Uninformed Search

CMPUT 261: Introduction to Artificial Intelligence

P&M §3.5

Logistics

- TA office hours begin next week
  - Moemen: Mondays at 10am
  - Shi-ang: Wednesdays at 12pm
  - Amir: Tuesdays at 2pm
  - See eClass page for meeting links
- Assignment #1 released next week
- Python tutorials next week during TA office hours

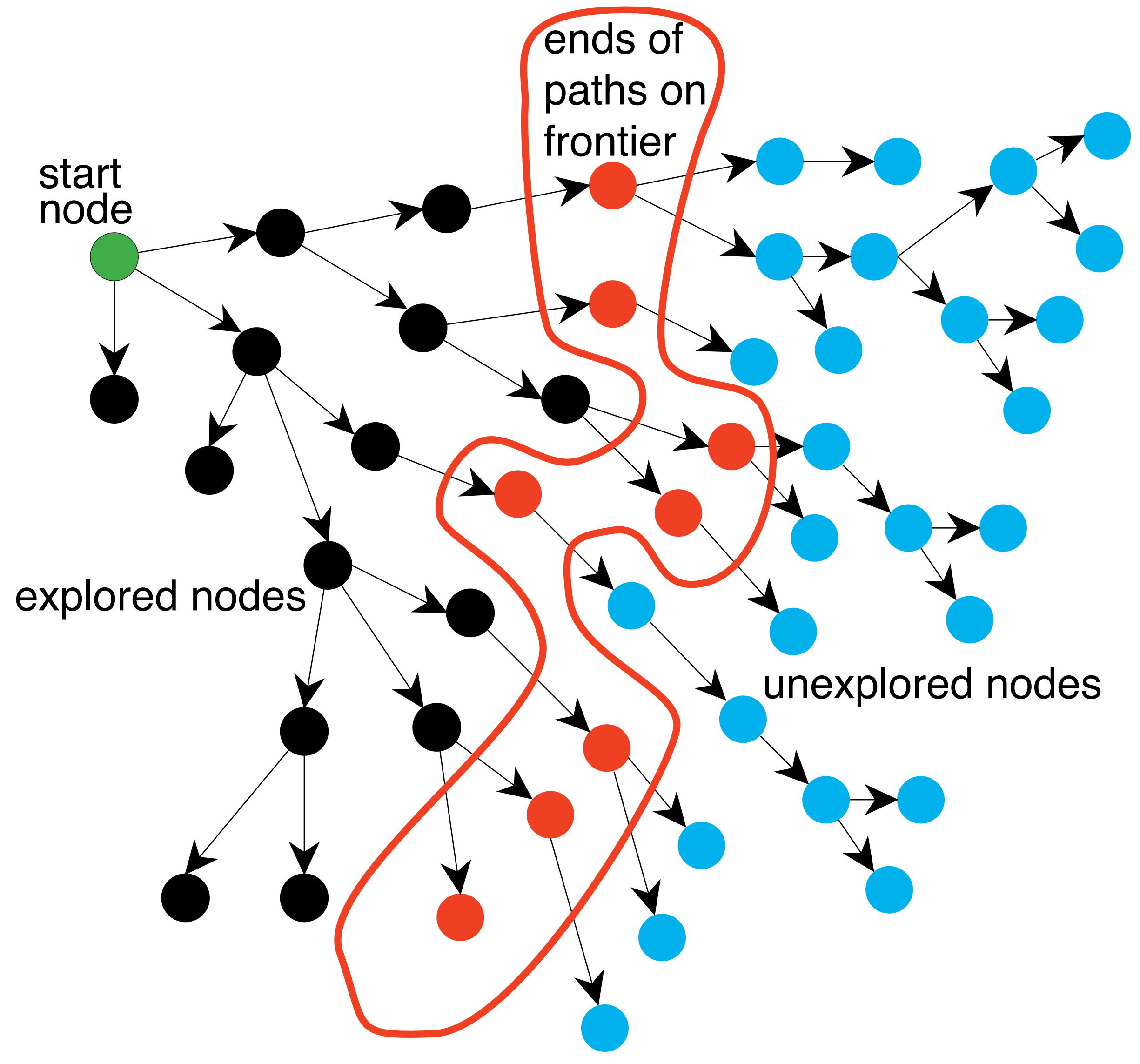
Recap: Graph Search

- Many AI tasks can be represented as **search problems**
    - A single generic **graph search algorithm** can then solve them all!
- A search problem consists of **states**, **actions**, **start states**, a **successor function**, a **goal** function, optionally a **cost** function
- **Solution quality** can be represented by labelling **arcs** of the search graph with **costs**

Recap: Generic Graph 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 a 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



https://artint.info/2e/html/ArtInt2e.Ch3.S4.html

Lecture Outline

1. Logistics & Recap
2. Properties of Algorithms and Search Graphs
3. Depth First and Breadth First Search
4. Iterative Deepening Search
5. Least Cost First Search

After this lecture, you should be able to:

- Demonstrate the operation of depth-first, breadth-first, iterative-deepening, and least-cost-first search on a graph
- Implement depth-first, breadth-first, iterative deepening, and least-cost first search
- Derive the time and space requirements for instantiations of the generic graph search algorithm

Algorithm Properties

