However, if we have access to a complete state heuristic, our definition is able to retain some of that additional information.

Proposition 3.7. Given a planning problem Π, a variable split V_E, V_S and a heuristic on the complete state space h, the deduced heuristic h_E inherits admissibility but might not inherit consistency from h.

Vice versa, given an explicit heuristic h'_E the associated complete heuristic h' inherits both admissibility and consistency from h'_E.

[11] We can also define the perfect explicit heuristic from the hybrid state space. Then, we first define a perfect hybrid heuristic $h^*_{E,S}$ that computes the shortest path of hybrid states (s_E, S_S) to a goal state and take $h^*_E(s_E) = \min_{S_S | (s_E, S_S) \in \mathcal{S}} h^*_{E,S}(s_E, S_S)$. We prove the equivalence of the two definition in Appendix B.

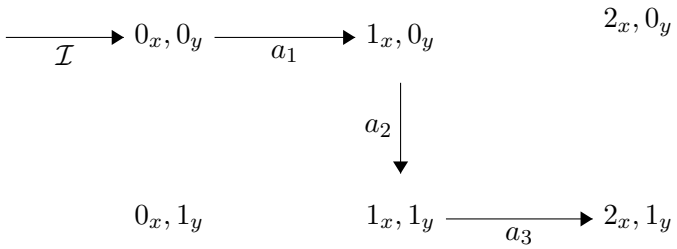

Figure 6: Consistency Counterexample: with y as the symbolic variable the perfect partial heuristic for an explicit state is the cheapest path to the goal (2,1) per column.

Proof. Let h be admissible, and let $s_E \in \mathcal{S}_E$ be arbitrary. Let $s_S \in \mathcal{S}_S$ be a symbolic state such that $h(s_E, s_S)$ is minimal. Then, by definition of the deduced heuristic and admissibility of h we get $h_E(s_E) = h(s_E, s_S) \leq h^*(s_E, s_S) = h^*_E(s_E)$.

The other direction is immediate, as for any state $s = s_E, s_S$ we have $h'(s_E, s_S) = h'_E(s_E) \leq h^*_E(s_E) \leq h^*(s_E, s_S)$.

For the consistency counterexample, consider the following planning problem with $V = \{x, y\}$ with $D_x = \{0, 1, 2\}$ and $D_y = \{0, 1\}$ and the following unit cost actions:

- $a_1 = \langle x = 0 \wedge y = 0, x = 1 \rangle$
- $a_2 = \langle x = 1, y = 1 \rangle$
- $a_3 = \langle x = 1 \wedge y = 1, x = 2 \rangle$

The initial state is $\{x = 0, y = 0\}$ and the goal state is $\{x = 2, y = 1\}$. Now, given the split $V_E = \{x\}$ and $V_S = \{y\}$, we claim that while the perfect heuristic h^* is consistent, its deduced counterpart h^*_E is not. We have $h^*_E(\{x = 0\}) = 3$ and $h^*_E(\{x = 1\}) = 1$, but $tr_E(\{x = 0\}, a_1) = \{x = 1\}$ and $c(a_1) = 1$.

Now, assume h'_E is consistent, and let $s = s_E, s_S$ be an arbitrary state and $a \in A$ any action such that $s \rightarrow^a s' \in T$ where $s' = s'_E, s'_S$. Then $tr_E(s_E, a) = s'_E$ and so $h'(s_E, s_S) = h'_E(s_E) \leq h'_E(s'_E) + c(a) = h'(s'_E, s'_S) + c(a)$. □

Since admissibility is inherited from the complete heuristic, the following claim about the algorithm is useful. Using the claim, we know that given any admissible complete heuristic, the hybrid algorithm will produce an optimal plan with its deduced heuristic.

Claim 3.8. The hybrid search algorithm (Alg. 7) is optimal when the explicit heuristic h_E is admissible.

Proof. First, we note that in the cases where all variables are explicit or symbolic we assumed hybrid search deferred to A* and blind symbolic search respectively. We know that these algorithms are optimal given an admissible heuristic.

Suppose we have explicit and symbolic variables $V_E, V_S \neq \emptyset$ and some admissible heuristic h_E on explicit states. Let (s_E, S_S) be any hybrid goal state that is reachable at optimal cost n and (s'_E, S'_S) a hybrid state on an optimal path to (s_E, S_S). Also, assume (s''_E, S''_S) is a goal state that is reachable at suboptimal cost $m > n$.

We will show that the hybrid algorithm will expand (s'_E, S'_S) before (s''_E, S''_S). We get:

$$\begin{aligned}
f((s'_E, S'_S)) &= g((s'_E, S'_S)) + h_E(s'_E) && \text{(def. } f\text{)} \\
&\leq g((s'_E, S'_S)) + h^*_E(s'_E) && \text{(admissibility)} \\
&\leq g((s_E, S_S)) && \text{(Prop. 3.7)} \\
&= n < m \\
&= g((s''_E, S''_S)) \\
&= f((s''_E, S''_S)) && \text{(admissibility)}
\end{aligned}$$

Where we use the reverse direction of the admissibility inheritance from Prop. 3.7 to show that $h^*_E(s'_E)$ must be upper-bounded by the actual cost of getting from any hybrid state with explicit part s'_E to a goal state, in particular one in (s_E, S_S).

So, we will always expand a node on an optimal path before expanding a goal state on a suboptimal path. □

We will now illustrate the algorithm via example.

Example 3.9. Consider a unit cost logistics example with one truck and one package. The road network is a binary tree of depth n for $n > 1$. Formally, $\Pi_n = (V_n, A_n, \mathcal{I}_n, \mathcal{G}_n)$ where:

- $V_n = \{depth_t, width_t, depth_p, width_p\}$ with $D_{depth_t} = \{0, ..., n\}$ and $D_{width_t} = \{0, 1, ..., 2^n - 1\}$ and $D_{depth_p} = \{0, ..., n\} \cup \{t\}$ and $D_{width_t} = \{0, 1, ..., 2^n - 1\} \cup \{t\}$
- $A_n = \{drive_{\text{left,down,k}}, drive_{\text{right,down,k}}, drive_{\text{up,j}}, load_l, unload_l\}$ where
  - $pre_{drive_{\text{left,down,k}}} = \{depth_t = k, width_t = j\}$ and $\mathit{eff}_{drive_{\text{left,down,k}}} = \{depth_t = k + 1, width_t = 2j\}$ for $k \in \{0, 1, ..., n - 1\}$ and $j \in \{0, 1, ..., 2^k - 1\}$
  - $pre_{drive_{\text{right,down,k}}} = \{depth_t = k, width_t = j\}$ and $\mathit{eff}_{drive_{\text{right,down,k}}} = \{depth_t = k + 1, width_t = 2j + 1\}$ for $k \in \{0, 1, ..., n - 1\}$ and $j \in \{0, 1, ..., 2^k - 1\}$
  - $pre_{drive_{\text{up,k}}} = \{depth_t = k, width_t = j\}$ and $\mathit{eff}_{drive_{\text{up,k}}} = \{depth_t = k - 1, width_t = \lfloor \frac{j}{2} \rfloor\}$ for $k \in \{1, ..., n\}$ and $j \in \{0, 1, ..., 2^k - 1\}$
  - $pre_{load_{d,w}} = \{depth_t = d, width_t = w, depth_p = d, width_p = w\}$ and $\mathit{eff}_{load_{d,w}} = \{depth_p = t, width_p = t\}$ for $d \in \{0, 1, ..., n\}$ and $w \in \{0, 1, ..., 2^n - 1\}$
  - $pre_{unload_{d,w}} = \{depth_t = d, width_t = w, depth_p = t, width_p = t\}$ and $\mathit{eff}_{unload_{d,w}} = \{depth_p = d, width_p = w\}$ for $d \in \{0, 1, ..., n\}$ and $w \in \{0, 1, ..., 2^n - 1\}$

- $\mathcal{I}_n = \{depth_t = 0, width_t = 0, depth_p = n, width_p = 0\}$
- $\mathcal{G} = \{depth_p = 0, width_p = 0\}$

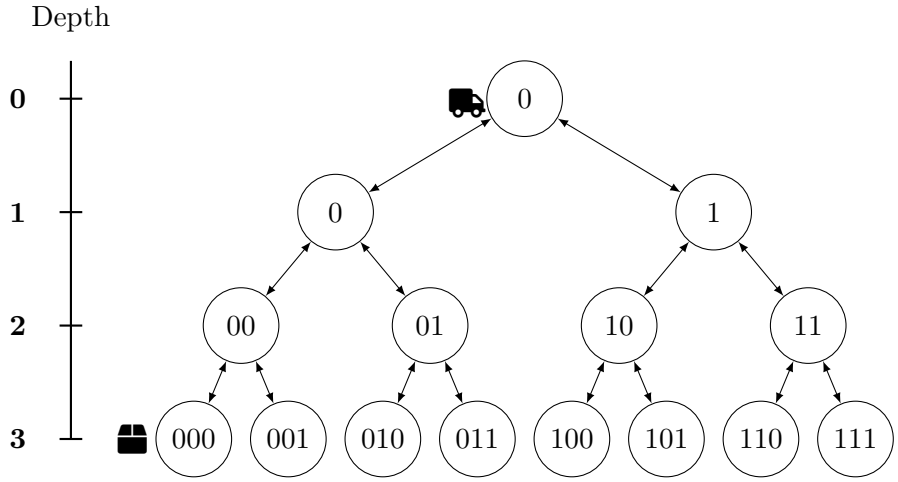

Figure 7: The initial state for $n = 3$ with the width variable in binary in each node

Also consider the following simple explicit heuristic h that assigns the value 0 when the package is at the goal location, the value 1 when it is in the truck and when the package is at any location at depth k the heuristic is k. It is easy to check that this heuristic is admissible though not consistent.[12]

Given the heuristic h does not change based on the location the location of the truck we can put the variables associated with the truck in the symbolic set as Alg 7 disregards the heuristic on the symbolic part. The package location variables we can make explicit.

