Lecture 14: Search

Mark Hasegawa-Johnson

Johannes Hevelius observing with one of his telescopes (1647).

Outline

- Planning in a Known, Observable, Deterministic Environment
- Dijkstra’s algorithm: Frontier, Explored set
- Explored set: Avoid expanding the same state
- DFS, BFS, and UCS

Planning in a Known, Fully Observable, Deterministic Environment

Suppose that you are at point A, and you want to get to point B.

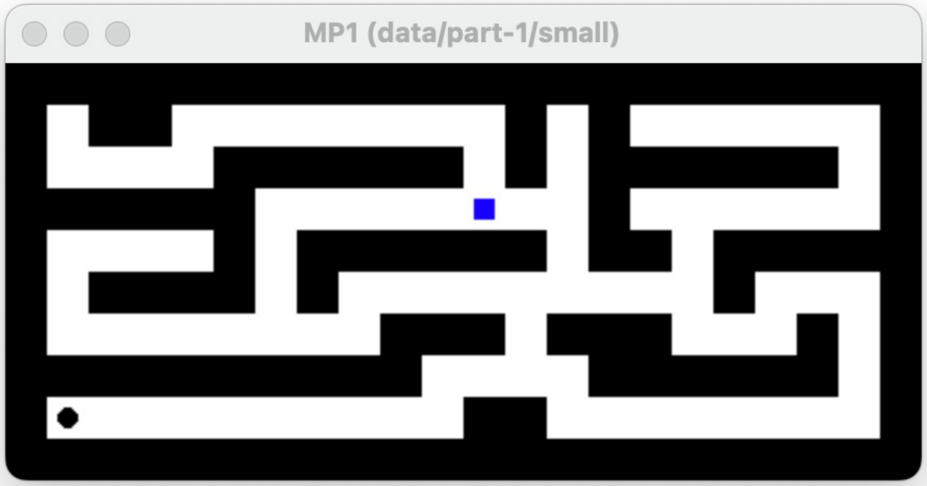
- **Known environment:** You have a valid map, telling you all the possible turns you could take between A and B.
- **Fully Observable:** You have thousands of internet strangers telling you about all the traffic jams between A and B, so you know exactly how long each road will take.
- **Deterministic:** the time required for each road is not random.

The rational thing to do is to (let your mapping software) plan your route in advance, so that you can get from A to B in the shortest possible time.

Today's subject: how much computation is required to find the ***optimal plan*** (the shortest path)?

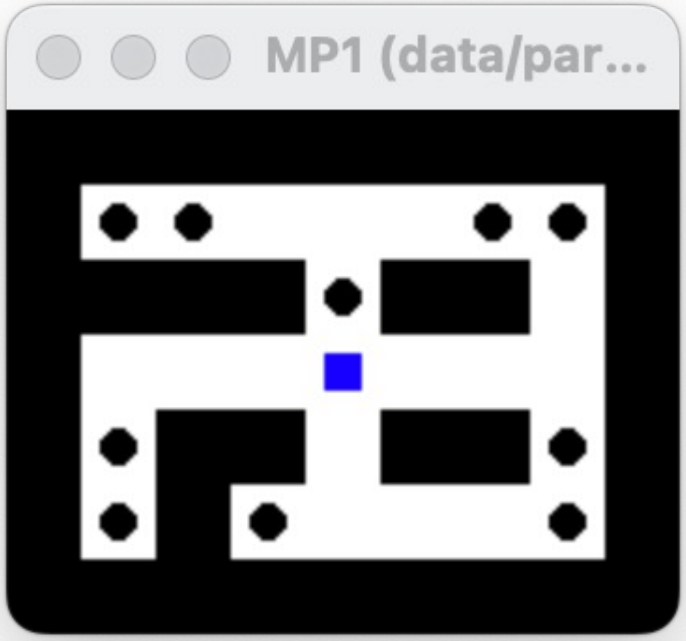
Search example: Solve this maze

- Starting point: Start at the square
- Goal: Reach the dot



Search example: Traveling salesman problem

- Starting point: Start at the square
- Goal: Reach ***<u>all the dots</u>***, in the shortest possible number of steps



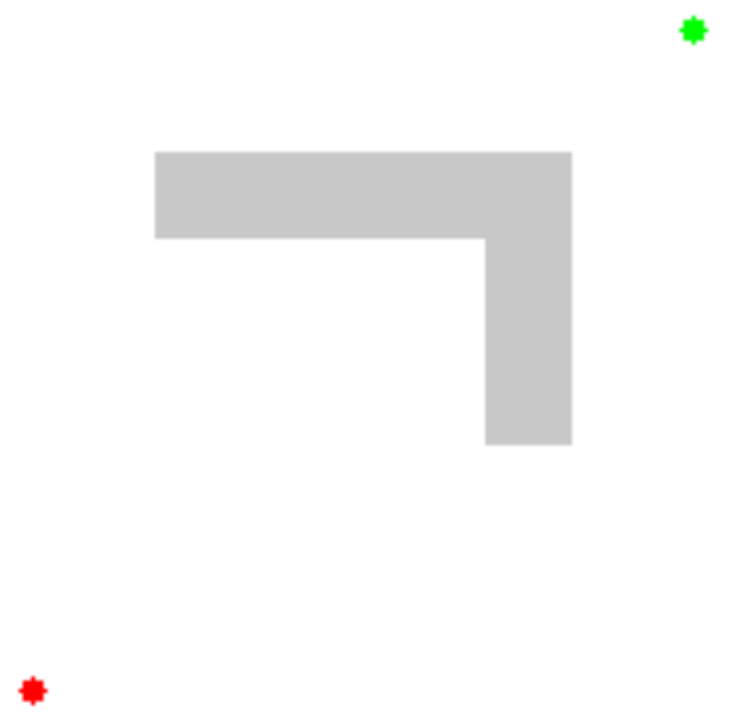
Outline

- Planning in a Known, Observable, Deterministic Environment
- Dijkstra’s algorithm: Frontier, Explored set
- Explored set: Avoid expanding the same state
- DFS, BFS, and UCS

Dijkstra’s algorithm

Dijkstra’s algorithm divides all states into three sets:

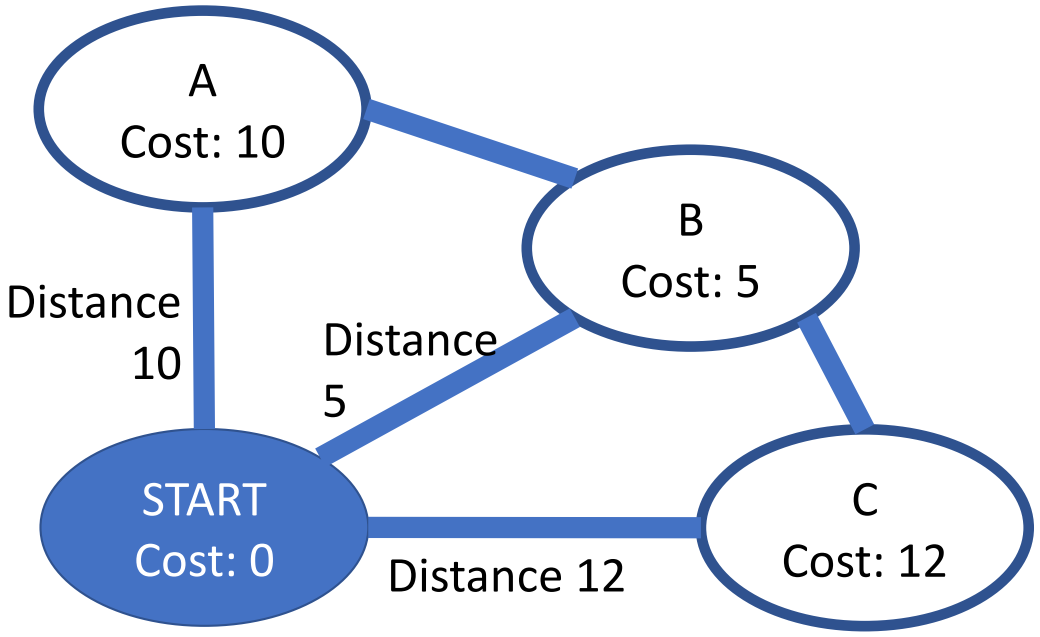
- The **explored s**et is the set of states to which you already know the shortest path (filled circles)
- The **unexplored** set are the states you don’t yet know how to reach (blank)
- The **frontier** are states you know how to reach, but you don’t yet know the shortest path (open circles)



