Additive–Disjunctive Heuristics for Optimal Planning

Andrew Coles, Maria Fox, Derek Long and Amanda Smith
Department of Computer and Information Sciences,
University of Strathclyde, Glasgow, G1 1XH, UK
email: firstname.lastname@cis.strath.ac.uk

Abstract

The development of informative, admissible heuristics for cost-optimal planning remains a significant challenge in domain-independent planning research. Two techniques are commonly used to try to improve heuristic estimates. The first is disjunction: taking the maximum across several heuristic values. The second is the use of additive techniques, taking the sum of the heuristic values from a set of evaluators in such a way that admissibility is preserved. In this paper, we explore how the two can be combined in a novel manner, using disjunction within additive heuristics. We define a general structure, the Additive–Disjunctive Heuristic Graph (ADHG), that can be used to define an interesting class of heuristics based around these principles. As an example of how an ADHG can be employed, and as an empirical demonstration, we then present a heuristic based on the well-known additive h^m heuristic, showing an improvement in performance when additive–disjunctive techniques are used.

1 Introduction

When performing cost-based optimal planning in a state-space search setting, an essential component of the system is an admissible heuristic: one which never over-estimates the cost of reaching a state. The accuracy of heuristic estimates is key to the performance of planners (Helmert & Mattmüller 2008). The challenge is to find an admissible heuristic that is as informative as possible and, subject to admissibility guarantees, a larger heuristic estimate is more informative. Two techniques are commonly used to maximise the heuristic values of states:

- Using *disjunctive heuristic* techniques, in which a state is evaluated by taking the maximum of several admissible heuristics.
- Using *additive heuristic* techniques (Haslum, Bonet, & Geffner 2005; Haslum *et al.* 2007), where the evaluation of a state is a sum across several admissible heuristic evaluators, where the evaluators are constrained in such a way that the resulting sum is admissible.

In practice, these are combined in a somewhat simplistic manner: the maximum value of several heuristics is computed, some of which may be additive. We refer to heuristics following this approach as *disjunctive–additive heuristics* (DAHs). The DAH approach fails to exploit the potential for disjunction *within* additive heuristics themselves: *additive–disjunctive heuristics*. In existing additive heuristics, each of the component heuristic evaluators uses the same approach. For example, the additive h^m heuristic (Haslum, Bonet, & Geffner 2005) is formed by taking the sum across h^m evaluators (Haslum & Geffner 2000). The motivation for disjunctive heuristics is that the best evaluator can vary on a per-state basis and the restriction of additive heuristics to families of similar evaluators fails to address this.

In this paper, we present a novel way for combining both additive- and disjunctive-heuristic techniques. We first discuss how disjunctive and additive techniques can be combined to form a simple additive–disjunctive heuristic, where each component evaluator within an additive heuristic employs disjunction. In doing so, we maximise the contribution of each component evaluator, and hence further exploit the synergy between the chosen evaluators. We then generalise the approach, defining a powerful, general structure for representing heuristic evaluators: an additive–disjunctive heuristic graph (ADHG). To demonstrate the potential power of the approach, we then explore a simple exemplar ADHG heuristic based around the extension of the additive h^m heuristic to include additive–disjunctive techniques and show, empirically, an improvement in performance compared to the original additive h^m heuristic.

2 Background

Optimal planning research can be divided into two streams: makespan-optimal planning, and cost-optimal planning (which includes length-optimal planning, by taking the costs of all actions to be the same). In the first of these, the aim is to find an optimal solution in terms of the total (parallel) plan execution time. A common approach is to encode a GraphPlan (Blum & Furst 1995) planning graph for use with a SAT (Kautz & Selman 1999), CSP (Do & Kambhampati 2001) or IP (van den Briel & Kambhampati 2005) solver, interleaving a search for a solution with an incremental increase in the planning graph size until the encoded representation is solvable.

In cost-optimal planning the aim is to minimise the cost of the plan found. Research in this area is currently dominated by heuristic state-space search, using the combination of an admissible heuristic and an optimal search algorithm (for in-

stance A* or IDA*). Our focus is on admissible heuristics for state-space cost-optimal planning. Defined formally, using the SAS+ representation, a cost-optimal planning problem is a tuple $\Pi = \langle V, O, s_0, s_\star, cost \rangle$ where:

- V is a set of variables. Each $v \in V$ can take a value from a corresponding finite domain D_v, or can be undefined (take the value *undef*).
- O is a set of operators, each $\langle prevail, pre\_post \rangle$. $prevail$ is the set of variable assignments that must hold for o to be applied, but are unchanged by o. pre_post is a set of transitions $v_n = (p \rightarrow q)$, where $v_n = p$ must hold for o to be applied, and the value of v_n is changed to q by its application.
- s_0 is the initial state of the planning problem, specified as variable–value pairs. We use $s[n]$ to denote the value of variable n in state s.
- $s_\star$ is the goal condition: variable–value pairs that must hold in a goal state. We can define the set of goal states $G = \{s \mid s_\star \subseteq s\}$.
- $cost$ is a cost function over operators. In this work, we assume action costs are constant non-negative values. Setting all costs to 1 is equivalent to seeking the shortest plan.

State-space search can proceed either forward or backward. In forward-chaining search, the vertices in the search space represent states and the edges between them represent the application of operators. The problem is to find the cost-optimal path from the initial state, s_0, to one of the goal states in G by following edges forwards from states, with the selected edges forming the plan.