What properties of algorithms do we want to analyze?

1. A search algorithm is **complete** if it is guaranteed to find a solution within a finite amount of time whenever a solution exists.

2. The **time complexity** of a search algorithm is a measure of how much **time** the algorithm will take to run, in the **worst case**.

   - In this section we measure by **total** number of paths added to the frontier.

3. The **space complexity** of a search algorithm is a measure of how much **space** the algorithm will use, in the **worst case**.

   - We measure by maximum number of paths in the frontier **at one time**.

Search Graph Properties

What properties of the **search graph** do algorithmic properties depend on?

- **Forward branch factor**: Maximum number of neighbours
  Notation: b

- **Maximum path length**. (Could be infinite!)
  Notation: m

- Presence of **cycles**

- Length of the **shortest** path to a **goal** node

Depth First Search

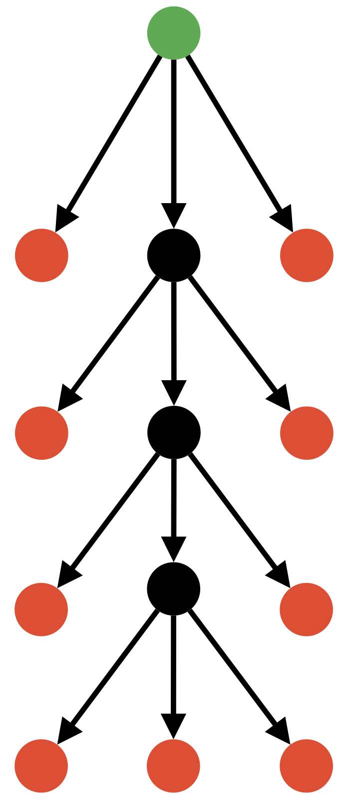
Input: a *graph*; a set of *start nodes*; a $goal$ function

$frontier$:= $\{\langle s \rangle \mid s$ *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 $goal(n_k)$:
    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

Question:

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

Depth First Search



Depth-first search always removes one of the **longest** paths from the frontier.

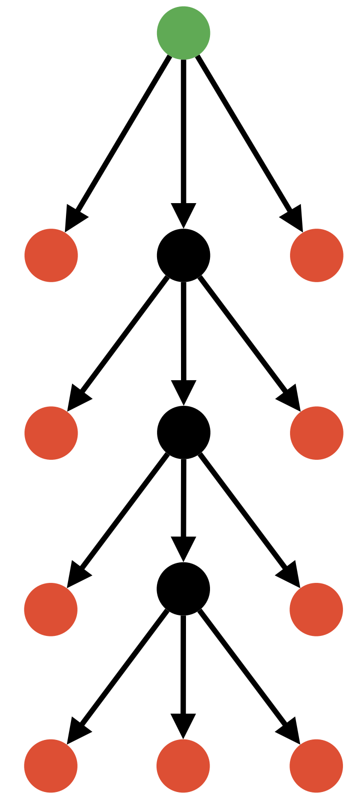
Example:
Frontier: $[p_1, p_2, p_3, p_4]$
$successors(p_1) = \{n_1, n_2, n_3\}$

What happens?

