Heuristic Search: Part II

CMPUT 366: Intelligent Systems

P&M §3.6

Recap: Heuristics

Definition:
A **heuristic function** is a function $h(n)$ that returns a non-negative estimate of the cost of the cheapest path from n to a goal node.

- e.g., Euclidean distance instead of travelled distance

Definition:
A heuristic function is **admissible** if $h(n)$ is always less than or equal to the cost of the cheapest path from n to a goal node.

- i.e., $h(n)$ is a **lower bound** on cost($\langle n, \ldots, g \rangle$) for any **goal node** g

Recap: A* Search

- A* search uses **both** path cost information and heuristic information to select paths from the frontier
- Let $f(p) = \text{cost}(p) + h(p)$
  - $f(p)$ **estimates** the total cost to the nearest goal node **starting from p**
- A* removes paths from the frontier with **smallest** $f(p)$
- When h is **admissible**,
  $p^* = \langle s, \ldots, n, \ldots, g \rangle$ is a **solution**, and
  $p = \langle s, \ldots, n \rangle$ is a **prefix** of p^*:
  - $f(p) \leq \text{cost}(p^*)$

$$\underbrace{\underbrace{\text{start} \xrightarrow{\text{actual}} n}_{\text{cost(p)}} \; \underbrace{\xrightarrow{\text{estimated}} \text{goal}}_{\text{h(n)}}}_{\text{f(p)}}$$

Recap: A* Search Algorithm

Input: a *graph*; a set of *start nodes*; a *goal* function

i.e., $f(\langle n_1, n_2, \ldots, n_k \rangle) \leq f(p)$
for all other paths $p \in$ *frontier*

frontier := { <*s*> | *s* is a start node}
while *frontier* is not empty:
    select heuristic minimizing path $\langle n_1, n_2, \ldots, n_k \rangle$ from *frontier*
    remove $\langle n_1, n_2, \ldots, n_k \rangle$ from *frontier*
    if *goal*(n_k):
        return $\langle n_1, n_2, \ldots, n_k \rangle$
    for each neighbour *n* of n_k:
        add $\langle n_1, n_2, \ldots, n_k, n \rangle$ to *frontier*
end while

A* Theorem

Theorem:

If there is a solution, A* using heuristic function h always returns an **optimal** solution (in **finite time**), if

1. The branching factor is **finite**,
2. All **arc costs** are greater than some $\epsilon > 0$, and
3. h is an **admissible** heuristic.

Proof:

1. The **optimal solution** is guaranteed to be **removed from the frontier** eventually
2. **No suboptimal solution** will be removed from the frontier whenever the frontier contains a **prefix of the optimal solution**

A* Theorem Proofs: A Lexicon

An **admissible heuristic**: $h(n)$

$f(\langle n_1, \ldots, n_k \rangle) = \text{cost}(\langle n_1, \ldots, n_k \rangle) + h(n_k)$

A **start node**: s

A **goal node**: z (i.e., $\text{goal}(z) = 1$)

The **optimal solution**: $p^* = \langle s, \ldots, a, b, \ldots z \rangle$

A **prefix** of the optimal solution: $p' = \langle s, \ldots, a \rangle$

A **suboptimal solution**: $g = \langle s, q, \ldots, z \rangle$

A* Theorem: Optimality

An **admissible heuristic**: $h(n)$
$f(\langle n_1, \ldots, n_k \rangle) = \text{cost}(\langle n_1, \ldots, n_k \rangle) + h(n_k)$
A **start node**: s
A **goal node**: z (i.e., goal$(z) = 1$)
The **optimal solution**: $p* = \langle s, \ldots, a, b, \ldots z \rangle$
A **prefix** of the optimal solution: $p' = \langle s, \ldots, a \rangle$
A **suboptimal solution**: $g = \langle s, q, \ldots, z \rangle$

Proof part 2: Optimality (no g is removed before $p*$)

1. $f(g) = \text{cost}(g)$ and $f(p*) = \text{cost}(p*)$

   (i) $f(\langle n_1, \ldots, n_k \rangle) = \text{cost}(\langle n_1, \ldots, n_k \rangle) + h(n_k)$, and $h(z) = 0$

2. $f(p') \leq f(g)$

   (i) $f(\langle s, \ldots, a \rangle) = \text{cost}(\langle s, \ldots, a \rangle) + h(a)$

   (ii) $f(\langle s, \ldots, a, b, \ldots, z \rangle) = \text{cost}(\langle s, \ldots, a, b, \ldots, z \rangle) + h(z) = \text{cost}(\langle s, \ldots, a \rangle) + \text{cost}(a, b, \ldots, z \rangle)$

   (iii) $h(a) \leq \text{cost}(\langle a, b, \ldots, z \rangle)$

   (iv) $f(p') \leq f(p*) < f(g)$ ∎

Comparing Heuristics

- Suppose that we have two **admissible** heuristics, h_1 and h_2
- Suppose that for every node n, $h_2(n) \geq h_1(n)$

Question: Which heuristic is better for search?