In regression search, the problem is inverted. Search is again over states, but edges denote *regression* through operators, rather than their application. The inversion of operators for regression is a straightforward manipulation of their descriptions. The problem is to find a path from $s_\star$ to a vertex, $s \subseteq s_0$. The plan is then the reverse sequence of the edges from $s_\star$ to s.

3 Heuristic Guidance in Practice

In both variants of search the challenge is to find informative heuristics for guidance. In optimal planning, this is particularly challenging, as it is essential that the heuristics do not sacrifice guarantees of optimality, so distance estimates must remain admissible. That is, in forward-chaining search, the heuristic value $h(s)$ must never exceed the cost of the optimal operator path from s to a state in G and in regression planning, $h(s')$ must not exceed the cost of the optimal operator regression path from s' to a state $s \subseteq s_0$.

There is a close relationship between forward-chaining and regression planning and their heuristics. Consider the task of evaluating a vertex in the search space, s. In forward-chaining planning, the heuristic value $h(s)$ is an estimate of the cost of satisfying the facts in $s_\star$ by applying operators forwards from s (initially s_0). In regression planning, although it is natural to consider regressing states through operators to produce a heuristic value, it is also possible to take $h(s)$ to be an estimate of the cost of satisfying the facts in s by applying operators to s_0. In practice, different heuristics are more or less appropriate in each setting. For instance, Pattern Database (PDB) heuristics (such as (Edelkamp 2002; Helmert, Haslum, & Hoffmann 2007)) are particularly effective in forward-chaining search: following preprocessing, the heuristic cost $h(s)$ from s to $s_\star$ can be computed cheaply. The h^m family of heuristics (Haslum & Geffner 2000), on the other hand, are effective with regression search: after preprocessing, the heuristic distance $h(s)$, from s_0 to s, can be calculated efficiently.

3.1 Disjunctive Heuristics

In all cases, the objective is the same: to find large, yet admissible, heuristic values. A larger heuristic value is more informative, leading to better discrimination between choices. A common way to achieve this is by the use of a *disjunctive heuristic*. A disjunctive admissible heuristic is defined as follows:

Definition 3.1 — Disjunctive Heuristic
A disjunctive admissible heuristic dh is one whose evaluation of a vertex is obtained by taking the maximum across several admissible heuristic evaluators:

$$dh(s) = \max_{i=0..n} h_i(s)$$

for an arbitrary collection of admissible heuristic evaluators, $h_0..h_n$. The admissibility of dh follows from the admissibility of $h_0..h_n$.

In effect, a disjunctive heuristic synthesises a more informative heuristic evaluator from a collection of evaluators of which some are better in some states or domains, and some in others. Its success in guiding search is clearly dependent on there being an effective evaluator within the collection and evaluating the disjunction carries a computational cost: several heuristics must be computed instead of just one. Nevertheless, with an appropriate mix of evaluators, the reduced search benefit outweighs the increase in vertex evaluation cost.

3.2 Additive Heuristics

A potentially more powerful way to combine estimates from a collection of heuristic evaluators is by using the sum operator instead of max. Clearly, one cannot sum arbitrary heuristic values and expect to maintain admissibility (otherwise a non-zero heuristic value could be added to itself arbitrarily many times). However, as observed in (Haslum, Bonet, & Geffner 2005) and further in (Felner, Korf, & Hanan 2004), under certain circumstances the sum can be admissible. The following definitions are provided to support an explanation of one way this can be achieved.

Definition 3.2 — Admissible Adjusted-Cost Heuristic
A heuristic evaluator, h, is an admissible adjusted-cost heuristic for a planning problem, $\Pi = \langle V, O, s_0, s_\star, cost \rangle$, if there is a cost function, $cost_h$, called the adjusted cost function for h, such that h is an admissible heuristic for $\Pi' = \langle V, O, s_0, s_\star, cost_h \rangle$, when it is applied to Π.

That is, an admissible adjusted-cost heuristic yields estimated costs for states in the original planning problem that are admissible estimates for the problem with an adjusted cost function.

Definition 3.3 — Additive Heuristic
A collection of admissible adjusted-cost heuristic evaluators $h_0..h_n$ can be combined additively if:

$$\forall o \in O \;\; cost(o) \geq \sum_{i=0..n} cost_{h_i}(o)$$

where $cost_{h_i}$ is the adjusted cost function for h_i. The additive heuristic ah is:

$$ah(s) = \sum_{i=0..n} h_i(s)$$

Theorem 1 *The additive heuristic, ah, using admissible adjusted-cost heuristic evaluators $h_0..h_n$ is admissible.*

Proof: For state, s, with actual goal-distance cost c, derived from operator sequence $o_1, ..., o_k$, the admissibility of h_i implies that $\sum_{j=1..k} cost_{h_i}(o_j) \geq h_i(s)$. Thus, $c = \sum_{j=1..k} cost(o_j) \geq \sum_{j=1..k} \sum_{i=0..n} cost_{h_i}(o_j) = \sum_{i=0..n} \sum_{j=1..k} cost_{h_i}(o_j) \geq \sum_{i=0..n} h_i(s) = ah(s)$.

A straightforward way to create admissible adjusted-cost heuristics is to partition the operators into sets $O_0, .., O_n$ and then to create an admissible adjusted-cost heuristic, h_i, using the adjusted cost function:

$$\begin{array}{ll} part_{O_i}(o) & = cost(o) \quad \text{if } o \in O_i \\ & = 0 \quad\quad\;\; \text{otherwise} \end{array}$$

We will refer to this strategy as a partitioning scheme and to an h_i based on partition $O_0, .., O_n$ as h_{O_i}.

