Uninformed Search Part 2
&
Heuristic Search

CMPUT 366: Intelligent Systems

P&M §3.6

Logistics

- TA office hours begin this week
  - See eClass page for times and meeting links
- Assignment #1 released next week

Lecture Outline

1. Logistics
2. Iterative Deepening Search
3. Least Cost First Search
4. Heuristics
5. A* Search
6. Comparing Heuristics

Recap:
Iterative Deepening Search

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

for *max_depth* from 1 to ∞:
    Perform **depth-first search** to a maximum depth *max_depth*
end for

Iterative Deepening Search

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

```
more_nodes := True
while more_nodes:
    frontier := { <s> | s is a start node}
    for max_depth from 1 to ∞:
        more_nodes := False
        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 goal(n_k):
                return <n_1, n_2, ..., n_k>
            if k < max_depth:
                for each neighbour n of n_k:
                    add <n_1, n_2, ..., n_k, n> to frontier
            else if n_k has neighbours:
                more_nodes := True
```

Iterative Deepening Search Analysis

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

1. What is the worst-case **time complexity**?
    - [A: $O(m)$] [B: $O(mb)$] [C: $O(b^m)$] [D: it depends]
2. When is iterative deepening search **complete**?
3. What is the worst-case **space complexity**?
    - [A: $O(m)$] [B: $O(mb)$] [C: $O(b^m)$] [D: it depends]

Bonus: Time Complexity of Iterated Deepening Search

- Breadth-first search requires $O(b^m)$ time, because in the worst case it visits **every path once**
- Iterative deepening search is **worse**, because it visits every path at least once, and many paths multiple times.
  But **how much** worse?

Claim: Iterated deepening search has time complexity no worse than $O(mb^m)$ (i.e., m **times worse** than breadth first search)

1. Paths of length 1 are visited *m* times; paths of length 2 are visited *m*-1 times; ... ; paths of length *m* are visited 1 time.

2. In other words, every path is visited m **times or fewer**

Note: This is a very **loose bound**. See the text for a much tighter bound.

When to Use
Iterative Deepening Search

- When is iterative deepening search **appropriate**?
  - Memory is limited, *and*
  - Both deep and shallow solutions may exist
    - or we prefer shallow ones
  - Tree may contain infinite paths

Optimality

Definition:
An algorithm is **optimal** if it is guaranteed to return an optimal (i.e., **minimal-cost**) solution **first**.

Question: Which of the three algorithms presented so far is optimal? Why?

Least Cost First Search

- *None* of the algorithms described so far is guided by **arc costs**
  - BFS and IDS are implicitly guided by **path length**, which can be the same for uniform-cost arcs
- They return a path to a goal node as soon as they happen to blunder across one, but it may not be the optimal one
- **Least Cost First Search** is a search strategy that is **guided by arc costs**

Least Cost First Search

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

i.e., cost(<n_1, n_2, ..., n_k>) ≤ cost(p) for all other paths $p \in$ *frontier*

frontier := { <s> | s is a start node}
while *frontier* is not empty:
    select the cheapest path <n_1, n_2, ..., n_k> from *frontier*
    remove <n_1, n_2, ..., n_k> from *frontier*
    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

Question:

What **data structure** for the frontier implements this search strategy?

Least Cost First Search Analysis

- Least Cost First Search is **complete** and **optimal** if there is $\varepsilon > 0$ with cost($<n_1,n_2>$) > ε for every arc $<n_1,n_2>$:

    1. Suppose $<n_1,n_2,\ldots,n_k>$ is the optimal solution
    2. Suppose that p is any non-optimal solution
       So, cost(p) > $<n_1,n_2,\ldots,n_k>$
    3. For every $1 \leq \ell \leq k$, cost($<n_1,n_2,\ldots,n_\ell>$) < cost(p)
    4. So p will never be removed from the frontier before $<n_1,n_2,\ldots,n_k>$

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

- When does Least Cost First Search have to expand **every node** of the graph?

