UNIVERSITÀ DEGLI STUDI DI NAPOLI PARTHENOPE

Artificial Intelligence

Search

LESSON 3

prof. Antonino Staiano

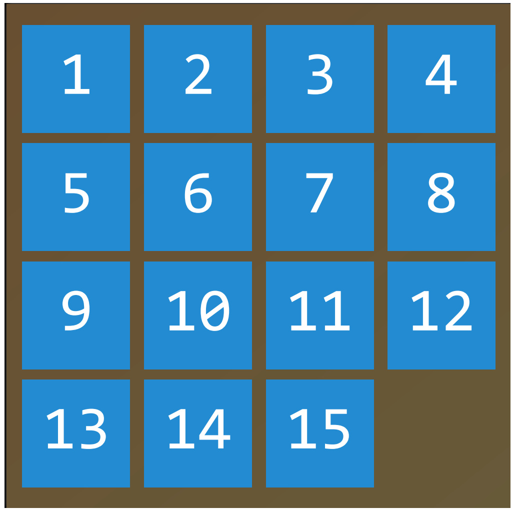
M.Sc. In "Machine Learning e Big Data" - University Parthenope of Naples

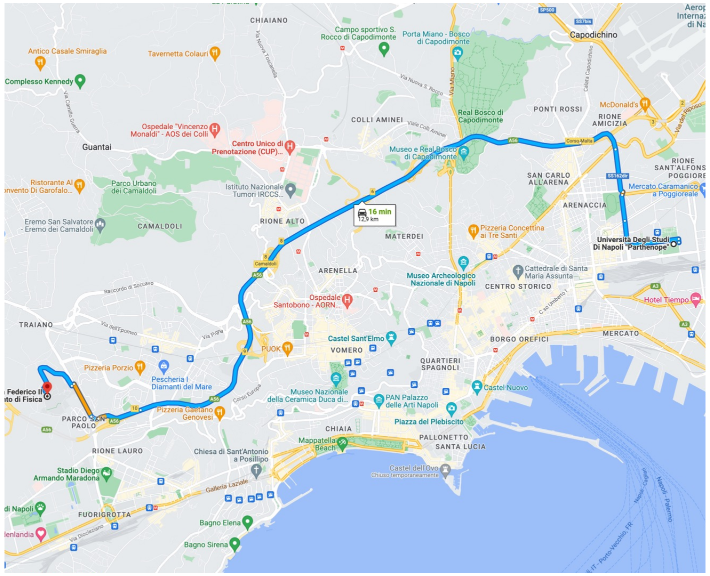
What a Search Problem is

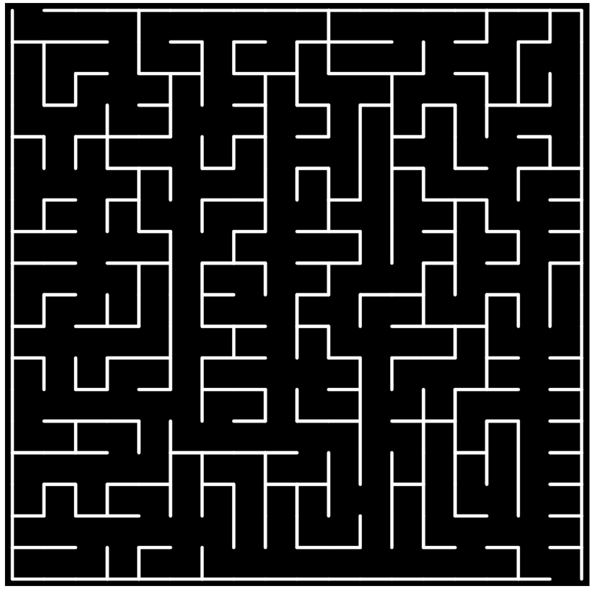
- Search problems involve an agent that
  - is given an initial state and a goal state
  - returns a solution of how to get from the former to the latter
- Example
  - A navigator app uses a typical search process, where the agent (the thinking part of the program) receives your current location and desired destination as input and returns a suggested path based on a search algorithm
- However, many other search problems exist, like puzzles or mazes

Designing Agents for Search Problems

- Consider the following problems, and assume that your goal is to design a rational agent (assume a computer program) capable of autonomously solving them
- Let's recall
  - A rational agent is a system that acts rationally, according to a well-defined objective







Missionaries and Cannibals

- A classical AI toy-problem
  - 3 missionaries and 3 cannibals on one side of a river
  - Goal: cross the river on a boat (or raft) to reach the other side of the river
  - Constraints
    - The boat can only hold two people
    - Do not leave more cannibals than missionaries on either side of the river
  - How can all six get across the river safely?

Game playing: 15-puzzle

- An array of tiles numbered from 1 to 15 and an empty cell
- Goal:
  - Transform the tiles from an initial configuration into a given desired configuration, by a sequence of moves of a tile into an adjacent empty cell
- A more challenging goal
  - Find the shortest of such sequences
- Example

13	10	11	6
5	7	4	8
1		14	9
3	15	2	12

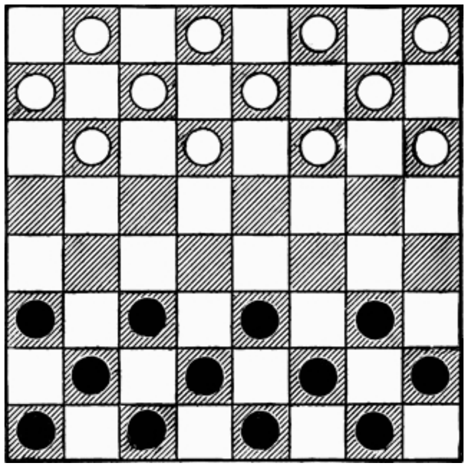
initial configuration

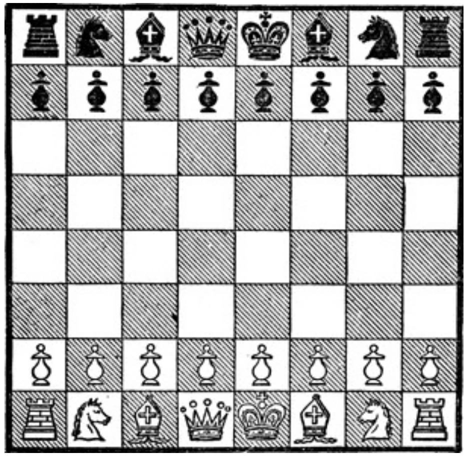
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

desired configuration

Game playing: checkers and chess

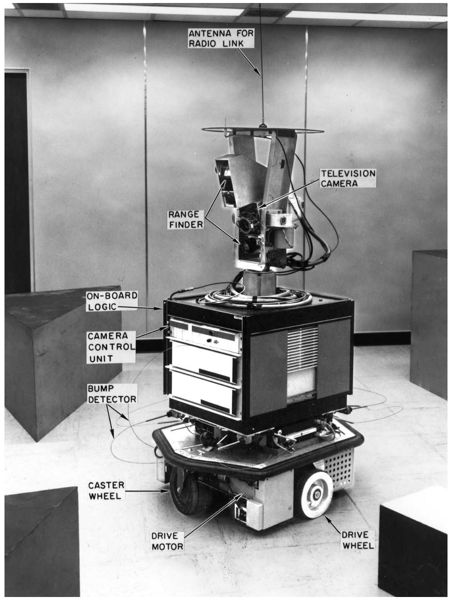
- Two historical problems addressed by many researchers since the early days of AI