Then, for every explicit state, all the (applicable) drive actions will be in the same equivalence class. Each load and unload action changes the explicit state, so they are in a singleton explicitly equivalent class when applicable.

In the hybrid algorithm, we generate the first successor by applying $drive_{\text{right,down,0}}$ and $drive_{\text{left,down,0}}$ simultaneously. So, the explicit part does not change from the initial state but the truck is at depth 1. Until we reach the nth successor, we continue to apply the applicable drive actions that positions the truck in the successive depth layers. If we do not do duplicate detection we can also go back layers, so the truck can be at every other depth.

After n steps, we can drive again but then reach an already explored hybrid state, which we will not enqueue. However, we can also load the package into the truck. Then, the resulting hybrid state is $(\{depth_p = t, width_p = t\}, \{\{depth_t = n, width_t = 0\}\})$ where we only have a single symbolic state due to the preconditions of the loading action.

[12]When the package is at depth $k > 2$ the unit cost load action, when applicable, reduces the heuristic value from k to 1 and $k - 1 > 1$.

After loading, the heuristics will prevent the states where we unload the package at a non-goal location from getting to the top of the priority queue as those unload actions will increase the f value to be at least as much as f values on optimal paths and we break ties in favor of higher g values. So, after $2n + 2$ steps we have the package unloaded at the desired location.

Moreover, the truck location BDDs can be represented concisely. Before picking up the package only depth variables are needed. After, we do also need to restrict to the left (lower indexed) width variables. This can also be done in a concise way by using an efficient binary encoding. For example, to specify that the truck can be at any of the four leftmost leaf nodes at depth 3, we need only specify that the first binary width variable is 0.

3.3 Comparing Planning Frameworks

3.3.1 The Logistics Example

We could also consider other strategies on Example 3.9. It is immediate that explicit A^* will need to consider many more search nodes. If you individually count the states in the BDD, the number of states explored by A^* is not larger, but the hybrid representation reduces a lot of overhead. For example, we do not need to specify the package location for all different truck locations and we can capture the truck being at a certain depth with one BDD-node as opposed to explicitly listing all the possible truck width values at that depth.

Moreover, the hybrid algorithm also does well compared to blind symbolic search. Up until the nth step blind search generates the same BDDs as hybrid search, except the package variables, which are fixed in these steps, are not in the BDDs in the hybrid model.

However, from this point on, the blind search algorithm will require branching on the package locations too. Namely, we also apply the unload actions at non-goal locations. Thus, the BDDs also need to capture the information that the truck can be at certain locations if we unloaded the package at location x and the truck can be at other locations if we had unloaded at location y. In the hybrid algorithm the successors of these suboptimal unload actions are never expanded because of their high heuristic value.

The better performance of the hybrid algorithm on Example 3.9 compared to BDDA* with the same heuristic relies on tie breaking. The hybrid algorithm, like A^*, can break ties in favor of higher g values when the f values are equivalent. BDDA* on the other hand, to avoid reexpansion and ensure uniqueness of the g, h- BDDs, explores state sets with lower g-values first given some fixed f value.

Now, in the example problem after loading the package into the truck we have taken $n + 1$ actions. Consider all the leftmost nodes at depth k with $1 \leq k \leq n$. To arrive with the package in the truck at such a node at depth k we need to take another $n - k$ drive actions. Unloading there

requires another action. So after unloading there we have a g-value of $2n + 2 - k$. The h value after unloading is k, so in all these cases we have an f value of $2n + 2$. This is equivalent to the f-value of the optimal plan. So, once BDDA* reaches this f-value, it will first expand the results of all these suboptimal unload actions before expanding the goal.

Another advantage of the hybrid approach over BDDA* in this case may be it's greater intuitiveness. Explicit states are easy to visualize as they are always fully specified, whereas symbolic representations can be more abstract. By keeping the package variables explicit in the hybrid model, it is always immediate where the package is, and it is clear that this is relevant for solving the problem. In a fully symbolic model like BDDA* we can still derive the package location(s) but it is less instantaneous.

However, the appeal of this intuition is lost when our aim is to let a computer or automated system perform the algorithms. Moreover, with the hybrid model we also need to devise a good variable split that is not necessary for BDDA*. We already argued that the extreme splits (all explicit or all symbolic) would be suboptimal in this case and will come back to the issue of this split in Chapter 4.

3.3.2 General Comparison Observations

For the simple logistics example we have shown that the hybrid algorithm can outperform other strategies like A*, blind symbolic search and BDDA*. However, this was not a very significant separation. For example, the number of BDDs expanded is only a linear factor more for BDDA* than the hybrid algorithm given the heuristic. This heuristic was not exceptionally informative, as it significantly underestimates the cost to the goal for most states.

When we have a better heuristic (i.e. a heuristic that is closer to the actual true cost to get to a goal) the hybrid algorithm inherits properties from A*. Specifically, it inherits that, given a fixed variable split, the number of search nodes cannot increase by having a heuristic that is better than the blind heuristic. The argument is analogous to that of Prop. 2.9. Since the symbolic part is not affected by the heuristic, having a better one will not affect the size of the hybrid BDDs. In contrast, for BDDA* using the perfect heuristic can cause an exponential blow-up in the size of the BDDs [22].

Moreover, when using the perfect heuristic, we get a BDD with exponentially many nodes, there must be a exponentially many states on optimal paths. This is reminiscent of the argument by Helmert and Röger [12] about when A^* requires exponentially many search nodes even given an almost perfect heuristic.

More generally, whenever, given some admissible heuristic h, the expansion size of BDDA* is exponential, it must be that A*, using any heuristic that is strictly dominated by h, expands exponentially many nodes too.

We define the expansion size of BDDA* as the sum of all the inner nodes of the BDDs that are expanded during the search.

Definition 3.10. Given a planning task Π and heuristic function h, we denote with $BDD_{\Pi,h(g=i,h=j)}$ the unique BDD that is expanded by BDDA* with g value i and h value j. The *expansion size* of BDDA* is the sum of the sizes of all $BDD_{\Pi,h(g=i,h=j)}$ with the property $i + j \leq h^*(\mathcal{I})$ and $i < h^*(\mathcal{I})$.

Definition 3.11. Given a planning problem Π and two admissible heuristics h, h'. We say that h *dominates* h' when for every state s in the state space associated with Π, $h(s) \geq h'(s)$.

We call the domination *strict* when for all state s that are not goal states $h(s) > h'(s)$.

Lemma 3.12. For a given planning problem Π and a heuristic h that strictly dominates over h', if there exists a variable ordering such that the expansion size of forward BDDA* with h is exponential in the input size, then the number of nodes expanded by A* with h' will also be exponential in the input size.

Proof. Given some variable ordering, if the expansion size of forward BDDA* with h is exponential in the input size, then there is some f-value pair, say g'', h'', such that $g'' + h'' \leq h^*(\mathcal{I})$ and $g'' < h^*(\mathcal{I})$ and $|BDD_{\Pi,h(g'',h'')}|$ is exponential in the number of variables of Π. This means that after g'' steps there are exponentially many states at least $h'' > 0$ steps away from the closest optimal goal state.

So, all of these exponentially many states described in $BDD_{\Pi,h(g'',h'')}$ must be expanded by A* with h', because in non-goal states s we have $h'(s) < h(s)$. Moreover, by optimality of A^* with an admissible heuristic, we know that for any state s it is expanded at its lowest g value. Thus, for any state $s \in BDD_{\Pi,h(g'',h'')}$, we have $g'(s) + h'(s) < g'' + h'' \leq h^*(\mathcal{I})$. The strict domination is necessary to ensure that A* actually expands the nodes because they will have an f value strictly smaller than the actual cheapest cost.

□

In particular, this means that if BDDA* with the perfect heuristic h^* has an exponential expansion size, then any almost perfect heuristic will result in the expansion of exponentially many nodes by A*.