Dijkstra’s algorithm, a.k.a. Uniform Cost Search

Step 1:

- Put the starting state in the explored set (you can reach it with a cost of 0).
- Put its neighbors into the frontier.
  - To each of them, specify the length of the best path you know.
  - Note: the path you know is not necessarily the best path!!



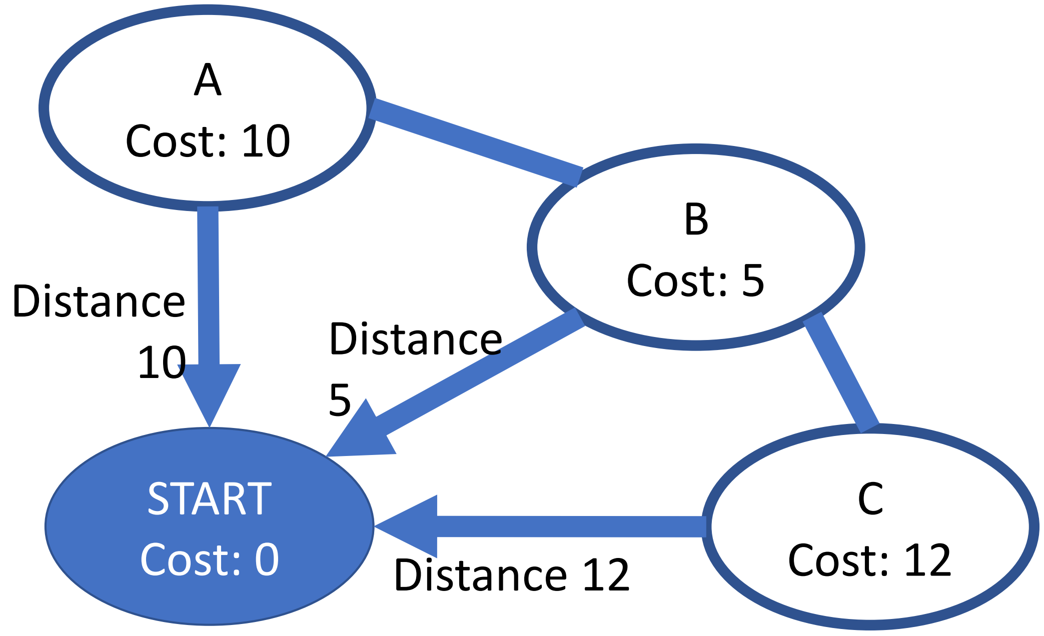
<u>explored set</u>: { START }

<u>frontier</u>: { (5,B), (10,A), (12,C) }

Dijkstra’s algorithm, a.k.a. Uniform Cost Search

Step 1:

- Each node (except the starting node) should have a pointer to its parent
- Each node has just **one parent**
- Later on we might change a node’s parent



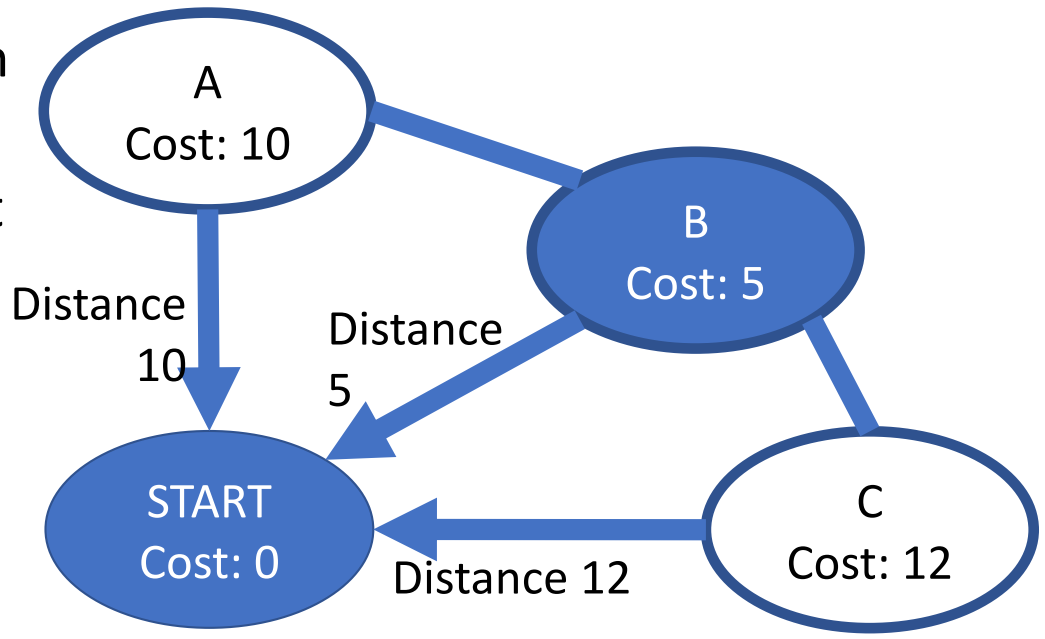
explored set: { START }

frontier: { (5,B,START), (10,A,START), (12,C,START) }

Dijkstra’s algorithm, a.k.a. Uniform Cost Search

Step 2:

- Move the lowest-cost state from frontier to explored,
- …because we know the shortest path to that node.
- All other nodes have a higher cost than this node, so…
- …we will never find a path to this node that’s less expensive than the path we already know.

