Branch & Bound

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

CMPUT 261: Introduction to Artificial Intelligence

P&M §3.7-3.8

Logistics

Assignment #1 was released last week

- Available on eClass
- Due: **Thursday, February 1 at 11:59pm**

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 node n to **some** 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 **actual 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

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** $f(p)$

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

Recap: A* Search Algorithm

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

$frontier$:= $\{\langle s \rangle \mid s$ *is a start node*$\}$
while $frontier$ is not empty:
    select f-minimizing path $\langle n_0, \ldots, n_k \rangle$ from *frontier*
    remove $\langle n_0, \ldots, n_k \rangle$ from $frontier$
    if $goal(n_k)$:
        return $\langle n_0, \ldots, n_k \rangle$
    for each neighbour n of n_k:
        add $\langle n_0, \ldots, n_k, n \rangle$ to *frontier*
end while

i.e., $f(\langle n_0, \ldots, n_k \rangle) \leq f(p)$ for all other paths $p \in frontier$

Recap: A* is Optimal

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

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

Proof:

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

"Recap": 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 of Last Lecture

- **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
- **Admissible** heuristics can be built from **relaxations** of the original problem
- *Simple* uses of heuristics do not guarantee improved performance
- **A* algorithm** for use of admissible heuristics with guarantees

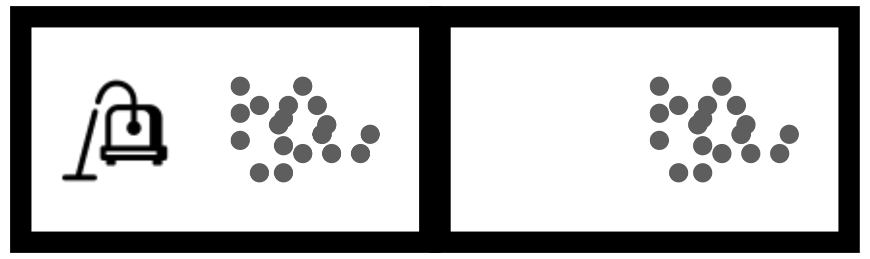
Lecture Outline

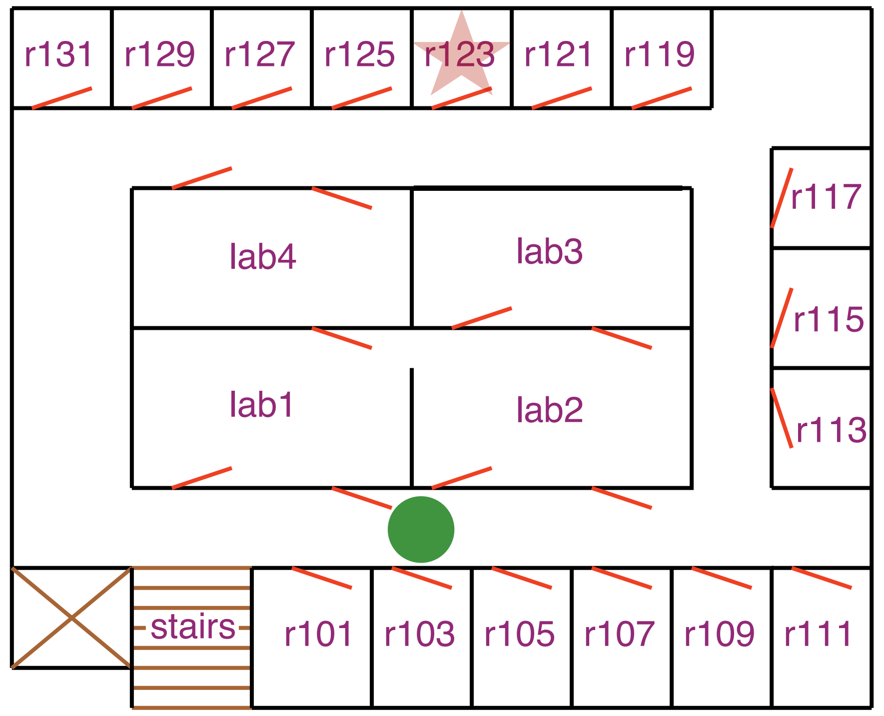
1. Recap & Logistics
2. Constructing Admissible Heuristics
3. Optimal Heuristic Usage
4. Branch & Bound
5. Cycle Pruning
6. Exploiting Search Direction