Proposition 3.13. For a given planning problem Π, if there exists a variable ordering such that the expansion size of forward BDDA* with the perfect heuristic is exponential in the input size, then the number of nodes expanded by A* with any almost perfect heuristic will also be exponential in the input size.

3.4 Exponential Separation

In this section we will describe a planning problem that exploits the weaknesses of explicit heuristic search, blind symbolic search and symbolic heuristic search. To construct this problem, we will combine two types of problems. One problem where using a heuristic actually requires more

effort than a concise symbolic representation without heuristic and one problem where the heuristic is really necessary to solve the problem concisely. The former problem is adapted from Speck, Geisser, and Mattmüller [22], while the latter is based upon an observation by Hung that BDDs need exponentially many nodes with respect to the input variables to describe permutations [13].

3.4.1 Perfect is Worse

For the first part of the problem we will also use this poor performance on permutations. Except now we ensure that the permutations only show up when using the perfect heuristic.

Example 3.14. Consider the following family of unit-cost problems $\Pi_n = (V_n, A_n, \mathcal{I}_n, \mathcal{G}_n)$ where $n = 2^k$ for some non-negative integer k:

- $V_n = \{x_0, ..., x_{2n}\} \cup \{v_1, ..., v_n\} \cup \{g_1, ..., g_n\}$ with x_i and g_i binary variables and $\mathcal{D}_{v_i} = \{1, .., n\}$ for $i \in \{1, ..., n\}$.
- $A_n = \{o_{i,j} \mid i, j \in [n]\} \cup \{a_{i,j} \mid i, j \in [n]\}$ such that:
  - $o_{i,j} = \langle x_{i-1} \wedge \neg x_i, x_i \wedge v_i = j \rangle$
  - $a_{i,j} = \langle x_{n+i-1} \wedge \neg x_{n+i} \wedge v_i = j, g_j \wedge v_i = 1 \wedge x_{n+i} \rangle$
- From the binary variables $\mathcal{I}_n$ only sets x_0 true and assigns 1 to all v_i
- $\mathcal{G}_n = \{g_1, ..., g_n\}$

The 'build' phase of this problem where we take $o_{i,j}$ actions for increasing values of i is depicted in Figure 8 where the first row consist of the binary x_i variables and the second row has the v_i variables. For each i-value, there are n $o_{i,j}$ actions to set the v_i variable to a value between 1 and n, while the x_i variables act as a counter to ensure we set each v_i once, one after another.

Then in the 'check' phase, we take the $a_{i,j}$ actions. The x_i's still act as a counter to ensure that at step $n + i$ we record the value of v_i and set the corresponding g_j to true. Importantly, upon recording the value of v_i we set it back to the default value 1. Then our goal is reached if all n distinct g_j values are set to true.

What we will now show is that for problem Π_n from Example 3.14 we get a polynomial expansion size when we use the blind heuristic, but an exponential one when using the perfect heuristic independent of variable ordering.

Lemma 3.15. For Π_n (Exm. 3.14), the expansion size of BDDA* with the blind heuristic is polynomial in n for every variable ordering.

Proof. The expansion size of BDDA* with a certain heuristic is the sum of the size of all the BDDs expanded in the process. First, we observe that BDDA* with the blind heuristic will expand $2n$ BDDs to solve Π_n, namely

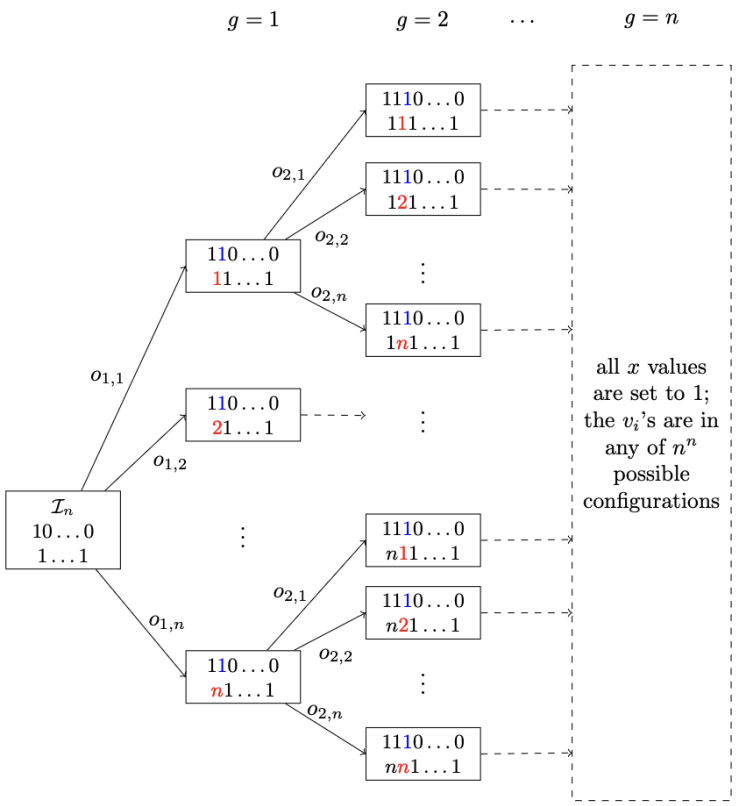

Figure 8: States reachable in n steps of Π_n. On first line the counter variables $x_0, ..., x_n$, on the second line $v_1, ..., v_n$.

$B_{\Pi_n,h^0}(0,0), B_{\Pi_n,h^0}(1,0), ...B_{\Pi_n,h^0}(2n-1,0)$. The first n of these BDDs are expanded in the 'build' phase where only $o_{i,j}$ actions are applicable. In this phase the g_i variables remain 0, while the value of the x_i variables is fixed for each step. Initially the v_i variables are also fixed, as we have $n = 2^k$ for some k, we use k variables for each v_i. So, without sink nodes, the initial BDD will have size $(3+k)n+1$.

At each subsequent build step, the variable v_i can take on any of its n values. This means that after step i we can get every possible combination of the k variables that represent v_i. So, the BDD can eliminate the nodes corresponding to these k variables. This means that the BDD size is only decreasing in the first i steps, where for each i from 1 to n we get a BDD of size $3n+1+(n-i)k$, regardless of variable ordering.

Now, in the check phase we apply each $a_{i,j}$ action one after another. For the first check this means that exactly of the g_j's gets set to 1 while the others stay 0. At step i, we get that between 1 and i g_j's are set to 1 while the others are 0. So, in this phase we are counting the number of g_j set to 1. A BDD can represent a count in polynomial size, namely in $\mathcal{O}(n^2)$ many nodes if there are n objects to be counted.

The BDD counting that between 1 and i g_j's are true is depicted in Figure 9 where at each depth there are at most $i+1$ nodes counting the number of g_j's set to true so far. When more than $i+1$ g_j's are set to true, we can immediately go to the sink node.

Moreover, while after each ith check step we need to consider the value of v_i again, as it is fixed it will only take it's k variables to do so. Similarly, the x_i values are also fixed at each step.

Finally, while the BDDs representing the check phase states would be smallest if we keep the g_j variables together, we only get up to n many duplicates of the other $(2+k)n+1$ variables when they are interspersed in the g_j counting BDDs. So, this means that for every variable ordering,[13] the BDDs expanded by BDDA* with the blind heuristic are polynomial. □

Lemma 3.16. For Π_n (Exm.3.14), the expansion size of BDDA* with the perfect heuristic is exponential in n for every variable ordering.

Proof. We will show that one of the BDDs expanded by BDDA* with the perfect heuristic for Π_n is exponential in n for every variable ordering. This suffices as this is a lower bound on the overall expansion size. Let the variable ordering be arbitrary and consider the BDD $B_{\Pi_n,h^*}(n,n)$ at the end of the build phase. This BDD must be expanded as all states in this BDD are actually on shortest paths. At this stage, the v_i variables collectively must be a permutation of the numbers between 1 and n. To encode a permutation of size n, a BDD always needs exponentially many nodes with respect to n for any variable ordering [13]. □

[13] Even the binary representation of the v_i variables need not be together, even though this is general practice.

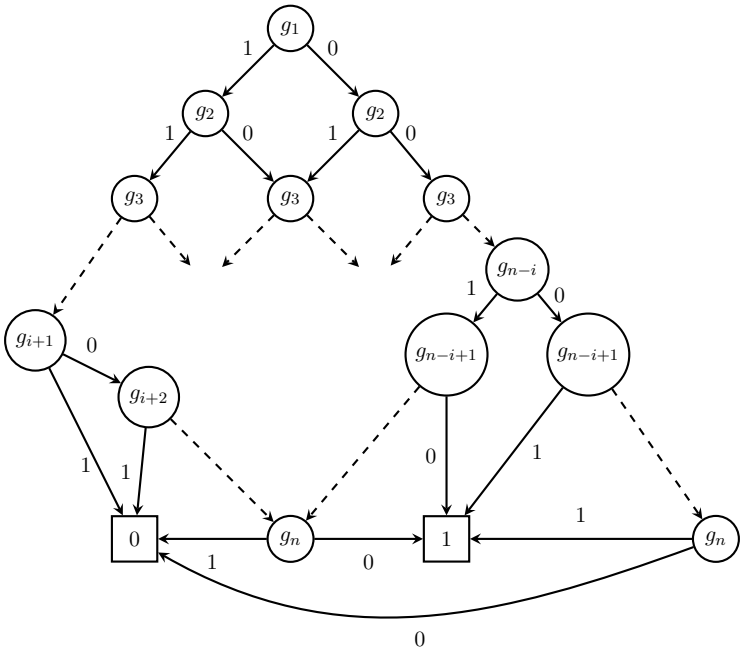

Figure 9: BDD to count between 1 and i of the g_k variables are set to 1.

