Branch & Bound

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

CMPUT 261: Introduction to Artificial Intelligence

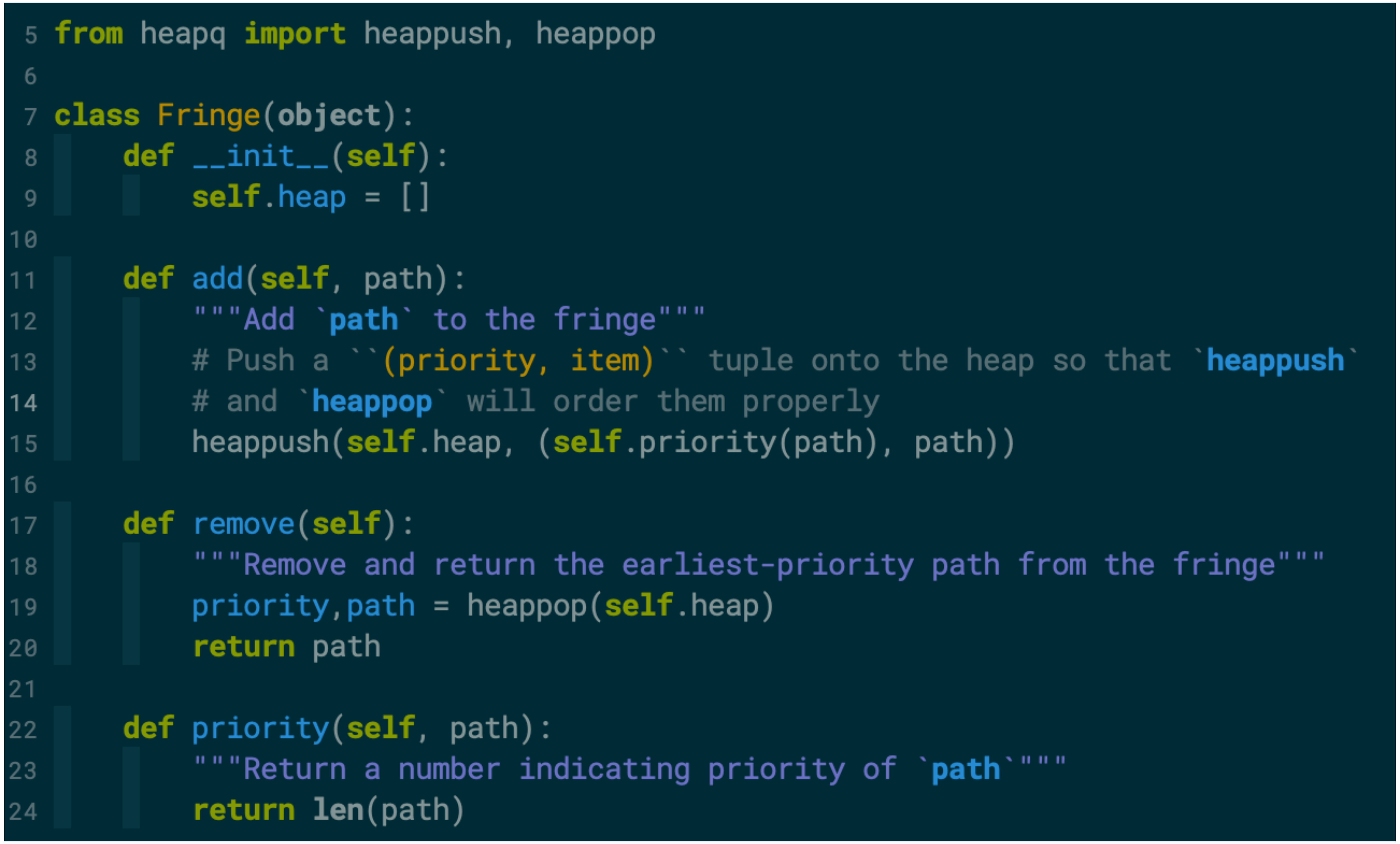
P&M §3.7-3.8

Logistics

Assignment #1 was released on Tuesday

- Available on eClass
- Due: Tuesday **September 27 at 11:59pm**

Example: Heaps in Python



```python
from heapq import heappush, heappop

class Fringe(object):
    def __init__(self):
        self.heap = []

    def add(self, path):
        """Add `path` to the fringe"""
        # Push a ``(priority, item)`` tuple onto the heap so that `heappush`
        # and `heappop` will order them properly
        heappush(self.heap, (self.priority(path), path))

    def remove(self):
        """Remove and return the earliest-priority path from the fringe"""
        priority,path = heappop(self.heap)
        return path

    def priority(self, path):
        """Return a number indicating priority of `path`"""
        return len(path)
```

