Journal of Artificial Intelligence Research 29 (2007) 153-190 Submitted 10/06; published 06/07

The Generalized A* Architecture

Pedro F. Felzenszwalb PFF@CS.UCHICAGO.EDU
Department of Computer Science
University of Chicago
Chicago, IL 60637

David McAllester MCALLESTER@TTI-C.ORG
Toyota Technological Institute at Chicago
Chicago, IL 60637

Abstract

We consider the problem of computing a lightest derivation of a global structure using a set of weighted rules. A large variety of inference problems in AI can be formulated in this framework. We generalize A* search and heuristics derived from abstractions to a broad class of lightest derivation problems. We also describe a new algorithm that searches for lightest derivations using a hierarchy of abstractions. Our generalization of A* gives a new algorithm for searching AND/OR graphs in a bottom-up fashion.

We discuss how the algorithms described here provide a general architecture for addressing the pipeline problem — the problem of passing information back and forth between various stages of processing in a perceptual system. We consider examples in computer vision and natural language processing. We apply the hierarchical search algorithm to the problem of estimating the boundaries of convex objects in grayscale images and compare it to other search methods. A second set of experiments demonstrate the use of a new compositional model for finding salient curves in images.

1. Introduction

We consider a class of problems defined by a set of weighted rules for composing structures into larger structures. The goal in such problems is to find a lightest (least cost) derivation of a global structure derivable with the given rules. A large variety of classical inference problems in AI can be expressed within this framework. For example the global structure might be a parse tree, a match of a deformable object model to an image, or an assignment of values to variables in a Markov random field.

We define a *lightest derivation problem* in terms of a set of statements, a set of weighted rules for deriving statements using other statements and a special goal statement. In each case we are looking for the lightest derivation of the goal statement. We usually express a lightest derivation problem using rule "schemas" that implicitly represent a very large set of rules in terms of a small number of rules with variables. Lightest derivation problems are formally equivalent to search in AND/OR graphs (Nilsson, 1980), but we find that our formulation is more natural for the applications we are interested in.

One of the goals of this research is the construction of algorithms for global optimization across many levels of processing in a perceptual system. As described below our algorithms can be used to integrate multiple stages of a processing pipeline into a single global optimization problem that can be solved efficiently.

Dynamic programming is a fundamental technique for designing efficient inference algorithms. Good examples are the Viterbi algorithm for hidden Markov models (Rabiner, 1989) and chart parsing methods for stochastic context free grammars (Charniak, 1996). The algorithms described here can be used to speed up the solution of problems normally solved using dynamic programming. We demonstrate this for a specific problem, where the goal is to estimate the boundary of a convex object in a cluttered image. In a second set of experiments we show how our algorithms can be used to find salient curves in images. We describe a new model for salient curves based on a compositional rule that enforces long range shape constraints. This leads to a problem that is too large to be solved using classical dynamic programming methods.

The algorithms we consider are all related to Dijkstra's shortest paths algorithm (DSP) (Dijkstra, 1959) and A* search (Hart, Nilsson, & Raphael, 1968). Both DSP and A* can be used to find a shortest path in a cyclic graph. They use a priority queue to define an order in which nodes are expanded and have a worst case running time of $O(M \log N)$ where N is the number of nodes in the graph and M is the number of edges. In DSP and A* the expansion of a node v involves generating all nodes u such that there is an edge from v to u. The only difference between the two methods is that A* uses a heuristic function to avoid expanding non-promising nodes.

Knuth gave a generalization of DSP that can be used to solve a lightest derivation problem with cyclic rules (Knuth, 1977). We call this Knuth's lightest derivation algorithm (KLD). In analogy to Dijkstra's algorithm, KLD uses a priority queue to define an order in which statements are expanded. Here the expansion of a statement v involves generating all conclusions that can be derived in a single step using v and other statements already expanded. As long as each rule has a bounded number of antecedents KLD also has a worst case running time of $O(M \log N)$ where N is the number of statements in the problem and M is the number of rules. Nilsson's AO* algorithm (1980) can also be used to solve lightest derivation problems. Although AO* can use a heuristic function, it is not a true generalization of A* — it does not use a priority queue, only handles acyclic rules, and can require $O(MN)$ time even when applied to a shortest path problem.[1] In particular, AO* and its variants use a backward chaining technique that starts at the goal and repeatedly refines subgoals, while A* is a forward chaining algorithm.[2]

Klein and Manning (2003) described an A* parsing algorithm that is similar to KLD but can use a heuristic function. One of our contributions is a generalization of this algorithm to arbitrary lightest derivation problems. We call this algorithm A* lightest derivation (A*LD). The method is forward chaining, uses a priority queue to control the order in which statements are expanded, handles cyclic rules and has a worst case running time of $O(M \log N)$ for problems where each rule has a small number of antecedents. A*LD can be seen as a true generalization of A* to lightest derivation problems. For a lightest derivation problem that comes from a shortest path problem A*LD is identical to A*.

Of course the running times seen in practice are often not well predicted by worst case analysis. This is specially true for problems that are very large and defined implicitly. For example, we can use dynamic programming to solve a shortest path problem in an acyclic graph in $O(M)$ time. This is better than the $O(M \log N)$ bound for DSP, but for implicit

1. There are extensions that handle cyclic rules (Jimenez & Torras, 2000).
2. AO* is backward chaining in terms of the inference rules defining a lightest derivation problem.

graphs DSP can be much more efficient since it expands nodes in a best-first order. When searching for a shortest path from a source to a goal, DSP will only expand nodes v with $d(v) \leq w^*$. Here $d(v)$ is the length of a shortest path from the source to v, and w^* is the length of a shortest path from the source to the goal. In the case of A* with a monotone and admissible heuristic function, $h(v)$, it is possible to obtain a similar bound when searching implicit graphs. A* will only expand nodes v with $d(v) + h(v) \leq w^*$.

The running time of KLD and A*LD can be expressed in a similar way. When solving a lightest derivation problem, KLD will only expand statements v with $d(v) \leq w^*$. Here $d(v)$ is the weight of a lightest derivation for v, and w^* is the weight of a lightest derivation of the goal statement. Furthermore, A*LD will only expand statements v with $d(v) + h(v) \leq w^*$. Here the heuristic function, $h(v)$, gives an estimate of the additional weight necessary for deriving the goal statement using a derivation of v. The heuristic values used by A*LD are analogous to the distance from a node to the goal in a graph search problem (the notion used by A*). We note that these heuristic values are significantly different from the ones used by AO*. In the case of AO* the heuristic function, $h(v)$, would estimate the weight of a lightest derivation for v.

An important difference between A*LD and AO* is that A*LD computes derivations in a bottom-up fashion, while AO* uses a top-down approach. Each method has advantages, depending on the type of problem being solved. For example, a classical problem in computer vision involves grouping pixels into long and smooth curves. We can formulate the problem in terms of finding smooth curves between pairs of pixels that are far apart. For an image with n pixels there are $\Omega(n^2)$ such pairs. A straight forward implementation of a top-down algorithm would start by considering these $\Omega(n^2)$ possibilities. A bottom-up algorithm would start with $O(n)$ pairs of nearby pixels. In this case we expect that a bottom-up grouping method would be more efficient than a top-down method.

The classical AO* algorithm requires the set of rules to be acyclic. Jimenez and Torras (2000) extended the method to handle cyclic rules. Another top-down algorithm that can handle cyclic rules is described by Bonet and Geffner (2005). Hansen and Zilberstein (2001) described a search algorithm for problems where the optimal solutions themselves can be cyclic. The algorithms described in this paper can handle problems with cyclic rules but require that the optimal solutions be acyclic. We also note that AO* can handle rules with non-superior weight functions (as defined in Section 3) while KLD requires superior weight functions. A*LD replaces this requirement by a requirement on the heuristic function.

A well known method for defining heuristics for A* is to consider an abstract or relaxed search problem. For example, consider the problem of solving a Rubik's cube in a small number of moves. Suppose we ignore the edge and center pieces and solve only the corners. This is an example of a problem abstraction. The number of moves necessary to put the corners in a good configuration is a lower bound on the number of moves necessary to solve the original problem. There are fewer corner configurations than there are full configurations and that makes it easier to solve the abstract problem. In general, shortest paths to the goal in an abstract problem can be used to define an admissible and monotone heuristic function for solving the original problem with A*.

Here we show that abstractions can also be used to define heuristic functions for A*LD. In a lightest derivation problem the notion of a shortest path to the goal is replaced by the notion of a lightest context, where a context for a statement v is a derivation of the goal with a "hole" that can be filled in by a derivation of v. The computation of lightest abstract contexts is itself a lightest derivation problem.

Abstractions are related to problem relaxations defined by Pearl (1984). While abstractions often lead to small problems that are solved through search, relaxations can lead to problems that still have a large state space but may be simple enough to be solved in closed form. The definition of abstractions that we use for lightest derivation problems includes relaxations as a special case.

Another contribution of our work is a hierarchical search method that we call HA*LD. This algorithm can effectively use a hierarchy of abstractions to solve a lightest derivation problem. The algorithm is novel even in the case of classical search (shortest paths) problem. HA*LD searches for lightest derivations and contexts at every level of abstraction simultaneously. More specifically, each level of abstraction has its own set of statements and rules. The search for lightest derivations and contexts at each level is controlled by a single priority queue. To understand the running time of HA*LD, let w^* be the weight of a lightest derivation of the goal in the original (not abstracted) problem. For a statement v in the abstraction hierarchy let $d(v)$ be the weight of a lightest derivation for v at its level of abstraction. Let $h(v)$ be the weight of a lightest context for the abstraction of v (defined at one level above v in the hierarchy). Let K be the total number of statements in the hierarchy with $d(v) + h(v) \leq w^*$. HAL*D expands at most $2K$ statements before solving the original problem. The factor of two comes from the fact that the algorithm computes both derivations and contexts at each level of abstraction.

Previous algorithms that use abstractions for solving search problems include methods based on pattern databases (Culberson & Schaeffer, 1998; Korf, 1997; Korf & Felner, 2002), Hierarchical A* (HA*, HIDA*) (Holte, Perez, Zimmer, & MacDonald, 1996; Holte, Grajkowski, & Tanner, 2005) and coarse-to-fine dynamic programming (CFDP) (Raphael, 2001). Pattern databases have made it possible to compute solutions to impressively large search problems. These methods construct a lookup table of shortest paths from a node to the goal at all abstract states. In practice the approach is limited to tables that remain fixed over different problem instances, or relatively small tables if the heuristic must be recomputed for each instance. For example, for the Rubik's cube we can precompute the number of moves necessary to solve every corner configuration. This table can be used to define a heuristic function when solving any full configuration of the Rubik's cube. Both HA* and HIDA* use a hierarchy of abstractions and can avoid searching over all nodes at any level of the hierarchy. On the other hand, in directed graphs these methods may still expand abstract nodes with arbitrarily large heuristic values. It is also not clear how to generalize HA* and HIDA* to lightest derivation problems that have rules with more than one antecedent. Finally, CFDP is related to AO* in that it repeatedly solves ever more refined problems using dynamic programming. This leads to a worst case running time of $O(NM)$. We will discuss the relationships between HA*LD and these other hierarchical methods in more detail in Section 8.

We note that both A* search and related algorithms have been previously used to solve a number of problems that are not classical state space search problems. This includes the traveling salesman problem (Zhang & Korf, 1996), planning (Edelkamp, 2002), multiple sequence alignment (Korf, Zhang, Thayer, & Hohwald, 2005), combinatorial problems on graphs (Felner, 2005) and parsing using context-free-grammars (Klein & Manning, 2003).

The work by Bulitko, Sturtevant, Lu, and Yau (2006) uses a hierarchy of state-space abstractions for real-time search.

1.1 The Pipeline Problem

A major problem in artificial intelligence is the integration of multiple processing stages to form a complete perceptual system. We call this the *pipeline problem.* In general we have a concatenation of systems where each stage feeds information to the next. In vision, for example, we might have an edge detector feeding information to a boundary finding system, which in turn feeds information to an object recognition system.

Because of computational constraints and the need to build modules with clean interfaces pipelines often make hard decisions at module boundaries. For example, an edge detector typically constructs a Boolean array that indicates weather or not an edge was detected at each image location. But there is general recognition that the presence of an edge at a certain location can depend on the context around it. People often see edges at places where the image gradient is small if, at higher cognitive level, it is clear that there is actually an object boundary at that location. Speech recognition systems try to address this problem by returning n-best lists, but these may or may not contain the actual utterance. We would like the speech recognition system to be able to take high-level information into account and avoid the hard decision of exactly what strings to output in its n-best list.

A processing pipeline can be specified by describing each of its stages in terms of rules for constructing structures using structures produced from a previous stage. In a vision system one stage could have rules for grouping edges into smooth curves while the next stage could have rules for grouping smooth curves into objects. In this case we can construct a single lightest derivation problem representing the entire system. Moreover, a hierarchical set of abstractions can be applied to the entire pipeline. By using HA*LD to compute lightest derivations a complete scene interpretation derived at one level of abstraction guides all processing stages at a more concrete level. This provides a mechanism that enables coarse high-level processing to guide low-level computation. We believe that this is an important property for implementing efficient perceptual pipelines that avoid making hard decisions between processing stages.

We note that the formulation of a complete computer vision system as a lightest derivation problem is related to the work by Geman, Potter, and Chi (2002), Tu, Chen, Yuille, and Zhu (2005) and Jin and Geman (2006). In these papers image understanding is posed as a parsing problem, where the goal is to explain the image in terms of a set of objects that are formed by the (possibly recursive) composition of generic parts. Tu et al. (2005) use data driven MCMC to compute "optimal" parses while Geman et al. (2002) and Jin and Geman (2006) use a bottom-up algorithm for building compositions in a greedy fashion. Neither of these methods are guaranteed to compute an optimal scene interpretation. We hope that HA*LD will provide a more principled computational technique for solving large parsing problems defined by compositional models.

1.2 Overview

We begin by formally defining lightest derivation problems in Section 2. That section also discusses dynamic programming and the relationship between lightest derivation problems and AND/OR graphs. In Section 3 we describe Knuth's lightest derivation algorithm. In Section 4 we describe A*LD and prove its correctness. Section 5 shows how abstractions can be used to define mechanically constructed heuristic functions for A*LD. We describe HA*LD in Section 6 and discuss its use in solving the pipeline problem in Section 7. Section 8 discusses the relationship between HA*LD and other hierarchical search methods. In Sections 9 and 10 we present some experimental results. We conclude in Section 11.