After this lecture, you should be able to:

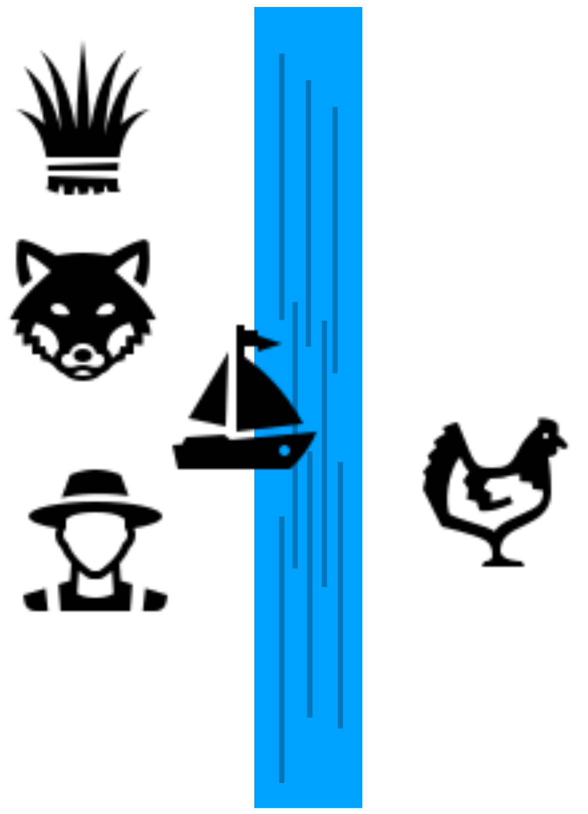
- Construct an admissible heuristic for an arbitrary search problem
- Define heuristic consistency, identify whether a heuristic is consistent
- Implement cycle pruning
- Explain when cycle pruning is and is not space- and time-efficient
- Implement branch & bound and IDA* and demonstrate their operation
- Derive the space and time complexity for branch & bound and IDA*
- Predict whether forward, backward, or bidirectional search are more efficient for a search problem

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?**)

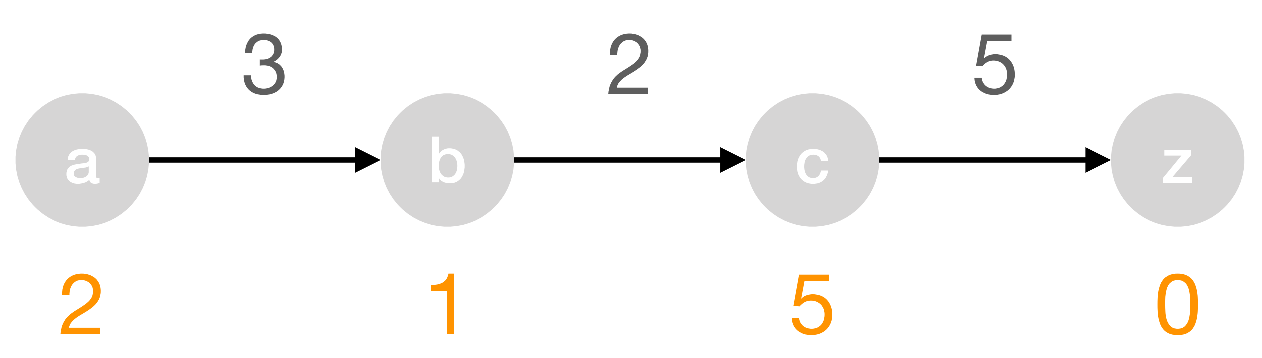


Consistent Heuristic

Definition:
A heuristic h is consistent if, for every pair of nodes $n, n' \in N$,

$$h(n') \leq \text{cost}(n, n') + h(n).$$

- That is, a heuristic never decides that things are "harder than it thought" along a given path
- **Question:** is h **consistent** on the graph below?
- **Question:** is h **admissible** on the graph below?