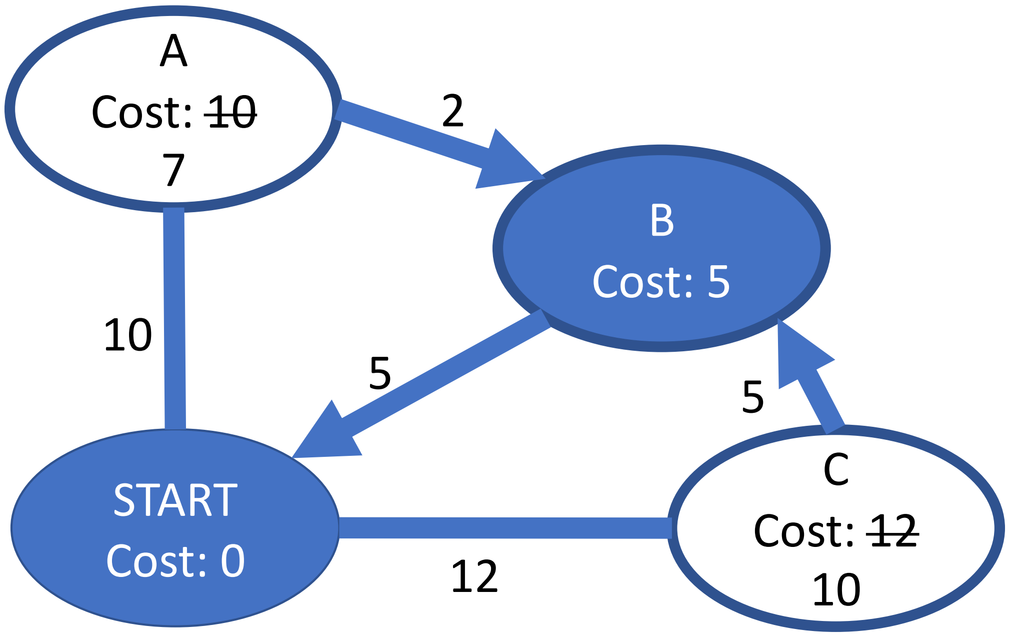


explored set: { START, B } **frontier**: { (10,A,START), (12,C,START) }

Dijkstra's algorithm, a.k.a. Uniform Cost Search

Step 2:

- If the newly explored node has neighbors that are still in the frontier, you might need to update their costs.
- If you update their costs, you should also update the "parent" pointer, so it points to the parent on the lowest-cost path.

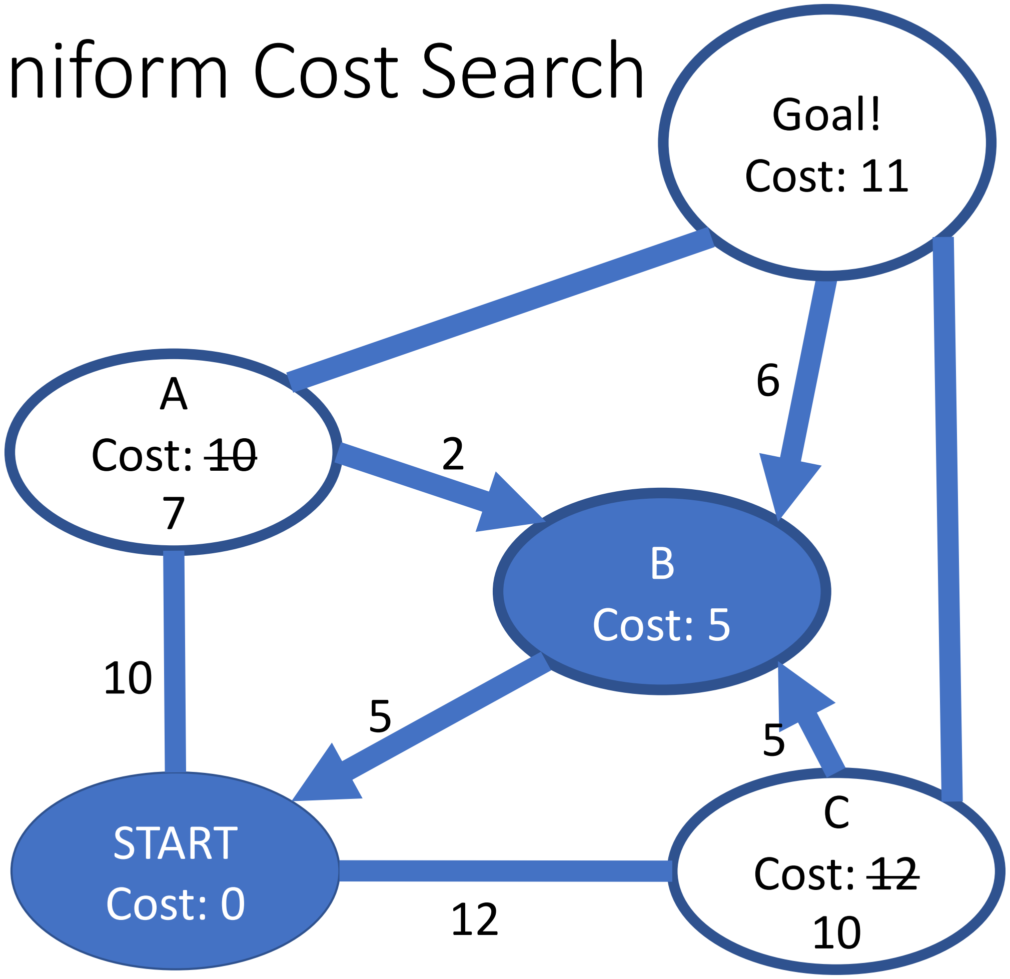


explored set: { START, B }   **frontier**: {(7,A,B), (10,C,B), (10,A,START), (12,C,START) }

Dijkstra's algorithm, a.k.a. Uniform Cost Search

Step 2:

- If the newly explored node has neighbors that are not in the frontier yet…
- …add them to the frontier.

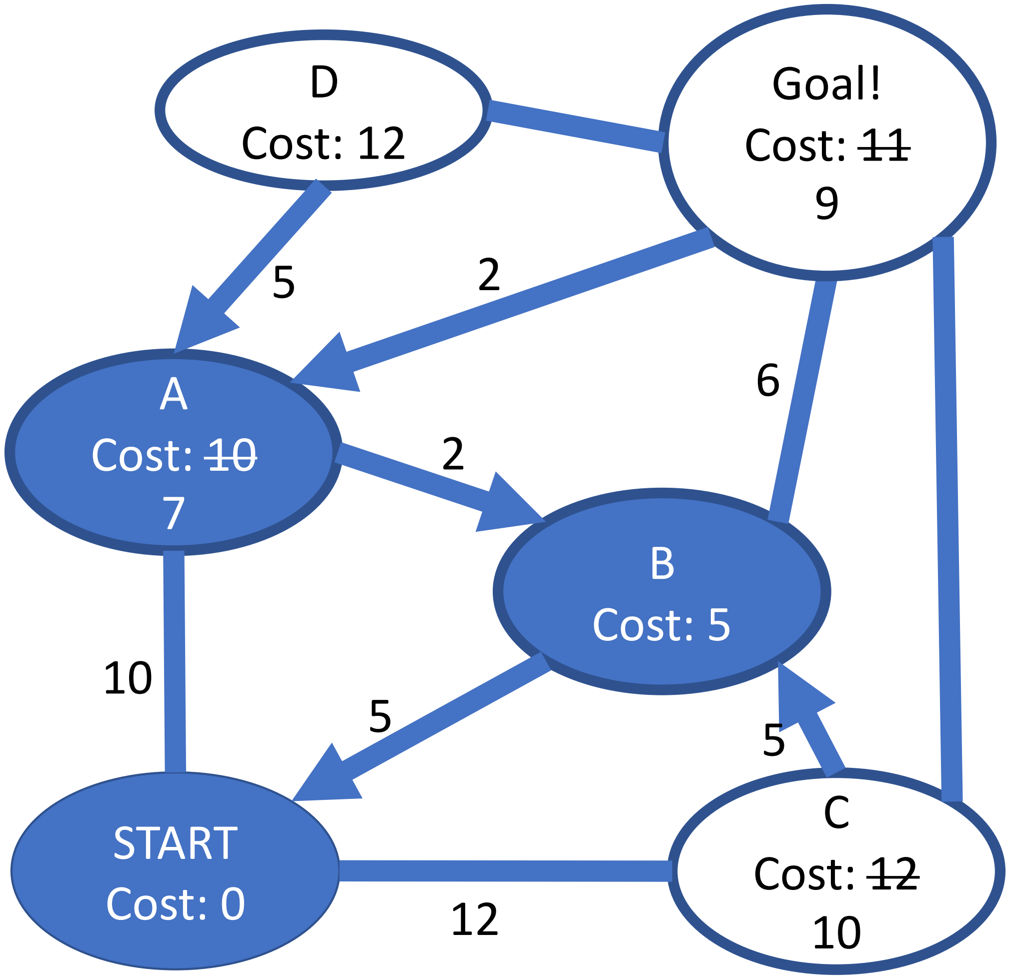


explored set: { START, B }   **frontier**: {(7,A,B), (10,C,B), (10,A,S), (11,G,B), (12,C,S) }