2. Lightest Derivation Problems

Let Σ be a set of statements and R be a set of inference rules of the following form,

$$\frac{\begin{array}{l} A_1 = w_1 \\ \quad\vdots \\ A_n = w_n \end{array}}{C = g(w_1, \ldots, w_n)}$$

Here the antecedents A_i and the conclusion C are statements in Σ, the weights w_i are non-negative real valued variables and g is a non-negative real valued weight function. For a rule with no antecedents the function g is simply a non-negative real value. Throughout the paper we also use $A_1, \ldots, A_n \rightarrow_g C$ to denote an inference rule of this type.

A *derivation* of C is a finite tree rooted at a rule $A_1, \ldots, A_n \rightarrow_g C$ with n children, where the i-th child is a derivation of A_i. The leaves of this tree are rules with no antecedents. Every derivation has a *weight* that is the value obtained by recursive application of the functions g along the derivation tree. Figure 1 illustrates a derivation tree.

Intuitively a rule $A_1, \ldots, A_n \rightarrow_g C$ says that if we can derive the antecedents A_i with weights w_i then we can derive the conclusion C with weight $g(w_1, \ldots, w_n)$. The problem we are interested in is to compute a lightest derivation of a special goal statement.

All of the algorithms discussed in this paper assume that the weight functions g associated with a lightest derivation problem are non-decreasing in each variable. This is a fundamental property ensuring that lightest derivations have an optimal substructure property. In this case lightest derivations can be constructed from other lightest derivations.

To facilitate the runtime analysis of algorithms we assume that every rule has a small number of antecedents. We use N to denote the number of statements in a lightest derivation problem, while M denotes the number of rules. For most of the problems we are interested in N and M are very large but the problem can be implicitly defined in a compact way, by using a small number of rules with variables as in the examples below. We also assume that $N \leq M$ since statements that are not in the conclusion of some rule are clearly not derivable and can be ignored.

2.1 Dynamic Programming

We say that a set of rules is *acyclic* if there is an ordering O of the statements in Σ such that for any rule with conclusion C the antecedents are statements that come before C in the ordering. Dynamic programming can be used to solve a lightest derivation problem if

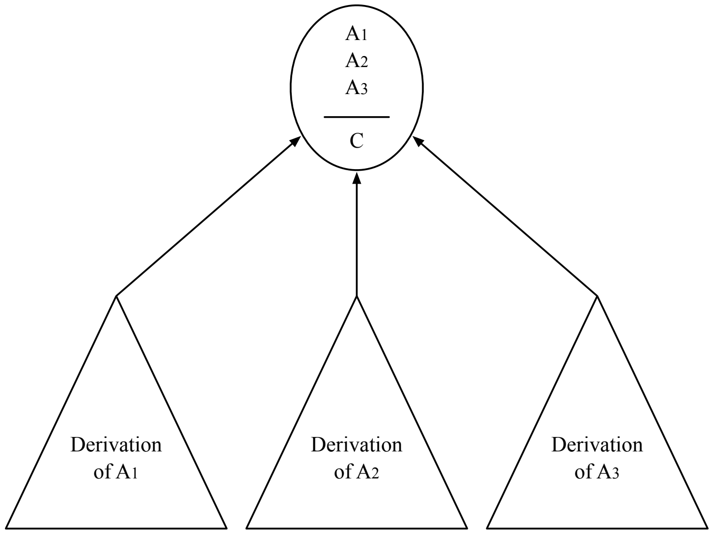


Figure 1: A derivation of C is a tree of rules rooted at a rule r with conclusion C. The children of the root are derivations of the antecedents in r. The leafs of the tree are rules with no antecedents.

the functions g in each rule are non-decreasing and the set of rules is acyclic. In this case lightest derivations can be computed sequentially in terms of an acyclic ordering O. At the i-th step a lightest derivation of the i-th statement is obtained by minimizing over all rules that can be used to derive that statement. This method takes $O(M)$ time to compute a lightest derivation for each statement in Σ.

We note that for cyclic rules it is sometimes possible to compute lightest derivations by taking multiple passes over the statements. We also note that some authors would refer to Dijkstra's algorithm (and KLD) as a dynamic programming method. In this paper we only use the term when referring to algorithms that compute lightest derivations in a fixed order that is independent of the solutions computed along the way (this includes recursive implementations that use memoization).

2.2 Examples

Rules for computing shortest paths from a single source in a weighted graph are shown in Figure 2. We assume that we are given a weighted graph $G = (V, E)$, where w_{xy} is a non-negative weight for each edge $(x, y) \in E$ and s is a distinguished start node. The first rule states that there is a path of weight zero to the start node s. The second set of rules state that if there is a path to a node x we can extend that path with an edge from x to y to obtain an appropriately weighted path to a node y. There is a rule of this type for each edge in the graph. A lightest derivation of $path(x)$ corresponds to shortest path from s to x. Note that for general graphs these rules can be cyclic. Figure 3 illustrates a graph

(1)

$$\frac{}{path(s) = 0}$$

(2) for each $(x, y) \in E$,

$$\frac{path(x) = w}{path(y) = w + w_{xy}}$$

Figure 2: Rules for computing shortest paths in a graph.

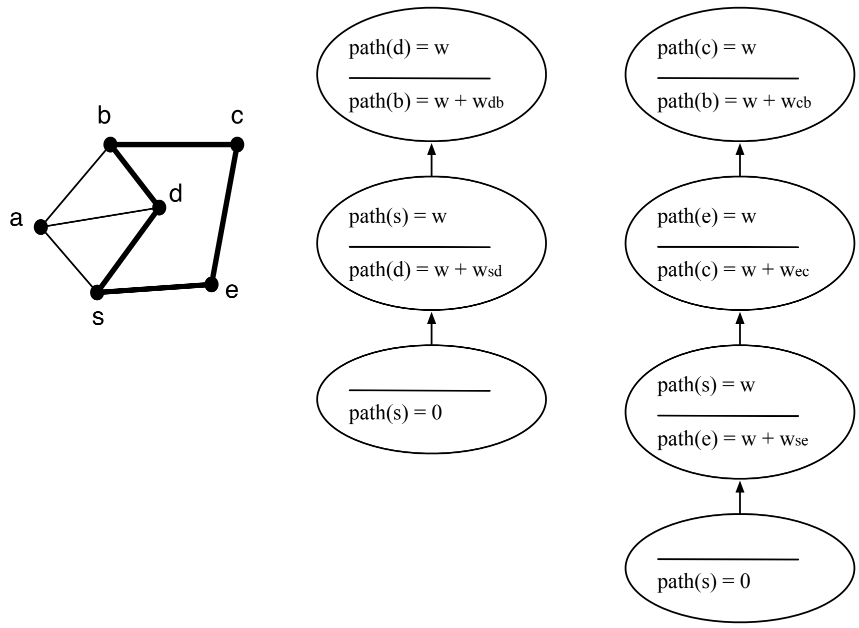


Figure 3: A graph with two highlighted paths from s to b and the corresponding derivations using rules from Figure 2.

and two different derivations of $path(b)$ using the rules just described. These corresponds to two different paths from s to b.

Rules for chart parsing are shown in Figure 4. We assume that we are given a weighted context free grammar in Chomsky normal form (Charniak, 1996), i.e., a weighted set of productions of the form $X \to s$ and $X \to Y Z$ where X, Y and Z are nonterminal symbols and s is a terminal symbol. The input string is given by a sequence of terminals $(s_1, \dots, s_n)$.

(1) for each production $X \to s_i$,

$$\frac{}{phrase(X, i, i+1) = w(X \to s_i)}$$

(2) for each production $X \to YZ$ and $1 \le i < j < k \le n+1$,

$$\frac{\begin{array}{l} phrase(Y, i, j) = w_1 \\ phrase(Z, j, k) = w_2 \end{array}}{phrase(X, i, k) = w_1 + w_2 + w(X \to YZ)}$$

Figure 4: Rules for parsing with a context free grammar.

The first set of rules state that if the grammar contains a production $X \to s_i$ then there is a phrase of type X generating the i-th entry of the input with weight $w(X \to s_i)$. The second set of rules state that if the grammar contains a production $X \to YZ$ and there is a phrase of type Y from i to j and a phrase of type Z from j to k then there is an, appropriately weighted, phrase of type X from i to k. Let S be the start symbol of the grammar. The goal of parsing is to find a lightest derivation of $phrase(S, 1, n+1)$. These rules are acyclic because when phrases are composed together they form longer phrases.

2.3 AND/OR Graphs

Lightest derivation problems are closely related to AND/OR graphs. Let Σ and R be a set of statements and rules defining a lightest derivation problem. To convert the problem to an AND/OR graph representation we can build a graph with a disjunction node for each statement in Σ and a conjunction node for each rule in R. There is an edge from each statement to each rule deriving that statement, and an edge from each rule to its antecedents. The leaves of the AND/OR graph are rules with no antecedents. Now derivations of a statement using rules in R can be represented by solutions rooted at that statement in the corresponding AND/OR graph. Conversely, it is also possible to represent any AND/OR graph search problem as a lightest derivation problem. In this case we can view each node in the graph as a statement in Σ and build an appropriate set of rules R.

3. Knuth's Lightest Derivation

Knuth (1977) described a generalization of Dijkstra's shortest paths algorithm that we call Knuth's lightest derivation (KLD). Knuth's algorithm can be used to solve a large class of lightest derivation problems. The algorithm allows the rules to be cyclic but requires that the weight functions associated with each rule be non-decreasing and superior. Specifically we require the following two properties on the weight function g in each rule,

non-decreasing: if $w_i' \ge w_i$ then $g(w_1, \dots, w_i', \dots, w_n) \ge g(w_1, \dots, w_i, \dots, w_n)$

superior: $g(w_1, \dots, w_n) \ge w_i$

For example,

$$g(x_1, \ldots, x_n) = x_1 + \cdots + x_n$$
$$g(x_1, \ldots, x_n) = \max(x_1, \ldots, x_n)$$

are both non-decreasing and superior functions.

Knuth's algorithm computes lightest derivations in non-decreasing weight order. Since we are only interested in a lightest derivation of a special goal statement we can often stop the algorithm before computing the lightest derivation of every statement.

A *weight assignment* is an expression of the form $(B = w)$ where B is a statement in Σ and w is a non-negative real value. We say that the weight assignment $(B = w)$ is derivable if there is a derivation of B with weight w. For any set of rules R, statement B, and weight w we write $R \vdash (B = w)$ if the rules in R can be used to derive $(B = w)$. Let $\ell(B, R)$ be the infimum of the set of weights derivable for B,

$$\ell(B, R) = \inf\{w : \ R \vdash (B = w)\}.$$

Given a set of rules R and a statement $goal \in \Sigma$ we are interested in computing a derivation of $goal$ with weight $\ell(goal, R)$.

We define a bottom-up logic programming language in which we can easily express the algorithms we wish to discuss throughout the rest of the paper. Each algorithm is defined by a set of rules with priorities. We encode the priority of a rule by writing it along the line separating the antecedents and the conclusion as follows,

$$\begin{array}{l} A_1 = w_1 \\ \quad\vdots \\ A_n = w_n \\ \hline C = g(w_1, \ldots, w_n) \end{array} \; p(w_1, \ldots, w_n)$$

We call a rule of this form a *prioritized rule*. The execution of a set of prioritized rules P is defined by the procedure in Figure 5. The procedure keeps track of a set $\mathcal{S}$ and a priority queue $\mathcal{Q}$ of weight assignments of the form $(B = w)$. Initially $\mathcal{S}$ is empty and $\mathcal{Q}$ contains weight assignments defined by rules with no antecedents at the priorities given by those rules. We iteratively remove the lowest priority assignment $(B = w)$ from $\mathcal{Q}$. If B already has an assigned weight in $\mathcal{S}$ then the new assignment is ignored. Otherwise we add the new assignment to $\mathcal{S}$ and "expand it" — every assignment derivable from $(B = w)$ and other assignments already in $\mathcal{S}$ using some rule in P is added to $\mathcal{Q}$ at the priority specified by the rule. The procedure stops when the queue is empty.

The result of executing a set of prioritized rules is a set of weight assignments. Moreover, the procedure can implicitly keep track of derivations by remembering which assignments were used to derive an item that is inserted in the queue.

Lemma 1. *The execution of a finite set of prioritized rules P derives every statement that is derivable with rules in P.*

Proof. Each rule causes at most one item to be inserted in the queue. Thus eventually $\mathcal{Q}$ is empty and the algorithm terminates. When $\mathcal{Q}$ is empty every statement derivable by a

Procedure *Run*(P)
1. $\mathcal{S} \leftarrow \emptyset$
2. Initialize $\mathcal{Q}$ with assignments defined by rules with no antecedents at their priorities
3. **while** $\mathcal{Q}$ is not empty
4. Remove the lowest priority element $(B = w)$ from $\mathcal{Q}$
5. **if** B has no assigned weight in $\mathcal{S}$
6. $\mathcal{S} \leftarrow \mathcal{S} \cup \{(B = w)\}$
7. Insert assignments derivable from $(B = w)$ and other assignments in $\mathcal{S}$ using some rule in P into $\mathcal{Q}$ at the priority specified by the rule
8. **return** $\mathcal{S}$

Figure 5: Running a set of prioritized rules.

single rule using antecedents with weight in $\mathcal{S}$ already has a weight in $\mathcal{S}$. This implies that every derivable statement has a weight in $\mathcal{S}$. □

Now we are ready to define Knuth's lightest derivation algorithm. The algorithm is easily described in terms of prioritized rules.

Definition 1 (Knuth's lightest derivation)**.** *Let R be a finite set of non-decreasing and superior rules. Define a set of prioritized rules $\mathcal{K}(R)$ by setting the priority of each rule in R to be the weight of the conclusion. KLD is given by the execution of $\mathcal{K}(R)$.*

We can show that while running $\mathcal{K}(R)$, if $(B = w)$ is added to $\mathcal{S}$ then $w = \ell(B, R)$. This means that all assignments in $\mathcal{S}$ represent lightest derivations. We can also show that assignments are inserted into $\mathcal{S}$ in non-decreasing weight order. If we stop the algorithm as soon as we insert a weight assignment for *goal* into $\mathcal{S}$ we will expand all statements B such that $\ell(B, R) < \ell(goal, R)$ and some statements B such that $\ell(B, R) = \ell(goal, R)$. These properties follow from a more general result described in the next section.

3.1 Implementation

The algorithm in Figure 5 can be implemented to run in $O(M \log N)$ time, where N and M refer to the size of the problem defined by the prioritized rules P.