Theorem 3.17. *Using h^* instead of h^0 can increase the expansion size of BDDA* exponentially in the size of the planning task $|\Pi_n|$ for all variable orderings.*

Proof. In Lemma 3.15, we showed that, regardless of variable ordering, the expansion size using h^0 is always polynomial with respect to the number of variables in Π_n. Moreover, in Lemma 3.16, we showed that, for all variable orderings, the expansion size using h^* is exponential with respect to the number of variables in Π_n. So, the perfect heuristic can exponentially increase the expansion size of BDDA* with respect to the blind heuristic for all variable orderings. □

We have now shown that if we have the perfect heuristic for Π_n, then BDDA* will have an expansion size exponential with respect to the input size. From Lemma 3.13 it then follows that A* with an almost perfect heuristic must also expand exponentially many states. However, the assumption that we have the perfect or even an almost perfect heuristic is quite strong.

Claim 3.18. For the problem Π_n (Exm.3.14), even if we do not have the perfect heuristic but rather h^{max} the expansion size of BDDA* will still be exponential.

Proof. We know that h^{max} computes for every state the cost of achieving the most expensive subgoal.[14] In Π_n we have n subgoals $g_1, ..., g_n$. Given

[14] h^{max} is formally defined on STRIPS rather than FDR problems wherein negative preconditions are not allowed. However, given the naive translation where each FDR variable, value pair is a STRIPS fact, the argument follows.

all the states reachable after n steps, any state where the variables v_i do not encode a permutation on $[n]$ will have a subgoal that is unsolvable. That is, if there is no v_i with the value j, then we can never make g_j true. So, h^{max} recognizes all non-permutations to be unsolvable and they will get assigned heuristic value ∞. Moreover, for all states that do encode permutations and where $v_n = j$ the most expensive subgoal is g_j which is reachable in n more steps since we check the v_i variable values in order of increasing i. This means that $B_{\Pi_n, h^{max}}(n, n)$ is still exponential in n and it must be expanded since it has the same f value as any optimal path. □

Now, from Lemma 3.12 we know that given any heuristic that is strictly dominated by h^{max}, A* will also expand exponentially many nodes. However, due to disadvantageous tie-breaking of BDDA* we cannot generally conclude this for equivalent (or simply dominated) heuristics. However, in this specific case, A* with h^{max} also expands exponentially many states.

Claim 3.19. For the problem Π_n (Exm.3.14), A^* with h^{max} will still expand exponentially many nodes.

Proof. In the initial state every subgoal g_j can be achieved in $n+1$ steps by first setting $v_1 = j$ then arbitrarily assigning the remaining v_i followed by $a_{1,j}$. The heuristic value remains 1 for each subsequent layer i' of reachable states up (excluding) the nth layer, because there has to be some value j that has not been assigned to any v_i where $i < i'$. So, to achieve g_j we then need to set $v_{i'} = j$ and then once we can take action $a_{i',j}$ after another n steps we will have achieved g_j. This also means that all states reachable within $n-1$ steps must be expanded by A^* with h^{max}, because for every state s_i in such reachable layer i, we will have $f(s_i) = g(s_i) + h^{max}(s_i) = i + n + 1$. So for every such layer the f-value increases and thus all states in the prior layer have to be explored first. Since there are n^{n-1} states in the layers reachable up to $n-1$, A^* must expand exponentially many nodes. □

3.4.2 A Different Permutation

In Section 3.4.1, we described a planning problem where having a good (even perfect) heuristic will exponentially worsen the performance of BDDA*. However, we also saw that for this problem, blind symbolic search resulted in BDDs of polynomial size which could also be generated in polynomial time. So, we still need to construct a problem where hybrid search is guaranteed to do better that blind symbolic search.

Example 3.20. Consider the following family of unit-cost problems $\Pi'_n = (V_n, A_n, \mathcal{I}_n, \mathcal{G}_n)$:

- $V'_n = \{v_1, ..., v_n\} \cup \{x\}$ such that for each $i \in [n]$ we have $\mathcal{D}_{v_i} = \{0, ..., n\}$ and also $\mathcal{D}_x = \{0, ..., n\}$.
- $A'_n = \{a_{i,j} \mid i, j \in [n]\}$ such that:
  - $a_{i,j} = \langle x = i - 1 \wedge v_j = 0, x = i \wedge v_j = i \rangle$

- In the initial state all variables are set to 0
- In the goal state for all $i \in [n]$ we have $v_i = i$

In this problem we assign the numbers 1 through n a slot in increasing order by maintaining a counter to indicate which variable gets assigned a location next.

Claim 3.21. For the problem Π'_n (Exm. 3.20) the v_i variables can take on every possible permutation on 0 through $n-1$ after $n-1$ actions from the initial state.

Proof. This follows immediately from the construction of the problem. □

Claim 3.22. There is a single optimal path that solves Π'_n (Exm. 3.20) and it has cost n.

Proof. Since we can only assign each of the possible v_j values once and the v_j's are fully specified in the goal, we must at each counter step i take action $a_{i,i}$. Before n steps we cannot satisfy the goal as any action the value n cannot have been assigned by restrictions on the counter. □

We know that BDDs cannot represent arbitrary permutations in a concise, i.e. subexponential, manner. So, by constructing a problem that builds permutations arbitrarily, we guarantee that symbolic search without any further heuristic information will perform badly.

Lemma 3.23. For Π'_n (Exm. 3.20), blind symbolic search has an expansion size exponential in n.

Proof. As the optimal path has length n (Claim 3.22), blind symbolic search must expand $n-1$ BDDs. The final expanded BDD must contain all permutations of 0 through $n-1$ (Claim 3.21). Since BDDs need exponentially many nodes in n to represent a permutation of size n, the statement holds. □

So, blind symbolic search will not perform well on this problem. However, to construct a grander problem where hybrid search will do well, we need to ensure that heuristic explicit search is a good strategy for this problem.

Lemma 3.24. For Π'_n (Exm. 3.20) determining which states are solvable can be done in time polynomial in n and the perfect heuristic restricted to the solvable states is computable in polynomial time.

Proof. Consider the goal count heuristic that adds 1 to each variable that has not yet reached it's goal value. For any solvable state, it must be that there is still a sequence of actions from that state that leads to a goal state. This means that before each step $i \leq n$, we must have $x = i - 1$ and the values less than i assigned to their goal value, i.e. $v_j = j$ for $j < i$, and the other v_j's still set to 0. In such a state $n - i + 1$ of the v_j's are not yet assigned and exactly as many actions are required to solve it. □

Lemma 3.25. A^* with the goal count heuristic expands n nodes to solve Π'_n (Exm. 3.20).

Proof. From the initial state ($h^{gc}(\mathcal{I}_n) = n$) all successors that do not assign the value 1 to it's goal location still have heuristic value n. The action $a_{1,1}$ that does assign $v_1 = 1$ has a successor with goal count heuristic value $n - 1$. So, this successor will be expanded next. Since this will hold for each subsequent step, that assigning the value to the correct location decreases the heuristic value while all other actions do not, we will expand exactly the $n - 1$ states on the optimal path before reaching the goal. □

3.4.3 Putting the Permutations Together

So, from the claims above we have constructed a problem Π_n in Section 3.4.1 whose search will blow-up exponentially when we use a common heuristic like h^{max} while blind symbolic search is guaranteed to be polynomial. Moreover, from Section 3.4.2, we have a problem Π'_n and a (goal count) heuristic that will require exponentially many nodes for blind symbolic search, but where A^* finds a plan in linear time.

Theorem 3.26. *There exists a planning problem Π''_n and a variable split such that:*

- *A^* on the full problem requires exponential time given any heuristic that is strictly dominated by the perfect heuristic as well as given a specific heuristic computable in polynomial time.*
- *BDDA* on the full problem requires exponential space given a perfect heuristic as well as a specific heuristic computable in polynomial time.*
- *Blind symbolic search on the full problem requires exponential space.*
- *The hybrid algorithm runs in polynomial time and space given the heuristic computable in polynomial time.*

Proof. Let Π''_n be the disjoint union of the problems Π_n and Π'_n described in Sections 3.4.1 and 3.4.2 respectively. Assign all variables from the subproblem 3.4.1 to V_S and those from 3.4.2 to V_E.