Dijkstra’s algorithm, a.k.a. Uniform Cost Search

Step 3:

- Take the cheapest node from the frontier, move it to the explored set. You now know its cheapest path.
- Add its neighbors to the frontier, with updated costs

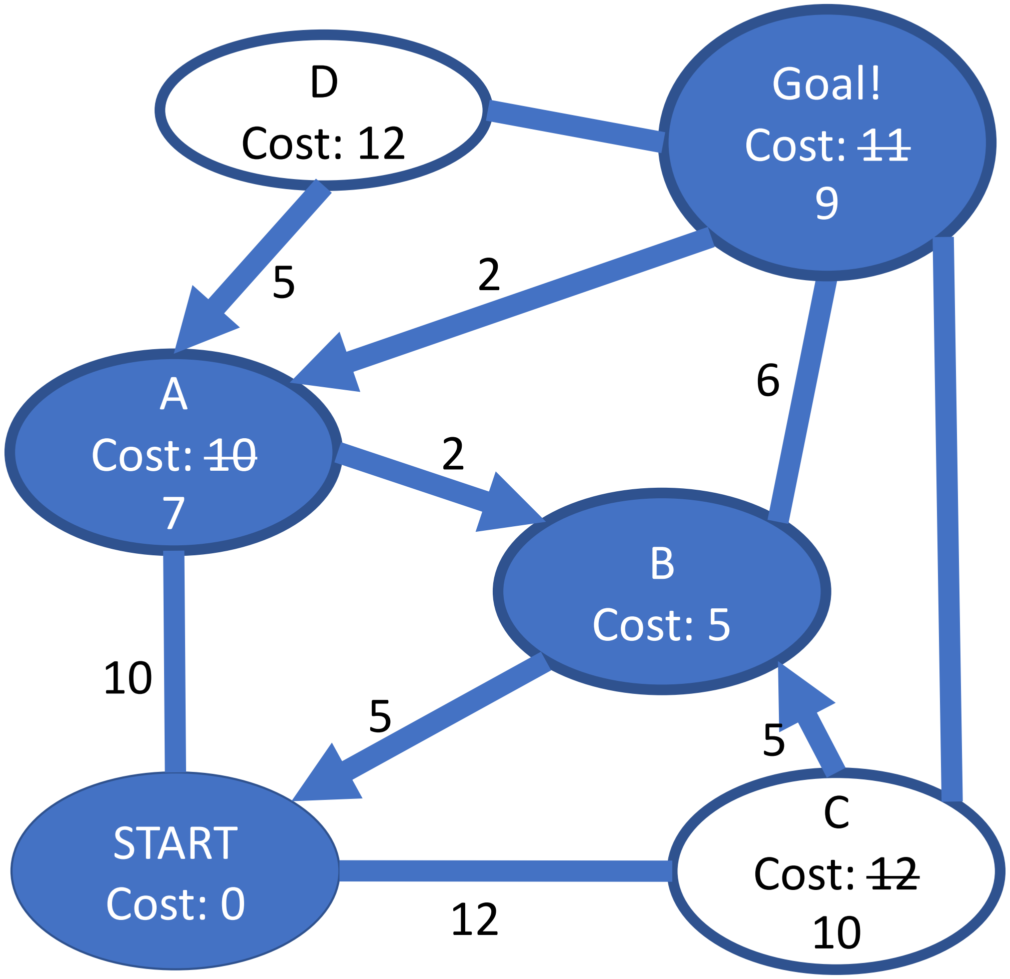


explored set: { START,B,A } **frontier**: { (9,G,A),(10,C,B),(10,A,S),(11,G,B),(12,C,S),(12,D,A) }

Dijkstra's algorithm, a.k.a. Uniform Cost Search

Last step:

- When you move the Goal state from Frontier to Explored, then you are done.
- There is no unexplored state with a lower cost, therefore…
- … it is guaranteed that you have found the lowest-cost path to the goal.

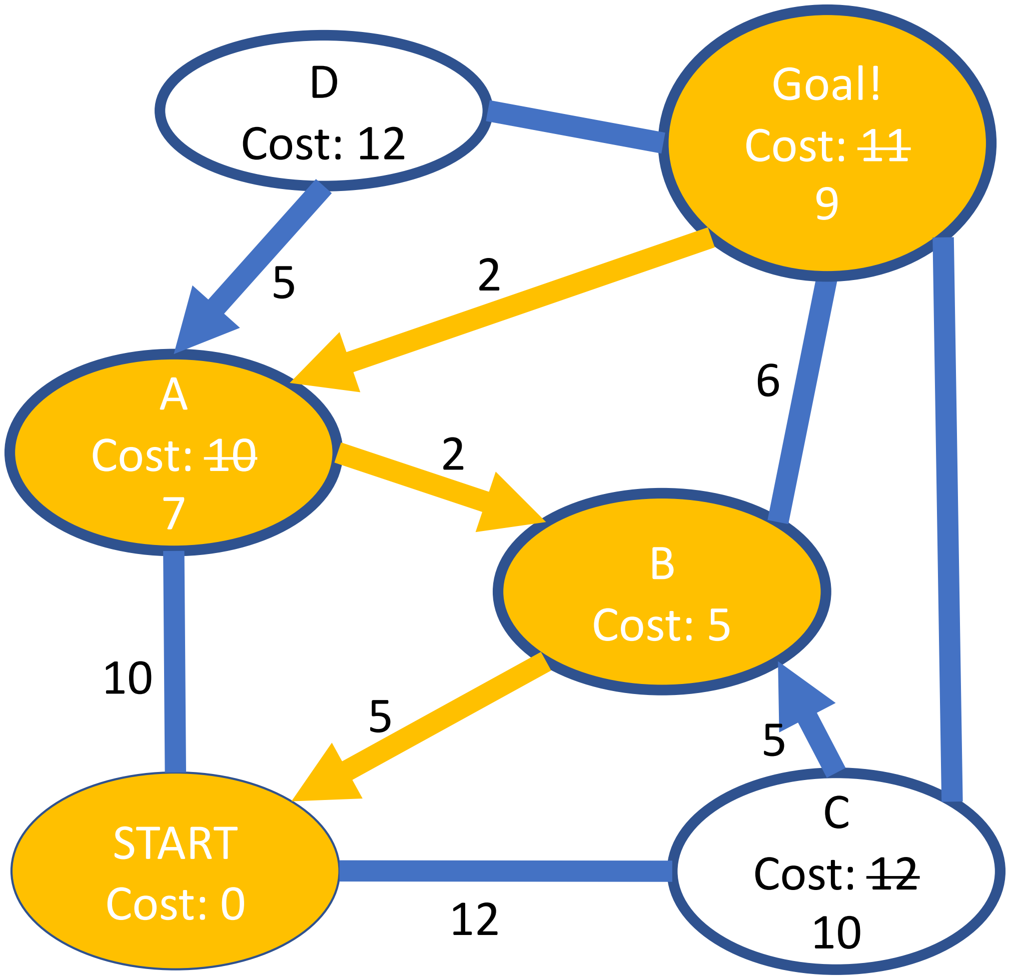


explored set: { START,B,A,GOAL }

frontier: { (10,C,B),(10,A,S),(11,G,B),(12,C,S),(12,D,A) }

Dijkstra's algorithm, a.k.a. Uniform Cost Search

The lowest-cost path is found by back-tracing from each node to its parent.