In practice the set of prioritized rules P is often specified implicitly, in terms of a small number of rules with variables. In this case the problem of executing P is closely related to the work on logical algorithms described by McAllester (2002).

The main difficulty in devising an efficient implementation of the procedure in Figure 5 is in step 7. In that step we need to find weight assignments in $\mathcal{S}$ that can be combined with $(B = w)$ to derive new weight assignments. The logical algorithms work shows how a set of inference rules with variables can be transformed into a new set of rules, such that every rule has at most two antecedents and is in a particularly simple form. Moreover, this transformation does not increase the number of rules too much. Once the rules are transformed their execution can be implemented efficiently using a hashtable to represent $\mathcal{S}$, a heap to represent $\mathcal{Q}$ and indexing tables that allow us to perform step 7 quickly.

Consider the second set of rules for parsing in Figure 4. These can be represented by a single rule with variables. Moreover the rule has two antecedents. When executing the parsing rules we keep track of a table mapping a value for j to statements $\mathit{phrase}(Y, i, j)$ that have a weight in $\mathcal{S}$. Using this table we can quickly find statements that have a weight in $\mathcal{S}$ and can be combined with a statement of the form $\mathit{phrase}(Z, j, k)$. Similarly we keep track of a table mapping a value for j to statements $\mathit{phrase}(Z, j, k)$ that have a weight in $\mathcal{S}$. The second table lets us quickly find statements that can be combined with a statement of the form $\mathit{phrase}(Y, i, j)$. We refer the reader to (McAllester, 2002) for more details.

4. A* Lightest Derivation

Our A* lightest derivation algorithm (A*LD) is a generalization of A* search to lightest derivation problems that subsumes A* parsing. The algorithm is similar to KLD but it can use a heuristic function to speed up computation. Consider a lightest derivation problem with rules R and goal statement *goal*. Knuth's algorithm will expand any statement B such that $\ell(B, R) < \ell(\mathit{goal}, R)$. By using a heuristic function A*LD can avoid expanding statements that have light derivations but are not part of a light derivation of *goal*.

Let R be a set of rules with statements in Σ, and h be a heuristic function assigning a weight to each statement. Here $h(B)$ is an estimate of the additional weight required to derive *goal* using a derivation of B. We note that in the case of a shortest path problem this weight is exactly the distance from a node to the goal. The value $\ell(B, R) + h(B)$ provides a *figure of merit* for each statement B. The A* lightest derivation algorithm expands statements in order of their figure of merit.

We say that a heuristic function is *monotone* if for every rule $A_1, \ldots, A_n \rightarrow_g C$ in R and derivable weight assignments $(A_i = w_i)$ we have,

$$w_i + h(A_i) \leq g(w_1, \ldots, w_n) + h(C). \tag{1}$$

This definition agrees with the standard notion of a monotone heuristic function for rules that come from a shortest path problem. We can show that if h is monotone and $h(\mathit{goal}) = 0$ then h is admissible under an appropriate notion of admissibility. For the correctness of A*LD, however, it is only required that h be monotone and that $h(\mathit{goal})$ be finite. In this case monotonicity implies that the heuristic value of every statement C that appears in a derivation of *goal* is finite. Below we assume that $h(C)$ is finite for every statement. If $h(C)$ is not finite we can ignore C and every rule that derives C.

Definition 2 (A* lightest derivation)**.** *Let R be a finite set of non-decreasing rules and h be a monotone heuristic function for R. Define a set of prioritized rules $\mathcal{A}(R)$ by setting the priority of each rule in R to be the weight of the conclusion plus the heuristic value, $g(w_1, \ldots, w_n) + h(C)$. A*LD is given by the execution of $\mathcal{A}(R)$.*

Now we show that the execution of $\mathcal{A}(R)$ correctly computes lightest derivations and that it expands statements in order of their figure of merit values.

Theorem 2. *During the execution of $\mathcal{A}(R)$, if $(B = w) \in \mathcal{S}$ then $w = \ell(B, R)$.*

Proof. The proof is by induction on the size of $\mathcal{S}$. The statement is trivial when $\mathcal{S} = \emptyset$. Suppose the statement was true right before the algorithm removed $(B = w_b)$ from $\mathcal{Q}$ and added it to $\mathcal{S}$. The fact that $(B = w_b) \in \mathcal{Q}$ implies that the weight assignment is derivable and thus $w_b \geq \ell(B, R)$.

Suppose T is a derivation of B with weight $w'_b < w_b$. Consider the moment right before the algorithm removed $(B = w_b)$ from $\mathcal{Q}$ and added it to $\mathcal{S}$. Let $A_1, \ldots, A_n \rightarrow_g C$ be a rule in T such that the antecedents A_i have a weight in $\mathcal{S}$ while the conclusion C does not. Let $w_c = g(\ell(A_1, R), \ldots, \ell(A_n, R))$. By the induction hypothesis the weight of A_i in $\mathcal{S}$ is $\ell(A_i, R)$. Thus $(C = w_c) \in \mathcal{Q}$ at priority $w_c + h(C)$. Let w'_c be the weight that T assigns to C. Since g is non-decreasing we know $w_c \leq w'_c$. Since h is monotone $w'_c + h(C) \leq w'_b + h(B)$. This follows by using the monotonicity condition along the path from C to B in T. Now note that $w_c + h(C) < w_b + h(B)$ which in turn implies that $(B = w_b)$ is not the weight assignment in $\mathcal{Q}$ with minimum priority. □

Theorem 3. *During the execution of $\mathcal{A}(R)$ statements are expanded in order of the figure of merit value $\ell(B, R) + h(B)$.*

Proof. First we show that the minimum priority of $\mathcal{Q}$ does not decrease throughout the execution of the algorithm. Suppose $(B = w)$ is an element in $\mathcal{Q}$ with minimum priority. Removing $(B = w)$ from $\mathcal{Q}$ does not decrease the minimum priority. Now suppose we add $(B = w)$ to $\mathcal{S}$ and insert assignments derivable from $(B = w)$ into $\mathcal{Q}$. Since h is monotone the priority of every assignment derivable from $(B = w)$ is at least the priority of $(B = w)$.

A weight assignment $(B = w)$ is expanded when it is removed from $\mathcal{Q}$ and added to $\mathcal{S}$. By the last theorem $w = \ell(B, R)$ and by the definition of $\mathcal{A}(R)$ this weight assignment was queued at priority $\ell(B, R) + h(B)$. Since we removed $(B = w)$ from $\mathcal{Q}$ this must be the minimum priority in the queue. The minimum priority does not decrease over time so we must expand statements in order of their figure of merit value. □

If we have accurate heuristic functions A*LD can be much more efficient than KLD. Consider a situation where we have a perfect heuristic function. That is, suppose $h(B)$ is exactly the additional weight required to derive *goal* using a derivation of B. Now the figure of merit $\ell(B, R) + h(B)$ equals the weight of a lightest derivation of *goal* that uses B. In this case A*LD will derive *goal* before expanding any statements that are not part of a lightest derivation of *goal*.

The correctness KLD follows from the correctness of A*LD. For a set of non-decreasing and superior rules we can consider the trivial heuristic function $h(B) = 0$. The fact that the rules are superior imply that this heuristic is monotone. The theorems above imply that Knuth's algorithm correctly computes lightest derivations and expands statements in order of their lightest derivable weights.

5. Heuristics Derived from Abstractions

Here we consider the case of additive rules — rules where the weight of the conclusion is the sum of the weights of the antecedents plus a non-negative value v called the weight of the rule. We denote such a rule by $A_1, \ldots, A_n \rightarrow_v C$. The weight of a derivation using additive rules is the sum of the weights of the rules that appear in the derivation tree.

A *context* for a statement B is a finite tree of rules such that if we add a derivation of B to the tree we get a derivation of *goal*. Intuitively a context for B is a derivation of *goal* with a "hole" that can be filled in by a derivation of B (see Figure 6).

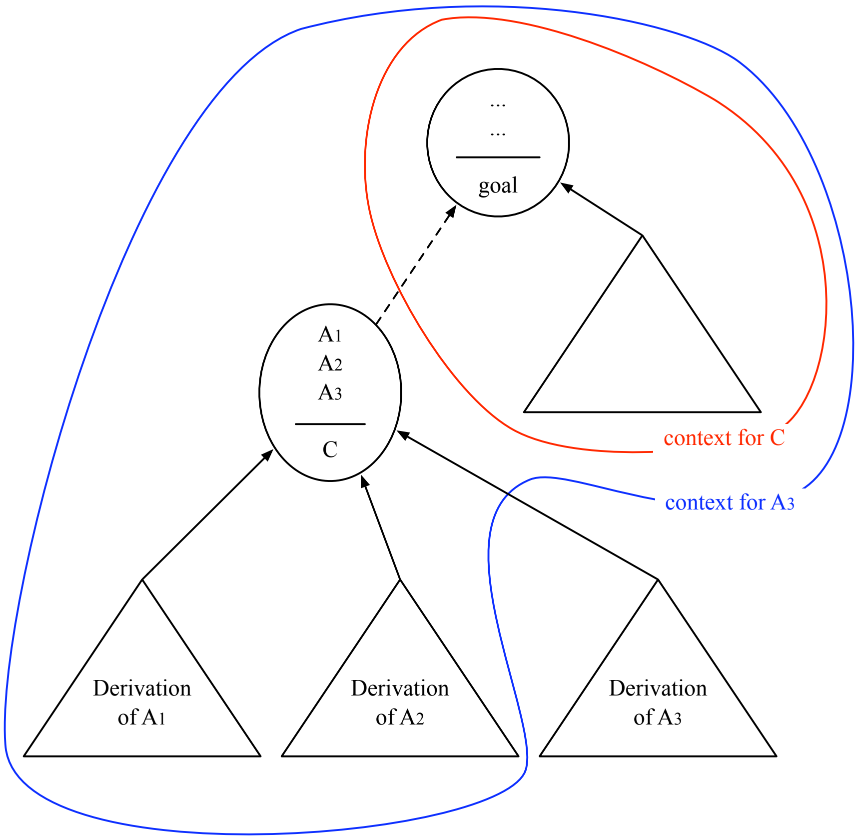


Figure 6: A derivation of *goal* defines contexts for the statements that appear in the derivation tree. Note how a context for C together with a rule $A_1, A_2, A_3 \to C$ and derivations of A_1 and A_2 define a context for A_3.

For additive rules, each context has a weight that is the sum of weights of the rules in it. Let R be a set of additive rules with statements in Σ. For $B \in \Sigma$ we define $\ell(\mathit{context}(B), R)$ to be the weight of a lightest context for B. The value $\ell(B, R) + \ell(\mathit{context}(B), R)$ is the weight of a lightest derivation of *goal* that uses B.

Contexts can be derived using rules in R together with context rules $c(R)$ defined as follows. First, *goal* has an empty context with weight zero. This is captured by a rule with no antecedents $\to_0 \mathit{context}(\mathit{goal})$. For each rule $A_1, \ldots, A_n \to_v C$ in R we put n rules in $c(R)$. These rules capture the notion that a context for C and derivations of A_j for $j \neq i$ define a context for A_i,

$$\mathit{context}(C), A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n \to_v \mathit{context}(A_i).$$

Figure 6 illustrates how a context for C together with derivations of A_1 and A_2 and a rule $A_1, A_2, A_3 \to C$ define a context for A_3.

We say that a heuristic function h is *admissible* if $h(B) \leq \ell(context(B), R)$. Admissible heuristic functions never over-estimate the weight of deriving *goal* using a derivation of a particular statement. The heuristic function is *perfect* if $h(B) = \ell(context(B), R)$. Now we show how to obtain admissible and monotone heuristic functions from abstractions.

5.1 Abstractions

Let (Σ, R) be a lightest derivation problem with statements Σ and rules R. An *abstraction* of (Σ, R) is given by a problem (Σ', R') and a map $abs : \Sigma \to \Sigma'$, such that for every rule $A_1, \ldots, A_n \to_v C$ in R there is a rule $abs(A_1), \ldots, abs(A_n) \to_{v'} abs(C)$ in R' with $v' \leq v$. Below we show how an abstraction can be used to define a monotone and admissible heuristic function for the original problem.

We usually think of *abs* as defining a coarsening of Σ by mapping several statements into the same abstract statement. For example, for a parser *abs* might map a lexicalized nonterminal NP_{house} to the nonlexicalized nonterminal NP. In this case the abstraction defines a smaller problem on the abstract statements. Abstractions can often be defined in a mechanical way by starting with a map *abs* from Σ into some set of abstract statements Σ'. We can then "project" the rules in R from Σ into Σ' using *abs* to get a set of abstract rules. Typically several rules in R will map to the same abstract rule. We only need to keep one copy of each abstract rule, with a weight that is a lower bound on the weight of the concrete rules mapping into it.

Every derivation in (Σ, R) maps to an abstract derivation so we have $\ell(abs(C), R') \leq \ell(C, R)$. If we let the goal of the abstract problem be $abs(goal)$ then every context in (Σ, R) maps to an abstract context and we see that $\ell(context(abs(C)), R') \leq \ell(context(C), R)$. This means that lightest abstract context weights form an admissible heuristic function,

$$h(C) = \ell(context(abs(C)), R').$$

Now we show that this heuristic function is also monotone.

Consider a rule $A_1, \ldots, A_n \to_v C$ in R and let $(A_i = w_i)$ be weight assignments derivable using R. In this case there is a rule $abs(A_1), \ldots, abs(A_n) \to_{v'} abs(C)$ in R' where $v' \leq v$ and $(abs(A_i) = w'_i)$ is derivable using R' where $w'_i \leq w_i$. By definition of contexts (in the abstract problem) we have,

$$\ell(context(abs(A_i)), R') \leq v' + \sum_{j \neq i} w'_j + \ell(context(abs(C)), R').$$

Since $v' \leq v$ and $w'_j \leq w_j$ we have,

$$\ell(context(abs(A_i)), R') \leq v + \sum_{j \neq i} w_j + \ell(context(abs(C)), R').$$

Plugging in the heuristic function h from above and adding w_i to both sides,

$$w_i + h(A_i) \leq v + \sum_{j} w_j + h(C),$$

which is exactly the monotonicity condition in equation (1) for an additive rule.