Additive variants have been constructed using a partitioning scheme with each of the heuristics mentioned at the start of Section 3, using h^m, to yield *additive* h^m (Haslum, Bonet, & Geffner 2005) and using PDB heuristics to yield additive PDBs (Haslum *et al.* 2007).

When creating additive heuristics with a partitioning scheme the challenge is to find partitions that maximise the component heuristics. At one extreme all operators are in a single partition and, at the other, each partition contains only a single operator. With the latter scheme the additive heuristic value is likely to be 0, since a fact will have a non-zero cost only if there is but one operator capable of achieving it. Otherwise, the cheapest solution for each evaluator will use zero-cost operators from outside its partition. For *additive* h^m a preprocessing technique is used, creating one partition for each variable with a value in $s_\star$ and assigning each operator to a single partition on the basis of the impact made upon the distance estimate from s_0 to $s_\star$.

4 Combining Disjunction and Addition

The combination of heuristics by disjunction or by addition has been presented as separate, if related, approaches. The approaches have been combined only in limited ways:

- In (Haslum, Bonet, & Geffner 2005), a disjunctive heuristic is defined over h^m and additive h^m.
- In (Haslum *et al.* 2007), a disjunctive heuristic is defined over a collection of additive PDB heuristics.

Such combinations are effective in practice, but are somewhat at odds with the motivation for disjunctive heuristics, which is to exploit the different strengths of a set of heuristics. We now propose a novel combination of heuristics, resulting in an Additive–Disjunctive Heuristic (ADH).

In existing additive heuristics, the component heuristic evaluators are all in the same family. For instance, additive h^m is a sum of h^m evaluators and the additive PDB heuristic is a sum of PDB evaluators. However, there is no reason to use only a single evaluator for each component heuristic. Hence, we propose a simple partitioning scheme ADH, where each partition is evaluated with a disjunctive heuristic.

Definition 4.1 — Simple ADH
A simple partition-based additive–disjunctive heuristic can be defined using the operator partition $O_0..O_n$ as:

$$adh(s) = \sum_{i=0..n} \max_{j=0..k} h_j^{O_i}(s)$$

where $h_j^{O_i}$ is an admissible adjusted-cost heuristic using cost function $part_{O_i}$.

This combination of addition and disjunction exploits the synergy in disjunctive heuristics within each partition. Indeed, the set of heuristic evaluators can be tailored to each partition: for O_i, we can use heuristics that are effective with its adjusted cost function.

Having defined a simple ADH, we now generalise the structure. In particular, we remove restrictions on the heuristic structure and generalise beyond the partitioning scheme.

4.1 A Generalised ADH

To generalise the ideas presented in Definition 4.1, we take two steps. First, we move beyond partitioning schemes, allowing an operator cost to be shared fractionally amongst many evaluators. Second, we define the heuristic structure using a directed acyclic graph (DAG): an Additive–Disjunctive Heuristic Graph (ADHG) (analogous to an and-or tree). An ADHG is a fully connected DAG with a single root vertex containing three types of vertices:

- Sum vertices ($\sum$);
- Max vertices ($\max$);
- Evaluator vertices, comprising an admissible adjusted-cost heuristic h_v, with adjusted cost function $cost_{h_v}$.

Evaluator vertices have no successors, while sum and max nodes always have successors. We denote the successors of a vertex, v, by $succ(v)$.

Directed edges denote the dependencies between vertices: an outgoing edge $v \rightarrow v'$ denotes that the expression evaluated at v' is used in the expression at v. The following definition indicates how an ADHG can be used to compute a heuristic value for a state:

Definition 4.2 — ADHG Heuristic
Given an ADHG with root vertex, r, the heuristic combination function hc is defined as $hc(s) = hc'(s, r)$ where:

$$\begin{array}{lll} hc'(s,v) & = \max_{c \in succ(v)} hc'(s,c), & \text{if } v \text{ is a } \max \\ & = \sum_{c \in succ(v)} hc'(s,c), & \text{if } v \text{ is a } \sum \\ & = h_v(s) & \text{if } v \text{ is an evaluator} \end{array}$$

To use an ADHG as a basis for an admissible heuristic, it must be *safely additive*. In generalising Definition 3.3, we cannot over-count the cost of an operator. We use the following definitions:

Definition 4.3 — Maximum Aggregate Cost
The *maximum aggregate cost* $mac(o, v)$ of an operator o at a vertex v in an ADHG is:

$$\begin{aligned} mac(o,v) &= \max_{c \in succ(v)} mac(o,c), && \text{if } v \text{ is a max} \\ &= \textstyle\sum_{c \in succ(v)} mac(o,c), && \text{if } v \text{ is a } \textstyle\sum \\ &= cost_{h_v}(o) && \text{otherwise} \end{aligned}$$

Definition 4.4 — Safely Additive ADHG
An ADHG is safely additive for a planning problem, $\Pi = \langle V, O, s_0, s_\star, cost \rangle$, if for all ADHG vertices v:

$$mac(o, v) \leq cost(o)$$

Theorem 2 *Given a safely additive ADHG, its heuristic combination function* hc *is an admissible heuristic.*

The proof of this result is a straightforward generalisation of the proof of theorem 1 using structural induction on the ADHG. We omit the details here for lack of space.

The safely additive ADHG generalises the way in which admissible adjusted-cost heuristics can be combined to form new admissible heuristics, using arbitrary combinations of the disjunctive and additive operations in order to exploit the power of both decomposition of structure and alternative views of structure.