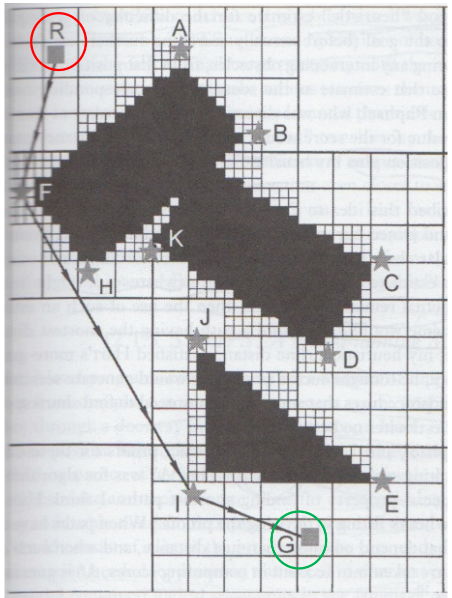




Robot navigation

- A real-world problem addressed since the '60s

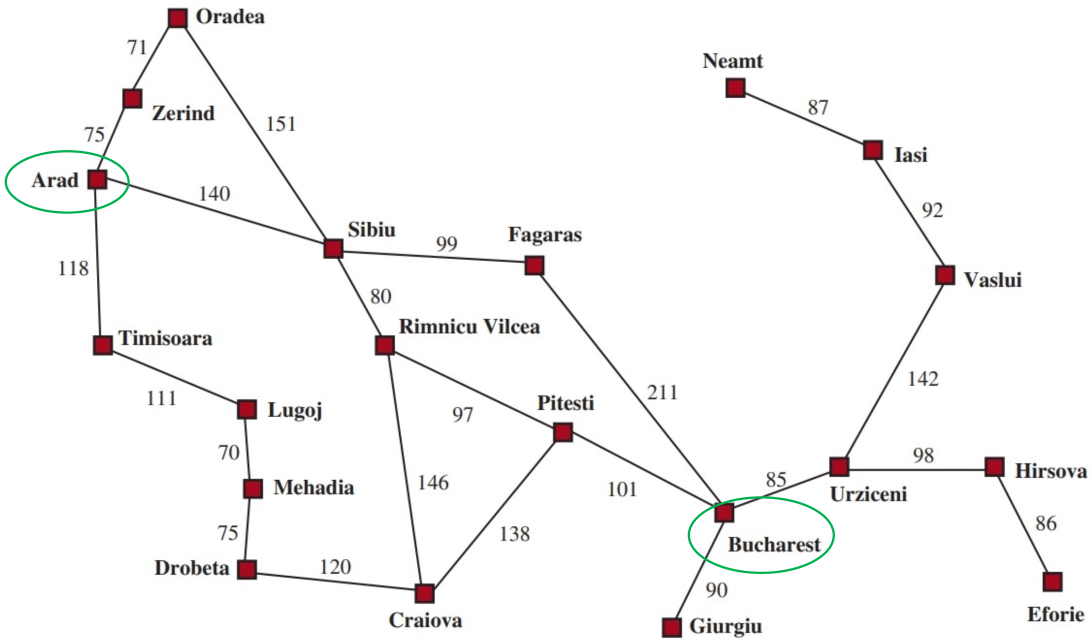




- A robot (left) and a problem to solve (right)
  - Find a route from R to G, possibly the shortest one, avoiding the black obstacles

Route finding in maps

- Example
  - Finding a route (more challenging, the shortest one) from Arad to Bucharest using the information shown on the map

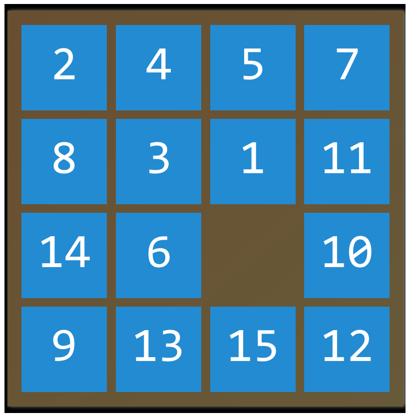


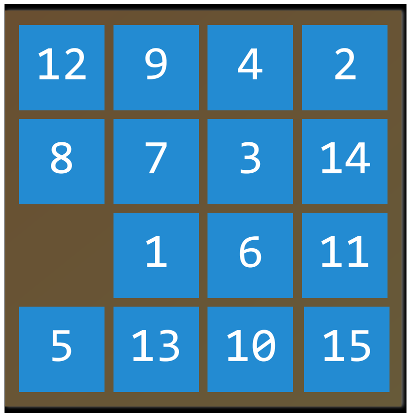
Searching problems

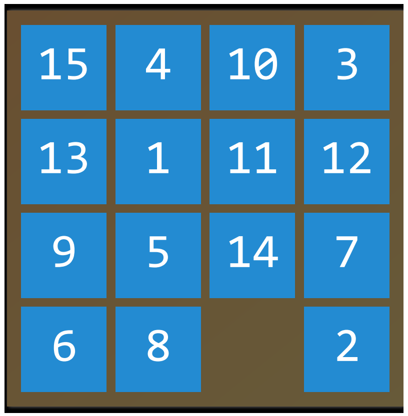
- The previous problems may seem very different from each other, nonetheless, they share some common characteristics allowing one to solve them using the same approach
- Main characteristic
  - A clear goal can be defined in terms of desired world states
- Having the goal, the task is to search for a sequence of actions leading to a goal state
- It requires suitably defining the actions and the states to be considered
- The solution to a problem is a sequence of actions that lead to a goal state
  - The process of looking for a solution is called search

Problem ingredients

- Agent
    - Entity that perceives its environment and acts upon that environment
- State
    - A configuration of the agent and its environment