Dominating Heuristics

Definition:
A heuristic h_2 **dominates** a heuristic h_1 if

1. $\forall n : h_2(n) \geq h_1(n)$, and

2. $\exists n : h_2(n) > h_1(n)$.

Theorem:
If h_2 dominates h_1, and both heuristics are admissible, then A* using h_2 will never remove more paths from the frontier than A* using h_1.

Question:
Which admissible heuristic dominates **all other** admissible heuristics?

A* Analysis

For a search graph with *finite* maximum branch factor *b* and *finite* maximum path length *m*...

1. What is the worst-case **space complexity** of A*?
   [A: $O(m)$] [B: $O(mb)$] [C: $O(b^m)$] [D: it depends]

2. What is the worst-case **time complexity** of A*?
   [A: $O(m)$] [B: $O(mb)$] [C: $O(b^m)$] [D: it depends]

Question: If A* has the same space and time complexity as least cost first search, then what is its advantage?

Summary

- **Domain knowledge** can help speed up graph search
- Domain knowledge can be expressed by a **heuristic function**, which **estimates** the cost of a path to the goal from a node
- A* considers both **path cost** and **heuristic cost** when selecting paths: $f(p) = \text{cost}(p) + h(p)$
- **Admissible** heuristics guarantee that A* will be **optimal**
- Admissible heuristics can be built from **relaxations** of the original problem
- The more **accurate** the heuristic is, the **fewer** the paths A* will explore

Branch & Bound

or, How I Learned to Stop Worrying and Love Depth First Search

CMPUT 366: Intelligent Systems

P&M §3.7-3.8

Lecture Outline

1. Recap / Heuristic Search Part II
2. Cycle Pruning
3. Branch & Bound
4. Exploiting Search Direction

Cycle Pruning

- Even on **finite graphs**, depth-first search may not be complete, because it can get trapped in a **cycle**.
- A search algorithm can **prune** any path that ends in a node already on the path **without missing an optimal solution** (**Why?**)

Questions:

1. Is depth-first search on with cycle pruning **complete** for finite graphs?
2. What is the **time complexity** for cycle checking in **depth-first search**?
3. What is the **time complexity** for cycle checking in **breadth-first search**?

Cycle Pruning
Depth First Search

Input: a *graph*; a set of *start nodes*; a *goal* function

frontier := { $<s>$ | s is a start node}
while *frontier* is not empty:
    select the newest **path** $<n_1, n_2, ..., n_k>$ **from** *frontier*
    remove $<n_1, n_2, ..., n_k>$ from *frontier*
    if $n_k \neq n_j$ for all $1 \leq j < k$:
        if *goal*(n_k):
            return $<n_1, n_2, ..., n_k>$
        for each neighbour n of n_k:
            add $<n_1, n_2, ..., n_k, n>$ to *frontier*
end while

Heuristic Depth First Search

	Heuristic Depth First	A*	Branch & Bound
Space complexity	$O(mb)$	$O(b^m)$	$O(mb)$
Heuristic Usage	Limited	Optimal	Optimal (if bound low enough)
Optimal?	No	Yes	Yes (if bound high enough)

Branch & Bound

- The *f*(*p*) function provides a **path-specific lower bound** on solution cost starting from *p*
- **Idea:** Maintain a **global upper bound** on solution cost also
  - Then prune any path whose lower bound **exceeds** the upper bound
- **Question:** Where does the upper bound come from?
  - **Cheapest** solution found so far
  - Before solutions found, specified on entry
  - Can increase the global upper bound **iteratively** (as in iterative deepening search)

Branch & Bound Algorithm

Input: a *graph*; a set of *start nodes*; a *goal* function; heuristic $h(n)$; $bound_0$

frontier := { <*s*> | *s* is a start node}
bound := $bound_0$
best := ∅
while *frontier* is not empty:
    select the newest path $\langle n_1, n_2, \ldots, n_k \rangle$ from *frontier*
    remove $\langle n_1, n_2, \ldots, n_k \rangle$ from *frontier*
    if $\text{cost}(\langle n_1, n_2, \ldots, n_k \rangle) + h(n_k) \leq bound$:
        if $goal(n_k)$:
            $bound := \text{cost}(\langle n_1, n_2, \ldots, n_k \rangle)$
            $best := \langle n_1, n_2, \ldots, n_k \rangle$
        else:
            for each neighbour *n* of n_k:
                add $\langle n_1, n_2, \ldots, n_k, n \rangle$ to *frontier*
end while
return *best*

Branch & Bound Analysis

- If *bound*$_0$ is set to just above the optimal cost, branch & bound will explore no more paths than A*
  (**Why?**)

- With **iterative increasing** of *bound*$_0$, will re-explore some lower-cost paths, but still similar time-complexity to A*
  Question: How *much* should the bound get increased by?

