# Lecture 4: Heuristic Search on Graphs

*10:30 AM, Feb 3, 2009*

## Contents

**1 Overview** **1**

**2 Examples** **3**

**3 Optimality** **5**

**4 Back to Trees** **6**

4.1 PathMax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

4.2 IDA* Search . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

## 1 Overview

A tree is a special case of a graph in which there is exactly one path to every node. Indeed an arbitrary graph can contain multiple paths to any given node. To extend heuristic search algorithms designed for trees to graphs, we maintain a *closed* list of nodes already visited, in addition to the open list of nodes yet to be visited. In best-$h$ search, nodes are inserted into the open list as they are encountered, if they are not yet on either the open or the closed lists. In addition, in best-$g$ search, nodes on the open list are promoted if they are re-encountered with lower priority. In addition, in A$^*$ search, nodes on the closed list are re-opened if they are re-encountered with lower priority.

**Best-$h$ Search on Graphs** It is straightforward to extend best-$h$ search to graphs: we maintain a closed list and do not open any nodes that already appear on either the open list or the closed list. Since the heuristic $h$ is a function (*i.e.*, each node's value is unique), nodes are never re-encountered with lower priority. Thus, open nodes need not ever be promoted and closed nodes need not be re-inserted into the open list. The best-$h$ search algorithm for graphs is depicted in Table 1.

**Best-$g$ Search on Graphs** The best-$g$ search algorithm is outlined in Table 2. Like best-$h$ search, best-$g$ search maintains a closed list and never re-opens any nodes on this list: since nodes are visited in priority order, closed nodes can never be re-encountered with lower priority. But open nodes could be re-encountered with lower priority, since there can be multiple paths to a single node in a graph. Best-$g$ search *promotes* nodes on the open list if ever they are re-encountered with lower priority.

**A$^*$ Search on Graphs** Table 3 describes A$^*$ search through arbitrary graphs. Like best-$g$ search, A$^*$ search on graphs promotes nodes on the open list whenever such nodes are re-encountered with lower priority. In addition, A$^*$ search on graphs maintains a closed list and re-opens nodes on this

BEST-$h(X, S, G, \delta, c, h)$

| | |
|---|---|
| Inputs | search problem |
| | heuristic function $h$ |
| Output | (path to) optimal goal node |
| Initialize | $O = S$ is the list of open nodes |
| | $C = \emptyset$ is the list of closed nodes |

`while` ($O$ is not empty) `do`

1. delete node $n \in O$ *s.t.* $h(n)$ is minimal
2. if $n \in G$, return (path to) $n$
3. for all $m \in \delta(n)$
   (a) compute $h'(m)$
   (b) if $m \notin C$ and $m \notin O$
      i. insert $m$ into $O$ with priority $h(m) = h'(m)$
4. insert $n$ into $C$

`fail`

Table 1: Best-$h$ Search on Graphs.

BEST-$g(X, S, G, \delta, c)$

| | |
|---|---|
| Inputs | search problem |
| Output | (path to) optimal goal node |
| Initialize | $O = S$ is the list of open nodes |
| | $C = \emptyset$ is the list of closed nodes |

`while` ($O$ is not empty) `do`

1. delete node $n \in O$ *s.t.* $g(n)$ is minimal
2. if $n \in G$, return (path to) $n$
3. for all $m \in \delta(n)$
   (a) $g'(m) = g(n) + c(m, n)$
   (b) if $m \notin C$ and $m \notin O$
      i. insert $m$ into $O$ with priority $g(m) = g'(m)$
   else if $m \in O$ and $g'(m) < g(m)$
      i. promote $m$ in $O$ to priority $g(m) = g'(m)$
4. insert $n$ into $C$

`fail`

Table 2: Best-$g$ Search on Graphs.