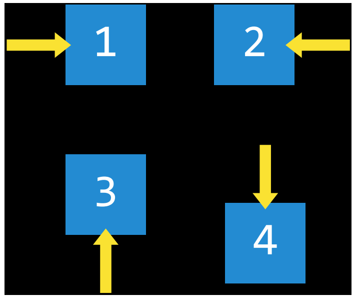




- Initial state
    - The state from which the search algorithm starts

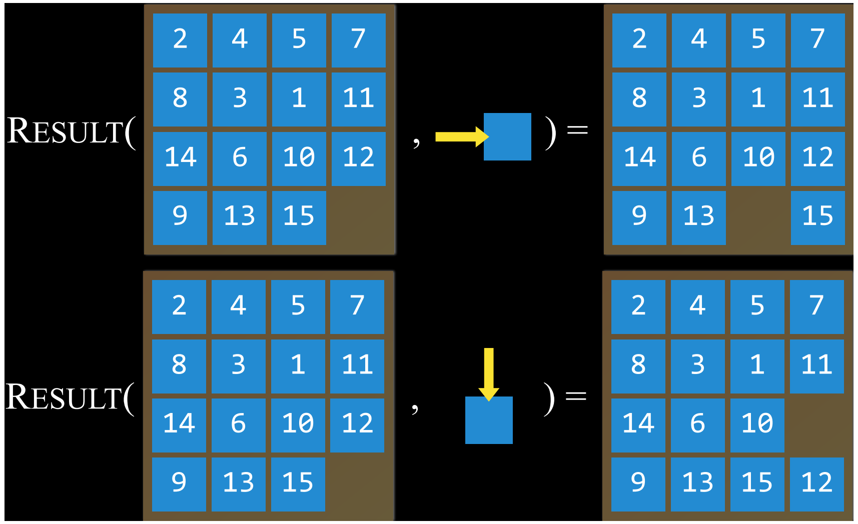
Problem ingredients

- Actions
  - Choices that can be made in a state
- Actions can be defined as a function
  - ACTIONS(s) returns the set of actions that can be executed in a state s



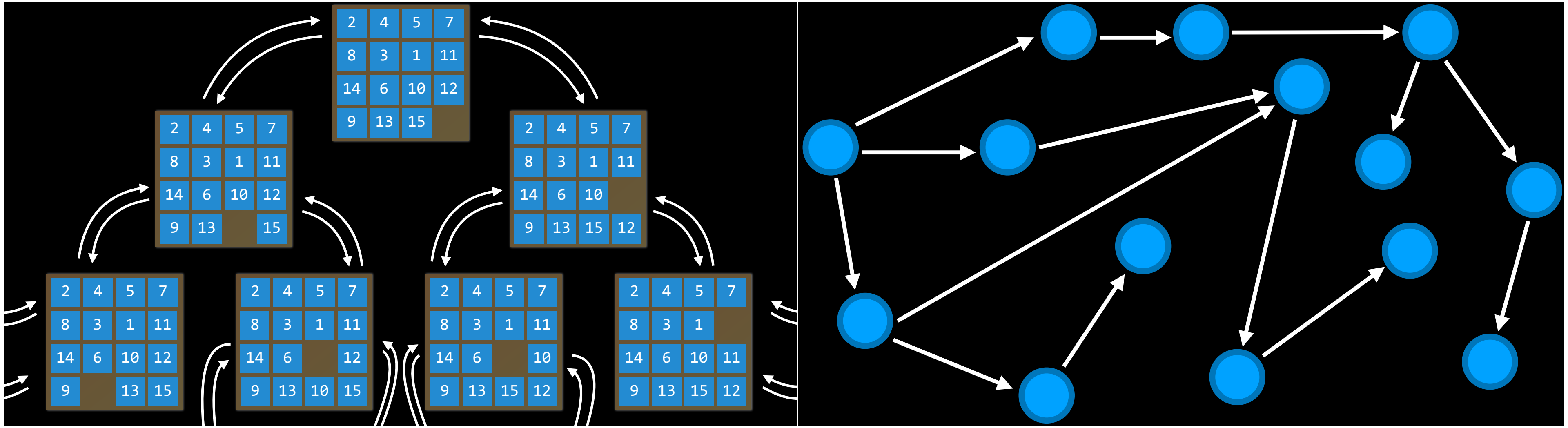
Problem ingredients

- Transition model
  - A description of what state results from performing any applicable action in any state
- Defined as a function
  - RESULTS(s,a) returns the state resulting from performing action a in state s



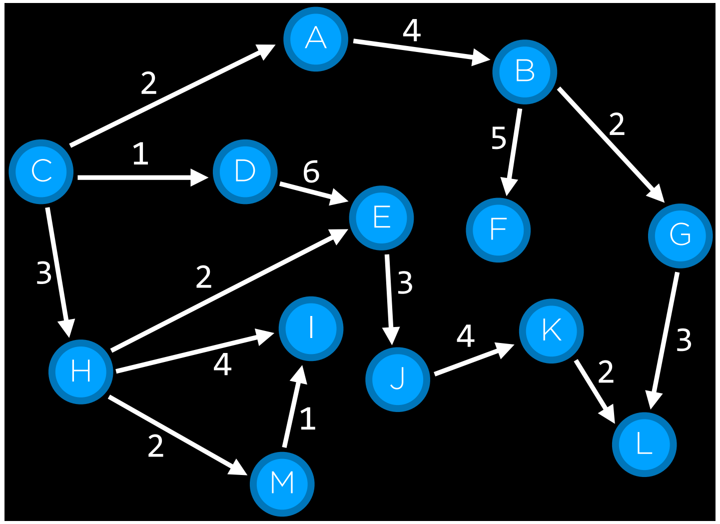
Problem ingredients

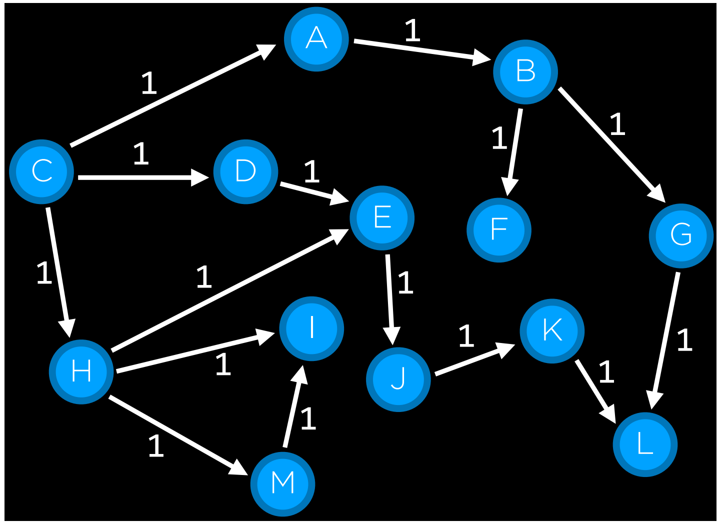
- State space
  - The set of all states reachable from the initial state by any sequence of actions
    - In a 15 puzzle, the state space consists of all the 16!/2 configurations on the board that can be reached from any initial state
    - The state space can be visualized as a directed graph with states, represented as nodes, and actions represented as arrows between nodes