Safely additive ADHG-based heuristics can be used in both forward-chaining and regression search. Again, the choice of which heuristic evaluators to use within the evaluator vertices will be influenced by the approach, but the idea remains general and suitable for use in either setting.

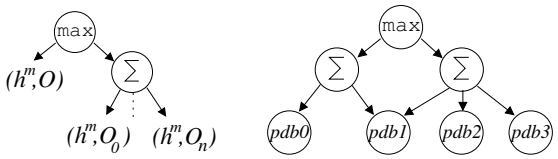
4.2 ADHGs in Context

The safely additive ADHG generalises the simple ADH (Definition 4.1) and subsumes the heuristic structure of previous additive heuristics for planning. By way of example, the heuristic used in (Haslum, Bonet, & Geffner 2005) is illustrated in Figure 1a: here, the maximum is taken of either h^m or the additive h^m heuristic. The sum node combines the partitioning scheme heuristics, $h^m_{O_0}, ..., h^m_{O_n}$.

Figure 1b shows the 'canonical heuristic function' from (Haslum *et al.* 2007), which takes the maximum across additive sums over underlying patterns associated with non-overlapping operator subsets. The canonical heuristic function is a safely additive ADHG (equivalent to a partitioning scheme in which some partitions are simply ignored)

Algorithm 1: Safe Merge

```
Data: old, new - roots of ADHGs
Result: M, a merged ADHG
1 if old = ∅ then return M ← new;
2 newops ← {(o ∈ O | mac(o, new) ≥ 0};
3 old′ ← old;
4 foreach evaluator vertex v in old′ do
5     foreach o ∈ newops do cost(o, h_v) ← 0;
6 if old = old′ then return M ← ∑(old, new);
7 else
8     old′ ← {v ∈ old′ | ∃o ∈ O.mac(o, v) > 0};
9     return M ← max(old, ∑(old′, new));
```



(a) (Haslum, Bonet, & Geffner 2005) (b) (Haslum *et al.* 2007)

Figure 1: ADHGs for Existing Additive Heuristics

because of the restriction of the additive sums to non-overlapping operator subsets.

4.3 Operations on ADHGs

It is possible to define a library of functions for the management of ADHGs. For reasons of space, we include only one important example: *safe merge*. The *safe merge* algorithm, shown in Algorithm 1 merges a new ADHG with root *new* into an existing ADHG with root *old*, preserving safe additivity (Definition 4.4). The key to the approach is determining which operators have non-zero maximum aggregate cost at the root of *new* (line 2), producing the set *newops*. We create a copy of *old* (*old′*), and change the evaluator cost functions to assign zero-cost to all the operators in *newops* (line 5). In doing so, *old′* is cleanly separated from *new*, so we can combine *old′* and *new* additively. Reducing the cost function values in *old* to produce *old′*, guarantees the merged ADHG yields an admissible heuristic, but offers no guarantee that it produces better heuristic values than the original ADHG. Therefore, the resulting merged ADHG is a $\max$ over:

- *old*, the existing tree; and,
- the sum of *old′* and *new*.

Two further considerations are made, to reduce the size of the ADHG produced. First, at line 6, if *old* and *old′* are identical (no operator costs are changed) we return just the sum of the two ADHGs: no change indicates *new* and *old* are disjoint. Second, at line 8, we eliminate redundant vertices: those which cannot contribute non-zero values to any heuristic computation, as the maximum aggregate cost across all operators is 0. Neither of these two steps changes the heuristic values produced, but the merged ADHG is more compact, reducing overheads in storage and evaluation.

5 An Example ADHG

Having now set out a general theoretical structure, the additive–disjunctive heuristic graph, that can be used to flexibly combine additive and disjunctive heuristic techniques, we now show an example of a heuristic using this approach. Whilst made possible through the use of an ADHG, the following heuristic by no means exhausts the possibilities available. It does, however, serve as an example of how embedding disjunction within an additive heuristic can improve heuristic estimates. Further, whilst the heuristic here is defined for use in regression search, the techniques are equally applicable in a forward-chaining search setting. We use regression alongside h^m heuristic techniques, which are particularly effective in this context.

Algorithm 2: Merging Sets for Goals

Data: bs - base sets, $S_\star$ - goal state
Result: bs, merged according to goals
1 **foreach** $(v = val) \in S_\star$ **do**
2 $\quad varops \leftarrow \{(labels, ops) \mid (\exists\, lbl \in labels \mid v \in lbl)$
3 $\quad\quad \wedge\, ops = bs[labels]\}$;
4 $\quad newset \leftarrow \cup_{(label,ops)\in varops}\, ops$;
5 $\quad newlabel \leftarrow \cup_{(label,ops)\in varops}\, label$;
6 $\quad$ erase bs sets containing any $o \in newset$;
7 $\quad bs[newlabel] \leftarrow newset$;

Our heuristic is formed from two components, combined using disjunctive techniques:

- A simple additive PDB heuristic: one pattern is formed for each SAS+ variable (corresponding to its domain transition graph), and action costs normalised as $pdb\_cost(o) = cost(o)/\mid pre\_post(o) \mid$.
- An additive–disjunctive heuristic, using nested disjunction and addition within the additive h^m heuristic.

Our focus will be on the latter of these, which serves to demonstrate how the ADHG can be employed elegantly to construct a flexible heuristic evaluator.

5.1 SAS+ Partitioning

Our approach begins with a definition of a partitioning scheme to be used with additive h^m according to SAS+ variables. Our approach differs from that originally employed in (Haslum, Bonet, & Geffner 2005) in two important respects: we infer partitions, rather than generating them by search, and, as we shall go on to show, we employ additive–disjunctive techniques. The effect of these is to allow the precise combination of partitions to be dependent on the specific search node being considered.