If the abstract problem defined by (Σ', R') is relatively small we can efficiently compute lightest context weights for every statement in Σ' using dynamic programming or KLD. We can store these weights in a "pattern database" (a lookup table) to serve as a heuristic function for solving the concrete problem using A*LD. This heuristic may be able to stop A*LD from exploring a lot of non-promising structures. This is exactly the approach that was used by Culberson and Schaeffer (1998) and Korf (1997) for solving very large search problems. The results in this section show that pattern databases can be used in the more general setting of lightest derivations problems. The experiments in Section 10 demonstrate the technique in a specific application.

6. Hierarchical A* Lightest Derivation

The main disadvantage of using pattern databases is that we have to precompute context weights for every abstract statement. This can often take a lot of time and space. Here we define a hierarchical algorithm, HA*LD, that searches for lightest derivations and contexts in an entire abstraction hierarchy simultaneously. This algorithm can often solve the most concrete problem without fully computing context weights at any level of abstraction.

At each level of abstraction the behavior HA*LD is similar to the behavior of A*LD when using an abstraction-derived heuristic function. The hierarchical algorithm queues derivations of a statement C at a priority that depends on a lightest abstract context for C. But now abstract contexts are not computed in advance. Instead, abstract contexts are computed at the same time we are computing derivations. Until we have an abstract context for C, derivations of C are "stalled". This is captured by the addition of $\mathit{context}(\mathit{abs}(C))$ as an antecedent to each rule that derives C.

We define an abstraction hierarchy with m levels to be a sequence of lightest derivation problems with additive rules (Σ_k, R_k) for $0 \leq k \leq m-1$ with a single abstraction function abs. For $0 \leq k < m-1$ the abstraction function maps Σ_k onto Σ_{k+1}. We require that (Σ_{k+1}, R_{k+1}) be an abstraction of (Σ_k, R_k) as defined in the previous section: if $A_1, \ldots, A_n \rightarrow_v C$ is in R_k then there exists a rule $\mathit{abs}(A_1), \ldots, \mathit{abs}(A_n) \rightarrow_{v'} \mathit{abs}(C)$ in R_{k+1} with $v' \leq v$. The hierarchical algorithm computes lightest derivations of statements in Σ_k using contexts from Σ_{k+1} to define heuristic values. We extend abs so that it maps Σ_{m-1} to a most abstract set of statements Σ_m containing a single element $\perp$. Since abs is onto we have $|\Sigma_k| \geq |\Sigma_{k+1}|$. That is, the number of statements decrease as we go up the abstraction hierarchy. We denote by abs_k the abstraction function from Σ_0 to Σ_k obtained by composing abs with itself k times.

We are interested in computing a lightest derivation of a goal statement $\mathit{goal} \in \Sigma_0$. Let $\mathit{goal}_k = \mathit{abs}_k(\mathit{goal})$ be the goal at each level of abstraction. The hierarchical algorithm is defined by the set of prioritized rules $\mathcal{H}$ in Figure 7. Rules labeled UP compute derivations of statements at one level of abstraction using context weights from the level above to define priorities. Rules labeled BASE and DOWN compute contexts in one level of abstraction using derivation weights at the same level to define priorities. The rules labeled START1 and START2 start the inference by handling the most abstract level.

The execution of $\mathcal{H}$ starts by computing a derivation and context for $\perp$ with START1 and START2. It continues by deriving statements in Σ_{m-1} using UP rules. Once the lightest derivation of goal_{m-1} is found the algorithm derives a context for goal_{m-1} with a

START1: $$\frac{}{\perp = 0}\ 0$$

START2: $$\frac{}{context(\perp) = 0}\ 0$$

BASE: $$\frac{goal_k = w}{context(goal_k) = 0}\ w$$

UP: $$\frac{\begin{array}{l} context(abs(C)) = w_c \\ A_1 = w_1 \\ \vdots \\ A_n = w_n \end{array}}{C = v + w_1 + \cdots + w_n}\ v + w_1 + \cdots + w_n + w_c$$

DOWN: $$\frac{\begin{array}{l} context(C) = w_c \\ A_1 = w_1 \\ \vdots \\ A_n = w_n \end{array}}{context(A_i) = v + w_c + w_1 + \cdots + w_n - w_i}\ v + w_c + w_1 + \cdots + w_n$$

Figure 7: Prioritized rules $\mathcal{H}$ defining HA*LD. BASE rules are defined for $0 \leq k \leq m-1$. UP and DOWN rules are defined for each rule $A_1, \ldots, A_n \rightarrow_v C \in R_k$ with $0 \leq k \leq m-1$.

BASE rule and starts computing contexts for other statements in Σ_{m-1} using DOWN rules. In general HA*LD interleaves the computation of derivations and contexts at each level of abstraction since the execution of $\mathcal{H}$ uses a single priority queue.

Note that no computation happens at a given level of abstraction until a lightest derivation of the goal has been found at the level above. This means that the structure of the abstraction hierarchy can be defined dynamically. For example, as in the CFDP algorithm, we could define the set of statements at each level of abstraction by refining the statements that appear in a lightest derivation of the goal at the level above. Here we assume a static abstraction hierarchy.

For each statement $C \in \Sigma_k$ with $0 \leq k \leq m-1$ we use $\ell(C)$ to denote the weight of a lightest derivation for C using R_k while $\ell(context(C))$ denotes the weight of a lightest context for C using R_k. For the most abstract level we define $\ell(\perp) = \ell(context(\perp)) = 0$.

Below we show that HA*LD correctly computes lightest derivations and lightest contexts at every level of abstraction. Moreover, the order in which derivations and contexts are expanded is controlled by a heuristic function defined as follows. For $C \in \Sigma_k$ with $0 \leq k \leq m-1$ define a heuristic value for C using contexts at the level above and a heuristic value for $context(C)$ using derivations at the same level,

$$h(C) = \ell(context(abs(C))),$$
$$h(context(C)) = \ell(C).$$

For the most abstract level we define $h(\perp) = h(context(\perp)) = 0$. Let a generalized statement Φ be either an element of Σ_k for $0 \leq k \leq m$ or an expression of the form $context(C)$ for $C \in \Sigma_k$. We define an intrinsic priority for Φ as follows,

$$p(\Phi) = \ell(\Phi) + h(\Phi).$$

For $C \in \Sigma_k$, we have that $p(context(C))$ is the weight of a lightest derivation of $goal_k$ that uses C, while $p(C)$ is a lower bound on this weight.

The results from Sections 4 and 5 cannot be used directly to show the correctness of HA*LD. This is because the rules in Figure 7 generate heuristic values at the same time they generate derivations that depend on those heuristic values. Intuitively we must show that during the execution of the prioritized rules $\mathcal{H}$, each heuristic value is available at an appropriate point in time. The next lemma shows that the rules in $\mathcal{H}$ satisfy a monotonicity property with respect to the intrinsic priority of generalized statements. Theorem 5 proves the correctness of the hierarchical algorithm.

Lemma 4 (Monotonicity)**.** *For each rule $\Phi_1, \ldots, \Phi_m \rightarrow \Psi$ in the hierarchical algorithm, if the weight of each antecedent Φ_i is $\ell(\Phi_i)$ and the weight of the conclusion is α then*

(a) the priority of the rule is $\alpha + h(\Psi)$.

(b) $\alpha + h(\Psi) \geq p(\Phi_i)$.

Proof. For the rules START1 and START2 the result follows from the fact that the rules have no antecedents and $h(\perp) = h(context(\perp)) = 0$.

Consider a rule labeled BASE with $w = \ell(goal_k)$. To see (a) note that α is always zero and the priority of the rule is $w = h(context(goal_k))$. For (b) we note that $p(goal_k) = \ell(goal_k)$ which equals the priority of the rule.

Now consider a rule labeled UP with $w_c = \ell(context(abs(C)))$ and $w_i = \ell(A_i)$ for all i. For part (a) note how the priority of the rule is $\alpha + w_c$ and $h(C) = w_c$. For part (b) consider the first antecedent of the rule. We have $h(context(abs(C))) = \ell(abs(C)) \leq \ell(C) \leq \alpha$, and $p(context(abs(C))) = w_c + h(context(abs(C))) \leq \alpha + w_c$. Now consider an antecedent A_i. If $abs(A_i) = \perp$ then $p(A_i) = w_i \leq \alpha + w_c$. If $abs(A_i) \neq \perp$ then we can show that $h(A_i) = \ell(context(abs(A_i))) \leq w_c + \alpha - w_i$. This implies that $p(A_i) = w_i + h(A_i) \leq \alpha + w_c$.

Finally consider a rule labeled DOWN with $w_c = \ell(context(C))$ and $w_j = \ell(A_j)$ for all j. For part (a) note that the priority of the rule is $\alpha + w_i$ and $h(context(A_i)) = w_i$. For part (b) consider the first antecedent of the rule. We have $h(context(C)) = \ell(C) \leq v + \sum_j w_j$ and we see that $p(context(C)) = w_c + h(C) \leq \alpha + w_i$. Now consider an antecedent A_j. If $abs(A_j) = \perp$ then $h(A_j) = 0$ and $p(A_j) = w_j \leq \alpha + w_i$. If $abs(A_j) \neq \perp$ we can show that $h(A_j) \leq \alpha + w_i - w_j$. Hence $p(A_j) = w_j + h(A_j) \leq \alpha + w_i$. $\square$

Theorem 5. *The execution of $\mathcal{H}$ maintains the following invariants.*

1. *If $(\Phi = w) \in \mathcal{S}$ then $w = \ell(\Phi)$.*
2. *If $(\Phi = w) \in \mathcal{Q}$ then it has priority $w + h(\Phi)$.*
3. *If $p(\Phi) < p(\mathcal{Q})$ then $(\Phi = \ell(\Phi)) \in \mathcal{S}$*

Here $p(\mathcal{Q})$ denotes the smallest priority in $\mathcal{Q}$.

Proof. In the initial state of the algorithm $\mathcal{S}$ is empty and $\mathcal{Q}$ contains only $(\perp = 0)$ and $(\mathit{context}(\perp) = 0)$ at priority 0. For the initial state invariant 1 is true since $\mathcal{S}$ is empty; invariant 2 follows from the definition of $h(\perp)$ and $h(\mathit{context}(\perp))$; and invariant 3 follows from the fact that $p(\mathcal{Q}) = 0$ and $p(\Phi) \geq 0$ for all Φ. Let $\mathcal{S}$ and $\mathcal{Q}$ denote the state of the algorithm immediately prior to an iteration of the while loop in Figure 5 and suppose the invariants are true. Let $\mathcal{S}'$ and $\mathcal{Q}'$ denote the state of the algorithm after the iteration.

We will first prove invariant 1 for $\mathcal{S}'$. Let $(\Phi = w)$ be the element removed from $\mathcal{Q}$ in this iteration. By the soundness of the rules we have $w \geq \ell(\Phi)$. If $w = \ell(\Phi)$ then clearly invariant 1 holds for $\mathcal{S}'$. If $w > \ell(\Phi)$ invariant 2 implies that $p(\mathcal{Q}) = w + h(\Phi) > \ell(\Phi) + h(\Phi)$ and by invariant 3 we know that $\mathcal{S}$ contains $(\Phi = \ell(\Phi))$. In this case $\mathcal{S}' = \mathcal{S}$.

Invariant 2 for $\mathcal{Q}'$ follows from invariant 2 for $\mathcal{Q}$, invariant 1 for $\mathcal{S}'$, and part (a) of the monotonicity lemma.

Finally, we consider invariant 3 for $\mathcal{S}'$ and $\mathcal{Q}'$. The proof is by reverse induction on the abstraction level of Φ. We say that Φ has level k if $\Phi \in \Sigma_k$ or Φ is of the form $\mathit{context}(C)$ with $C \in \Sigma_k$. In the reverse induction, the base case considers Φ at level m. Initially the algorithm inserts $(\perp = 0)$ and $(\mathit{context}(\perp) = 0)$ in the queue with priority 0. If $p(\mathcal{Q}') > 0$ then $\mathcal{S}'$ must contain $(\perp = 0)$ and $(\mathit{context}(\perp) = 0)$. Hence invariant 3 holds for $\mathcal{S}'$ and $\mathcal{Q}'$ with Φ at level m.

Now we assume that invariant 3 holds for $\mathcal{S}'$ and $\mathcal{Q}'$ with Φ at levels greater than k and consider level k. We first consider statements $C \in \Sigma_k$. Since the rules R_k are additive, every statement C derivable with R_k has a lightest derivation (a derivation with weight $\ell(C)$). This follows from the correctness of Knuth's algorithm. Moreover, for additive rules, subtrees of lightest derivations are also lightest derivations. We show by structural induction that for any lightest derivation with conclusion C such that $p(C) < p(\mathcal{Q}')$ we have $(C = \ell(C)) \in \mathcal{S}'$. Consider a lightest derivation in R_k with conclusion C such that $p(C) < p(\mathcal{Q}')$. The final rule in this derivation $A_1, \ldots, A_n \rightarrow_v C$ corresponds to an UP rule where we add an antecedent for $\mathit{context}(\mathit{abs}(C))$. By part (b) of the monotonicity lemma all the antecedents of this UP rule have intrinsic priority less than $p(\mathcal{Q}')$. By the induction hypothesis on lightest derivations we have $(A_i = \ell(A_i)) \in \mathcal{S}'$. Since invariant 3 holds for statements at levels greater than k we have $(\mathit{context}(\mathit{abs}(C)) = \ell(\mathit{context}(\mathit{abs}(C)))) \in \mathcal{S}'$. This implies that at some point the UP rule was used to derive $(C = \ell(C))$ at priority $p(C)$. But $p(C) < p(\mathcal{Q}')$ and hence this item must have been removed from the queue. Therefore $\mathcal{S}'$ must contain $(C = w)$ for some w and, by invariant 1, $w = \ell(C)$.

Now we consider Φ of the form $\mathit{context}(C)$ with $C \in \Sigma_k$. As before we see that $c(R_k)$ is additive and thus every statement derivable with $c(R_k)$ has a lightest derivation and subtrees of lightest derivations are lightest derivations themselves. We prove by structural induction that for any lightest derivation T with conclusion $context(C)$ such that $p(context(C)) < p(\mathcal{Q}')$ we have $(context(C) = \ell(context(C))) \in \mathcal{S}'$. Suppose the last rule of T is of the form,

$$context(C), A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n \rightarrow_v context(A_i).$$

This rule corresponds to a DOWN rule where we add an antecedent for A_i. By part (b) of the monotonicity lemma all the antecedents of this DOWN rule have intrinsic priority less than $p(\mathcal{Q}')$. By invariant 3 for statements in Σ_k and by the induction hypothesis on lightest derivations using $c(R_k)$, all antecedents of the DOWN rule have their lightest weight in $\mathcal{S}'$. So at some point $(context(A_i) = \ell(context(A_i)))$ was derived at priority $p(A_i)$. Now $p(A_i) < p(\mathcal{Q}')$ implies the item was removed from the queue and, by invariant 1, we have $(context(A_i) = \ell(context(A_i))) \in \mathcal{S}'$.