Problem ingredients

- Goal test
  - Way to determine whether a given state is a goal state
- Path cost
  - Numerical cost associated with a given path





Example: 15-puzzle

- goal:
  - getting to the desired tile configuration (possibly, by the shortest sequence of moves)
- states:
  - each possible 16!/2 tile configurations
- actions:
  - moving the *n*-th tile (n = 1,...,15) to one of the adjacent cells (two, three or four), if empty

13	10	11	6
5	7	4	8
1		14	9
3	15	2	12

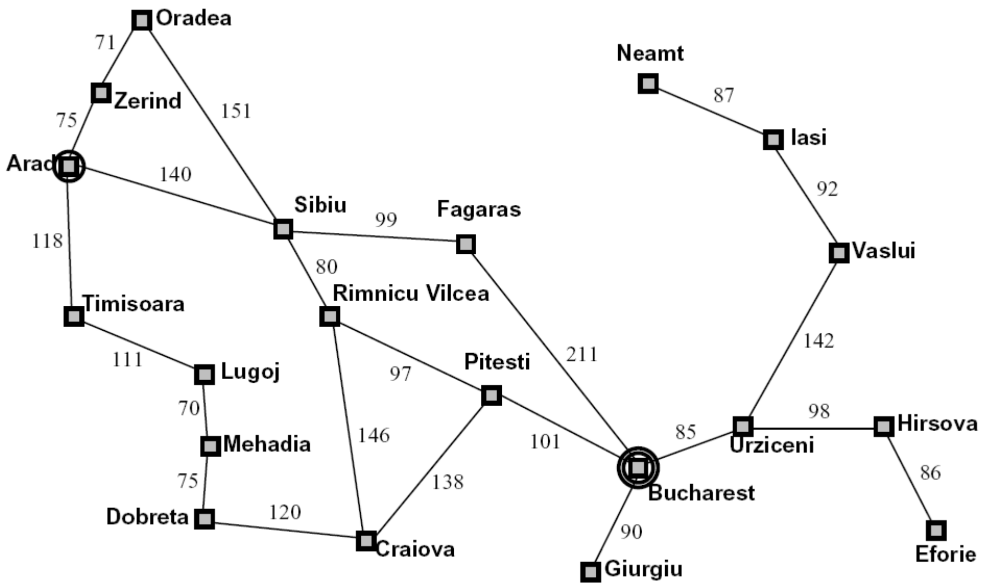
initial configuration

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

desired configuration

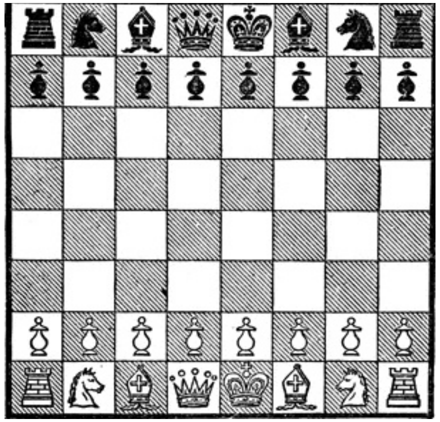
Example: Route finding on maps

- goal:
  - getting from a given city to a destination (possibly, through the shortest route)
- states:
  - Being in each possible city
- actions:
  - Moving between two adjacent cities



Example: Chess

- Goal:
  - To checkmate (possible in many chessboard configurations)
- States:
  - Each possible chessboard configuration
- Actions:
  - All legal moves



Properties of Search Problems

- Static vs dynamic
  - does the environment change over time? Examples: 15-puzzle and chess are static; robot navigation is dynamic if the position of obstacles changes over time
- Fully vs partially observable:
  - is the current state completely known? Examples: 15-puzzle and chess are fully observable; robot navigation is partially observable if sensors are not "perfect"
- Discrete vs continuous sets of states and actions
  - Examples: 15-puzzle and chess are discrete, robot navigation is continuous
- Deterministic vs non-deterministic
  - is the outcome (the resulting state) of any sequence of actions certain. i.e., known in advance? Examples: 15-puzzle is deterministic, chess is not (due to the opponent's move, which is unknown when deciding one's own)

Real-world Search Problems

- Many challenging real-world problems can be formulated as search problems
  - Traveling salesperson problem
    - Finding the shortest tour that allows one to visit every city on a given map exactly once
  - Route-finding
    - In computer networks airline travel planning, etc.
  - VLSI design
    - Cell layout, channel routing

Solving Search Problems

- Solution
  - A sequence of actions that leads from the initial state to a goal state
- Optimal solution
  - A solution that has the lowest path cost among all solutions

Data structures

- In a search process, data is often stored in a node
- Node
  - a data structure that keeps track of
    - A state
    - Its parent node, through which the current node was generated
    - The action that was applied to the state of the parent to get to the current node
    - The path cost from the initial state to this node
- Frontier
  - A mechanism that manages the nodes, that is, the set of nodes to be explored
  - The frontier starts by containing an initial state and an empty set of explored items

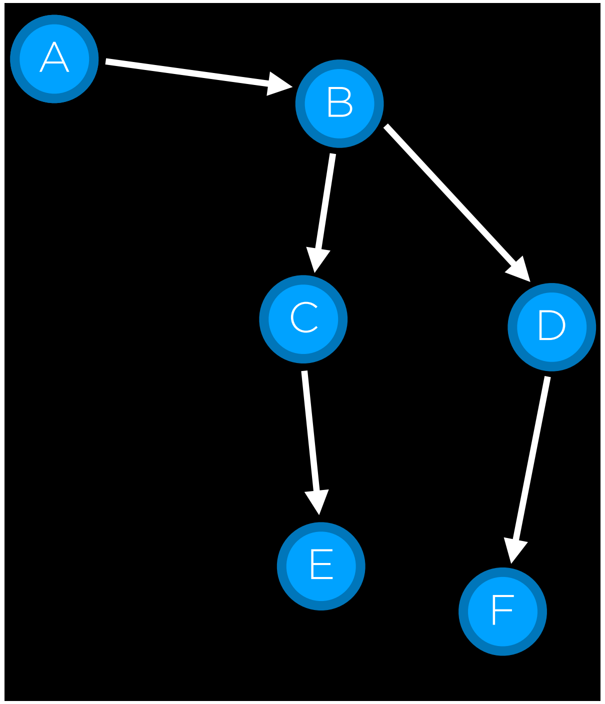
Approach

- Start with a frontier that contains the initial state
- Repeat:
  - If the frontier is empty, then stop, there is no solution
  - Remove a node from the frontier
  - If node contains the goal state, return the solution and stop
    - Else expand node, add resulting nodes to the frontier