Working in a SAS+ formalism, we begin by defining *base sets*:

Definition 5.1 — Base Sets
The base sets are labelled sets of operators. Each operator $o \in O$ is inserted into a base set labelled:

$$bs[\{v \in V \mid (v = (p \rightarrow q)) \in pre\_post(o)\}]$$

In this manner, the base set $bs[vars]$ contains the operators acting upon the set of variables $vars$.

For domains where operators never modify the value of more than one variable (e.g. a simple logistics problem, with only trucks and packages), one base set is created per variable. In other cases, many base sets can be created. As discussed in Section 3.2, an excess of sets in a partition is detrimental to the quality of the resulting heuristic. Hence, our next step is to reduce the number of base sets forming the partition by merging them according to the goal assignments, using Algorithm 2. For each goal $(v = k) \in s_\star$, the base sets affecting v are merged, and the label updated accordingly (lines 2–5). Then, at line 6 we ensure the operator sets are disjoint: sets containing any of the operators in the newly created set, *newset*, are erased. Intuitively, this merging ensures that all the operators affecting a given goal variable, v, are placed in a common (merged) base set p. We shall see the importance of this later.

5.2 Additive Partitioning Examples

Since the base sets are (by construction) disjoint, we could build an additive h^m heuristic, at this point, using the partitioning scheme implied by the base sets to dictate the cost function for each evaluator. The heuristic would be a sum of evaluators, $h^m_{O_i}(s)$, for the base sets O_i. To motivate and inform a more general heuristic construction, we shall now consider two example domains.

First, consider the Gripper domain as an example of the importance of the merging according to goal variables described in Section 5.1. Without the merging step, we have two base sets for each ball: one, B_0, labelled $\{ball_i, left\}$ and the other, B_1 labelled $\{ball_i, right\}$. Using this partitioning would yield two h^m evaluators for each ball as part of an additive heuristic, with cost functions $part_{B_0}$ and $part_{B_1}$ obtained using the first and second of these sets respectively. Now consider the evaluation of the initial state: the distance from s_0 to $s_\star$. According to each of these two evaluators, the cost of moving $ball_i$ to the destination room is zero: the 'left' partition uses the zero-cost actions for the right gripper, and vice versa. In general, this problem arises when the operators affecting a variable are split across multiple partitions: the induced cost functions allow zero-cost paths to be found. By merging sets in the partition according to goal variables we can avoid this in many cases.

Having discussed set merging, we now consider a simple logistics problem, with a number of trucks t and a number of packages p. From the SAS+ representation, we form $t + p$ base sets: t sets, each containing operators affecting a single truck (denoted $truck_1..truck_t$) and p partitions for packages (denoted $pkg_1..pkg_p$). Using the partitioning scheme these can be used to give us operator cost functions $part_{truck_1}(o)..part_{truck_t}(o)$, $part_{pkg_1}(o)..part_{pkg_p}(o)$ and the resulting additive heuristic evaluator:

$$\begin{aligned} h(s) \quad &= h^m_{truck_1}(s) + .. + h^m_{truck_t}(s) \\ &+ h^m_{pkg_1}(s) + .. + h^m_{pkg_p}(s) \end{aligned}$$

When evaluating a state, st, in which locations for all trucks are specified, this heuristic will provide an informative measure: a sum of the estimates of the costs of the necessary truck movement from s_0 to st. Such states occur in regression search even if no truck goals are specified in $s_\star$ as truck variable assignments are introduced, as needed, to support load and unload operators. However, consider evaluating a state sp where only goals for package variables are specified. Here, the heuristic will have a clear weakness. Specifically, just as in Gripper where there were multiple grippers, a consequence of the presence of multiple trucks is that the cost of truck movement is not taken into account. According to each evaluator with cost function $part_{truck_i}(o)$ a truck other than $truck_i$ can be used, at zero cost. To alleviate this problem, a more informative estimate can be found by merging the truck variable base sets, building a new cost function $part_{trucks}(o)$, and a revised heuristic:

$$\begin{aligned} mh(s) \quad &= h^m_{trucks}(s) \\ &+ h^m_{pkg_1}(s) + .. + h^m_{pkg_p}(s) \end{aligned}$$

Across the entire search-space, some states, s, might be better evaluated by $h(s)$ and others by $mh(s)$. A final

Algorithm 3: Ordering Goal Variables

Data: CG' - variable dependency graph
Result: $order$, a list of variable sets
1 **while** $CG' \neq \emptyset$ **do**
2 $min \leftarrow \min_{v \in CG'} \text{fanin}(v)$;
3 $next \leftarrow \{v \in CG' \mid \text{fanin}(v) = min\}$;
4 remove vertices for $layer$ from CG';
5 $grp \leftarrow next$, split by successors in CG';
6 **foreach** $L \in grp$ **do** Add L to the back of $order$;

additive–disjunctive heuristic can be defined to exploit the strengths of both, $fh(s)$:

$$\begin{aligned} fh(s) \quad &= \max[h^m_{trucks}(s), \\ &\quad \sum(h^m_{truck_1}(s)..h^m_{truck_t}(s))] \\ &+h^m_{pkg_1}(s) + .. + h^m_{pkg_p}(s) \end{aligned}$$

5.3 Domain-Independent Use of Additive–Disjunctive Partitioning

We have shown one example of the use of safely additive ADHGs. We now consider how they can be used in a domain-independent way. To do so, we need a tool that can be used to guide the construction of an appropriate ADHG and the causal graph (Helmert 2006) is useful in this role. The causal graph (CG) contains one vertex v for each variable in a planning problem and directed edges between pairs of vertices to represent dependencies between them, as follows:

1. If v' appears as a $prevail$ condition in an operator $o \in O$ with a pre_post condition affecting v, then v is a *parent* of v', and a directed edge $v' \rightarrow v$ is added to the CG.
2. If an operator $o \in O$ has multiple pre_post conditions, and hence affects both v and v', the two are *co-modified*, and bidirectional edges $v \leftrightarrow v'$ added to the CG.