Uninformed Search Summary

- Different **search strategies** have different properties and behaviour
  - **Depth first search** is space-efficient but not always complete or time-efficient
  - **Breadth first search** is complete and always finds the shortest path to a goal, but is not space-efficient
  - **Iterative deepening search** can provide the benefits of both, at the expense of some time-efficiency
  - All three strategies must potentially expand **every node**, and are not guaranteed to return an **optimal solution**
- **Least cost first** is essentially breadth-first search with an optimality guarantee

Recap: Search Strategies

	Depth First	Breadth First	Iterative Deepening	Least Cost First
Selection	Newest	Oldest	Newest, multiple	Cheapest
Data structure	Stack	Queue	Stack, counter	Priority queue
Complete?	Finite graphs only	Complete	Complete	Complete if cost(p) > ε
Space complexity	$O(mb)$	$O(b^m)$	$O(mb)$	$O(b^m)$
Time complexity	$O(b^m)$	$O(b^m)$	$O(mb^m)$ **	$O(b^m)$
Optimal?	No	No	No	Optimal

Domain Knowledge

- Domain-specific knowledge can help speed up search by identifying **promising directions** to explore

- We will encode this knowledge in a function called a **heuristic function** which **estimates** the cost to get from a node to a goal node

- The search algorithms in this lecture take account of this heuristic knowledge when **selecting** a path from the frontier

Heuristic Function

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.

- For paths: $h(\langle n_1, n_2, \ldots, n_k \rangle) = h(n_k)$
- Uses only **readily-available** information about a node (i.e., easy to compute)
- **Problem-specific**

Admissible Heuristic

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 any goal node.

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

Example Heuristics

- **Euclidean distance** for DeliveryBot
  (ignores that it can't go through walls)

- **Number of dirty rooms** for VacuumBot
  (ignores the need to move between rooms)

- **Points** for chess pieces
  (ignores positional strength)

Constructing Admissible Heuristics

- Search problems try to find a cost-minimizing path, subject to **constraints** encoded in the search graph

- How to construct an easier problem? **Drop** some constraints.

  - This is called a **relaxation** of the original problem

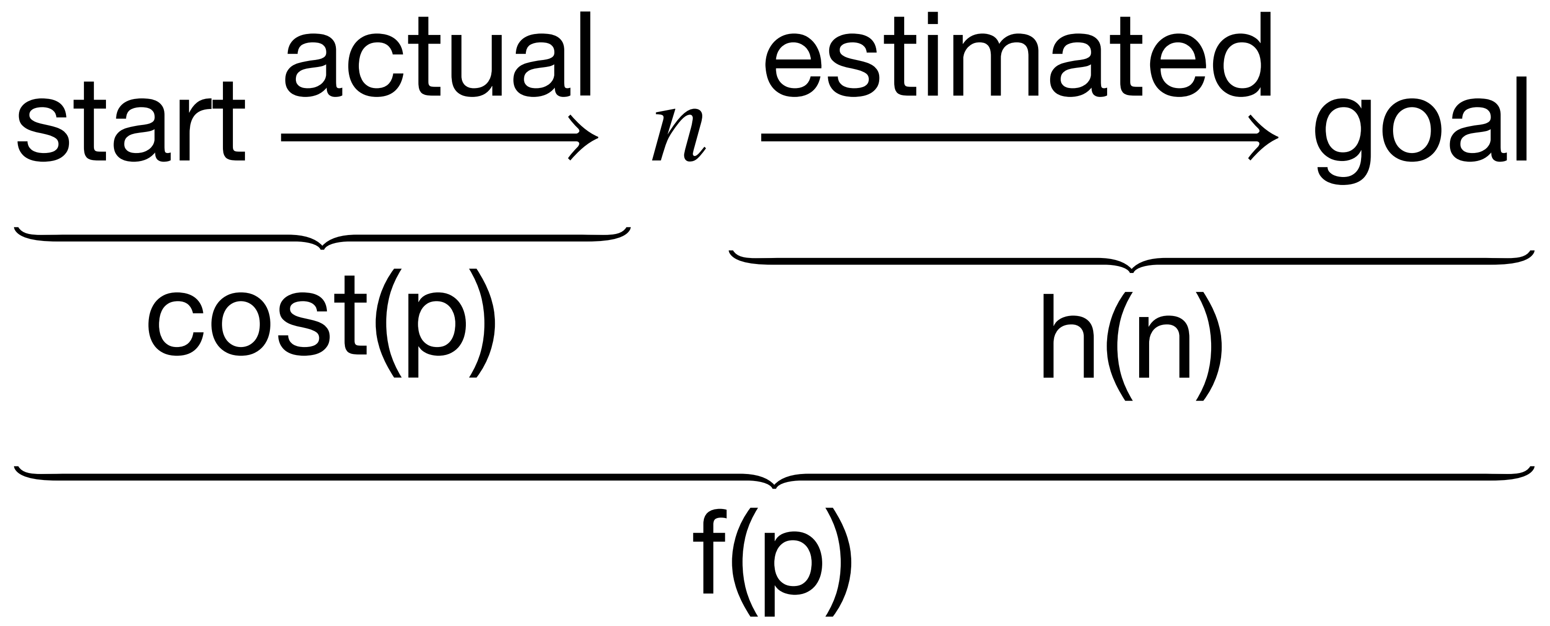
- The cost of the optimal solution to the relaxation will always be an **admissible heuristic** for the original problem (**Why?)**