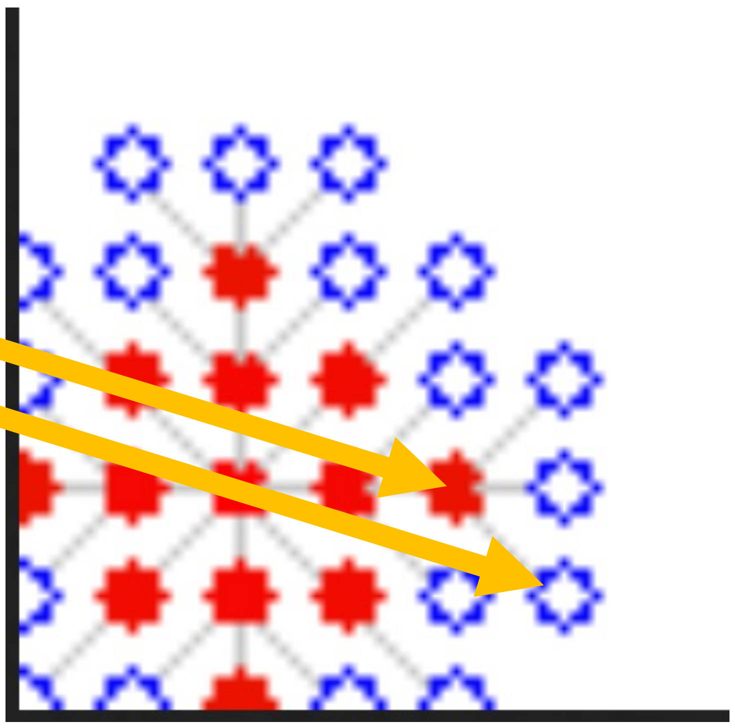


Outline

- Planning in a Known, Observable, Deterministic Environment
- Dijkstra’s algorithm: Frontier, Explored set
- Explored set: Avoid expanding the same state
- DFS, BFS, and UCS

Purpose of the explored set

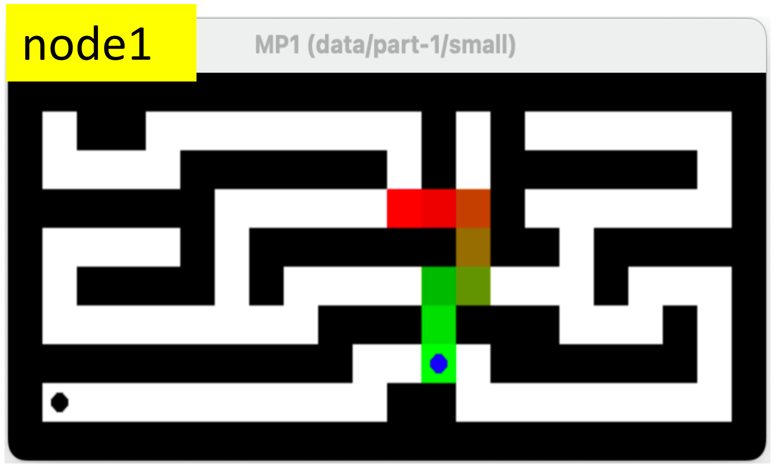
- Once a state has been inserted into the explored set, it should not be evaluated again.
- For example, this state…
- …is a neighbor of this state…
- …but moving backward from the right-hand state to the left-hand state would be a waste of time!

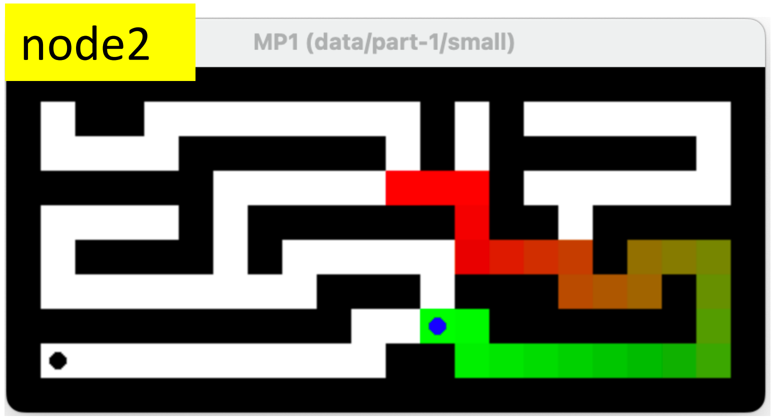


Purpose of the explored set

The purpose of the explored set is to make sure that:

- if we have already found the path shown on top,
- we don't waste time evaluating the path on the bottom.
- The path on the bottom is a worse way to get to exactly the same state





How to avoid wasted time: Hashing

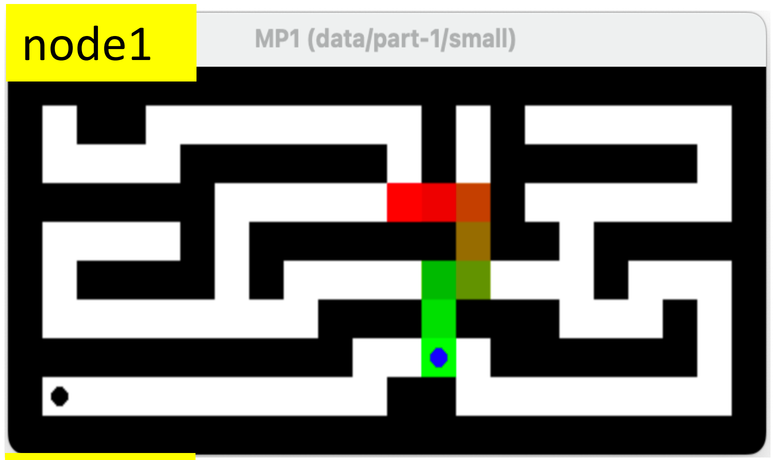
By defining an appropriate hash() function, we can make these two nodes appear to be the same node. For example:

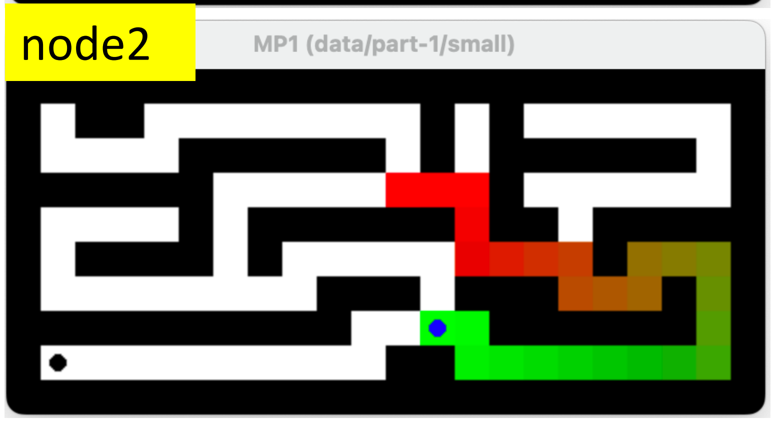
```
class Node():
    def hash(self):
        return hash((self.y,self.x))
```

If you then do:

```
explored = set([node1])
node2 in explored
```

The answer will be True, and so you will know that you don't need to waste computation on node2.