In the simple logistics problem it is the dependency of changes in a package variable on the values of many truck variables that causes the failure in the initial base sets partition. The CG contains an edge from each truck to each package, highlighting precisely this dependency. This motivates the generalisation of the approach taken in logistics, which is to introduce disjunction over two options: an h^m evaluator obtained by merging the partitions affecting the variables on which a variable v depends, and an additive h^m evaluator obtained by not merging them. This substructure can be represented as a component of an ADHG, which we refer to as the merge-or-not disjunction:

Definition 5.2 — Merge-Or-Not Disjunction
A merge-or-not disjunction ADHG component, built from a partition $O_0..O_n$, is a $\max$ vertex with two children:

- an h^m evaluator, using cost function $part_{\cup_{i=0..n} O_i}$;
- a $\sum$ vertex, under which are the $h^m_{O_i}$ evaluators ($i = 0..n$).

To use this disjunction, it remains to identify useful candidates for potential merging-or-not, and to build a corresponding ADHG. We perform this in a three-stage process: variable dependency analysis, variable consideration ordering and ADHG construction.

1) Variable Dependency Analysis. We begin by taking a subset of the causal graph, $CG' \subseteq CG$, containing only the vertices for variables appearing in $s_\star$. We then reverse the direction of any edges in CG': $v \rightarrow v'$ is replaced with $v' \rightarrow v$. The resulting structure represents the parental/co-modification dependencies between variables, and the vertices with lowest fan-in represent those on which the fewest other variables depend. In the logistics domain, for example, vertices for packages have zero fan-in and those for trucks have fan-in of p, the number of packages.

2) Variable Consideration Ordering. Having defined CG', we now define the order in which to visit variables in the ADHG construction process. We achieve this using a topological order traversal which, for reasons of efficiency, we extend to visit a sequence of *sets of* vertices. We visit sets of vertices from CG' in increasing fan-in order, where the sets contain vertices that have the same predecessors (and therefore the same fan-in).

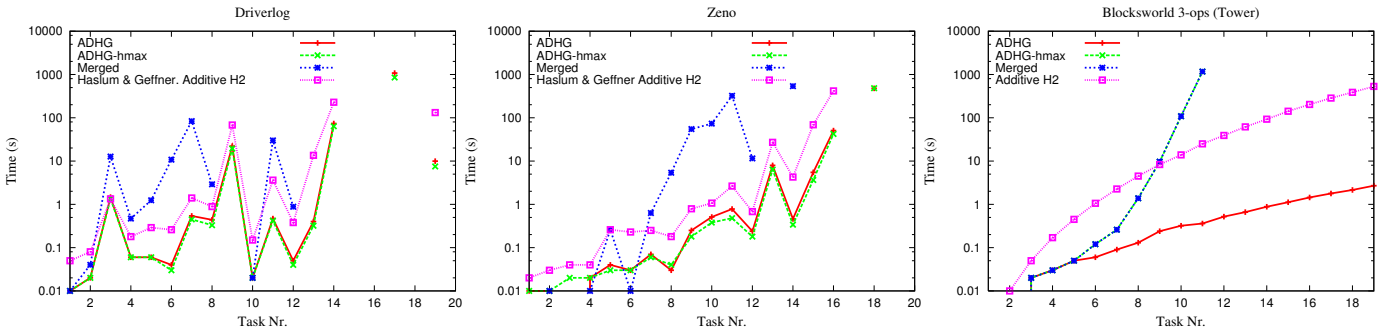
The algorithm is shown in Algorithm 3. At line 3, we extract the vertices with the lowest fan-in from CG', forming a set $next$. The vertices are then split into a number of sets, at line 5, based on their succesors in CG' (i.e. parents in CG), before being added to $order$.

3) ADHG Construction The final stage is to construct an ADHG from the goal-merged partition sets. The Algorithm is presented in Algorithm 4, and employs the 'Safe Merge' algorithm (Algorithm 1) to repeatedly merge newly constructed ADHG components into a developing ADHG (initially empty). It follows the strategy set out at the start of this subsection: forming a merge-or-not disjunction for the parents (inc CG) of the current variables, L (lines 8–13). This component is then additively combined with evaluators formed from the partitions for the variables L, $layerparts$. The resulting ADHG substructure is then merged (safely) into the developing ADHG, G.