- **Neat trick**: If you have two admissible heuristics h_1 and h_2, then $h_3(n) = \max\{h_1(n), h_2(n)\}$ is admissible too! (**Why?**)

Simple Uses of Heuristics

- **Heuristic depth first search:** Add neighbours to the fringe in **decreasing order** of their heuristic values, then run depth first search as usual
  - Will explore most promising successors first, but
  - Still explores **all paths** through a successor before considering other successors
  - Not complete, not optimal
- **Greedy best first search:** Select path from the frontier with the **lowest heuristic** value
  - Not guaranteed to work any better than breadth first search (**why?**)

A* Search



- A* search uses **both** path cost information and heuristic information to select paths from the frontier
- Let $f(p) = \text{cost}(p) + h(p)$
- 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^*:
  - $f(p) \leq \text{cost}(p^*)$
  - **Why?**

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*

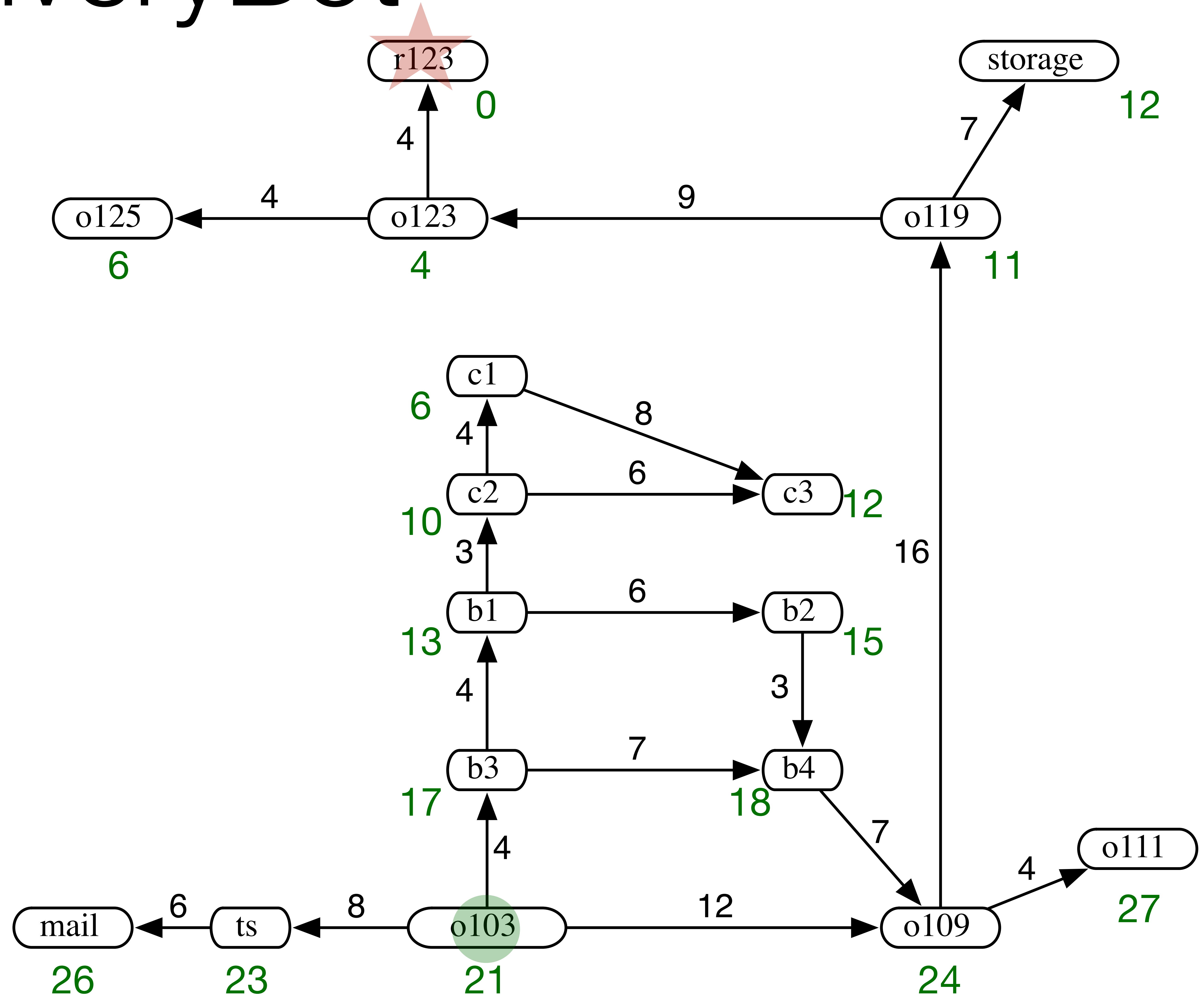
frontier := $\{ \langle s \rangle \mid s$ is a start node$\}$
while *frontier* is not empty:
    select heuristic minimizing path $\langle n_1, n_2, \ldots, n_k \rangle$ from *frontier*
    remove $\langle n_1, n_2, \ldots, n_k \rangle$ from *frontier*
    if $goal(n_k)$:
        return $\langle n_1,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

Question:

What **data structure** for the frontier implements this search strategy?

A* Search Example: DeliveryBot

- Heuristic: **Euclidean distance**
- **Question:** What is f(b3)? f(109)?
- A* will spend a bit of time exploring paths in the labs before trying to go around via o109
- At that point the heuristic starts helping more
- **Question:** Does breadth-first search explore paths in the lab too?
- **Question:** Does breadth-first search explore any paths that A* does not?



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.

A* Theorem: Completeness

Proof part 1: A* is complete

- Since arc costs are larger than ϵ, every path in the frontier will eventually have cost larger than k, for any finite k
- So every path in the frontier will eventually have cost larger than the cost of the optimal solution
- So the optimal solution will eventually be removed from the frontier

A* Theorem: Optimality

Proof part 2: Optimality

- If path g is a **solution**, then $f(g)$ is equal to cost(g) **(Why?)**

i.e., $p = \langle s, n_1, \ldots, n_k \rangle$,
$p^* = \langle s, n_1, \ldots, n_k, n_{k+1}, \ldots, z >$,
and p^* is **optimal**

- If a path p **leads to an optimal solution**, and path g is **any solution**, then $f(p) \leq f(g)$ (**Why?**)

- So no **sub-optimal solution** will be removed from the frontier while a **path that leads to an optimal solution** is on the frontier.

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