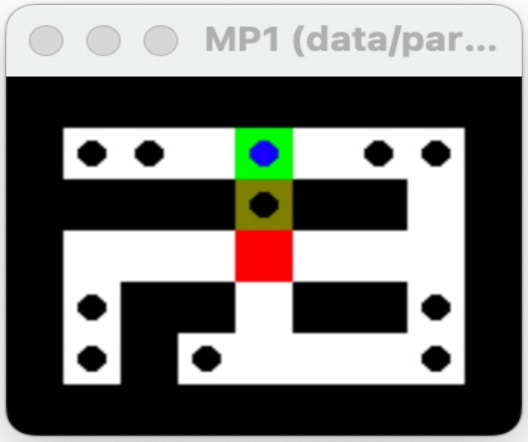


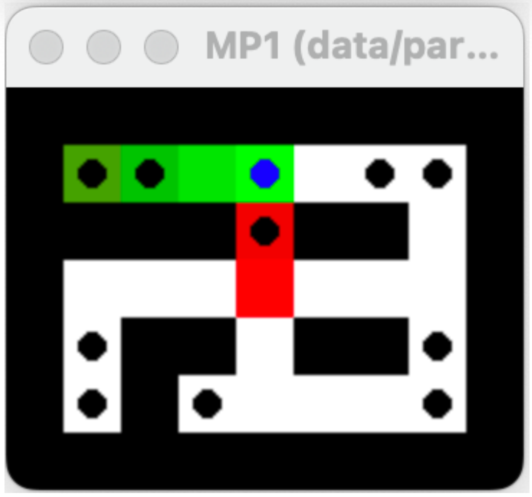
But watch out!!!

In a multi-waypoint maze, the hash function needs to consider

- (y, x) location
- Which waypoints have been reached

If you ignore the waypoints, the hash function will incorrectly tell you that these two are the same, so the lower path will be discarded





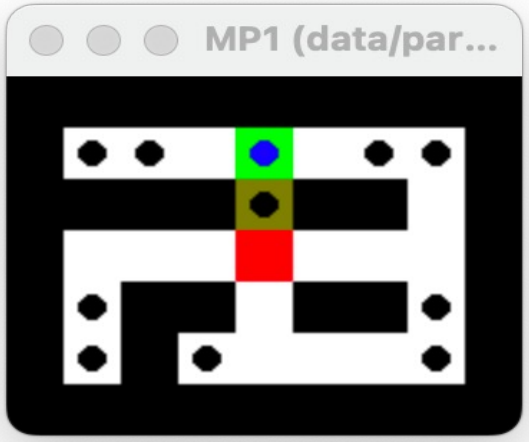
Advantages and disadvantages of explored set

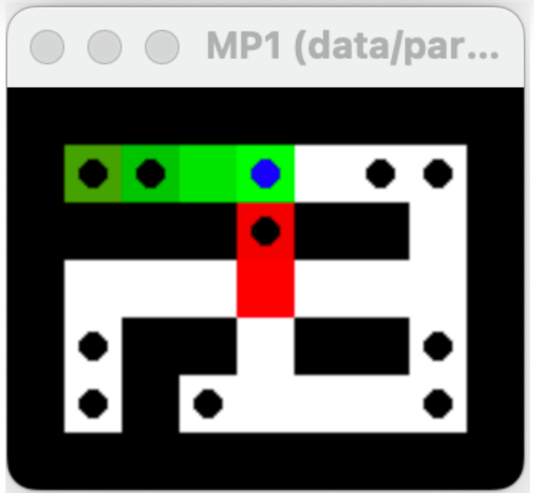
Advantage of an explored set:

- It saves computation, because you don't re-expand states you have already expanded.

Disadvantage of an explored set:

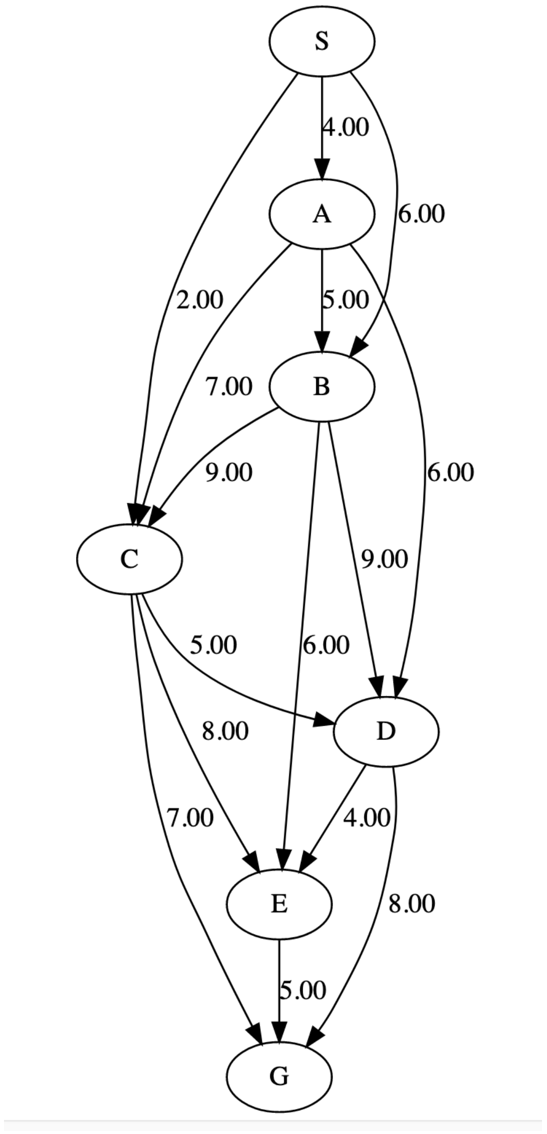
- If you define the explored set in the wrong way, then you might accidentally call these two the same state, and then your algorithm will never find an optimal solution.