1. Remove p_1; test p_1 for goal
2. Add $\{\langle p_1, n_1\rangle, \langle p_1, n_2\rangle, \langle p_1, n_3\rangle\}$ to **front** of frontier
3. New frontier: $[\langle p_1, n_1\rangle, \langle p_1, n_2\rangle, \langle p_1, n3\rangle, p_2, p_3, p_4]$
4. p_2 is selected only after **all paths starting with p_1** have been explored

Question: When is $\langle p_1, n_3\rangle$ selected?

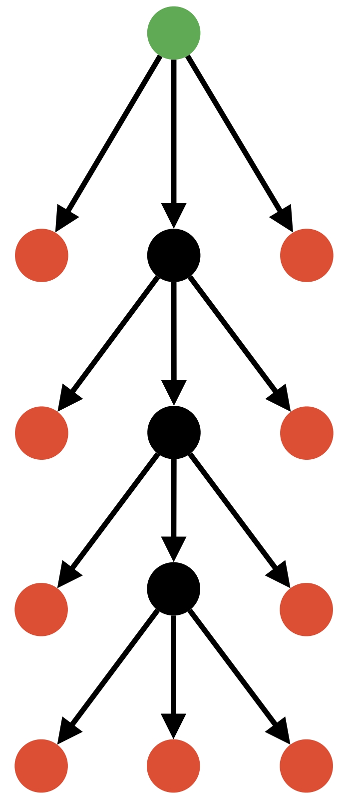
Depth First Search Analysis



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

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

When to Use
Depth First Search



- When is depth-first search **appropriate**?
  - Memory is restricted
  - All solutions at same approximate depth
  - Order in which neighbours are searched can be tuned to find solution quickly
- When is depth-first search **inappropriate**?
  - Infinite paths exist
  - When there are likely to be shallow solutions
    - Especially if some other solutions are very deep

Breadth First Search

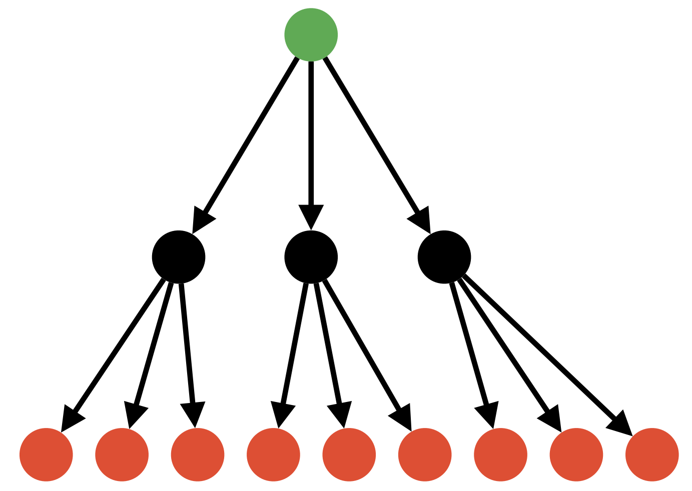
Input: a *graph*; a set of *start nodes*; a $goal$ function

$frontier := \{\langle s \rangle \mid s \text{ is a start node}\}$
while $frontier$ is not empty:
    select the oldest 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

Question:

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

Breadth First Search



Breadth-first search always removes one of the **shortest** paths from the frontier.

Example:
Frontier: $[p_1, p_2, p_3, p_4]$
$successors(p_1) = \{n_1, n_2, n_3\}$

What happens?