list if ever they are are re-encountered with lower priority. How can closed nodes be re-encountered with lower priority? Recall that A* search is guided by the evaluation function $f = g + h$: *i.e.*, nodes are visited in *roughly* $f$-order. Unlike in best-$g$ search, where nodes are visited in $g$-order, a closed node in A* search can be re-encountered with lower a $g$-cost, and thus a lower $f$-priority than its $f-$priority during any previous encounters.

```
A*(X, S, G, δ, c, h)
  Inputs      search problem
              heuristic function h
  Output      (path to) optimal goal node
  Initialize  O = S is the list of open nodes
              C = ∅ is the list of closed nodes

while (O is not empty) do
  1. delete node n ∈ O s.t. f(n) is minimal
  2. if n ∈ G, return (path to) n
  3. for all m ∈ δ(n)
     (a) compute h'(m)
     (b) g'(m) = g(n) + c(m, n)
     (c) f'(m) = g'(m) + h'(m)
     (d) if m ∉ C and m ∉ O
               insert m into O with priority f(m) = f'(m)
         else if m ∈ O and f'(m) < f(m)
               promote m in O to priority f(m) = f'(m)
         else if m ∈ C and f'(m) < f(m)
               re-open m in O with priority f(m) = f'(m)
  4. insert n into C with priority f(n)
fail
```

Table 3: A* Search on Graphs. Note that $f'(m) < f(m)$ iff $g'(m) < g(m)$, since $h(m)$ is constant, for all nodes $n$.

## 2 Examples

A sample graph appears in Figure 1. The start state is $S$ and the unique goal node is $G$, but there are two paths to this goal node: $SACG$ and $SBCG$.

**Best-$h$ Search** Best-$h$ search maintains the following sequence of priority queues in its search through this graph: $S_0$, $A_1B_2$, $C_0B_2$, $G_0B_2$, GOAL! The path returned is suboptimal: $SACG$.

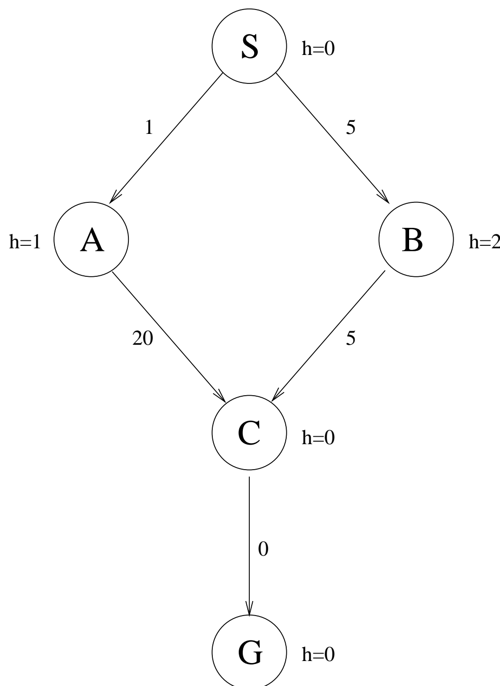


Figure 1: Sample DAG: edges are labeled with costs $g$; nodes are labeled with heuristic values $h$. The start state is $S$ and the unique goal node is $G$. The optimal path to this goal is $SBCG$. Only best-$g$ search and A$^*$ search return this optimal path; best-$h$ search returns the suboptimal path $SACG$.

**Best-$g$ Search** Best-$g$ search maintains the following sequence of priority queues in its search through this graph: $S_0$, $A_1B_5$, $B_5C_{21}$, $C_{10}$, $G_{10}$, GOAL! The path returned is optimal: $SBCG$.

Best-$g$ search first encounters node $C$ via node $A$, traversing a path of cost 21. Later, it encounters node $C$ again via node $B$, traversing a path of cost only 10. At this point, node $C$'s priority is promoted from 21 to 10.

**A$^*$ Search** A$^*$ search maintains the following sequence of priority queues in its search through the graph in Figure 1: $S_0$, $A_2B_7$, $B_7C_{21}$, $C_{10}$, $G_{10}$, GOAL! The path to the goal that is returned is optimal: $SBCG$.

Figure 2 (LHS):

- Best-$g$: $S_0$, $D_1A_1$, $A_1G_{11}$, $B_2G_{11}$, $C_3G_{11}$, $G_{11}$, GOAL! (After $C_3$ is popped, $G_{13}$ is not pushed, since $G_{11}$ is already on the queue.) The optimal path $SDG$ is returned.
- Best-$h$: $S_0$, $A_3D_{10}$, $B_2D_{10}$, $C_1D_{10}$, $G_0D_{10}$, GOAL! The path returned, namely $SABCG$, is not optimal.
- A$^*$: $S_0$, $A_4D_{11}$, $B_4D_{11}$, $C_4D_{11}$, $D_{11}G_{13}$, $G_{11}$, GOAL! (After $D_{11}$ is popped, the node $G_{13}$ is promoted to $G_{11}$.) The optimal path $SDG$ is returned.