Consider three heuristics, first the polynomially computable h_{poly} which for any state s in the state space of Π''_n returns the sum of h^{max} for the part of the state from Π_n from 3.4.1 and goal count for the part of the state from Π'_n from 3.4.2. Additionally, we consider the perfect heuristic h^* and lastly any strictly dominated heuristic, $h^{<*}$.

The exponential blow-up in the expansion size of BDDA* given heuristics h^* and h_{poly} follows from 3.16 and 3.18 respectively.

The exponential blow-up of states expanded by A* given $h^{<*}$ follow from the the result for BDDA* with h^* and 3.12. For h_{poly} it follows from 3.19.

The exponential space required by blind symbolic search follows from 3.23.

Finally, it follows from 3.25 and 3.15 that the hybrid search algorithm with the proposed variable split will find an optimal plan in polynomial time given h_{poly}. We will first generate the specific permutation explicitly, as performing these actions will reduce the heuristic while applying the symbolic only actions keep the heuristic value the same. Then once the explicit part is in the goal state we will apply the symbolic only actions resulting in the consecutively smaller BDDs in the build phase and quadratic counting BDDs thereafter. □

4 Splitting the Variables

In the previous chapter we have worked out a problem instance where hybrid planning outperforms other search strategies. From the construction it was clear for which variable split the hybrid algorithm would perform well. It is also clear that with a different split the hybrid algorithm would perform worse. For example, if we swap the explicit and the symbolic variables, we need to expand exponentially many search nodes or hybrid states since we already need to go through exponentially many explicit states. Additionally, in these search nodes we will also expand BDDs of exponential size.

In general, given a planning problem, we want to be able to find the appropriate variable split.

4.1 The Search Effort

In order to quantify what constitutes a good variable split, we need to be able to compare the performance of hybrid search given different variable splits.

Definition 4.1. Given a planning problem $\Pi = (V, A, c, \mathcal{I}, \mathcal{G})$ and a variable split V_E, V_S, we define the *search effort* of hybrid search *effort*(Π, V_E, V_S) as $|V_E|$ times the number of search nodes expanded during the search plus the size of all non-terminal nodes in the expanded BDDs[15].

The reasoning behind this definition is that each expanded search node has an explicit and a symbolic part. For the explicit part each explicit variable value has to be specified, so $|V_E|$ values need to be stored. For the symbolic part we need to store the BDD representing the possible symbolic states.

We observe that given this definition specifying a single state with V variables has effort $|V|$ whereas a single state in a BDD requires $\sum_{v \in V} \log_2(|D_v|)$ inner nodes. So, if we use binary values, the *effort* of representing a single state explicitly is equivalent to doing so in a BDD.

Now, in order to make our reasoning more concise we will assume that we have a hybrid algorithm implementation with duplicate state detection. For explicit search the implementation of this is easy. If we are working with consistent heuristics, we need to keep a set of states already explored. Then, whenever we generate a successor already in that list, we do not need to explore it. If we have inconsistent heuristics but also for plan reconstruction, we need a mapping of states to their minimum g-value. Then, when we find a state again at a higher path cost, we do not explore it again. If we find a state again at a lower path cost, which can happen with inconsistent heuristics, we still need to re-expand it to guarantee that we find optimal plans. This is illustrated in Figure 10.

For symbolic planning it is not necessarily the case that re-expanding states at a higher path cost will result in smaller BDDs. For example,

[15] A BDD is expanded when we compute it's non-zero cost successors, the expanded BDD consists of all symbolic states reachable at that

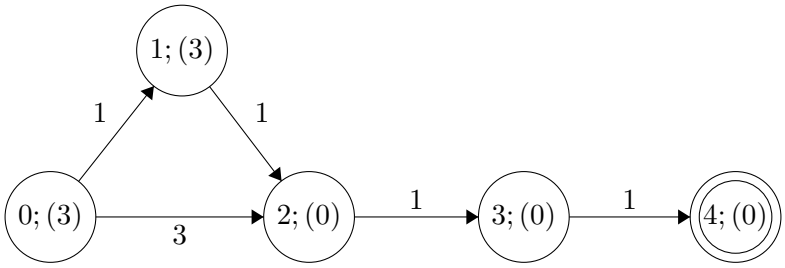

Figure 10: Planning problem with state; (heuristic value) where states 2 and 3 need to be re-expanded by A* due to the inconsistent heuristic.

when we can reach a set of states at some cost where certain variables can take on any of their values but some have already been reached at lower cost. Then, without duplicate detection the variables do no need any nodes in the BDD whereas the filtered version does need nodes associated with those variables. Still, the described implementations of general symbolic search (Alg. 4), BDDA* (Alg. 5) and hybrid search (Alg. 7) do filter out states that have been reached at a lower path cost.

The assumption of duplicate detection allows us to reason about the search effort with greater ease as we can upper bound the number of the (original problem) states explored and so also upper bound the size of the BDDs that represent the symbolic parts of these states.

Namely, that to represent m states over k binary variables, unless $m, k = 1$ we will always use fewer nodes than $m \cdot k$. This holds because no variable can occur more than m times because otherwise the BDD is not reduced. For the first $i \leq k$ variables in the ordering we can only have $\max(2^{i-1}, m)$ nodes, i.e. the first variable needs to be a single root node and thereafter it can grow like a binary tree at most. So, we must have fewer than $m \cdot k$ nodes in the BDD.

Using this, it follows that given a problem with binary variables and no further heuristic information, the best way to split the variables is to make all variables symbolic.

Proposition 4.2. Given a planning problem $\Pi = (V, A, c, \mathcal{I}, \mathcal{G})$ where for all $v \in V$ we have $|D_v| = 2$ and no further heuristic information, the search effort is minimal when $V_S = V$.

Proof. Let Π be such a problem. We will first assume that there are no zero-cost actions in A. Let V'_E, V'_S be any other split of the variables. As we have some explicit variables, this means that $|V'_E| = k > 0$. Also fix some arbitrary ordering on V'_S. Now, if the problem is solvable, there must be some plan with minimal cost n. As we have no further heuristic information, this means all original problem states with cost less than n need to be explored. We assume advantageous tie-breaking so that the first hybrid state explored with cost n has a goal state, so no states at this cost need to be expanded.

Let m_i be the number of distinct explicit states reachable with cost $i < n$. Then the search effort is the total number of hybrid states ex-

panded multiplied by the number of explicit variables plus the number of BDD nodes of each expanded hybrid state. That is $\mathit{effort}(\Pi, V'_E, V'_S) = k \cdot \sum_{i<n} m_i + \sum_{i<n} \sum_{j \leq m_i} |BDD_{V'_S,i,j}|$ where $BDD_{V'_S,i,j}$ is the BDD over the variables in V'_S expanded at cost i from the initial state corresponding to j which is one of the m_i explicit states at depth i.

Now, if we make all the variables symbolic, we only expand one BDD for each of the $i < n$ search layers. Then, if we take any ordering on the now symbolic variables and prepend that ordering to the one on V'_S, we will need at most than $k \cdot m_i$ nodes to represent the formerly explicit states of each layer i. Then, when we attach the $BDD_{V'_S,i,j}$ for each relevant $j \leq m_i$, we have a (potentially reducible) BDD that represents the states expanded at depth i. So, the total search effort must be upper-bounded by that of the V'_E, V'_S split.

If the problem is unsolvable, there must be some maximal cost n' at which the last new state can be reached. Then all states with path cost up to and including n' need to be expanded, and the argument is analogous.

Finally, if there are zero-cost actions in A, we will also expand, for each $i < n$, the zero-cost layers within i until no new states are generated. Due to the elimination of duplicate states an analogous argument holds. □

However, when we do have access to a heuristic, we generally cannot make any claims about the search effort. In that case, it becomes a trade-off between the strength of the heuristic and the compactness of the BDD representation. That is to say, given a good heuristic that will mostly expand states on shortest goal paths first it might not be beneficial to make a lot of variables symbolic. While BDDs can represent large state sets concisely, this might not way up against the potential drop in search nodes that the heuristic can provide.

We can generalize the result from Proposition 4.2 in case of heuristics that are blind to a subset of the variables.

Definition 4.3. Given a planning problem $\Pi = (V, A, c, \mathcal{I}, \mathcal{G})$, a heuristic h and a subset of the variables $V' \subseteq V$. We say that h is *blind with respect to* V', when for all (reachable) states s and any state s' that is like s except that variables in V' may have changed value that $h(s) = h(s')$.

Pattern databases, by construction, are an example of heuristics that are blind to the variables that are not in the pattern.

Proposition 4.4. Given a planning problem $\Pi = (V, A, c, \mathcal{I}, \mathcal{G})$ where for all $v \in V$ we have $|D_v| = 2$ and a heuristic h that is blind to the variables in $V' \subseteq V$ then the search effort can never increase by putting variables in V' in V_S.