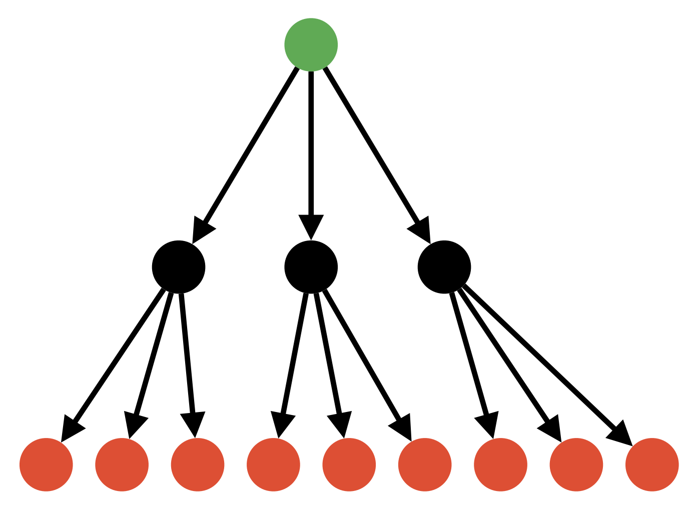
1. Remove p_1; test p_1 for goal

2. Add $\{\langle p_1, n_1 \rangle, \langle p1{,}n2 \rangle, \langle p1{,}n3 \rangle\}$ to **end** of frontier:

3. New frontier: $[p_2, p_3, p_4, \langle p_1, n_1 \rangle, \langle p1{,}n2 \rangle, \langle p1{,}n3 \rangle]$

4. p_2 is selected **next**

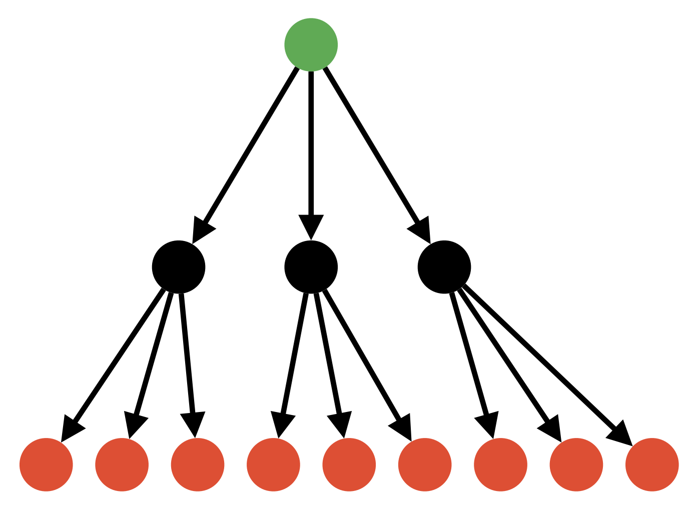
Breadth First Search Analysis



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

When to Use Breadth First Search



- When is breadth-first search **appropriate?**
  - When there might be infinite paths
  - When there are likely to be shallow solutions, *or*
  - When we want to guarantee a solution with fewest arcs
- When is breadth-first search **inappropriate?**
  - Large branching factor
  - All solutions located deep in the tree
  - Memory is restricted

Comparing DFS vs. BFS

	Depth-first	**Breadth-first**
Complete?	Only for finite graphs	Complete
Space complexity	$O(mb)$	$O(b^m)$
Time complexity	$O(b^m)$	$O(b^m)$

- Can we get the space benefits of depth-first search without giving up completeness?
- Run depth-first search to a maximum depth
  - then try again with a larger maximum
  - until either goal found or graph completely searched

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

```
for max_depth from 1 to ∞:
    more_nodes := False
    frontier := {⟨s⟩ | s is a start node}
    while frontier is not empty:
        select the newest path ⟨n_0, …, n_k⟩ from frontier
        remove ⟨n_0, …, n_k⟩ from frontier
        if goal(n_k):
            return ⟨n_0, …, n_k⟩
        if k < max_depth:
            for each neighbour n of n_k:
                add ⟨n_0, …, n_k, n⟩ to frontier
        else if n_k has neighbours:
            more_nodes := True
    end-while
    if more_nodes = False:
        return None
```