Source: https://jrwright.info/introai/examples/fringe.py

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^*:
  - $f(p) \leq \text{cost}(p^*)$

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

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, 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**

Lecture Outline

1. Recap & Logistics
2. Cycle Pruning
3. Branch & Bound
4. Exploiting Search Direction

After this lecture, you should be able to:

- 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

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

```
$frontier := \{\langle s \rangle \mid s \text{ is a start node}\}$
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
```

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 $h(n)$; $bound_0$

```
$frontier := \{\langle s \rangle \mid s$ 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 not 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!
- Solution: **iteratively increase** $bound_0$ (like with iterative deepening search)
  - 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?
   - 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?**)
   - 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 Search

	Heuristic Depth First	**A***	**Branch & Bound**	**IDA***
Space complexity	$O(mb)$	$O(b^m)$	$O(mb)$	$O(mb)$
Time Complexity	$O(b^m)$	$O(b^m)$	$O(b^m)$	$O(b^{2m})$
Heuristic Usage	Limited	Optimal	Optimal (if bound *low* enough)	Close to Optimal
Optimal?	No	Yes	Yes (if bound *high* enough)	Yes

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**

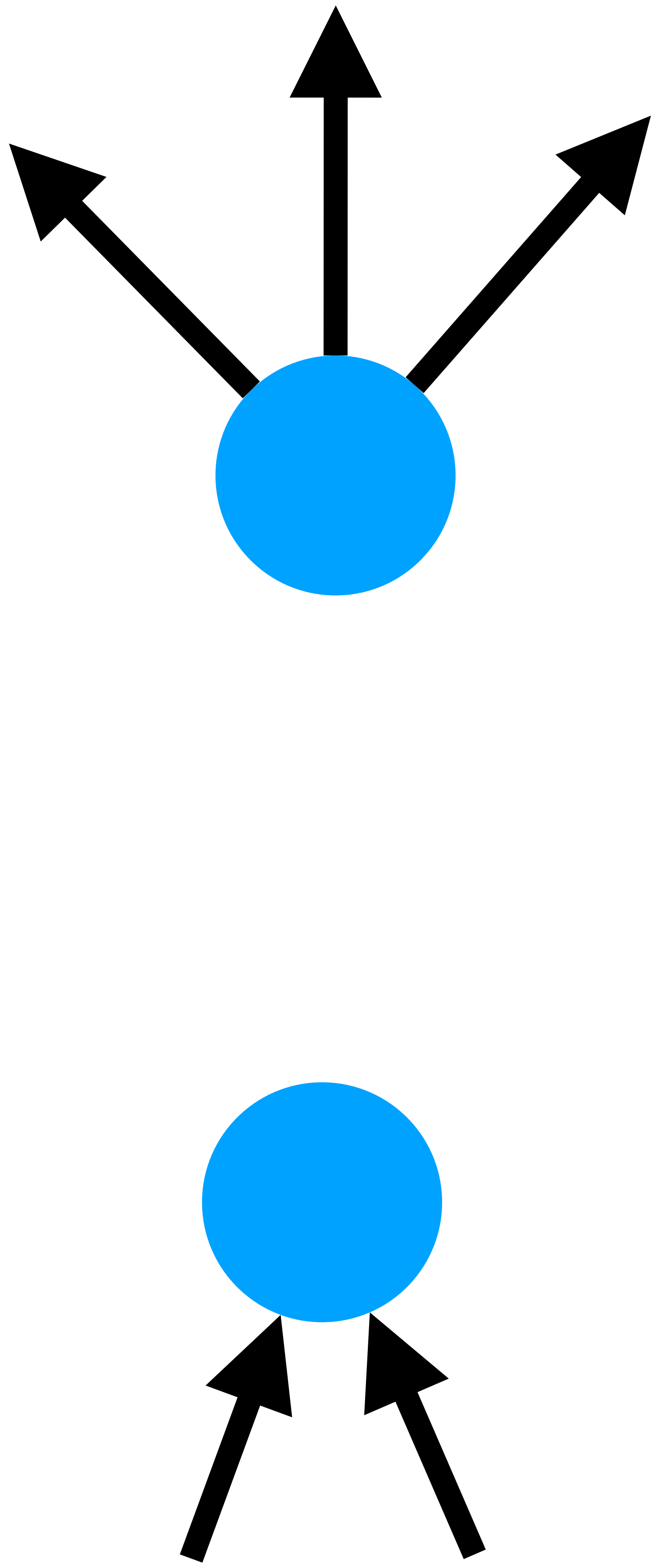
- Given a search graph $G = (N, A)$, **known** goal node g, and set of start nodes S, can construct a **reverse search problem** $G = (N, A^r)$:

  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

- The more **accurate** the heuristic is, the **fewer** the paths A* will explore
- **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