We note that the search effort is not necessarily minimal when we set $V_S = V'$, as it may still be the case that the compression of the state representation on on other variables outweigh the heuristic's potential to reduce the number of search nodes.

4.2 When to Compute a Heuristic

The search effort, as we have defined it, may not fully capture the actual effort of finding (the existence) of plans for problems. For the symbolic part we only consider the number internal nodes in expanded BDDs and not the computations required to get those BDDs in the first place. For the explicit part we assume that variables with larger domains do not take more effort to store.

Moreover, when using explicit search we need to precompute a heuristic. The effort of computing this heuristic is not taken into account into the search effort. So, if we would want to take this into account, it would be useful to know when we should compute a heuristic. That is, when computing the heuristic actually leads to a reduced search effort.

From Helmert and Röger [12], we know that good heuristics perform poorly when there are many states on optimal paths. There are heuristics that are 'worse', for example, an inadmissible heuristic that assigns a value greater than the cheapest path to all states except for states on some fixed cheapest path where it is admissible. In that case, we would expand the states on this goal path before expanding any of the other states.

However, from Lemma 3.12, we know that whenever a heuristic h is strictly dominated[16] by the perfect heuristic all states s on cheapest paths need to be explored by A^* since $h(s) + g(s) < h^*(s) + g(s) = h^*(\mathcal{I})$.

So, intuitively, if we get an input problem with a lot of states on cheapest paths and projecting away a subset of the variables greatly reduces the number of states on cheapest paths, we might want to make those variables symbolic. While we need to specify how to define 'a lot' of states and a 'great reduction', we argue that we would want to make these variables symbolic because even with a good heuristic we would need to expand a lot of combinations of these variables.

Claim 4.5. Let $\Pi = (V, A, c, \mathcal{I}, \mathcal{G})$ be a planning problems and let CPS be the set of states on cheapest paths that are not goal states. For any $V' \subsetneq V$ with, let $\Pi \restriction_{V \setminus V'}$ be the problem projected on the remaining variables, if the number of non-goal states on cheapest paths in this projection $|CPS \restriction_{V \setminus V'}| \approx \frac{|CPS|}{\Pi_{v \in V'} |D_v|}$, then no heuristic strictly dominated by h^* will significantly reduce the search effort when $V' \subseteq V_E$.

Proof. The total number of states over V' is $\Pi_{v \in V'} |D_v|$. When many of these partial states can be extended to states on some cheapest path, then all these partial states need to be explored by any algorithm with any heuristic h that is strictly dominated by h^*. □

However, that a heuristic will not reduce the explicit search effort does not give any guarantees that the symbolic search effort will be low. Consider a unit cost binary counter with n variables where we want to go from the initial all zeroes to all 1, using actions that correspond to increasing the binary value by 1 for each state. Then all the 2^n possible states are on

[16] without strict domination some of the states may not be explored due to tiebreaking.

a cheapest path where the last state is the goal state. Making any subset of variables symbolic will however not reduce the search effort since at each cost layer only a single state can be reached.

Conversely, if the number of states on shortest paths is low, the search effort can still be reduced by putting variables in the symbolic set. Consider a problem with n light switch variables that we can toggle on and off and a stick variable such that when we have the stick we can flip all switches at once if all switches are in the same position. $V = \{l_1, ..., l_1, r\}, A = \{t_{1,on}, ..., t_{n,on}, grab, t_{all}\}$ where $t_{i,on} = \langle l_i = 0, l_i = 1\rangle, grab = \langle r = 0, r = 1\rangle, t_{all} = \langle r = 1 \wedge \bigwedge_{i\leq n} l_i = 0, \bigwedge_{i\leq n} l_i = 1\rangle$ and all actions cost 1 except t_{all} costing $n - 2$.

Then the cheapest path is to grab the stick and then flip all switches at once costing $n - 1$. However, given any heuristic strictly dominated by h^* all states reachable at cost $n - 2$ must also be explored. So, these $\sum_{0\leq i\leq n-2} \binom{n}{i} = 2^n - \binom{n}{n-1} - \binom{n}{n} = 2^n - n - 1$ states need to be explored. This means that the explicit search effort will be $n + 1$ times this number of states.

However, if we make the light variables l_i symbolic, then at each layer costing j we need a BDD to represent that j of the lights are switched on, which we can do in BDDs with a size in $\mathcal{O}(n^2)$, so as we expand $n - 2$ of such BDDs the search effort with the switches symbolic will be $\mathcal{O}(n^3)$.

Furthermore, we assumed that we have access to the number of states on cheapest paths. However, if we could efficiently decide for any state whether it is on a cheapest path, we would solve any planning problem efficiently by simply querying whether the initial state is on a shortest path. So, it is not feasible to get only a problem as input and then decide whether to compute a heuristic, let alone which one.

4.3 The Variable Ordering Returns

So, then the question remains, given a problem and a heuristic, what variable split and ordering will minimize the search effort. This also points to another flaw in the definition of the search effort. We observed that the variable ordering can have a substantial impact on the number of nodes in a BDD. So, it is clear that the search effort would be minimal when we choose the optimal variable ordering at each expanded hybrid search node.

However, while existing implementation of symbolic search do employ some measure of dynamic variable reordering, the runtime overhead of optimizing the variable ordering is often exponential [16, 15]. This tracks with the finding of Bollig and Wegener [2] on the NP-hardness of finding an optimal variable ordering for any BDD.

However, if we fix the variable ordering in advance, we may come up with a different optimal split.

Claim 4.6. The optimality of a variable split is dependent on the variable ordering.

Proof. Consider the following clas of unit cost planning problems $\Pi_n = (V, A, \mathcal{I}, \mathcal{G})$ with $V = \{v_1, ..., v_n, v'_1, ..., v'_n\} \cup \{x_0, ..., x_n\} \cup \{goal\}$ and $A =$

$A = \{a_{i,1}, a'_{i,1} | 1 \leq i \leq n\} \cup \{b_j, b'_j, c_j, c'_j | 2 \leq j \leq n\} \cup \{d, d'\}$ such that the unprimed actions are defined as follows:

- $a_{i,1} = \langle x_0 \wedge \neg x_1 \wedge \neg v_i \wedge \neg v'_i, x_1 \wedge v_i \rangle$
- $b_j = \langle x_{j-1} \wedge \neg x_j \wedge \neg v_{j-1} \wedge \neg v'_{j-1}, x_j \wedge v_{j-1} \rangle$
- $c_j = \langle x_{j-1} \wedge \neg x_j \wedge v_{j-1}, x_j \rangle$
- $d = \langle \bigwedge_{i \leq n} v_i, goal \rangle$

The prime actions are equivalent except that each occurence of a primed v variable is swapped for its unprimed counterpart and vice versa. In the initial state $\mathcal{I}$ all variables except x_0 are set to 0. To be a goal state, the variable $\{goal\}$ must be true.

This means that from the initial state we can apply $a_{i,1}$ or $a'_{i,1}$ to set exactly one v_i or v'_i to true, resulting in $2n$ different states. Thereafter, at each step j we must apply a corresponding (primed or unprimed) b or c action. If v_{j-1} or v'_{j-1} has been set, we use c_j or c'_j respectively to just increase the x counter. If neither is set we can choose b_j to set v_{j-1} and b'_j to set v'_{j-1}.

Also consider the following heuristic h that is $3n^2$ whenever any primed and unprimed v_i, v'_i variable pair are true at the same time. Otherwise it has the value $n - \#v_i - \#v'_i$.

Now, we claim (1) that for variable ordering where all the primed variables occur before the first unprimed one or vice versa, $V_S = V$ is not an optimal split.

However, (2) for the variable ordering $\langle g, x_0, ..., x_1, v_1, v'_1, ..., v_n, v'_n \rangle$ the split $V_S = V$ is optimal.

To show (1), we will first show that the search effort of $V_E = V$ is strictly less than $V_S = V$. Consider $V_E = V$. Then, because of the heuristic, no state with both a primed and an unprimed v variable true will be explored. However, as the heuristic is strictly dominated by h^* all states on shortest paths will. The initial state and its $2n$ immediate successors are on shortest paths. Thereafter, each state has exactly one successor on a shortest path. In fact at each path cost $1 < g \leq n$ there are still $2n - 2(g-2)$ states on shortest paths. At cost $n+1$, we reach a goal state with a single d action. So, the total number of states explored is $1 + 2n + \sum_{1<g\leq n} 2n - 2(g-2) = 1 + 2n + 2n^2 - n(n+1) + 4n = n^2 + 5n + 1$.

This means the total search effort is in $\mathcal{O}(n^3)$ as it is the number of variables $3n+2$ times the number of explored states.

Wlog assume that all unprimed variables occur before the first primed one. When we have $V_S = V$, we must explore all reachable states at cost less than $n+1$. To represent the set of states reachable at cost n, with a variable ordering where all unprimed variables precede the first primed one, we must use at least 2^n nodes. This is the case because the unprimed variables can have all possible combinations of values, and we need to keep track of them to ensure that they are opposite to the primed variables, i.e. $v'_i = \neg v_i$.