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]

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

$frontier$:= $\{\langle s \rangle \mid s$ *is a start node*$\}$
while $frontier$ is not empty:
  select the cheapest 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., $cost(\langle n_0, \ldots, n_k \rangle) \le cost(p)$ for all other paths $p \in frontier$

Question:

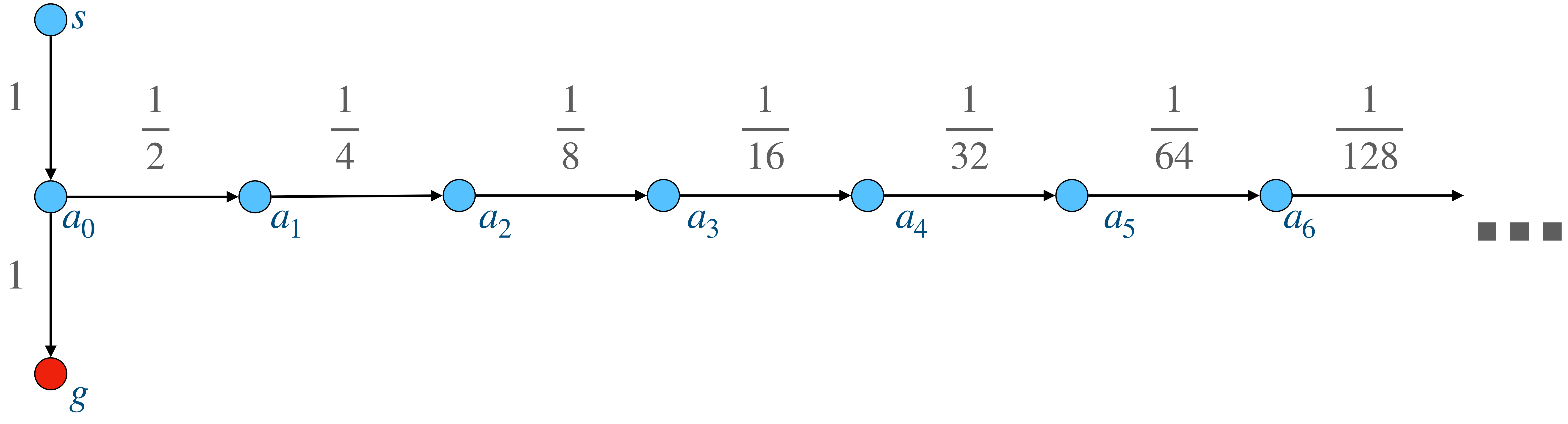
What **data structure** for the frontier implements this search strategy?

Least Cost First Search Analysis

- **Theorem:** Least Cost First Search is **complete** and **optimal** if there is $\epsilon > 0$ with $cost(\langle n_1, n_2 \rangle) > \epsilon$ for every arc $\langle n_1, n_2 \rangle$:
  1. Suppose $\langle n_0, \ldots, n_k \rangle$ is the optimal solution
  2. Suppose that p is any non-optimal solution
     So, $cost(p) > \langle n_0, \ldots, n_k \rangle$
  3. For every $0 \leq \ell \leq k$, $cost(\langle n_0, \ldots, n_\ell \rangle) < cost(p)$
  4. So p will never be removed from the frontier before $\langle n_0, \ldots, n_k \rangle$
- 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?

Why $c(n_1, n_2) > \epsilon > 0$ instead of just $c(n_1, n_2) > 0$?

- Consider the infinite search graph below
- Every cost is larger than 0
- But there's no **single positive value** that is smaller than all costs
    - Can make arc costs arbitrarily small by following the right-hand path far enough
- But then $c\left(\langle s, a_0, g\rangle\right) > c\left(\langle s, a_0, a_1, \ldots, a_n\rangle\right)$ for **all** values of n
    - The solution $\langle s, a_0, g\rangle$ will **never be removed** from the frontier



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 search** is **optimal** (under some conditions), but still must potentially expand every node