The algorithm constructs a safe additive ADHG based around additive h^m evaluators, combining addition with dis-

Algorithm 4: ADHG Construction

Data: CG - causal graph, $order$ - variable ordering, bps - the goal-merged partitions
Result: G, an ADHG
1 $G \leftarrow \emptyset$;
2 **foreach** *set of variables* $L \in order$ **do**
3 $current\_var\_parts \leftarrow \emptyset$;
4 **foreach** $(label, ops) \in bps$ **do**
5 **if** $\exists\ lbl \in label \mid L \cap lbl \neq \emptyset$ **then**
6 add $\{ops\}$ to $current\_var\_parts$;
7 $pvars \leftarrow \{v' \in (V \setminus L) \mid \exists v \in L.(v' \rightarrow v) \in CG\}$;
8 $pparts \leftarrow \emptyset$;
9 **foreach** $(label, ops) \in bps$ **do**
10 **if** $\exists\ lbl \in label \mid pvars \cap lbl \neq \emptyset$ **then**
11 add $\{ops\}$ to $pparts$;
12 $D \leftarrow$ merge-or-not disjunction for $pparts$;
13 $E \leftarrow \sum_{p \in current\_var\_parts} h^m_p(s)$;
14 $new \leftarrow \sum(D, E)$;
15 Safely merge new into G;
16 **return** G;



(a) Driverlog (b) ZenoTravel (c) Blocksworld 3-ops 'Tower'

Figure 2: Time Taken to Find an Optimal Plan in Driverlog, Zeno and Blocksworld 3-ops 'Tower' Problems

junction within the strategy of considering either merging or not merging parent variables. The resulting approach is domain-independent, relying on the guidance of the causal graph. By employing disjunction within the additive h^m framework we improve the heuristic value at each node over that achieved by considering a fixed set of partitions.

6 Evaluation of the Exemplar Heuristic

To illustrate the potential power of additive–disjunctive heuristics, we shall compare our new heuristic to that around which it is constructed: additive h^m. To evaluate the heuristics, each was used with the IDA* search algorithm to direct an optimal regression planner, solving cost SAS+ planning problems with all action costs fixed to 1. A range of benchmark planning domains was used, and each invocation of search was subject to a limit of 30 minutes and 1.5GB of RAM on a 3.4GHz Pentium IV PC. Two measurements were taken: the number of nodes expanded and the total time taken. To perform a thorough evaluation, four configurations were considered:

1. ADHG: the exemplar ADHG heuristic, in full (see Section 5), with both the additive–disjunctive h^m and the simple PDB evaluators. Here, in the h^m evaluators, we take $m = 1$; i.e. h^{max} (Haslum & Geffner 2000).
2. ADHG-hmax: the ADHG, without the PDB branch and again, $m = 1$.
3. Merged: as ADHG-hmax, but replacing Algorithm 4 line 13, with '$D \leftarrow h^m_{\cup pparts}(s)$'; i.e. do not consider parent variable partitions individually.
4. Additive h^2, as presented originally in (Haslum, Bonet, & Geffner 2005). Additive h^{max} gave rise to consistently worse performance, so whereas the other configurations take $m = 1$, here we take $m = 2$, which is in its favour. We return to this towards the end of this section.

Comparing configurations 2,3 and 4 allows clean investigation into the power of mixed additive–disjunctive techniques: all use the same basic additive h^m evaluators. Configuration 1 serves as an over-arching measure of the power of being able to define complex heuristics within the clean ADHG representation. Note that with configuration 4, we disregard the time taken to perform partition preprocessing, so results in terms of time for configuration 3 are artificially deflated.

Across these configurations, we present results from three domains. Several domains were considered, but for reasons of space we restrict our attention to a sample of these: Driverlog and ZenoTravel from IPC3 (Long & Fox 2003) and Blocksworld 3-operator 'Tower' problems, where the goal is to stack n blocks from a table into tower such that block n is on block $n-1$. The latter of these is chosen because it was used as a test set for additive h^m (and featured somewhat more recently in the context of the optimal planner CPT (Vidal & Geffner 2006)). For the first two, due to the limited scalability of the additive h^2 heuristic, we define the 20 problems in each as follows: odd-numbered problems $1, 3, 5..$ correspond to IPC problems $1, 2, 3..$ and even-numbered problems correspond to a problem generated using the same parameters as the previous problem file, but with a different random seed.

We begin with a discussion of results in terms of time: data for Driverlog and Zeno are shown in Figures 2a and 2b. As can be seen, 'Merged' is weak heuristic in general; as discussed in Section 5.2, merging parent variables is not always the best thing to do, and these results reflect this. Indeed, compared to Haslum & Geffner's fixed-partitions, it falls behind, so if partition decisions is fixed, it clearly is not the best option to pursue. Within an ADHG heuristic, however, the power of considering whether to merge evaluators or not at each state becomes apparent: the ADHG heuristics outperform additive h^2 consistently. Looking at the heuristic estimates at each node during search, the distances from s_0 to $s_\star$ are often similar when comparing Merged, the ADHG heuristic and additive h^2. Once search moves backwards away from $s_\star$ the gap widens: very rapidly, in the case of Merged, but also in the case of additive h^2. Due to its nature, the partitioning strategy of additive h^2 suffers from over-fitting: the partitions are a good estimator for the initial heuristic cost, but are less effective at other points. By employing disjunction within additive h^{max}, the ADHG heuristic overcomes this obstacle, leading to a better estimator across the search space as a whole. Also notable is the fact that ADHG-hmax outperforms ADHG marginally in terms of time: in these two domains, the value from the additive–disjunctive h^{max} heuristic branch dominates the other.