  - Iteratively increase bound to the **lowest-f-value** node that was **pruned**

  - Worse than A* by no more than a **linear** factor of *m*,
    by the same argument as for iterative deepening search

Exploiting Search Direction

- When we care about finding the path to a known goal node, we can search forward, but we can often search **backward**

- Given a search graph $G=(N,A)$, **known** goal node g, and set of start nodes S, can construct a **reverse search problem** $G=(N, A^r)$:

    1. Designate g as the start node
    2. $A^r = \{ \langle n_2,n_1 \rangle \mid \langle n_1,n_2 \rangle \in A \}$
    3. $\text{goal}^r(n) = \text{True if } n \in S$
       (i.e., if n is a start node of the original problem)

Questions:

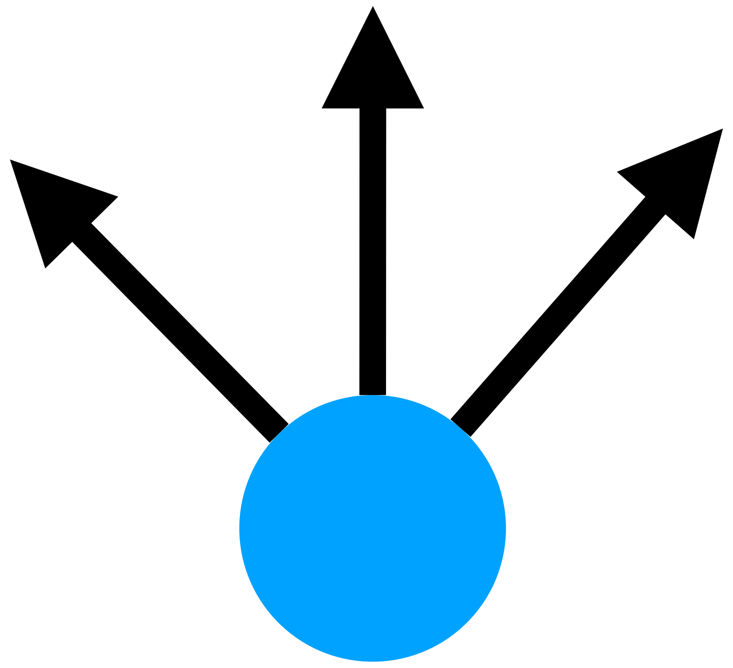
1. When is this **useful**?
2. When is this **infeasible**?

Reverse Search

Definitions:

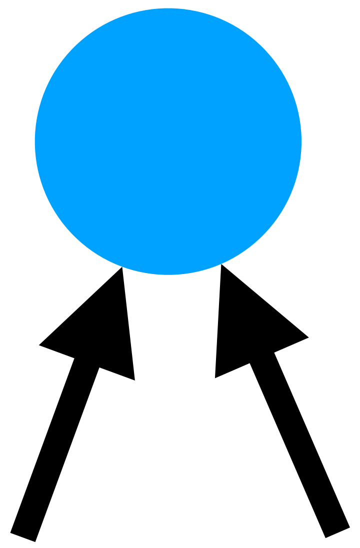
1. **Forward branch factor**: Maximum number of **outgoing** neighbours
   Notation: *b*

   - Time complexity of forward search: $O(b^m)$



2. **Reverse branch factor**: Maximum number of **incoming** neighbours
   Notation: r

   - Time complexity of reverse search: $O(r^m)$



When the reverse branch factor is **smaller** than the forward branch factor, reverse search is more **time-efficient**.

Bidirectional Search

- **Idea:** Search backward from from goal and forward from start **simultaneously**
- Time complexity is **exponential in path length**, so exploring half the path length is an exponential improvement
  - Even though must explore half the path length **twice**
- Main problems:
  - **Ensuring** that the frontiers meet
  - **Checking** that the frontiers have met

Questions:

What bidirectional **combinations** of search algorithm make sense?

- Breadth first + Breadth first?
- Depth first + Depth first?
- Breadth first + Depth first?

Summary

- **Cycle pruning** can guarantee the **completeness** of depth-first search on **finite** graphs
  - Although depth first search is really most useful on very large or **infinite** graphs...
- **Branch & bound** combines the **optimality** guarantee and **heuristic efficiency** of A* with the space efficiency of depth-first search
- Tweaking the **direction of search** can yield efficiency gains