Example: Find a path from A to E

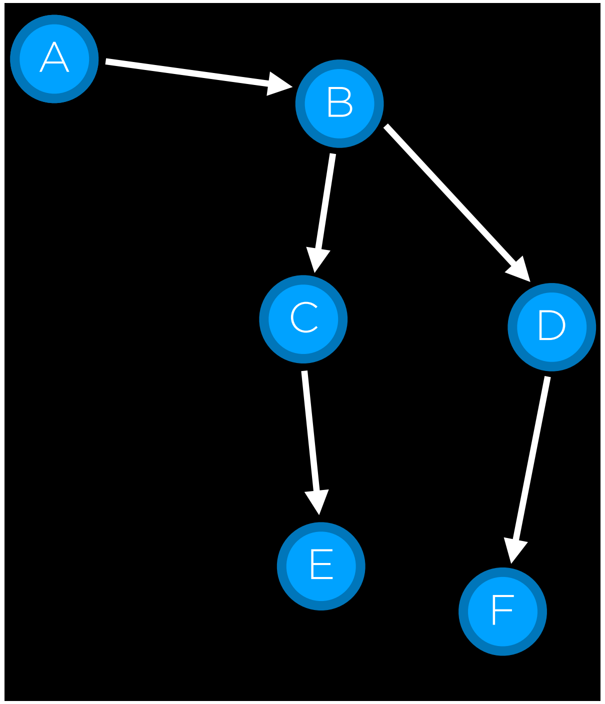
- Start with a frontier that contains the initial state
- Repeat:
  - If the frontier is empty, then no solution
  - Remove a node from the frontier
  - If node contains goal state, return the solution
  - Expand node, add resulting nodes to the frontier

Frontier



Example: Find a path from A to E

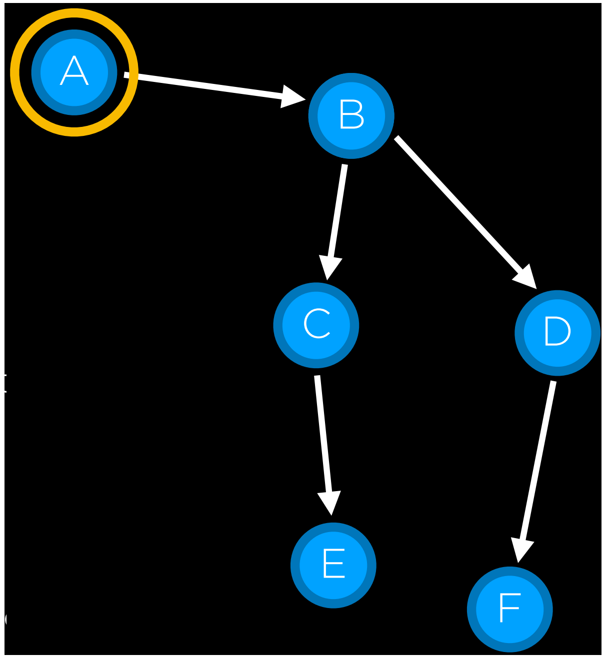
- Start with a **frontier** that contains the initial state
- Repeat:
  - If the frontier is empty, then no solution
  - Remove a node from the frontier
  - If node contains goal state, return the solution
  - Expand node, add resulting nodes to the frontier



Example: Find a path from A to E

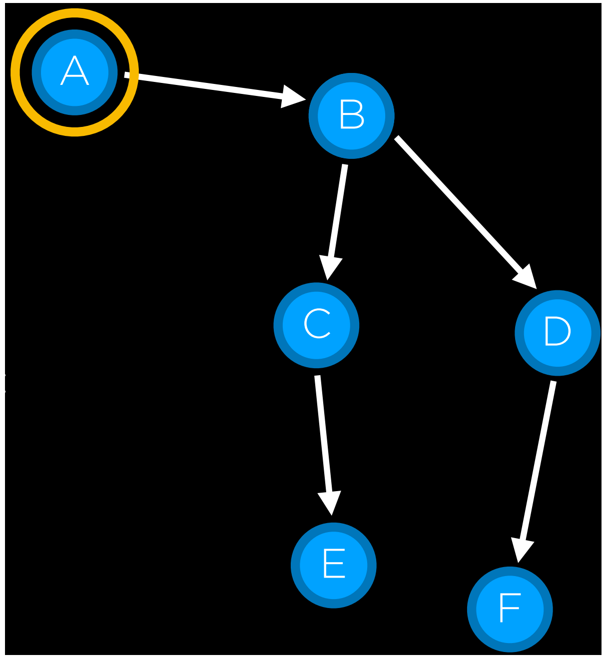
- Start with a frontier that contains the initial state
- Repeat:
  - If the frontier is empty, then no solution
  - Remove a node from the frontier
  - If node contains goal state, return the solution
  - Expand node, add resulting nodes to the frontier

Frontier



Example: Find a path from A to E

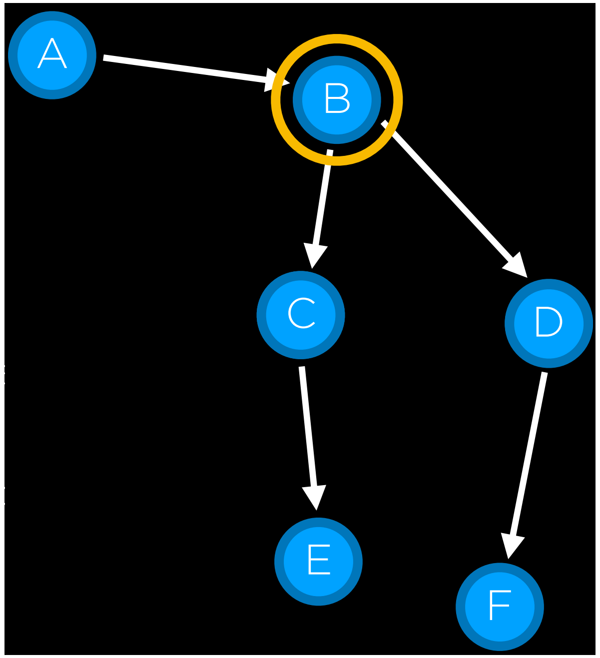
- Start with a frontier that contains the initial state
- Repeat:
  - If the frontier is empty, then no solution
  - Remove a node from the frontier
  - If node contains goal state, return the solution
  - Expand node, add resulting nodes to the frontier