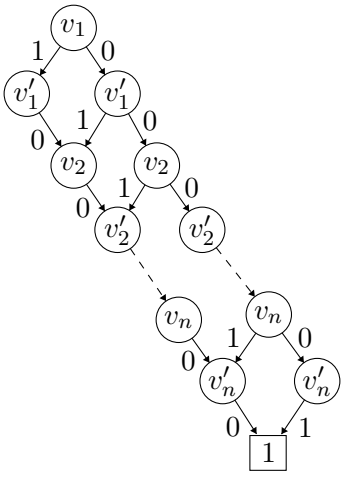

Figure 11: BDD of v, v' states reachable at cost 1 with $v_1 \prec v'_1 \prec v_2 \prec v'_2 \prec ... \prec v_n \prec v'_n$ using $4n - 1$ internal nodes. Paths to 0 sink removed for clarity.

Now, we will show (2). First, the initial state takes $3n + 2$ nodes. At cost 1, exactly one v or v' variable is set to true. This is is represented in Figure 11. At each cost $i + 1$ thereafter, we know that the first i v, v' variables have been set and that they are not both set. So, we get 3 nodes for each such v_i, v'_i pair. For the remaining pairs at most one has v_j or v'_j set. This is represented by the same BDD as in Figure 11, except rooted at the first v_j and the rightmost v'_n is not necessary anymore as it will also be a valid state if that v'_n is 0.

So, at cost $i + 1$ for $1 \leq i < n - 1$ we get $3i + 4(n - i) - 2 = 4n - i - 2$ v, v' nodes. The x and *goal* variables have a fixed value at each depth so will only add one node per variable. Since all BDDs are linear in n, we get a search effort that is at most quadratic in n.

Finally, we can not do better that $V = V_S$. We can make any of the x and goal variables explicit as they are fixed at each cost, but this will keep the search effort the same. If we make any number k of v or v' variable explicit, then we reduce each BDD of reachable states after the initial state by at most $2k$ variables, but we always need to keep at least 2 more BDDs at each depth. So, we get an explicit contribution to the search effort of at least $2k$. So, any split with v or v' variables in V_E will not be optimal. □

Moreover, if we fix the ordering in advance, it may make an exponential difference to the search effort. For example, if we consider the problem from Claim 4.6 without any heuristic and the two described (types of) variable orders.

However, by assuming both a fixed variable ordering as well as a heuristic, we can concisely describe the problem of the optimal split as a decision problem:

OPT-VAR-SPLIT: Given a planning problem Π, a heuristic h, a total

order on the variables of Π and a size bound k, does there exist a split V_E, V_S such that $\mathit{effort}(\Pi, V_E, V_S) \leq k$.

While we have an intuition that this problem too is difficult, i.e. at least NP-hard, we have no reduction for it. The intuition for the difficulty of OPT-VAR-SPLIT stems from the fact that it is difficult to pre-empt the effects of the heuristic and the variable ordering on the search effort. To know whether and if so how many states a heuristic will avoid, we generally need to construct a lot of the state space. In addition, we need to compute how the reduction in search effort by the heuristic relates to the possibilities of compression by BDDs.

Similarly, attempts to come up with a reduction fell short because from for example SAT we need to encode a planning problem and a heuristic and a variable ordering and the effects the latter two have on the search effort of the problem, using just clauses and literals. We presume this will require several gadgets.

5 Conclusion

In this thesis we have investigated several canonical planning algoritms from the literature and introduced a new hybrid algorithm. Explicit A* state based search using heuristics finds optimal plans without increasing the search space, but may still expand exponentially many nodes when the heuristic is almost perfect and there are exponentially many states on cheapest paths. This occurs, among others, when we are counting objects but the order of objects does not matter. In contrast, symbolic search algorithms using BDDs are quite efficient at such counting, but require exponential space to represent permutations of objects.

In general, symbolic search has a lot of potential to explore a vast number of states efficiently. Unfortunately, when we use heuristics with symbolic search, we may exponentially increase the search effort. So, by using a hybrid explicit-symbolic search algorithm we can still use (partial) heuristics while also exploring more than one state at the same time.

In fact, we have demonstrated that there is a class of problems and a polynomial time computable heuristic such that the hybrid algorithm finds a plan in polynomial time and space, while A* requires polynomial time and blind symbolic search as well as BDDA*, symbolic search with a heuristic, require exponential space.

While this shows that hybrid search expands the set of problems that can be solved efficiently, we have also noted some limitations in the use of the hybrid algorithm. In order to solve part of the problem explicitly and part of it symbolically, we need to decide how to split the variables into their explicit and symbolic partition. In order to compare the one such split against another we defined a notion of search effort. However, this search effort was not just dependent on the variable split, but also on the variable ordering, as they yield (possibly exponetially) differing BDD sizes, and the heuristic as it may guide the planner towards the goal.

So, while our intuition is that determining this split, even when we control for the variable ordering and the heuristic, is difficult, this remains to be shown formally.

5.1 Future Research

First, we already observed some flaws in the hybrid search effort as a theoretical notion because it ignores some possibly expensive computations. So, the first practical avenue of future research is to actually implement the hybrid algorithm. This will allow us to compare the performance, as measured by computation time, of truly hybrid search, i.e. there are both explicit and symbolic variables, against both explicit and symbolic search as well as other state-of-the-art planners.

Moreover, an implementation may be useful to get a better understanding of what constitutes a good splitting of the variables. This may be helpful because with more computational power we can get preliminary evidence of whether, for example, causal graphs that represent the dependencies between variables based on the actions, can aid finding optimal

variable splits.

This line of research may also have links to decoupled search [11, 23] where such causal graphs are used to partition the problem variables into several sets that can be explored efficiently due to the lack of interaction between the variables in different sets. This also means that the decoupled approach is limited to cases where these causal graphs have specific structures. Possibly, hybrid search can be used when these graphs do not have the correct topologies as its application is not restricted by variable dependencies.

Finally, another future research direction is to formally show the difficulty of finding a good variable split. If this is the case, we can also investigate the possibilities for good approximations and heuristics[17] that will, on average, result in a variable split where the hybrid algorithm performs well.

[17]Heuristics to estimate good variable splits rather than guide the planner in finding plans.

References

[1] Christer Bäckström and Bernhard Nebel. "Complexity results for SAS+ planning". In: *Computational Intelligence* 11.4 (Nov. 1995), pp. 625–655. DOI: 10.1111/j.1467-8640.1995.tb00052.x.

[2] B. Bollig and I. Wegener. "Improving the variable ordering of obdds is NP-complete". In: *IEEE Transactions on Computers* 45.9 (1996), pp. 993–1002. DOI: 10.1109/12.537122.

[3] Randal E. Bryant. "Graph-based algorithms for boolean function manipulation". In: *IEEE Transactions on Computers* C–35.8 (Aug. 1986), pp. 677–691. DOI: 10.1109/tc.1986.1676819.

[4] Tom Bylander. "The computational complexity of propositional STRIPS planning". In: *Artificial Intelligence* 69.1-2 (1994), pp. 165–204.

[5] Joseph C. Culberson and Jonathan Schaeffer. "Pattern Databases". In: *Computational Intelligence* 14.3 (Aug. 1998), pp. 318–334. DOI: 10.1111/0824-7935.00065.

[6] Stefan Edelkamp. "Planning with Pattern Databases". In: *Proceedings of the 6th European Conference on Planning (ECP'01)* (2001). Ed. by A. Cesta and D. Borrajo, pp. 13–24.

[7] Stefan Edelkamp and Peter Kissmann. "Limits and Possibilities of BDDs in State Space Search". In: *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 46–53. URL: https://doi.org/10.1007/978-3-540-85845-4_6.

[8] Richard E Fikes and Nils J Nilsson. "STRIPS: A new approach to the application of theorem proving to problem solving". In: *Artificial intelligence* 2.3-4 (1971), pp. 189–208.

[9] Yang Gao and Steve Chien. "Review on space robotics: Toward top-level science through space exploration". In: *Science Robotics* 2.7 (2017), eaan5074.

[10] Fernando Garrido and Paulo Resende. "Review of decision-making and planning approaches in automated driving". In: *IEEE Access* 10 (2022), pp. 100348–100366.

[11] Daniel Gnad and Jörg Hoffmann. "Star-topology decoupled state space search". In: *Artificial Intelligence* 257 (2018), pp. 24–60.

[12] Malte Helmert and Gabriele Röger. "How good is almost perfect?" In: *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*. AAAI'08. Chicago, Illinois: AAAI Press, 2008, pp. 944–949. ISBN: 9781577353683.

[13] Ngai Ngai William Hung. "Exploiting symmetry for formal verification". MA thesis. University of Texas at Austin, 1997.

[14] Erez Karpas and Daniele Magazzeni. "Automated planning for robotics". In: *Annual Review of Control, Robotics, and Autonomous Systems* 3.1 (2020), pp. 417–439.