Quiz

- Try the quiz: https://us.prairielearn.com/pl/course_instance/129874/assessment/2333250
- Here, "no explored set" means "do not prevent yourself from re-expanding nodes that you have already expanded."
- The variant at left ends with the following:
- Explored: {0:S,2:C,4:A,6:B,7:D,9:B,9:G}
- Frontier: [10:D,10:E,11:C,11:E, 12:E,15:C,15:D,15:E,15:G,18:C,18:D]



Outline

- Planning in a Known, Observable, Deterministic Environment
- Dijkstra’s algorithm: Frontier, Explored set
- Explored set: Avoid expanding the same state
- DFS, BFS, and UCS

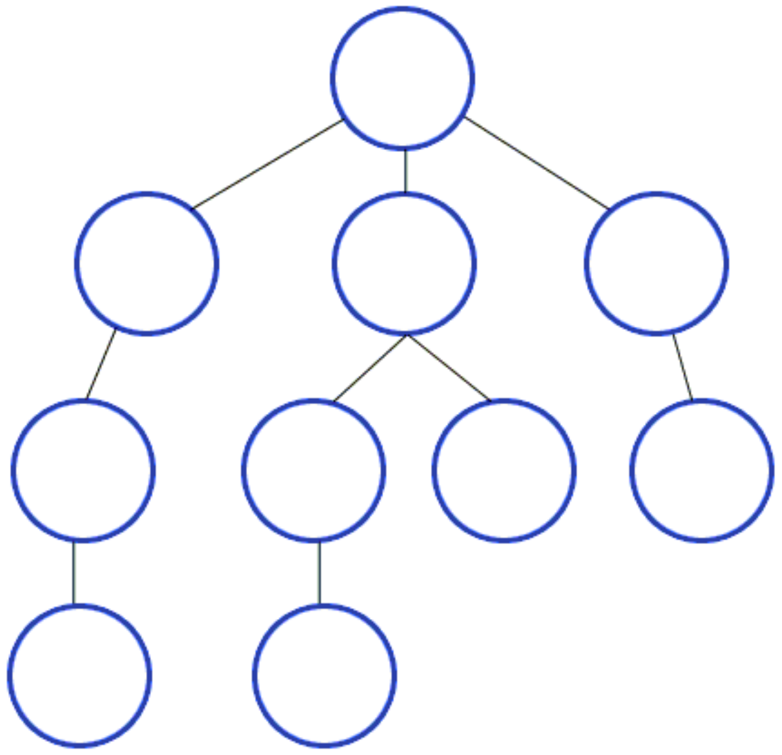
In which order should you pick nodes from the frontier?

- LIFO (last-in, first-out) = Depth-First Search (DFS):
  - the next node you expand will always be the one **most recently** added to the frontier.
- FIFO (first-in, first-out) = Breadth-First Search (BFS):
  - the next node you expand will always be the one **least recently** added to the frontier.
- PriorityQueue (lowest-cost, first-out) = Uniform Cost Search (UCS, a.k.a. Dijkstra's algorithm):
  - the next node you expand will always be the one with the **lowest cost**

Depth-first search (DFS)

Expand frontier in LIFO order (last in, first out).

Result: most recently discovered path is pursued, all the way to the end.



Depth-first-search. CC-BY-SA 3.0, Mre, 2009
https://commons.wikimedia.org/wiki/File:Depth-First-Search.gif

Analysis of search strategies

- Strategies are evaluated along the following criteria:
  - **Completeness:** does it always find a solution if one exists?
  - **Optimality:** does it always find a least-cost solution?
  - **Time complexity:** number of nodes generated
  - **Space complexity:** maximum number of nodes in memory
- Time and space complexity are measured in terms of
  - ***b**:* maximum branching factor of the search tree
  - ***d**:* depth of the optimal solution
  - ***m***: maximum length of any path in the state space (may be infinite)

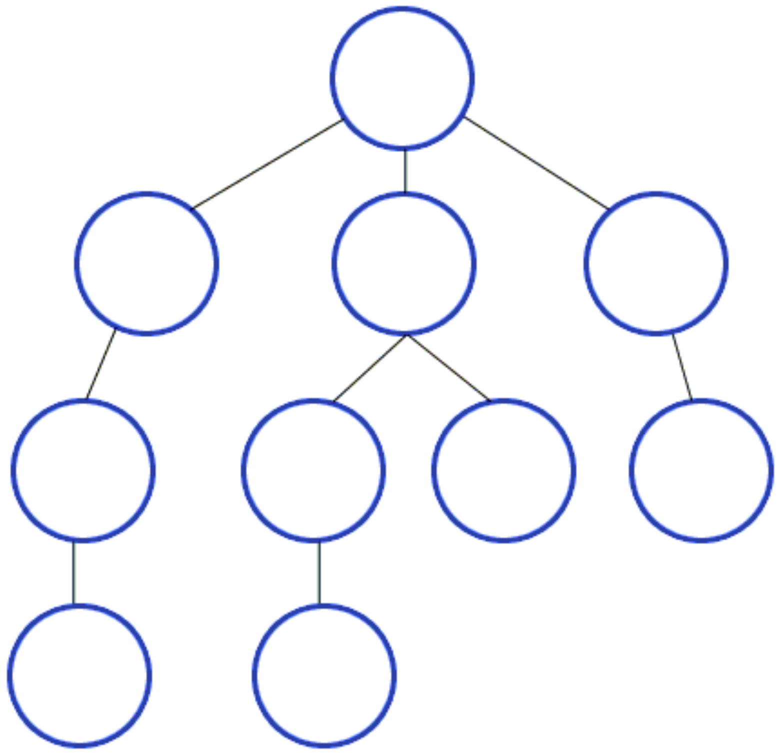
Depth-first search (DFS)

Incomplete: If there are an infinite number of states, DFS might go down a path of infinite length, and might never find a solution.

Suboptimal: DFS returns the first path it finds, which might not be the shortest path.

Time Complexity: $O\{b^m\}$, where m is the longest possible path length.

Space Complexity: only $O\{m\}$! Once you've finished a path, you can delete it from the tree!

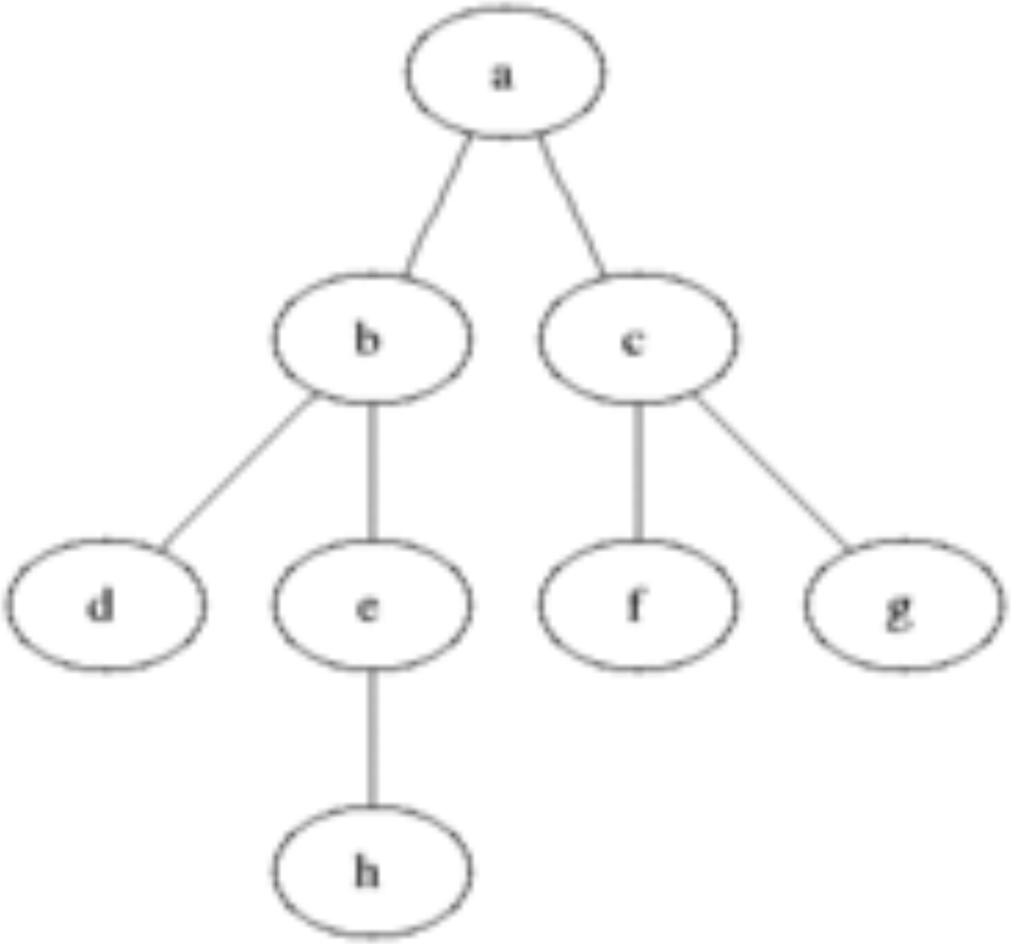


Depth-first-search. CC-BY-SA 3.0, Mre, 2009
https://commons.wikimedia.org/wiki/File:Depth-First-Search.gif

Breadth-first search (BFS)

Expand the frontier in FIFO order (first-in, first-out).