Now suppose the last (and only) rule in T is $\rightarrow_0 context(goal_k)$. This rule corresponds to a BASE rule where we add $goal_k$ as an antecedent. Note that $p(goal_k) = \ell(goal_k) = p(context(goal_k))$ and hence $p(goal_k) < p(\mathcal{Q}')$. By invariant 3 for statements in Σ_k we have $(goal_k = \ell(goal_k))$ in $\mathcal{S}'$ and at some point the BASE rule was used to queue $(context(goal_k) = \ell(context(goal_k)))$ at priority $p(context(goal_k))$. As in the previous cases $p(context(goal_k)) < p(\mathcal{Q}')$ implies $(context(goal_k) = \ell(context(goal_k))) \in \mathcal{S}'$. $\square$

The last theorem implies that generalized statements Φ are expanded in order of their intrinsic priority. Let K be the number of statements C in the entire abstraction hierarchy with $p(C) \leq p(goal) = \ell(goal)$. For every statement C we have that $p(C) \leq p(context(C))$. We conclude that HA*LD expands at most $2K$ generalized statements before computing a lightest derivation of *goal*.

6.1 Example

Now we consider the execution of HA*LD in a specific example. The example illustrates how HA*LD interleaves the computation of structures at different levels of abstraction. Consider the following abstraction hierarchy with 2 levels.

$$\Sigma_0 = \{X_1, \ldots, X_n, Y_1, \ldots, Y_n, Z_1, \ldots, Z_n, goal_0\}, \quad \Sigma_1 = \{X, Y, Z, goal_1\},$$

$$R_0 = \left\{ \begin{array}{l} \rightarrow_i X_i, \\ \rightarrow_i Y_i, \\ X_i, Y_j \rightarrow_{i*j} goal_0, \\ X_i, Y_i \rightarrow_5 Z_i, \\ Z_i \rightarrow_i goal_0, \end{array} \right\}, \quad R_1 = \left\{ \begin{array}{l} \rightarrow_1 X, \\ \rightarrow_1 Y, \\ X, Y \rightarrow_1 goal_1, \\ X, Y \rightarrow_5 Z, \\ Z \rightarrow_1 goal_1, \end{array} \right\},$$

with $abs(X_i) = X$, $abs(Y_i) = Y$, $abs(Z_i) = Z$ and $abs(goal_0) = goal_1$.

1. Initially $\mathcal{S} = \emptyset$ and $\mathcal{Q} = \{(\bot = 0)$ and $(context(\bot) = 0)$ at priority $0\}$.

2. When $(\bot = 0)$ comes off the queue it gets put in $\mathcal{S}$ but nothing else happens.

3. When $(context(\bot) = 0)$ comes off the queue it gets put in $\mathcal{S}$. Now statements in Σ_1 have an abstract context in $\mathcal{S}$. This causes UP rules that come from rules in R_1 with no antecedents to "fire", putting $(X = 1)$ and $(Y = 1)$ in $\mathcal{Q}$ at priority 1.

4. When $(X = 1)$ and $(Y = 1)$ come off the queue they get put in $\mathcal{S}$, causing two UP rules to fire, putting $(goal_1 = 3)$ at priority 3 and $(Z = 7)$ at priority 7 in the queue.

5. We have,
$$\mathcal{S} = \{(\bot = 0),\ (context(\bot) = 0),\ (X = 1),\ (Y = 1)\}$$
$$\mathcal{Q} = \{(goal_1 = 3) \text{ at priority } 3,\ (Z = 7) \text{ at priority } 7\}$$

6. At this point $(goal_1 = 3)$ comes off the queue and goes into in $\mathcal{S}$. A BASE rule fires putting $(context(goal_1) = 0)$ in the queue at priority 3.

7. $(context(goal_1) = 0)$ comes off the queue. This is the base case for contexts in Σ_1. Two DOWN rules use $(context(goal_1) = 0)$, $(X = 1)$ and $(Y = 1)$ to put $(context(X) = 2)$ and $(context(Y) = 2)$ in $\mathcal{Q}$ at priority 3.

8. $(context(X) = 2)$ comes off the queue and gets put in $\mathcal{S}$. Now we have an abstract context for each $X_i \in \Sigma_0$, so UP rules to put $(X_i = i)$ in $\mathcal{Q}$ at priority $i + 2$.

9. Now $(context(Y) = 2)$ comes off the queue and goes into $\mathcal{S}$. As in the previous step UP rules put $(Y_i = i)$ in $\mathcal{Q}$ at priority $i + 2$.

10. We have,
$$\begin{aligned}\mathcal{S} \;=\; &\{(\bot = 0),\ (context(\bot) = 0),\ (X = 1),\ (Y = 1),\ (goal_1 = 3),\\ &(context(goal_1) = 0),\ (context(X) = 2),\ (context(Y) = 2)\}\end{aligned}$$
$$\mathcal{Q} = \{(X_i = i) \text{ and } (Y_i = i) \text{ at priority } i + 2 \text{ for } 1 \le i \le n,\ (Z = 7) \text{ at priority } 7\}$$

11. Next both $(X_1 = 1)$ and $(Y_1 = 1)$ will come off the queue and go into $\mathcal{S}$. This causes an UP rule to put $(goal_0 = 3)$ in the queue at priority 3.

12. $(goal_0 = 3)$ comes off the queue and goes into $\mathcal{S}$. The algorithm can stop now since we have a derivation of the most concrete goal.

Note how HA*LD terminates before fully computing abstract derivations and contexts. In particular $(Z = 7)$ is in $\mathcal{Q}$ but Z was never expanded. Moreover $context(Z)$ is not even in the queue. If we keep running the algorithm it would eventually derive $context(Z)$, and that would allow the Z_i to be derived.

7. The Perception Pipeline

Figure 8 shows a hypothetical run of the hierarchical algorithm for a processing pipeline of a vision system. In this system weighted statements about edges are used to derive weighted statements about contours which provide input to later stages ultimately resulting in statements about recognized objects.

It is well known that the subjective presence of edges at a particular image location can depend on the context in which a given image patch appears. This can be interpreted in the perception pipeline by stating that higher level processes — those later in the pipeline — influence low-level interpretations. This kind of influence happens naturally in a lightest

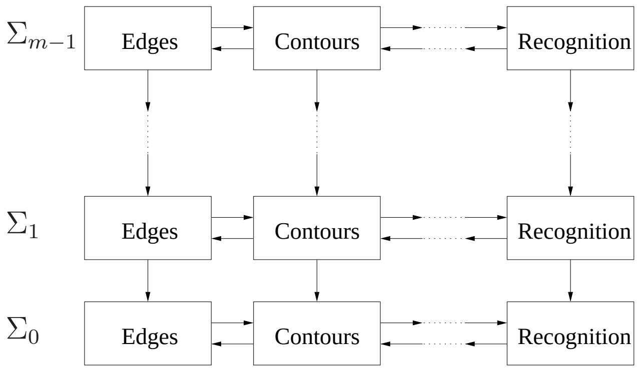


Figure 8: A vision system with several levels of processing. Forward arrows represent the normal flow of information from one stage of processing to the next. Backward arrows represent the computation of contexts. Downward arrows represent the influence of contexts.

derivation problem. For example, the lightest derivation of a complete scene analysis might require the presence of an edge that is not locally apparent. By implementing the whole system as a single lightest derivation problem we avoid the need to make hard decisions between stages of the pipeline.

The influence of late pipeline stages in guiding earlier stages is pronounced if we use HA*LD to compute lightest derivations. In this case the influence is apparent not only in the structure of the optimal solution but also in the flow of information across different stages of processing. In HA*LD a complete interpretation derived at one level of abstraction guides all processing stages at a more concrete level. Structures derived at late stages of the pipeline guide earlier stages through abstract context weights. This allows the early processing stages to concentrate computational efforts in constructing structures that will likely be part of the globally optimal solution.

While we have emphasized the use of admissible heuristics, we note that the A* architecture, including HA*LD, can also be used with inadmissible heuristic functions (of course this would break our optimality guarantees). Inadmissible heuristics are important because admissible heuristics tend to force the first few stages of a processing pipeline to generate too many derivations. As derivations are composed their weights increase and this causes a large number of derivations to be generated at the first few stages of processing before the first derivation reaches the end of the pipeline. Inadmissible heuristics can produce behavior similar to beam search — derivations generated in the first stage of the pipeline can flow through the whole pipeline quickly. A natural way to construct inadmissible heuristics is to simply "scale-up" an admissible heuristic such as the ones obtained from abstractions. It is then possible to construct a hierarchical algorithm where inadmissible heuristics obtained from one level of abstraction are used to guide search at the level below.

8. Other Hierarchical Methods

In this section we compare HA*LD to other hierarchical search methods.

8.1 Coarse-to-Fine Dynamic Programming

HA*LD is related to the coarse-to-fine dynamic programming (CFDP) method described by Raphael (2001). To understand the relationship consider the problem of finding the shortest path from s to t in a trellis graph like the one shown in Figure 9(a). Here we have k columns of n nodes and every node in one column is connected to a constant number of nodes in the next column. Standard dynamic programming can be used to find the shortest path in $O(kn)$ time. Both CFDP and HA*LD can often find the shortest path much faster. On the other hand the worst case behavior of these algorithms is very different as we describe below, with CFDP taking significantly more time than HA*LD.

The CFDP algorithm works by coarsening the graph, grouping nodes in each column into a small number of supernodes as illustrated in Figure 9(b). The weight of an edge between two supernodes A and B is the minimum weight between nodes $a \in A$ and $b \in B$. The algorithm starts by using dynamic programming to find the shortest path P from s to t in the coarse graph, this is shown in bold in Figure 9(b). The supernodes along P are partitioned to define a finer graph as shown in Figure 9(c) and the procedure repeated. Eventually the shortest path P will only go through supernodes of size one, corresponding to a path in the original graph. At this point we know that P must be a shortest path from s to t in the original graph. In the best case the optimal path in each iteration will be a refinement of the optimal path from the previous iteration. This would result in $O(\log n)$ shortest paths computations, each in fairly coarse graphs. On the other hand, in the worst case CFDP will take $\Omega(n)$ iterations to refine the whole graph, and many of the iterations will involve finding shortest paths in large graphs. In this case CFDP takes $\Omega(kn^2)$ time which is much worst than the standard dynamic programming approach.

Now suppose we use HA*LD to find the shortest path from s to t in a graph like the one in Figure 9(a). We can build an abstraction hierarchy with $O(\log n)$ levels where each supernode at level i contains 2^i nodes from one column of the original graph. The coarse graph in Figure 9(b) represents the highest level of this abstraction hierarchy. Note that HA*LD will consider a small number, $O(\log n)$, of predefined graphs while CFDP can end up considering a much larger number, $\Omega(n)$, of graphs. In the best case scenario HA*LD will expand only the nodes that are in the shortest path from s to t at each level of the hierarchy. In the worst case HA*LD will compute a lightest path and context for every node in the hierarchy (here a context for a node v is a path from v to t). At the i-th abstraction level we have a graph with $O(kn/2^i)$ nodes and edges. HA*LD will spend at most $O(kn\log(kn)/2^i)$ time computing paths and contexts at level i. Summing over levels we get at most $O(kn\log(kn))$ time total, which is not much worst than the $O(kn)$ time taken by the standard dynamic programming approach.

8.2 Hierarchical Heuristic Search

Our hierarchical method is also related to the HA* and HIDA* algorithms described by Holte et al. (1996) and Holte et al. (2005). These methods are restricted to shortest paths problems but they also use a hierarchy of abstractions. A heuristic function is defined for each level of abstraction using shortest paths to the goal at the level above. The main idea is to run A* or IDA* to compute a shortest path while computing heuristic values on-demand. Let *abs* map a node to its abstraction and let g be the goal node in the concrete

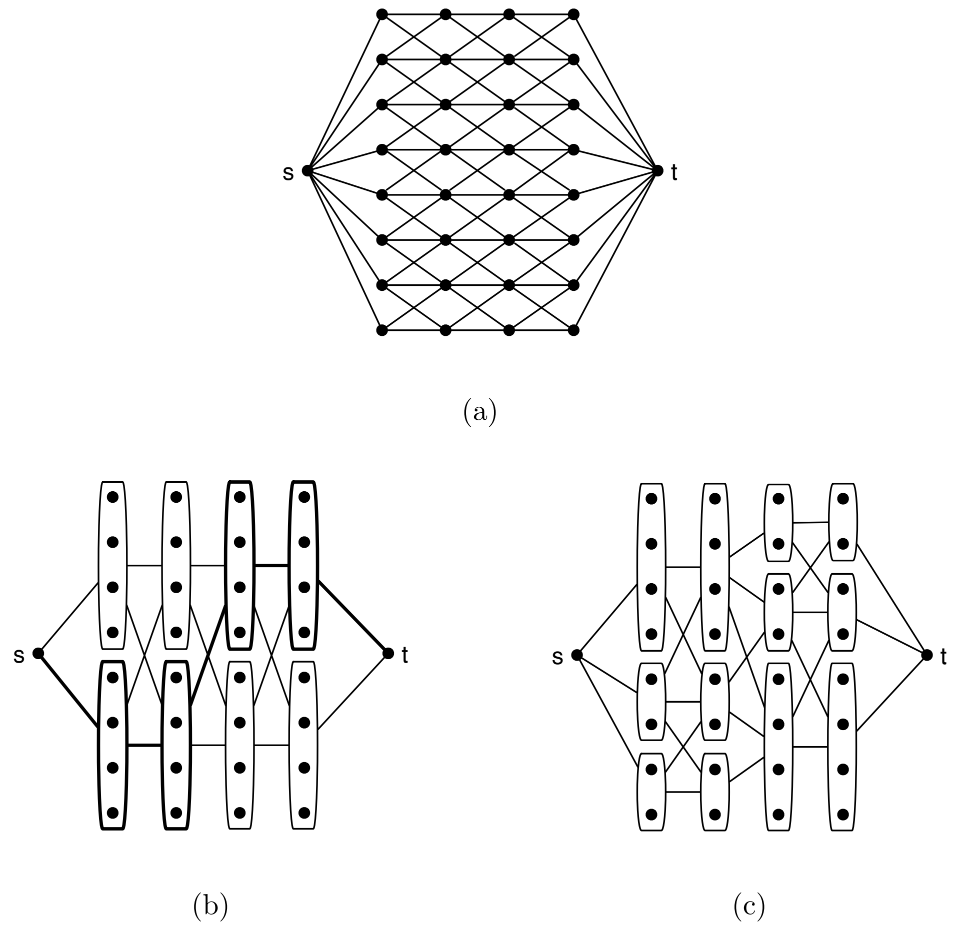


Figure 9: (a) Original dynamic programming graph. (b) Coarse graph with shortest path shown in bold. (c) Refinement of the coarse graph along the shortest path.

graph. Whenever the heuristic value for a concrete node v is needed we call the algorithm recursively to find the shortest path from $abs(v)$ to $abs(g)$. This recursive call uses heuristic values defined from a further abstraction, computed through deeper recursive calls.