Heuristic Usage of A*

Definition:

Let p^* be an optimal solution.

A path p is **surely removed** by A* if $f(p) < f(p^*)$.

Theorem:

Any path that is surely removed by A* using a consistent heuristic h will also be removed from the frontier by **any other optimal graph search algorithm** using h.

I.e., there is no way to use a given consistent heuristic that is guaranteed to find an optimal solution faster than A*, "up to tie-breaking"

Space Complexity of A*

- A* makes use of heuristic information to improve **time complexity**
  - Focuses on parts of the search graph that are likely to contain solution
- Explores paths in order of f-value
  - Frontier might need to contain all paths of the same cost as the solution at some point
- Using heuristic to change the order that **depth-first-search** puts paths go into the frontier doesn't reliably improve its time complexity
  - In general, DFS with heuristic-ordering will expand more paths than A* with same heuristic
  - **Can we use a heuristic in some other way to improve DFS's time complexity without giving up its good space complexity?**

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

Branch & Bound Algorithm

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

$frontier := \{\langle s \rangle \mid s \text{ is a start node}\}$
$bound := bound_0$
$best := \emptyset$
while $frontier$ is not empty:
  select the newest path $\langle n_0, \ldots, n_k \rangle$ from $frontier$
  remove $\langle n_0, \ldots, n_k \rangle$ from $frontier$
  if $f(\langle n_0, \ldots, n_k \rangle) \leq bound$:
    if $goal(n_k)$:
      $bound := cost(\langle n_0, \ldots, n_k \rangle)$
      $best := \langle n_0, \ldots, n_k \rangle$
    else:
      for each neighbour n of n_k:
        add $\langle n_0, \ldots, n_k, n \rangle$ to $frontier$
end while
return $best$

Question: Why $cost$ instead of f here?

Choosing $bound_0$

- If $bound_0$ is set to *just* above the optimal cost, branch & bound will explore **no more paths than A***
    - Won't explore any paths p' that are more costly than the optimal solution, because $f(p') > bound_0$
    - Will eventually find the optimal solution path p^* because $f(p^*) < bound_0$
- But we don't (in general) know the cost of the optimal solution!
- One possibility: Initialize $bound_0 = \infty$
    - What problems could this have?
- Solution: **iteratively increase** $bound_0$ (like with IDS)
    - This algorithm is sometimes called **IDA***
    - Some lower-cost paths will be re-explored

Initialize $bound_0$

until solution found:

  Perform **branch & bound** using $bound_0$

  Increase $bound_0$

Iterative Deepening A* (IDA*)

1. What should we **initialize** $bound_0$ to?

2. **How much** should we increase $bound_0$ by at each step?

   - **One idea:**
     Iteratively increase bound to the **lowest f-value** path that was **pruned**
     - Guarantees at least one more path will be explored
     - Can stop immediately after finding a solution (**why?**)
     - Time complexity can be **much worse** than A*:
       $O(b^{2m})$ instead of $O(b^m)$ (**why?**)
   - Need to increase $bound_0$ by **enough** (else won't explore enough), but **not too much** (else won't prune enough)
   - Choosing next f-limit is an active area of research (see https://www.movingai.com/SAS/IDA/)

Initialize $bound_0$

until solution found:

  Perform **branch & bound** using $bound_0$

  Increase $bound_0$

	Heuristic Depth First	A*	Branch & Bound	IDA*
Space complexity	*$O(mb)$*	$O(b^m)$	*$O(mb)$*	*$O(mb)$*
Time Complexity	$O(b^m)$	$O(b^m)$	$O(b^m)$	*(depends on how bound increases)*
Heuristic Usage	Limited	Optimal (up to tie-breaking, for consistent *h*)	Optimal (if bound *low* enough)	Close to Optimal
Optimal?	No	Yes	Yes (if bound *high* enough)	Yes