Result: all paths of length d are explored, then all paths of length d+1, and so on.



Animated-BFS. CC-SA 3.0, Blake Matheny, 2007
https://commons.wikimedia.org/wiki/File:Animated_BFS.gif

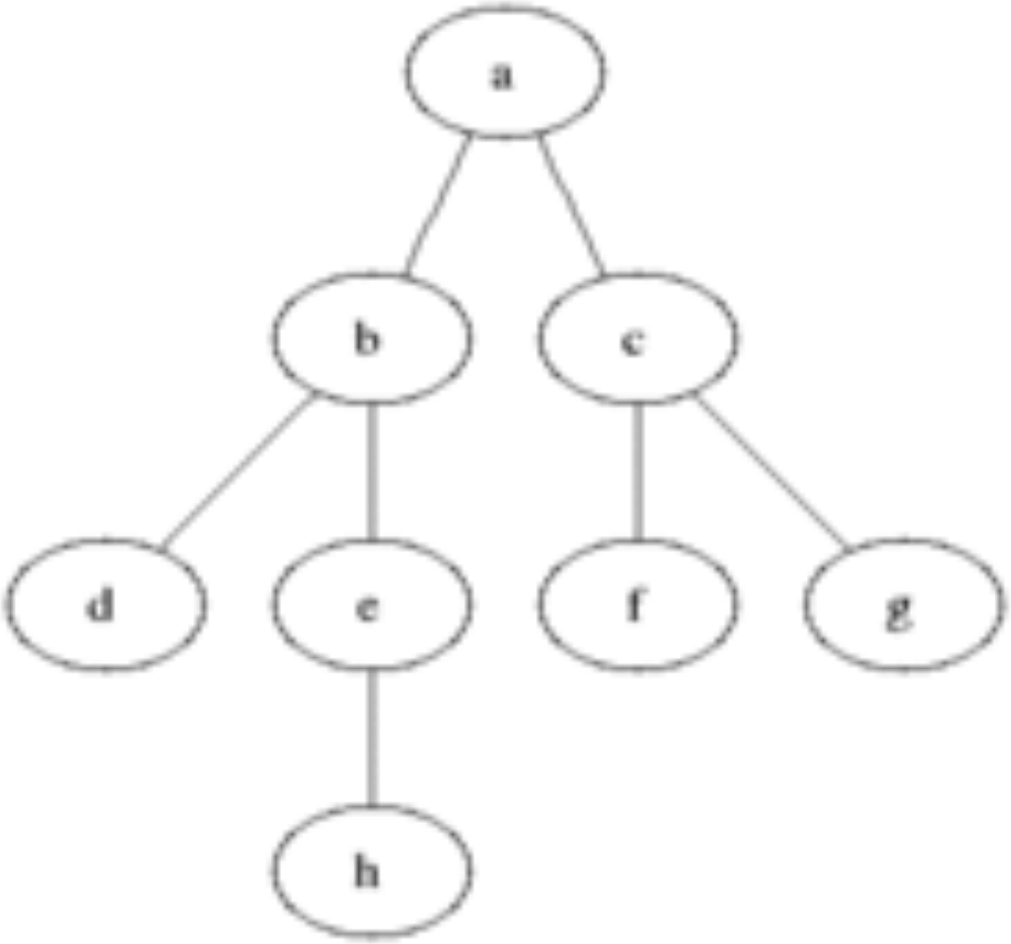
Breadth-first search (BFS)

Complete: if a finite-length path exists, BFS will find it.

Optimal: BFS returns the first solution it finds, which is always the shortest path.

Time Complexity: $O\{b^d\}$, where d is the length of the best path. This is usually much less than $O\{b^m\}$, because $d < m$.

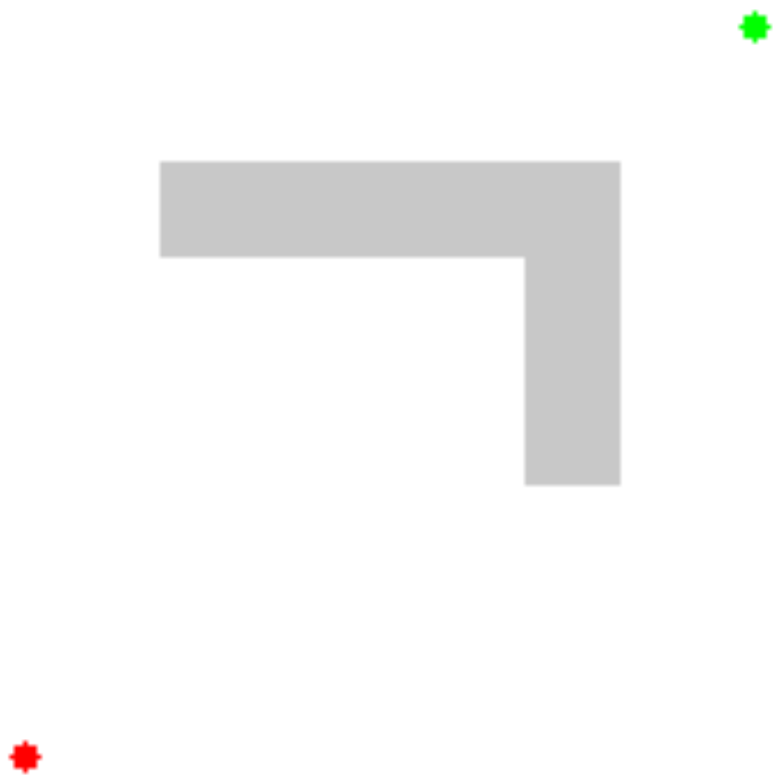
Space Complexity: $O\{b^d\}$. No part of the tree can be deleted until you've found the solution.



Animated-BFS. CC-SA 3.0, Blake Matheny, 2007
https://commons.wikimedia.org/wiki/File:Animated_BFS.gif

BFS = UCS with equal step sizes

- If every step has the same cost, then…
- …the lowest-cost node in the frontier is always the one that entered the frontier first.
- Therefore, if every step has the same cost, then UCS = BFS.


Dijkstra’s progress, CC-BY 3.0, Subh83, 2011
https://commons.wikimedia.org/wiki/File:Dijkstras_progress_animation.gif

Outline

- Planning in a Known, Observable, Deterministic Environment
- Dijkstra’s algorithm: Frontier, Explored set
- Explored set: Avoid expanding the same state
- DFS, BFS, and UCS