[15] Peter Kissmann, Stefan Edelkamp, and Jörg Hoffmann. "Gamer and dynamic-gamer–symbolic search at ipc 2014". In: *IPC-8 planner abstracts* (2014), pp. 77–84.

[16] Peter Kissmann and Jörg Hoffmann. "BDD ordering heuristics for classical planning". In: *Journal of Artificial Intelligence Research* 51 (2014), pp. 779–804.

[17] Peter Kissmann and Jörg Hoffmann. "What's in it for my BDD? On causal graphs and variable orders in planning". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 23. 2013, pp. 327–331.

[18] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A modern approach (Third Edition)*. Prentice-Hall, 2010.

[19] Shermin Sherkat, Thomas Wortmann, and Andreas Wortmann. "Two Decades of Automated AI Planning Methods in Construction and Fabrication: a Systematic Review". In: *ACM Computing Surveys* (2025).

[20] Detlef Sieling and Ingo Wegener. "Reduction of OBDDs in linear time". In: *Information Processing Letters* 48.3 (1993), pp. 139–144. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(93)90256-9. URL: https://www.sciencedirect.com/science/article/pii/0020019093902569.

[21] AR Sousa and JJPZS Tavares. "Toward automated planning algorithms applied to production and logistics". In: *IFAC Proceedings Volumes* 46.24 (2013), pp. 165–170.

[22] David Speck, Florian Geißer, and Robert Mattmüller. "When perfect is not good enough: On the search behaviour of symbolic heuristic search". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 30. 2020, pp. 263–271.

[23] David Speck and Daniel Gnad. "Decoupled search for the masses: A novel task transformation for classical planning". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 34. 2024, pp. 546–554.

A Hybrid Plan Reconstruction

You can find a pseudocode implementation of the reconstruction algorithm in 8.

The presence of zero cost actions makes the plan reconstruction a bit tedious. Given an explicit state s_E, any set of symbolic states that is newly reached by a zero cost action will not just be put in the relevant g, S_E bucket, but also get an extra zero-cost action index i. For each subsequent set of symbolic states with explicit state s_E that are reached at cost g, we increase the i counter.

We call the plan reconstruction when we first find that $reached[g][s_E]$ contains a hybrid goal state. If this is first achieved by a zero-cost action, then there is no goal state in the first $reached[g][s_E]$ that is constructed of non-zero cost action. So, if $i_{max}(g, s_E)$ is defined at the start, the last applied action must have been zero cost. In that case, we take the current goals *curr* to be the set of symbolic goal states in that final layer, and decrease the $i_{max}(g, s_E)$.

If the last action was not zero-cost we also set the appropriate *curr* goal set.

The procedure then is not essentially different if we last applied zero or non-zero cost actions except that we look through the respective set of actions.

At the start of each iteration we check if we are already in the initial (hybrid) state. Then for each hybrid $s_E, curr$ we compute its hybrid predecessor for the action a. If it is defined, i.e. $s'_E \neq \bot$ and $sym\_pre \neq \emptyset$, we check whether $reached[g - c(a)][s'_E] \cap sym\_pre$ has any states. If it does, we know that a is part of a cheapest plan and we need to find whether the action preceding it was zero-cost or not.

However, even if $i_{max}(g - c(a), s'_E)$ is defined, it is not necessarily the latest (or any) zero cost action that will have led to a current symbolic goal state by application of a. As we may have applied several zero cost actions consecutively until convergence of the set of states $reached[g - c(a)][s'_E]$. So, we decrement the i index until we find a hybrid predecessor. If we do not, in the zero-cost added layers, then the hybrid predecessor was obtained by a non zero-cost action. We know this since the intersection with $reached[g - c(a)][s'_E]$ was non-empty.

In both cases, whether in a zero-cost layer or not, we update the new symbolic goal set *curr* as well as the hybrid state s_E and prepend a to the plan.

This algorithm too could be improved for efficiency. If the last action was zero-cost, then, unless it was in the first zero-cost layer (with index 0), we know that the predecessor must also be zero cost and so we only need to check the last triple indexed reached.

It is guaranteed to terminate because we only call this algorithm when we have reached a hybrid goal state and we must have generated this by applying sets of explicitly equivalent actions and keeping track of which state could be (newly) reached at which cost.

Algorithm 8: Plan Reconstruction: Hybrid Search

Input: *reached*, *exp_goals*, *sym_goals*, A, g, s_E (the path cost and explicit state at which the goal was reached)

Output: A plan as a sequence of actions.

Assume that a query of *reached* with any undefined index maps to the empty set.

```
1  if reached[g][s_E][i_max(g, s_E)] = ∅ then
2      curr ← reached[g][s_E] ∩ sym_goals
3      last_zero ← 0
4  else
5      curr = reached[g][s_E][i_max(g, s_E)] ∩ sym_goals
6      i_max(g, s_E) ← i_max(g, s_E) − 1
7      last_zero ← 1
8  plan ← []
9  while s_E ≠ I_E ∨ I_S ∉ curr do
10     if last_zero = 0 then
11         for a ∈ A^{>0} do
12             s'_E ← pre_im(s_E, a)
13             if s'_E ≠ ⊥ then
14                 sym_pre ← pre_im_S(curr, a)
15                 inter ← reached[g − c(a)][s'_E] ∩ sym_pre
16             if inter ≠ ∅ then
17                 if i_max(g − c(a), s'_E) is defined then
18                     while reached[g − c(a)][s'_E][i_max(g − c(a), s'_E)] ∩
                        sym_pre = ∅ do
19                         reached[g−c(a)][s'_E] ← reached[g−c(a)][s'_E] \
                            reached[g − c(a)][s'_E][i_max(g − c(a), s'_E)]
20                         i_max(g − c(a), s'_E) ← i_max(g − c(a), s'_E) − 1
21                         if i_max(g − c(a), s'_E) < 0 then
22                             curr ← reached[g − c(a)][s'_E] ∩ sym_pre
23                             last_zero ← 0
24                             go to line 28
25                     curr ←
                        reached[g−c(a)][s'_E][i_max(g−c(a), s'_E)]∩sym_pre
26                     i_max(g − c(a), s'_E) ← i_max(g − c(a), s'_E) − 1
27                     last_zero ← 1
28                 s_E ← s'_E
29                 plan ← [a] + +plan
30     if last_zero = 1 then
31         for a ∈ A^0 do
32             // equivalent procedure as in lines 12-29 //
```

B Perfect Explicit Heuristic Equivalence

We prove the equivalence of the two definitions of the perfect explicit heuristic. We will first provide them again.

Definition B.1. We distinguish the deduced and direct perfect explicit heuristics.

- The ***deduced*** *perfect explicit heuristic* $h^*_E : \mathcal{S}_E \to \mathbb{R}^+_0 \cup \{\infty\}$ is deduced from the perfect heuristic h^* such that $h^*_E(s_E) := \min_{s_S \in \mathcal{S}_S} h^*(s_E, s_S)$.
- The ***direct*** *perfect explicit heuristic* $h^{*'}_E := \min_{S_S | (s_E, S_S) \in \mathcal{S}} h^*_{E,S}(s_E, S_S)$, where $h^*_{E,S}$ computes for every hybrid state (s_E, S_S) the cheapest path to a hybrid goal state.

Claim B.2. For any planning problem Π and variable split V_E, V_S and any explicit state s_E we get equivalence of the deduced and direct perfect explicit heuristics, $h^*_E(s_E) = h^{*'}_E(s_E)$.

Proof. Let s_E be arbitrary, and pick any s_S such that $h^*(s_E, s_S)$ is minimal at cost n. Then there must be a sequence of actions $\langle a_1, ..., a_m \rangle$ such that $\mathcal{G} \subseteq tr((s_E, s_S), \langle a_1, ..., a_m \rangle)$. Now, if we take the explicitly equivalent class A_i for each a_i and the relevant explicit state, then all transitions $(s_E, s_S) \to^{A_1} (s_{E,1}, S_{S,1}) \to^{A_2} ... \to^{A_m} (s_{E,m}, S_{S,m})$ must be in the hybrid state space and the last state must be a hybrid goal state. So, $h^{*'}_E(s_E) \geq n$

Moreover, if there is a hybrid state S'_S such that $h_{E,S}(s_E, S'_S) < n$, then there is a sequence of explicitly equivalent action classes $A'_1, ..., A'_k$ costing less than n that result in a hybrid goal state. So, then there exists a sequence of actions $a'_1, ..., a'_k$ costing less than n and hybrid states $(s_E, s'_S), (s_{E,1}, s_{S,1}), ..., (s_{E,k}, s_{S,k})$ with $s'_S \in S'_S$ and each $(s_{E,i}, s_{S,i})$ in the complete state after applying $a'_i \in A'_i$, such that $\mathcal{G} \subseteq (s_{E,k}, s_{S,k})$. So, then $h^*(s_E, s'_S) < n$ which cannot be.

The reasoning for opposite direction is analogous. □