It is not clear how to generalize HA* and HIDA* to lightest derivation problems that have rules with multiple antecedents. Another disadvantage is that these methods can potentially "stall" in the case of directed graphs. For example, suppose that when using HA* or HIDA* we expand a node with two successors x and y, where x is close to the goal but y is very far. At this point we need a heuristic value for x and y, and we might have to spend a long time computing a shortest path from $abs(y)$ to $abs(g)$. On the other hand, HA*LD would not wait for this shortest path to be fully computed. Intuitively HA*LD would compute shortest paths from $abs(x)$ and $abs(y)$ to $abs(g)$ simultaneously. As soon as the shortest path from $abs(x)$ to $abs(g)$ is found we can start exploring the path from x to g, independent of how long it would take to compute a path from $abs(y)$ to $abs(g)$.

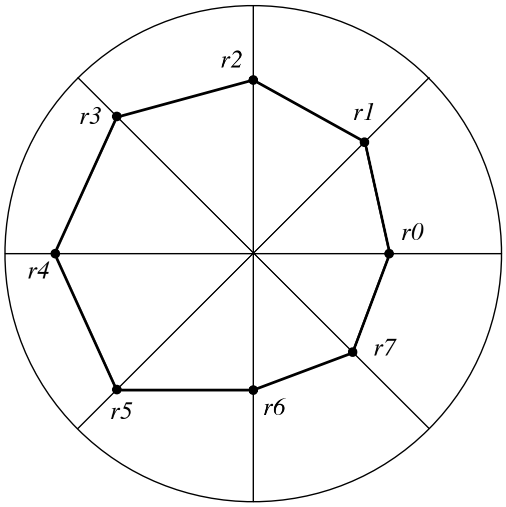


Figure 10: A convex set specified by a hypothesis $(r_0, \ldots, r_7)$.

9. Convex Object Detection

Now we consider an application of HA*LD to the problem of detecting convex objects in images. We pose the problem using a formulation similar to the one described by Raphael (2001), where the optimal convex object around a point can be found by solving a shortest path problem. We compare HA*LD to other search methods, including CFDP and A* with pattern databases. The results indicate that HA*LD performs better than the other methods over a wide range of inputs.

Let x be a reference point inside a convex object. We can represent the object boundary using polar coordinates with respect to a coordinate system centered at x. In this case the object is described by a periodic function $r(\theta)$ specifying the distance from x to the object boundary as a function of the angle θ. Here we only specify $r(\theta)$ at a finite number of angles $(\theta_0, \ldots, \theta_{N-1})$ and assume the boundary is a straight line segment between sample points. We also assume the object is contained in a ball of radius R around x and that $r(\theta)$ is an integer. Thus an object is parametrized by $(r_0, \ldots, r_{N-1})$ where $r_i \in [0, R-1]$. An example with $N = 8$ angles is shown in Figure 10.

Not every hypothesis $(r_0, \ldots, r_{N-1})$ specifies a convex object. The hypothesis describes a convex set exactly when the object boundary turns left at each sample point (θ_i, r_i) as i increases. Let $C(r_{i-1}, r_i, r_{i+1})$ be a Boolean function indicating when three sequential values for $r(\theta)$ define a boundary that is locally convex at i. The hypothesis $(r_0, \ldots, r_{N-1})$ is convex when it is locally convex at each i.[3]

Throughout this section we assume that the reference point x is fixed in advance. Our goal is to find an "optimal" convex object around a given reference point. In practice reference locations can be found using a variety of methods such as a Hough transform.

3. This parametrization of convex objects is similar but not identical to the one used by Raphael (2001).

Let $D(i, r_i, r_{i+1})$ be an image data cost measuring the evidence for a boundary segment from (θ_i, r_i) to (θ_{i+1}, r_{i+1}). We consider the problem of finding a convex object for which the sum of the data costs along the whole boundary is minimal. That is, we look for a convex hypothesis minimizing the following energy function,

$$E(r_0, \ldots, r_{N-1}) = \sum_{i=0}^{N-1} D(i, r_i, r_{i+1}).$$

The data costs can be precomputed and specified by a lookup table with $O(NR^2)$ entries. In our experiments we use a data cost based on the integral of the image gradient along each boundary segment. Another approach would be to use the data term described by Raphael (2001) where the cost depends on the contrast between the inside and the outside of the object measured within the pie-slice defined by θ_i and θ_{i+1}.

An optimal convex object can be found using standard dynamic programming techniques. Let $B(i, r_0, r_1, r_{i-1}, r_i)$ be the cost of an optimal partial convex object starting at r_0 and r_1 and ending at r_{i-1} and r_i. Here we keep track of the last two boundary points to enforce the convexity constraint as we extend partial objects. We also have to keep track of the first two boundary points to enforce that $r_N = r_0$ and the convexity constraint at r_0. We can compute B using the recursive formula,

$$\begin{aligned} B(1, r_0, r_1, r_0, r_1) &= D(0, r_0, r_1), \\ B(i+1, r_0, r_1, r_i, r_{i+1}) &= \min_{r_{i-1}} B(i, r_0, r_1, r_{i-1}, r_i) + D(i, r_i, r_{i+1}), \end{aligned}$$

where the minimization is over choices for r_{i-1} such that $C(r_{i-1}, r_i, r_{i+1}) = \textbf{true}$. The cost of an optimal object is given by the minimum value of $B(N, r_0, r_1, r_{N-1}, r_0)$ such that $C(r_{N-1}, r_0, r_1) = \textbf{true}$. An optimal object can be found by tracing-back as in typical dynamic programming algorithms. The main problem with this approach is that the dynamic programming table has $O(NR^4)$ entries and it takes $O(R)$ time to compute each entry. The overall algorithm runs in $O(NR^5)$ time which is quite slow.

Now we show how optimal convex objects can be defined in terms of a lightest derivation problem. Let $\mathit{convex}(i, r_0, r_1, r_{i-1}, r_i)$ denote a partial convex object starting at r_0 and r_1 and ending at r_{i-1} and r_i. This corresponds to an entry in the dynamic programming table described above. Define the set of statements,

$$\Sigma = \{\mathit{convex}(i, a, b, c, d) \mid i \in [1, N],\ a, b, c, d \in [0, R-1]\} \cup \{\mathit{goal}\}.$$

An optimal convex object corresponds to a lightest derivations of *goal* using the rules in Figure 11. The first set of rules specify the cost of a partial object from r_0 to r_1. The second set of rules specify that an object ending at r_{i-1} and r_i can be extended with a choice for r_{i+1} such that the boundary is locally convex at r_i. The last set of rules specify that a complete convex object is a partial object from r_0 to r_N such that $r_N = r_0$ and the boundary is locally convex at r_0.

To construct an abstraction hierarchy we define L nested partitions of the radius space $[0, R-1]$ into ranges of integers. In an abstract statement instead of specifying an integer value for $r(\theta)$ we will specify the range in which $r(\theta)$ is contained. To simplify notation we

(1) for $r_0, r_1 \in [0, R-1]$,

$$\frac{}{convex(1, r_0, r_1, r_0, r_1) = D(0, r_0, r_1)}$$

(2) for $r_0, r_1, r_{i-1}, r_i, r_{i+1} \in [0, R-1]$ such that $C(r_{i-1}, r_i, r_{i+1}) = \textbf{true}$,

$$\frac{convex(i, r_0, r_1, r_{i-1}, r_i) = w}{convex(i+1, r_0, r_1, r_i, r_{i+1}) = w + D(i, r_i, r_{i+1})}$$

(3) for $r_0, r_1, r_{N-1} \in [0, R-1]$ such that $C(r_{N-1}, r_0, r_1) = \textbf{true}$,

$$\frac{convex(N, r_0, r_1, r_{N-1}, r_0) = w}{goal = w}$$

Figure 11: Rules for finding an optimal convex object.

assume that R is a power of two. The k-th partition P^k contains $R/2^k$ ranges, each with 2^k consecutive integers. The j-th range in P^k is given by $[j * 2^k, (j+1) * 2^k - 1]$.

The statements in the abstraction hierarchy are,

$$\Sigma_k = \{convex(i, a, b, c, d) \mid i \in [1, N],\ a, b, c, d \in P^k\} \cup \{goal_k\},$$

for $k \in [0, L-1]$. A range in P^0 contains a single integer so $\Sigma_0 = \Sigma$. Let f map a range in P^k to the range in P^{k+1} containing it. For statements in level $k < L-1$ we define the abstraction function,

$$abs(convex(i, a, b, c, d)) = convex(i, f(a), f(b), f(c), f(d)),$$
$$abs(goal_k) = goal_{k+1}.$$

The abstract rules use bounds on the data costs for boundary segments between (θ_i, s_i) and (θ_{i+1}, s_{i+1}) where s_i and s_{i+1} are ranges in P^k,

$$D^k(i, s_i, s_{i+1}) = \min_{\substack{r_i \in s_i \\ r_{i+1} \in s_{i+1}}} D(i, r_i, r_{i+1}).$$

Since each range in P^k is the union of two ranges in P^{k-1} one entry in D^k can be computed quickly (in constant time) once D^{k-1} is computed. The bounds for all levels can be computed in $O(NR^2)$ time total. We also need abstract versions of the convexity constraints. For $s_{i-1}, s_i, s_{i+1} \in P^k$, let $C^k(s_{i-1}, s_i, s_{i+1}) = \textbf{true}$ if there exist integers r_{i-1}, r_i and r_{i+1} in s_{i-1}, s_i and s_{i+1} respectively such that $C(r_{i-1}, r_i, r_{i+1}) = \textbf{true}$. The value of C^k can be defined in closed form and evaluated quickly using simple geometry.

The rules in the abstraction hierarchy are almost identical to the rules in Figure 11. The rules in level k are obtained from the original rules by simply replacing each instance of $[0, R-1]$ by P^k, C by C^k and D by D^k.

Standard DP	6718.6 seconds
CFDP	13.5 seconds
HA*LD	8.6 seconds
A* with pattern database in Σ_2	14.3 seconds
A* with pattern database in Σ_3	29.7 seconds

Table 1: Running time comparison for the example in Figure 12.

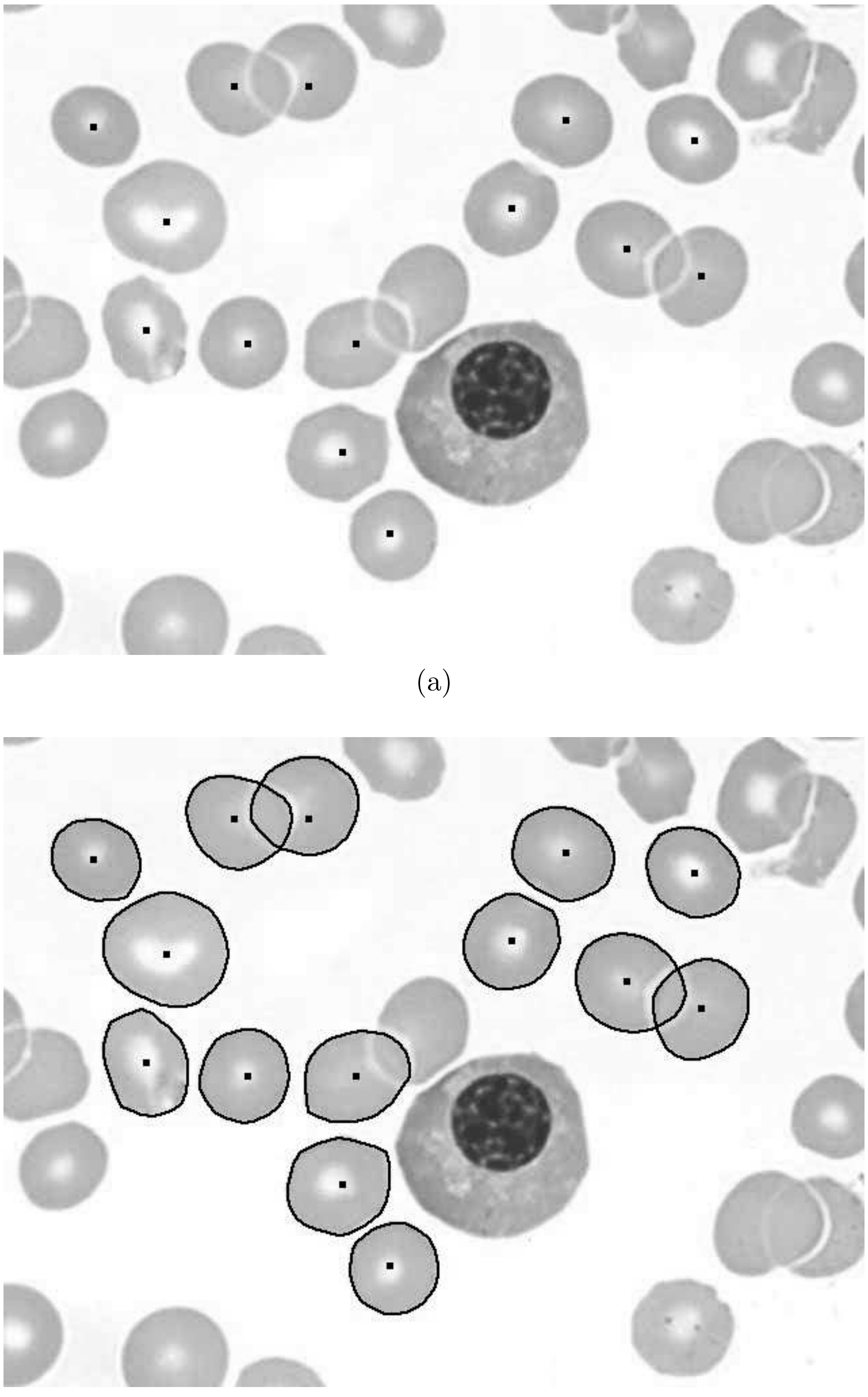
9.1 Experimental Results

Figure 12 shows an example image with a set of reference locations that we selected manually and the optimal convex object found around each reference point. There are 14 reference locations and we used $N = 30$ and $R = 60$ to parametrize each object. Table 1 compares the running time of different optimization algorithms we implemented for this problem. Each line shows the time it took to solve all 14 problems contained in the example image using a particular search algorithm. The standard DP algorithm uses the dynamic programming solution outlined above. The CFDP method is based on the algorithm by Raphael (2001) but modified for our representation of convex objects. Our hierarchical A* algorithm uses the abstraction hierarchy described here. For A* with pattern databases we used dynamic programming to compute a pattern database at a particular level of abstraction, and then used this database to provide heuristic values for A*. Note that for the problem described here the pattern database depends on the input. The running times listed include the time it took to compute the pattern database in each case.