Figure 2 (RHS):

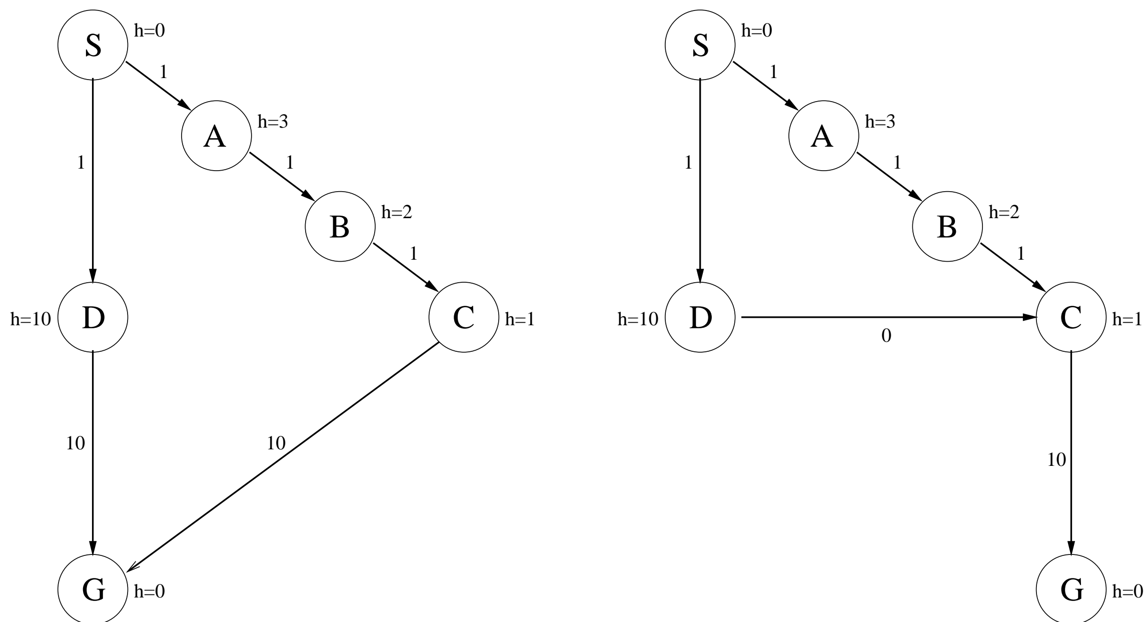


Figure 2: Sample DAGs: edges are labeled with costs $g$; nodes are labeled with heuristic values $h$. The start state is $S$ and the unique goal is $G$. (LHS) The optimal path to the goal is $SDG$. (RHS) The optimal path to the goal is $SDCG$.

- Best-$g$: $S_0$, $D_1A_1$, $C_1A_1$, $A_1G_{11}$, $B_2G_{11}$, $C_3G_{11}$, $G_{11}$, GOAL! (After $C_3$ is popped, $G_{13}$ is not pushed, since $G_{11}$ is already on the queue.) The optimal path $SDCG$ is returned.
- Best-$h$: $S_0$, $A_3D_{10}$, $B_2D_{10}$, $C_1D_{10}$, $G_0D_{10}$, GOAL! The path returned, namely $SABCG$, is not optimal.
- A*: $S_0$, $A_4D_{11}$, $B_4D_{11}$, $C_4D_{11}$, $D_{11}G_{13}$, $C_2G_{13}$, $G_{11}$, GOAL! (After $C$ with cost 4 is closed, it is re-opened with cost 2. And after $C_2$ is popped, the node $G_{13}$ is promoted to $G_{11}$.) The optimal path $SDCG$ is returned.

## 3 Optimality

The proof of optimality of best-$g$ search relies on the fact that nodes are visited in $g$-order (that is, in order of nondecreasing $g$-costs). Hence, the first goal node that best-$g$ reaches is necessarily one of minimal $g$-cost. A* search, on the other hand, need not visit nodes in $f$-order (that is, in order of nondecreasing $f$-costs). (See Figure 3.)

Thus, the first goal node that A* reaches is not obviously one of minimal $f$-cost. A property called consistency, however, ensures that $f$ is a monotonically, nondecreasing function of depth: i.e., $f(m) \geq f(n)$, for all $m \in \delta(n)$, so that A* search does visit nodes in $f$-order. Consequently, assuming consistency, the first goal node that is reached is one of minimal $f$-cost.