Example: Find a path from A to E

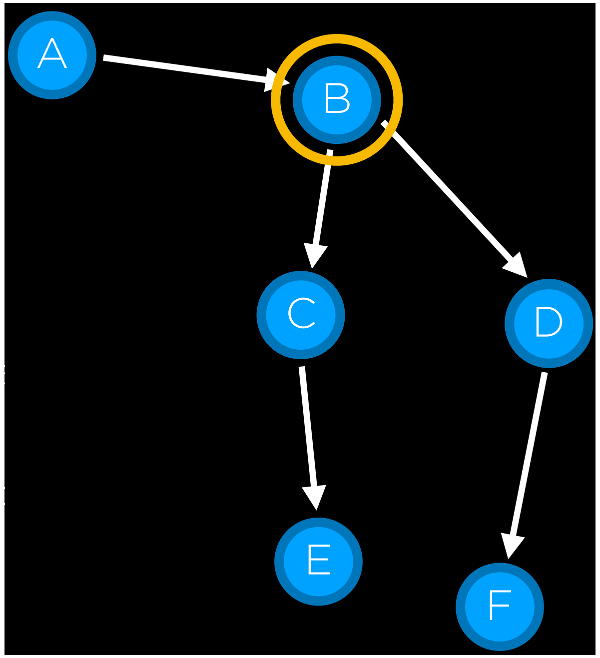
- Start with a frontier that contains the initial state
- Repeat:
  - If the frontier is empty, then no solution
  - Remove a node from the frontier
  - If node contains goal state, return the solution
  - Expand node, add resulting nodes to the frontier

Frontier



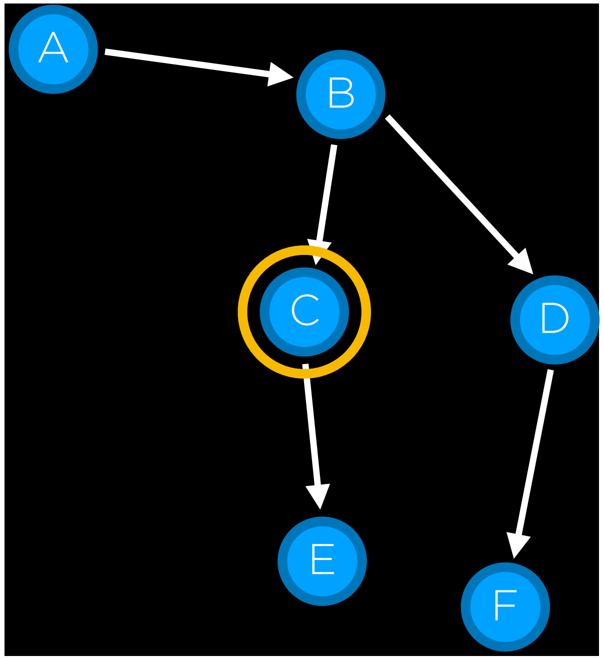
Example: Find a path from A to E

- Start with a frontier that contains the initial state
- Repeat:
    - If the frontier is empty, then no solution
    - Remove a node from the frontier
    - If node contains goal state, return the solution
    - Expand node, add resulting nodes to the frontier



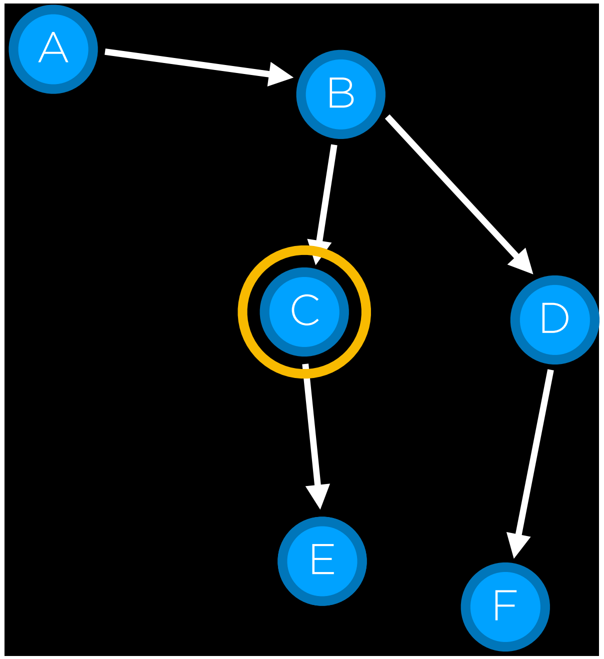
Example: Find a path from A to E

- Start with a frontier that contains the initial state
- Repeat:
  - If the frontier is empty, then no solution
  - Remove a node from the frontier
  - If node contains goal state, return the solution
  - Expand node, add resulting nodes to the frontier



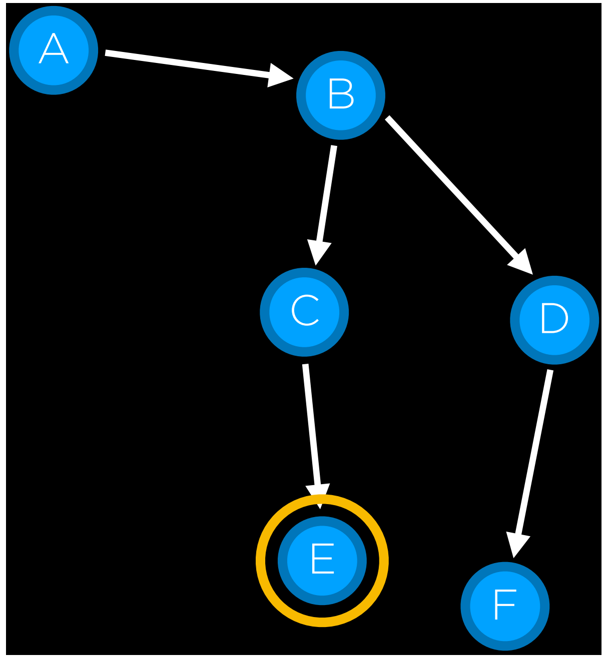
Example: Find a path from A to E

- Start with a frontier that contains the initial state
- Repeat:
  - If the frontier is empty, then no solution
  - Remove a node from the frontier
  - If node contains goal state, return the solution
  - Expand node, add resulting nodes to the frontier



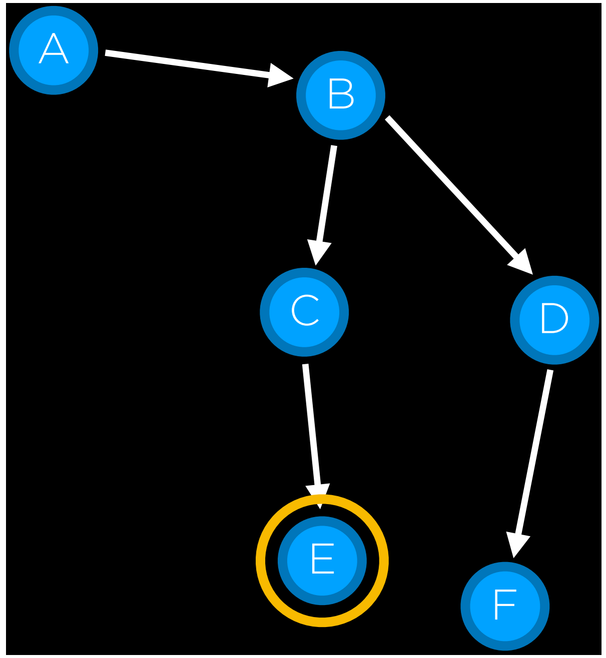
Example: Find a path from A to E

- Start with a frontier that contains the initial state
- Repeat:
  - If the frontier is empty, then no solution
  - Remove a node from the frontier
  - If node contains goal state, return the solution
  - Expand node, add resulting nodes to the frontier