We see that CFDP, HA*LD and A* with pattern databases are much more efficient than the standard dynamic programming algorithm that does not use abstractions. HA*LD is slightly faster then the other methods in this example. Note that while the running time varies from algorithm to algorithm the output of every method is the same as they all find globally optimum objects.

For a quantitative evaluation of the different search algorithms we created a large set of problems of varying difficulty and size as follows. For a given value of R we generated square images of width and height $2 * R + 1$. Each image has a circle with radius less than R near the center and the pixels in an image are corrupted by independent Gaussian noise. The difficulty of a problem is controlled by the standard deviation, σ, of the noise. Figure 13 shows some example images and optimal convex object found around their centers.

The graph in Figure 14 shows the running time (in seconds) of the different search algorithms as a function of the noise level when the problem size is fixed at $R = 100$. Each sample point indicates the average running time over 200 random inputs. The graph shows running times up to a point after which the circles can not be reliably detected. We compared HA*LD with CFDP and A* using pattern databases (PD2 and PD3). Here PD2 and PD3 refer to A* with a pattern database defined in Σ_2 and Σ_3 respectively. Since the pattern database needs to be recomputed for each input there is a trade-off in the amount of time spent computing the database and the accuracy of the heuristic it provides. We see that for easy problems it is better to use a smaller database (defined at a higher level of abstraction) while for harder problems it is worth spending time computing a bigger database. HA*LD outperforms the other methods in every situation captured here.



(a)

(b)

Figure 12: (a) Reference locations. (b) Optimal convex objects.

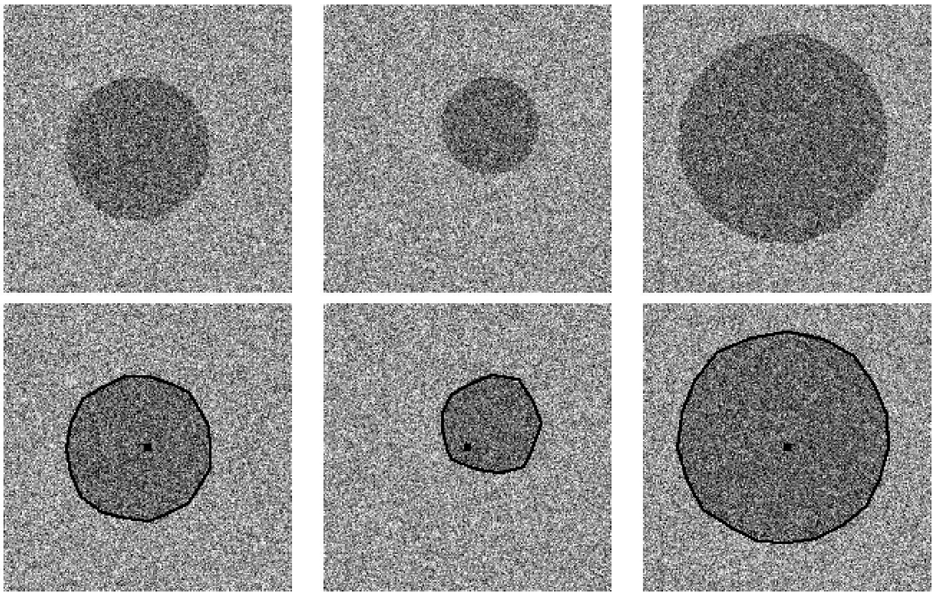


Figure 13: Random images with circles and the optimal convex object around the center of each one (with $N = 20$ and $R = 100$). The noise level in the images is $\sigma = 50$.

Figure 15 shows the running time of the different methods as a function of the problem size R, on problems with a fixed noise level of $\sigma = 100$. As before each sample point indicates the average running time taken over 200 random inputs. We see that the running time of the pattern database approach grows quickly as the problem size increases. This is because computing the database at any fixed level of abstraction takes $O(NR^5)$ time. On the other hand the running time of both CFDP and HA*LD grows much slower. While CFDP performed essentially as well as HA*LD in this experiment, the graph in Figure 14 shows that HA*LD performs better as the difficulty of the problem increases.

10. Finding Salient Curves in Images

A classical problem in computer vision involves finding salient curves in images. Intuitively the goal is to find long and smooth curves that go along paths with high image gradient. The standard way to pose the problem is to define a saliency score and search for curves optimizing that score. Most methods use a score defined by a simple combination of local terms. For example, the score usually depends on the curvature and the image gradient at each point of a curve. This type of score can often be optimized efficiently using dynamic programming or shortest paths algorithms (Montanari, 1971; Shashua & Ullman, 1988; Basri & Alter, 1996; Williams & Jacobs, 1996).

Here we consider a new compositional model for finding salient curves. An important aspect of this model is that it can capture global shape constraints. In particular, it looks for curves that are almost straight, something that can not be done using local constraints alone. Local constraints can enforce small curvature at each point of a curve, but this is

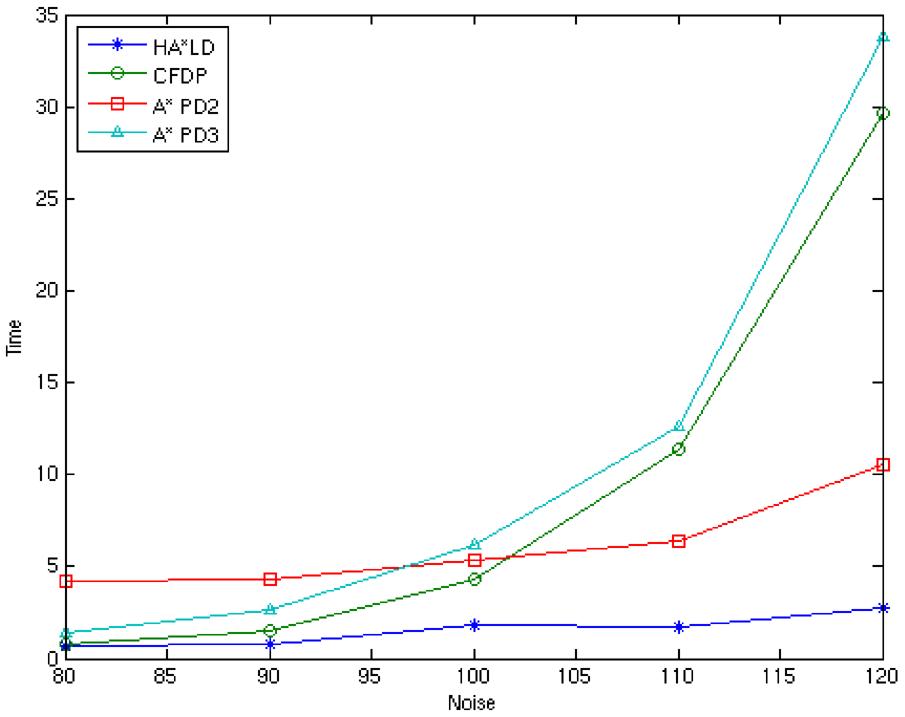


Figure 14: Running time of different search algorithms as a function of the noise level σ in the input. Each sample point indicates the average running time taken over 200 random inputs. In each case $N = 20$ and $R = 100$. See text for discussion.

not enough to prevent curves from turning and twisting around over long distances. The problem of finding the most salient curve in an image with the compositional model defined here can be solved using dynamic programming, but the approach is too slow for practical use. Shortest paths algorithms are not applicable because of the compositional nature of the model. Instead we can use A*LD with a heuristic function derived from an abstraction (a pattern database).

Let C_1 be a curve with endpoints a and b and C_2 be a curve with endpoints b and c. The two curves can be composed to form a curve C from a to c. We define the weight of the composition to be the sum of the weights of C_1 and C_2 plus a shape cost that depends on the geometric arrangement of points (a, b, c). Figure 16 illustrates the idea and the shape costs we use. Note that when C_1 and C_2 are long, the arrangement of their endpoints reflect non-local geometric properties. In general we consider composing C_1 and C_2 if the angle formed by $\overline{ab}$ and $\overline{bc}$ is at least $\pi/2$ and the lengths of C_1 and C_2 are approximately equal. These constraints reduce the total number of compositions and play an important role in the abstract problem defined below.

Besides the compositional rule we say that if a and b are nearby locations, then there is a short curve with endpoints a and b. This forms a base case for creating longer curves. We

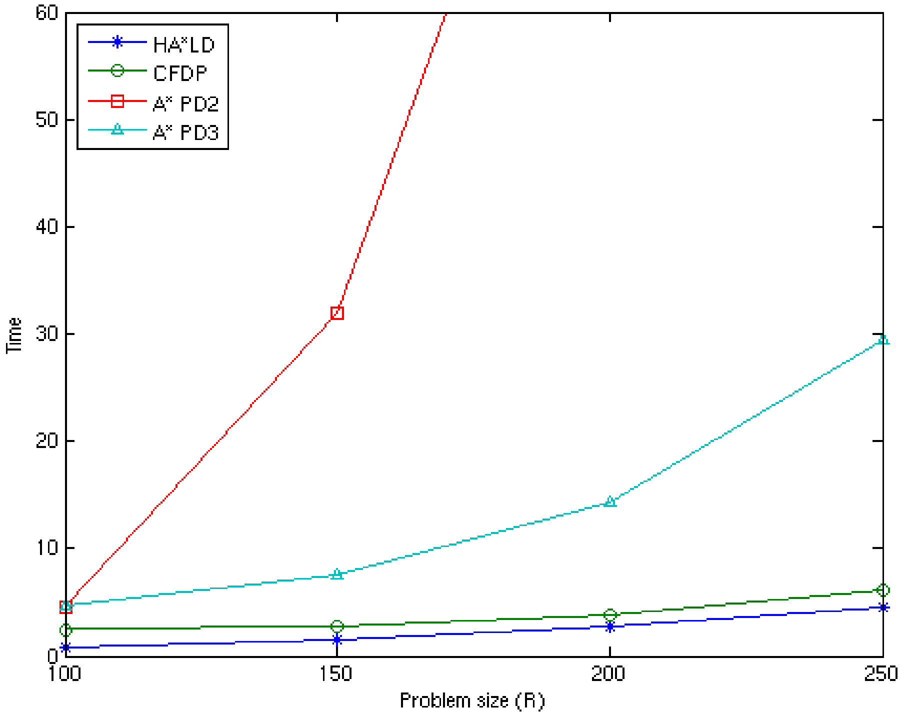


Figure 15: Running time of different search algorithms as a function of the problem size R. Each sample point indicates the average running time taken over 200 random inputs. In each case $N = 20$ and $\sigma = 100$. See text for discussion.

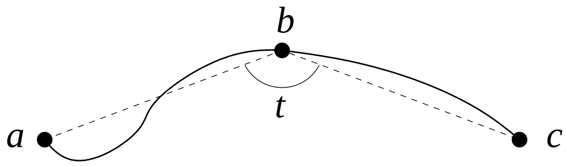


Figure 16: A curve with endpoints (a, c) is formed by composing curves with endpoints (a, b) and (b, c). We assume that $t \geq \pi/2$. The cost of the composition is proportional to $\sin^2(t)$. This cost is scale invariant and encourages curves to be relatively straight.

assume that these short curves are straight, and their weight depends only on the image data along the line segment from a to b. We use a data term, $seg(a, b)$, that is zero if the image gradient along pixels in $\overline{ab}$ is perpendicular to $\overline{ab}$, and higher otherwise.

Figure 17 gives a formal definition of the two rules in our model. The constants k_1 and k_2 specify the minimum and maximum length of the base case curves, while L is a constant

(1) for pixels a, b, c where the angle between $\overline{ab}$ and $\overline{bc}$ is at least $\pi/2$ and for $0 \leq i \leq L$,

$$\frac{\begin{array}{l} \mathit{curve}(a,b,i) = w_1 \\ \mathit{curve}(b,c,i) = w_2 \end{array}}{\mathit{curve}(a,c,i+1) = w_1 + w_2 + \mathit{shape}(a,b,c)}$$

(2) for pixels a, b with $k_1 \leq ||a - b|| \leq k_2$,

$$\frac{}{\mathit{curve}(a,b,0) = \mathit{seg}(a,b)}$$

Figure 17: Rules for finding "almost straight" curves between a pair of endpoints. Here L, k_1 and k_2 are constants, while $\mathit{shape}(a,b,c)$ is a function measuring the cost of a composition.

controlling the maximum depth of derivations. A derivation of $\mathit{curve}(a,b,i)$ encodes a curve from a to b. The value i can be seen as an approximate measure of arclength. A derivation of $\mathit{curve}(a,b,i)$ is a full binary tree of depth i that encodes a curve with length between $2^i k_1$ and $2^i k_2$. We let $k_2 = 2k_1$ to allow for curves of any length.

The rules in Figure 17 do not define a good measure of saliency by themselves because they always prefer short curves over long ones. We can define the saliency of a curve in terms of its weight *minus* its arclength, so that salient curves will be light and long. Let λ be a positive constant. We consider finding the lightest derivation of *goal* using,

$$\frac{\mathit{curve}(a,b,i) = w}{\mathit{goal} = w - \lambda 2^i}$$

For an $n \times n$ image there are $\Omega(n^4)$ statements of the form $\mathit{curve}(a,c,i)$. Moreover, if a and c are far apart there are $\Omega(n)$ choices for a "midpoint" b defining the two curves that are composed in a lightest derivation of $\mathit{curve}(a,c,i)$. This makes a dynamic programming solution to the lightest derivation problem impractical. We have tried using KLD but even for small images the algorithm runs out of memory after a few minutes. Below we describe an abstraction we have used to define a heuristic function for A*LD.