A heuristic function $h$ is called *consistent* iff (i) $h(n) \leq c(n,m)+h(m)$, for all nodes $n$ with successor nodes $m \in \delta(n)$; and (ii) $h(n^*) = 0$, for all goal nodes $n^*$. The first part of this definition can be

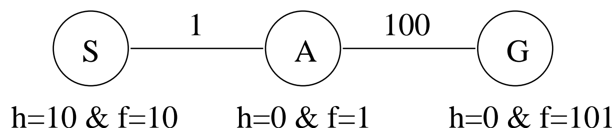


Figure 3: Simple search tree: $S$ is the start node; $A$ is an intermediate node; $G$ is the goal node. Edges are labeled with costs $c$. Nodes are labelled with $h$-costs and $f$-costs. Note that $f$ is *not* monotonically nondecreasing in depth, but that A* search visits nodes in order of depth. Thus, A* search does not visit nodes in $f$-order.

understood as a form of the **triangle inequality**, which stipulates that the length of a side of a triangle cannot exceed the sum of the lengths of the two other sides.

**Lemma:** If $h$ is consistent, then $f$ is monotonically, nondecreasing in depth: *i.e.*, for all $n \in X$, for all $m \in \delta(n)$, $f(m) \geq f(n)$.

**Proof:** Note that $g$ (and therefore $f$) is not a function. For each value of $g(n)$,

$$\begin{aligned} f(n) &= g(n) + h(n) \\ &= g(m) - c(n,m) + h(n) \\ &\leq g(m) + h(m) \\ &= f(m) \end{aligned}$$

$\diamond$

This lemma implies that A* search with a consistent heuristic is optimal: since nodes are visited in nondecreasing order of their $f$-costs, the first goal node visited will be one of minimal $f$-cost. Since the $h$-value at goal nodes is 0, this node will also be a goal of minimal $g$-cost.

**Theorem:** A* search (on trees and graphs) with a consistent heuristic is optimal.

# 4 Back to Trees

Consistency implies admissibility, but the converse is not true. Still, most admissible heuristics that are useful in practice happen to also be consistent. That said, if you ever find yourself trying to solve a search problem on a *tree* and your heuristic is not consistent, it can be made consistent using the pathmax optimization routine. Doing so would guarantee monotonicity, so that A* would visit nodes in nondecreasing order according to their $f$-costs.

## 4.1 PathMax

The **pathmax** equation is an optimization routine applicable to heuristic search through a tree in which the heuristic $h$ is inductively updated to yield a new heuristic $\hat{h}$: beginning at the root node $n$, $\hat{h}(n) = h(n)$; then, for all successor nodes $m \in \delta(n)$, $\hat{h}(m) = \max\{\hat{h}(n) - c(n,m), h(m)\}$. Equivalently, by adding $g(m)$ to both $\hat{h}(n) - c(n,m)$ and $h(m)$, $\hat{f}(m) = \max\{\hat{f}(n), f(m)\}$. (Check: $\hat{h}(n) - c(n,m) + g(m) = \hat{h}(n) + g(n) = \hat{f}(n)$.) For example, if $g(n) = 1$ and $h(n) = 10$, while $g(m) = 3$ and $h(m) = 3$ for some child $m$ of $n$, then $n$'s $f$-cost 11, while $m$'s $f$-cost is 6. The

| | |
|---|---|
| $A^*(X, S, G, \delta, c, h)$ | |
| Inputs | search problem |
| | heuristic function $h$ |
| Output | (path to) optimal goal node |
| Initialize | $O = S$ is the list of open nodes |
| | $C = \emptyset$ is the list of closed nodes |

```
while (O is not empty) do
  1. delete node n ∈ O s.t. f(n) is minimal
  2. if n ∈ G, return (path to) n
  3. for all m ∈ δ(n)
     (a) compute h'(m)
     (b) g'(m) = g(n) + c(m, n)
     (c) f'(m) = g'(m) + h'(m)
     (d) if m ∉ C and m ∉ O
             insert m into O with priority f(m) = f'(m)
         else if m ∈ O and f'(m) < f(m)
             promote m in O to priority f(m) = f'(m)
  4. insert n into C
fail
```

Table 4: $A^*$ Search on Graphs with a Consistent Heuristic.

pathmax equation increases $m$'s $f$-cost to 11 because $m$ is on the same path to a goal as $n$, so $m$'s costs cannot be any less than $n$'s.