Example: Find a path from A to E

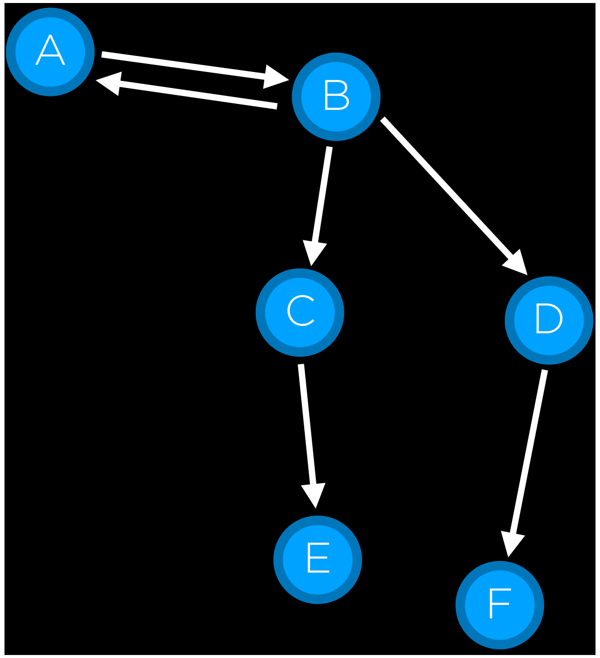
- Start with a frontier that contains the initial state
- Repeat:
  - If the frontier is empty, then no solution
  - Remove a node from the frontier
  - If node contains goal state, return the solution
  - Expand node, add resulting nodes to the frontier



Any problem here?

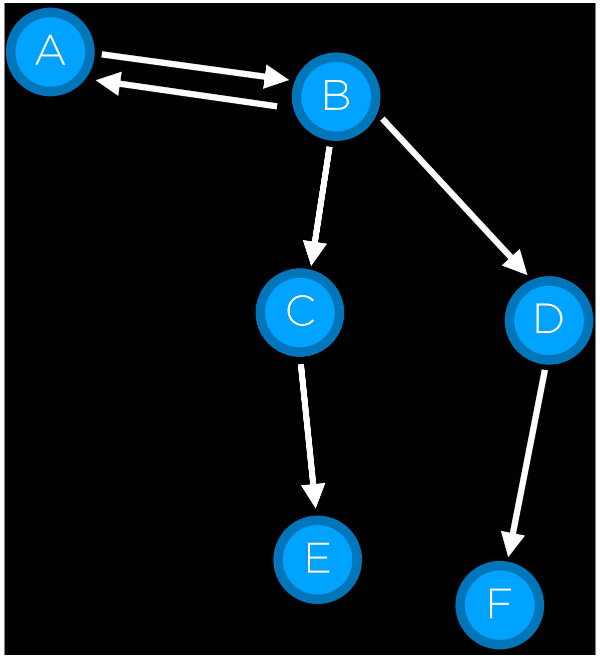
- Find a path from A to E

Frontier



Any problem here?

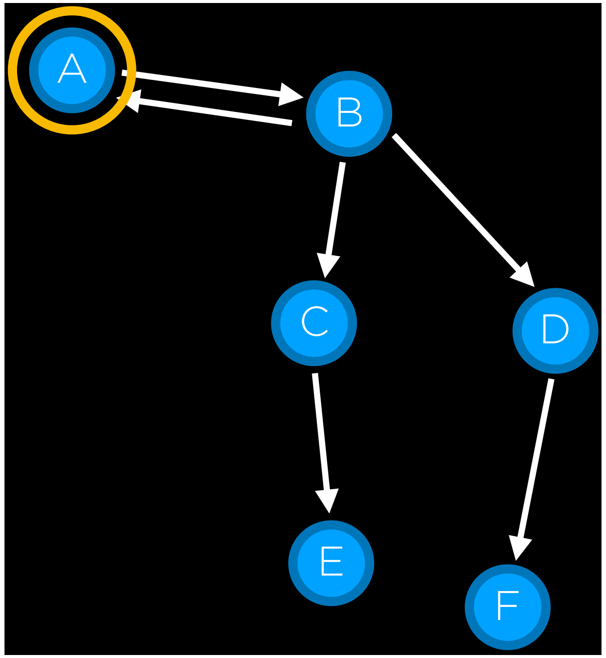
- Find a path from A to E



Any problem here?

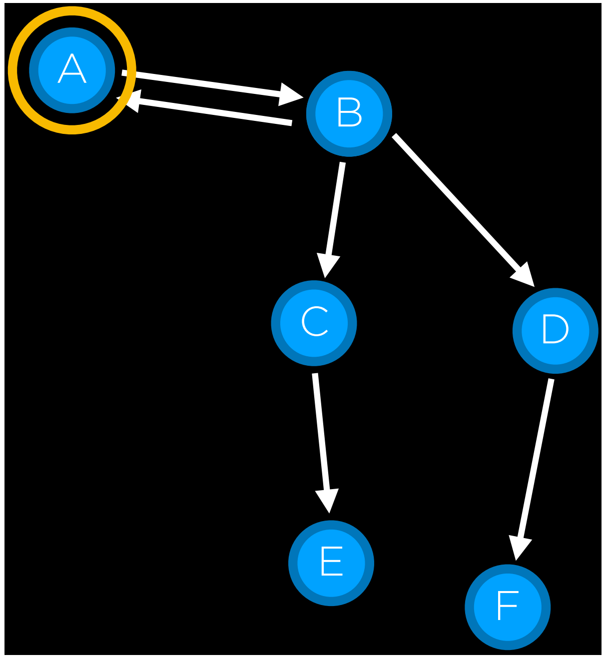
- Find a path from A to E

Frontier



Any problem here?