Consider a hierarchical set of partitions of an image into boxes. The i-th partition is defined by tiling the image into boxes of $2^i \times 2^i$ pixels. The partitions form a pyramid with boxes of different sizes at each level. Each box at level i is the union of 4 boxes at the level below it, and the boxes at level 0 are the pixels themselves. Let $f_i(a)$ be the box containing a in the i-th level of the pyramid. Now define

$$\mathit{abs}(\mathit{curve}(a,b,i)) = \mathit{curve}(f_i(a), f_i(b), i).$$

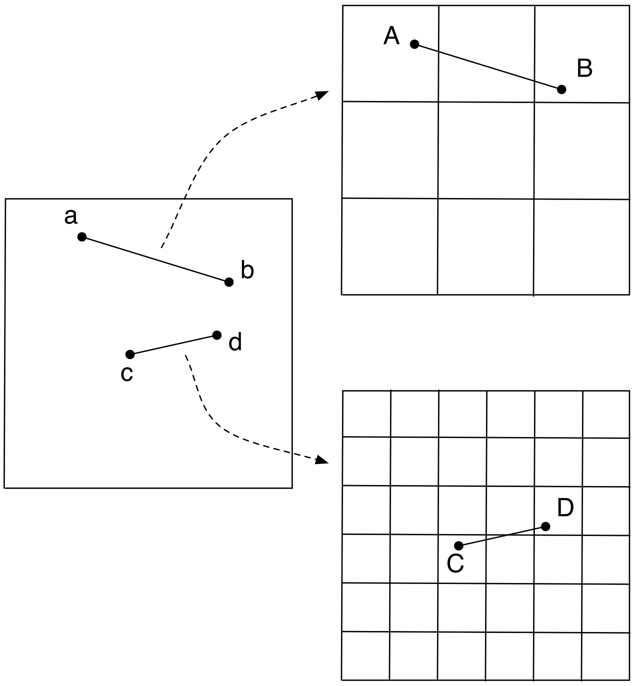


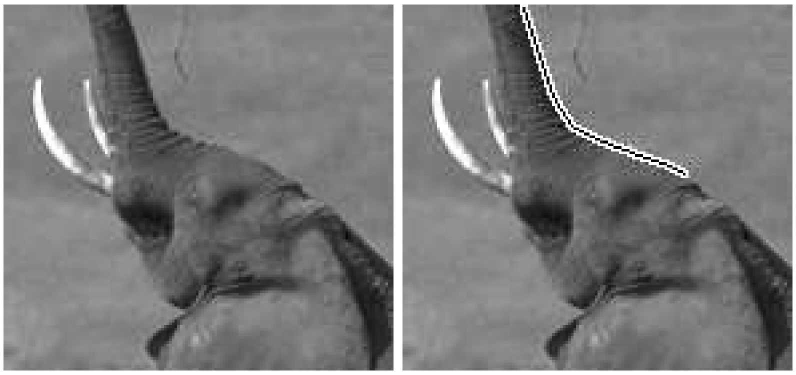
Figure 18: The abstraction maps each curve statement to a statement about curves between boxes. If $i > j$ then $curve(a, b, i)$ gets coarsened more than $curve(c, d, j)$. Since light curves are almost straight, $i > j$ usually implies that $||a - b|| > ||c - d||$.

Figure 18 illustrates how this map selects a pyramid level for an abstract statement. Intuitively *abs* defines an adaptive coarsening criteria. If a and b are far from each other, a curve from a to b must be long, which in turn implies that we map a and b to boxes in a coarse partition of the image. This creates an abstract problem that has a small number of statements without losing too much information.

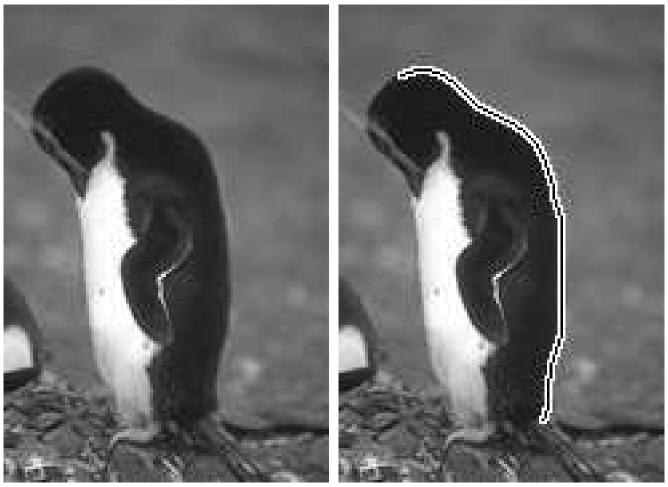
To define the abstract problem we also need to define a set of abstract rules. Recall that for every concrete rule r we need a corresponding abstract rule r' where the weight of r' is at most the weight of r. There are a small number of rules with no antecedents in Figure 17. For each concrete rule $\rightarrow_{seg(a,b)} curve(a, b, 0)$ we define a corresponding abstract rule, $\rightarrow_{seg(a,b)} abs(curve(a, b, 0))$. The compositional rules from Figure 17 lead to abstract rules for composing curves between boxes,

$$curve(A, B, i), curve(B, C, i) \rightarrow_v curve(A', C', i + 1),$$

where A, B and C are boxes at the i-th pyramid level while A' and C' are the boxes at level $i + 1$ containing A and C respectively. The weight v should be at most $shape(a, b, c)$ where a, b and c are arbitrary pixels in A, B and C respectively. We compute a value for v by bounding the orientations of the line segments $\overline{ab}$ and $\overline{bc}$ between boxes.



146 × 137 pixels. Running time: 50 seconds (38 + 12).



122 × 179 pixels. Running time: 65 seconds (43 + 22).

226 × 150 pixels. Running time: 73 seconds (61 + 12).

Figure 19: The most salient curve in different images. The running time is the sum of the time spent computing the pattern database and the time spent solving the concrete problem.

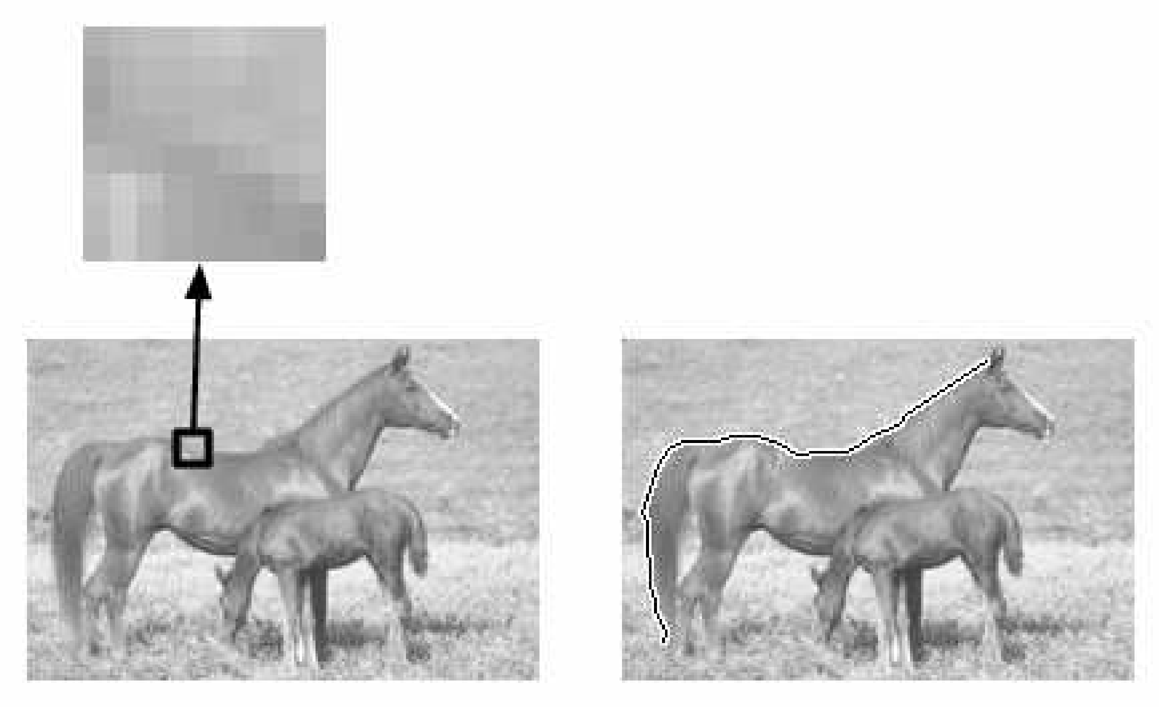


Figure 20: An example where the most salient curve goes over locations with essentially no local evidence for a the curve at those locations.

The abstract problem defined above is relatively small even in large images, so we can use the pattern database approach outlined in Section 5.1. For each input image we use KLD to compute lightest context weights for every abstract statement. We then use these weights as heuristic values for solving the concrete problem with A*LD. Figure 19 illustrates some of the results we obtained using this method. It seems like the abstract problem is able to capture that most short curves can not be extended to a salient curve. It took about one minute to find the most salient curve in each of these images. Figure 19 lists the dimensions of each image and the running time in each case.

Note that our algorithm does not rely on an initial binary edge detection stage. Instead the base case rules allow for salient curves to go over any pixel, even if there is no local evidence for a boundary at a particular location. Figure 20 shows an example where this happens. In this case there is a small part of the horse back that blends with the background if we consider local properties alone.

The curve finding algorithm described in this section would be very difficult to formulate without A*LD and the general notion of heuristics derived from abstractions for lightest derivation problems. However, using the framework introduced in this paper it becomes relatively easy to specify the algorithm.

In the future we plan to "compose" the rules for computing salient curves with rules for computing more complex structures. The basic idea of using a pyramid of boxes for defining an abstract problem should be applicable to a variety of problems in computer vision.

11. Conclusion

Although we have presented some preliminary results in the last two sections, we view the main contribution of this paper as providing a general architecture for perceptual inference. Dijkstra's shortest paths algorithm and A* search are both fundamental algorithms with many applications. Knuth noted the generalization of Dijkstra's algorithm to more general problems defined by a set of recursive rules. In this paper we have given similar generalizations for A* search and heuristics derived from abstractions. We have also described a new method for solving lightest derivation problems using a hierarchy of abstractions. Finally, we have outlined an approach for using these generalizations in the construction of processing pipelines for perceptual inference.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0535174 and 0534820.

References

Basri, R., & Alter, T. (1996). Extracting salient curves from images: An analysis of the saliency network. In *IEEE Conference on Computer Vision and Pattern Recognition*.

Bonet, B., & Geffner, H. (2005). An algorithm better than AO*. In *Proceedings of the National Conference on Artificial Intelligence*.

Bulitko, V., Sturtevant, N., Lu, J., & Yau, T. (2006). State abstraction in real-time heuristic search. *Technical Report, University of Alberta, Department of Computer Science*.

Charniak, E. (1996). *Statistical Language Learning*. MIT Press.

Culberson, J., & Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, *14*(3), 318–334.

Dijkstra, E. (1959). A note on two problems in connection with graphs. *Numerical Mathematics*, *1*, 269–271.

Edelkamp, S. (2002). Symbolic pattern databases in heuristic search panning. In *International Conference on AI Planning and Scheduling*.

Felner, A. (2005). Finding optimal solutions to the graph partitioning problem with heuristic search. *Annals of Mathematics and Artificial Intelligence*, *45*(3-4), 293–322.

Geman, S., Potter, D., & Chi, Z. (2002). Composition systems. *Quarterly of Applied Mathematics*, 707–736.

Hansen, E., & Zilberstein, S. (2001). LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, *129*, 35–62.

Hart, P., Nilsson, N., & Raphael, B. (1968). A formal basis for the heuristic determination of minimal cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107.

Holte, R., Grajkowski, J., & Tanner, B. (2005). Hierarchical heuristic search revisited. In *Symposium on Abstraction, Reformulation and Approximation*.

Holte, R., Perez, M., Zimmer, R., & MacDonald, A. (1996). Hierarchical A*: Searching abstraction hierarchies efficiently. In *Proceedings of the National Conference on Artificial Intelligence*.

Jimenez, P., & Torras, C. (2000). An efficient algorithm for searching implicit AND/OR graphs with cycles. *Artificial Intelligence*, *124*, 1–30.

Jin, Y., & Geman, S. (2006). Context and hierarchy in a probabilistic image model. In *IEEE Conference on Computer Vision and Pattern Recognition*.

Klein, D., & Manning, C. (2003). A* parsing: Fast exact viterbi parse selection. In *Proceedings of the HLT-NAACL*.

Knuth, D. (1977). A generalization of Dijkstra's algorithm. *Information Processing Letters*, *6*(1), 1–5.

Korf, R. (1997). Finding optimal solutions to Rubik's cube using pattern databases. In *Proceedings of the National Conference on Artificial Intelligence*.

Korf, R., & Felner, A. (2002). Disjoint pattern database heuristics. *Artificial Intelligence*, *134*, 9–22.

Korf, R., Zhang, W., Thayer, I., & Hohwald, H. (2005). Frontier search. *Journal of the ACM*, *52*(5), 715–748.

McAllester, D. (2002). On the complexity analysis of static analyses. *Journal of the ACM*, *49*(4), 512–537.

Montanari, U. (1971). On the optimal detection of curves in noisy pictures. *Communications of the ACM*, *14*(5).

Nilsson, N. (1980). *Principles of Artificial Intelligence*. Morgan Kaufmann.

Pearl, J. (1984). *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley.

Rabiner, L. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, *77*(2), 257–286.

Raphael, C. (2001). Coarse-to-fine dynamic programming. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *23*(12), 1379–1390.

Shashua, A., & Ullman, S. (1988). Structural saliency: The detection of globally salient structures using a locally connected network. In *IEEE International Conference on Computer Vision*.

Tu, Z., Chen, X., Yuille, A., & Zhu, S. (2005). Image parsing: Unifying segmentation, detection, and recognition. *International Journal of Computer Vision*, *63*(2), 113–140.

Williams, L., & Jacobs, D. (1996). Local parallel computation of stochastic completion fields. In *IEEE Conference on Computer Vision and Pattern Recognition*.

Zhang, W., & Korf, R. (1996). A study of complexity transitions on the asymmetric traveling salesman problem. *Artificial Intelligence*, *81*, 1–2.