**Lemma:** Applying the pathmax operation beginning at the root of a tree and working down towards the leaves yields a consistent heuristic: i.e., for all $n, m \in \delta(n)$, $\hat{h}(n) \leq \hat{h}(m) + c(n, m)$.

**Proof:** Two cases arises. Either $\hat{h}(m) = h(m) \geq \hat{h}(n) - c(n, m)$, or $\hat{h}(m) = \hat{h}(n) - c(n, m)$. $\diamond$

**Corollary:** Pathmax implies monotonicity (i.e., $A^*$ visits nodes in nondecreasing order according to their $f$-costs).

**Lemma:** The pathmax operation preserves admissibility: *i.e.*, if $h$ is admissible, then $\hat{h}(m) = \max\{h(m), h(n) - c(n, m)\}$, for all $n$ and for all $m \in \delta(n)$ is also admissible.

**Proof:** Fix an arbitrary $n \in X$. To show $\hat{h}(m) \leq h^*(m)$, for all $m \in \delta(n)$, it suffices to show: (i) $h(m) \leq h^*(m)$, and (ii) $h(n) - c(n, m) \leq h^*(m)$. Condition (i) follows immediately from the admissibility of $h$. Condition (ii) breaks down into two cases: $h(n) - c(n, m) \leq h^*(n) - c(n, m) = h^*(m)$, if $m$ is on a path from $n$ to its nearest goal; otherwise, $h(n) - c(n, m) \leq h^*(n) - c(n, m) < h^*(m)$. Therefore, $h(n) - c(n, m) \leq h^*(m)$.

**Theorem:** $A^*$ search with pathmax is optimal.

## 4.2 IDA$^*$ Search

Iterative deepening A$^*$ (IDA$^*$) is a search algorithm with the performance properties of $A^*$—it is complete and optimal—and the space requirements of DFS—(essentially) linear in depth. The main idea of iterative deepening A$^*$ (or iterative deepening Best-$g$ or Best-$h$) is to repeatedly search in depth-first fashion, over subgraphs with $f$-cost (or $g$-cost or $h$-cost) less than $\alpha$, less than $2\alpha$, less than $3\alpha$, and so on, until a goal is found, where $\alpha$ is a lower bound on the cost between nodes and their successors: *i.e.*, $0 < \alpha \leq c(n, m)$, for all $n, m \in \delta(n)$. IDA$^*$ is usually implemented with pathmax, so that the algorithm visits nodes in contours of greater and greater $f$-costs.

Recall that the space complexity of ID is $O(bd)$, where $d$ is the depth of the goal node. Similarly, the space complexity of IDA$^*$ is $O(bg^*/\alpha)$, where $g^*$ is the optimal cost. The time complexity of IDA$^*$, however, can exceed that of A$^*$. In particular, in search spaces where the $f$-cost is different at every state, only one additional state is expanded during each iteration. In such a search space, if A$^*$ expands $N$ nodes, IDA$^*$ expands $1 + \ldots + N = O(N^2)$ nodes. The typical solution to this problem is to fix an increment $\beta > \alpha$ such that several nodes $n$ have cost $f_i < f(n) \leq f_i + \beta$, where $f_i$ is the $i$th incremental value of the $f$-cost. This strategy reduces search time, since the total number of iterations is proportional to $1/\beta < 1/\alpha$, and returns solutions that are at worst $\beta$-optimal: *i.e.*, if the algorithm returns $m^*$, then $g(m^*) < g^* + \beta$.

| IDA$^*(X, S, G, \beta, c, h)$ | |
|---|---|
| Inputs | search problem |
| | admissible heuristic $h$ |
| Output | (path to) optimal goal node |
| Initialize | $\beta$ is the increment |
| | $i = 0$ is the cut-off $f$-value |
| | $O = S$ is the priority queue of open nodes |

`while` (1) `do`

1. `while` ($O$ is not empty) `do`
   - (a) delete *first* node $n \in O$
   - (b) if $n \in G$, return (path to) goal $n$
   - (c) for all $m \in \beta(n)$
     - i. compute $h(m)$
     - ii. $g(m) = g(n) + c(m, n)$
     - iii. $f(m) = g(m) + h(m)$
     - iv. if $f(m) \leq i$, insert $m$ in *front* of $O$
2. increment $i$ by $\beta$, $O = S$

Table 5: Iterative Deepening A$^*$.