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

$frontier$:= $\{\langle s \rangle \mid s$ *is a start node*$\}$
while $frontier$ is not empty:
  select the newest path $\langle n_0, \ldots, n_k \rangle$ from $frontier$
  remove $\langle n_0, \ldots, n_k \rangle$ from $frontier$
  if $n_k \neq n_j$ for all $0 \leq j < k$:
    if $goal(n_k)$:
      return $\langle n_0, \ldots, n_k \rangle$
    for each neighbour n of n_k:
      add $\langle n_0, \ldots, n_k, n \rangle$ to $frontier$
end while

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. $goal^r(n) = 1$ if $n \in S$
       (i.e., if n is a start node of the original problem)

Questions:

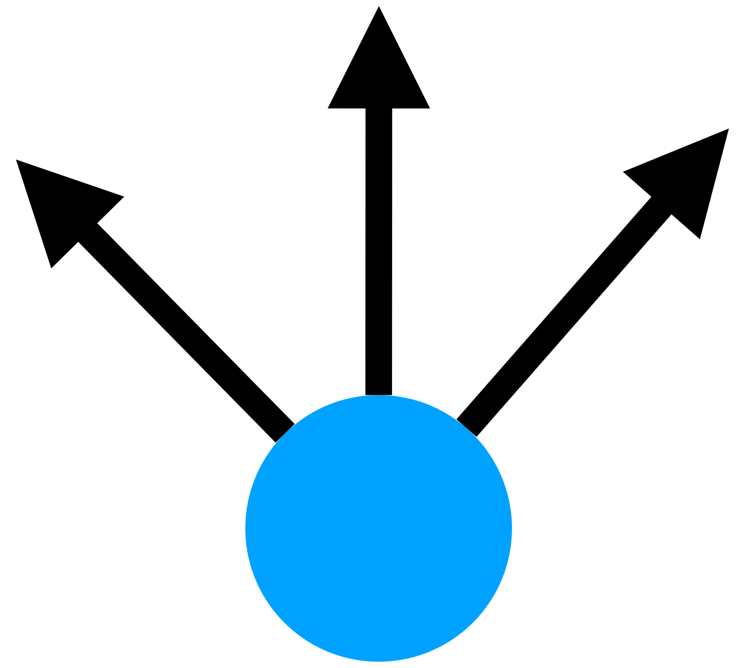
1. When is this **useful**?
2. When is this **infeasible**?

Reverse Search

Definitions:

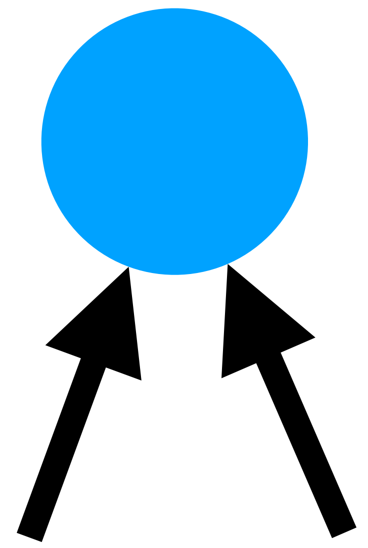
1. **Forward branch factor:** Maximum number of **outgoing** neighbours
   Notation: b

   - 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:
  - **Guaranteeing** 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

- A* uses **consistent** heuristics **optimally** ("up to tie breaking")
- **Branch & bound** combines the **optimality** guarantee and **heuristic efficiency** of A* with the space efficiency of depth-first search
- **IDA*** is an iterative-deepening version of branch & bound that doesn't require that you get the initial bound "right"
  - But its time complexity can be significantly worse
- Tweaking the **direction of search** can yield efficiency gains