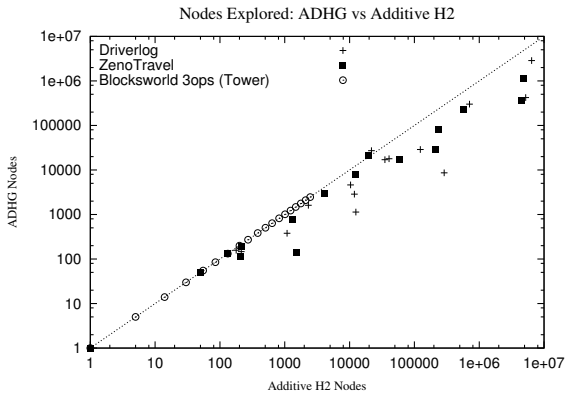


Figure 3: Nodes Expanded: ADHG vs Additive h^2

Data for the Blocksworld 3-ops 'Tower' problems are shown in Figure 2c. In this domain, we see a somewhat different picture. As before, ADHG outperforms additive h^2 substantially, but the partitioning used within additive h^2 is better suited to this domain than the approach used for ADHG-hmax. Inspecting the ADHG-hmax structure, the tightly-connected causal graph of the blocksworld problem has a harmful effect on the induction of the ADHG used. This is also reflected in 'Merged', which produces identical heuristic estimates. The other component of the exemplar ADHG heuristic — the single-variable-projection additive PDB approach — works well in this domain, however, allowing large problems to be solved in seconds.

Figure 3 shows the scatter plot comparing nodes evaluated using additive h^2 compared with ADHG across the three domains, with points plotted for problems solved by both approaches. As can be seen, the number of nodes explored in Blocksworld 3-ops Tower problems is identical (the heuristic estimates are identical) but, as we have shown, ADHG exhibits better performance in terms of time. For the other domains, on all but a few problems the ADHG heuristic explores fewer nodes than if additive h^2 is used.

Overall, the exemplar ADHG heuristic demonstrates the power gained by employing disjunction with addition. As a further note on this comparison, the ADHG was built around h^{max}, whereas additive h^2 employed h^2 evaluators. In the latter, using h^2 rather than h^{max} markedly improves the quality of the heuristic: the computational cost of finding the most costly subset of size 2 within each evaluator at each point in the search-space is outweighed by the improved heuristic. With the exemplar ADHG, however, the same benefits were not realised: there are, in general, more evaluators in the ADHG than in additive h^2 so whilst occasionally improvement is seen in terms of number of nodes, the heuristic was never better in terms of time. However, the fact that it is built around fundamentally weaker evaluators was more than offset by the power of the ADHG techniques.

7 Conclusions

In this paper, we have introduced a generalised structure for combining disjunctive- and additive-heuristic techniques in planning: the Additive–Disjunctive Heuristic Graph (ADHG). The ADHG framework presents an intuitive framework for heuristic construction, allowing disjunction and addition to be combined whilst guaranteeing admissibility. Further, whilst an ADHG could then be compiled down to a flat max-over-sum graph (such as that of (Haslum *et al.* 2007)), the representation allows efficient heuristic evaluation. In doing so, the potential is opened for the construction of powerful, flexible aheuristics: an important part of any optimal planner. To illustrate some of the potential of the approach, an exemplar ADHG heuristic is provided, illustrating how an ADHG variant of additive h^m can lead to better heuristic estimates: disjunction provides better cross-search-space heuristic estimates, avoiding the problem of over-fitting to the goal-set. The ADHG framework is a powerful general approach, and extends far beyond the exemplar heuristic given here. In particular, we are now extending it to use recently developed powerful additive PDB techniques (Helmert, Haslum, & Hoffmann 2007) in a forward-chaining setting.

Acknowledgements

We would like to thank Malte Helmert for providing code for Downward, including the SAS+ preprocessor, and Patrik Haslum for providing the additive h^m partitioning code.

References

Blum, A., and Furst, M. 1995. Fast Planning through Planning Graph Analysis. In *Proc. Int. Joint Conf. on Art. Int. (IJCAI-95)*.

Do, M. B., and Kambhampati, S. 2001. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *J. Art. Int.* 132:151–182.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In *Proc. 6th Int. Conf. on Art. Int. Planning Systems (AIPS)*, 274–283.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive Pattern Database Heuristics. *J. Art. Int. Res.* 22:279–318.

Haslum, P., and Geffner, H. 2000. Admissible Heuristics for Optimal Planning. In *Proc. 5th Int. Conf. on AI Plan. and Sched. (AIPS)*, 140–149.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Kœnig, S. 2007. Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning. In *Proc. Nat. Conf. on Art. Int. (AAAI)*.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New Admissible Heuristics for Domain-Independent Planning. In *Proc. 20th Nat. Conf. on Art. Int. (AAAI)*, 1343–1348.

Helmert, M., and Mattmüller, R. 2008. Accuracy of Admissible Heuristic Functions in Selected Planning Domains. In *Proc. Conf. of Ass. Adv. AI (AAAI)*.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *Proc. Int. Conf. on AI Plan. and Sched. (ICAPS)*, 176–183.

Helmert, M. 2006. The Fast Downward Planning System. *J. Art. Int. Res.* 26:191–246.

Kautz, H., and Selman, B. 1999. Unifying SAT-based and Graph-Based Planning. In *Proc. Int. Joint Conf. on Art. Int. (IJCAI-99)*.

Long, D., and Fox, M. 2003. The 3rd International Planning Competition: Results and Analysis. *J. Art. Int. Res.* 20:1–59.

van den Briel, M., and Kambhampati, S. 2005. Optiplan: Unifying IP-based and Graph-based Planning. *J. Art. Int. Res.* 24:919–931.

Vidal, V., and Geffner, H. 2006. Branching and Pruning: An Optimal Temporal POCL Planner based on Constraint Programming. *J. Art. Int.* 170(3):298–335.