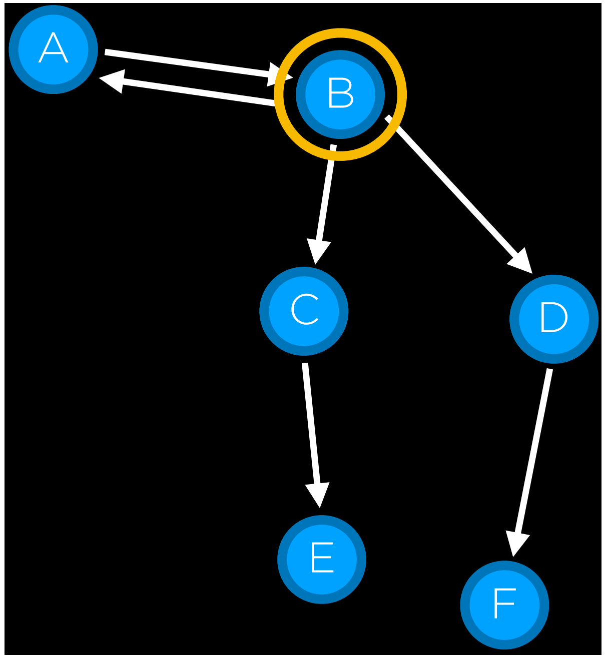
- Find a path from A to E



Any problem here?

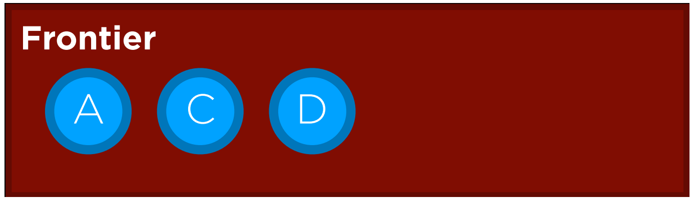
- Find a path from A to E

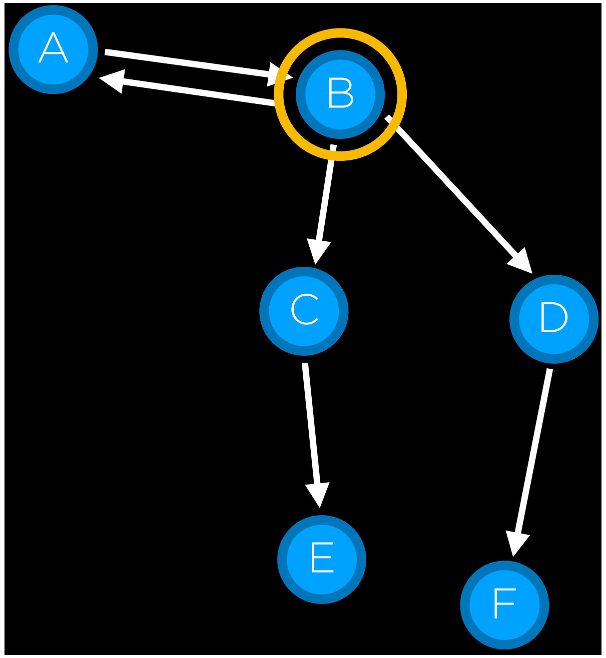
Frontier



Any problem here?

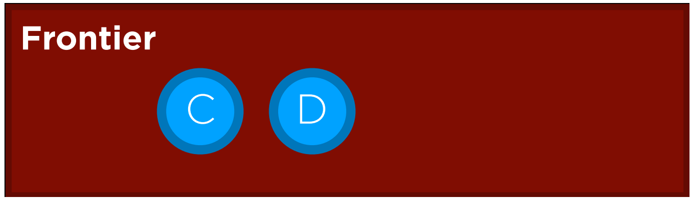
- Find a path from A to E

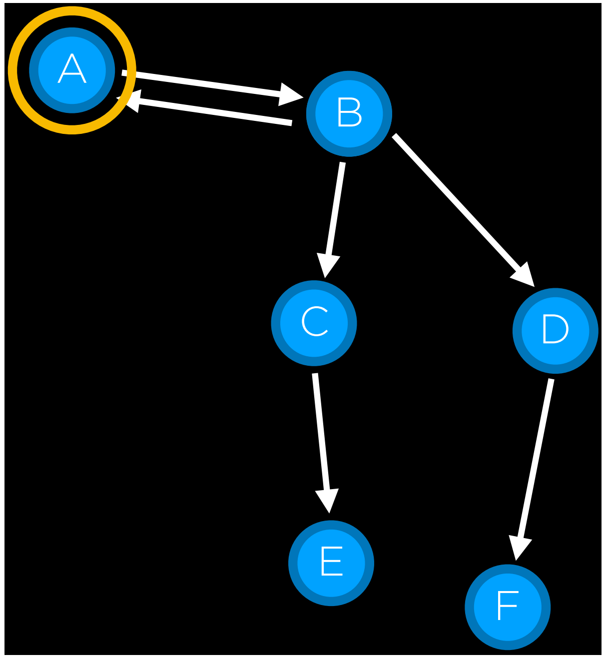




Any problem here?

- Find a path from A to E





A Cleaver Approach

- Start with a frontier that contains the initial state
- Start with an empty explored set
- Repeat:
  - If the frontier is empty, then no solution
  - Remove a node from the frontier
  - If a node contains goal state, return solution
  - Add the node to the explored set
  - Expand node, add resulting nodes to the frontier if they aren't already in the frontier or the explored set

Which node should be removed from the frontier?

- The choice of the nodes to be removed impacts the quality of the solution and how fast it is achieved
- There are multiple ways to choose, two of which can be represented by the data structures of
  - stack (in *depth-first* search) and
  - queue (in *breadth-first search*)

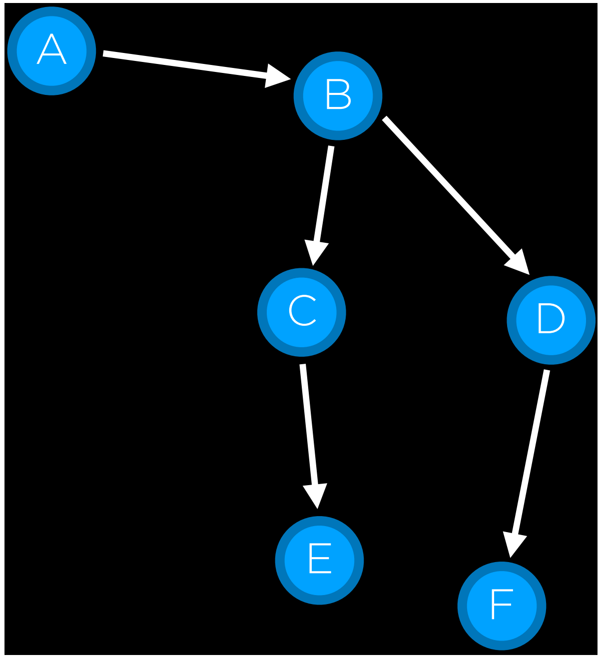
Depth-First Search

- A *depth-first* search algorithm exhausts every single direction before trying another direction
- In these cases, the frontier is managed as a *stack* data structure
  - *last-in first-out* mode
- After nodes are added to the frontier, the first node to be removed and considered is the last node added
- This results in a search algorithm that goes as deep as possible in the first direction that gets in its way while leaving all other directions for later

Example: Find a path from A to E

Frontier

Explored Set

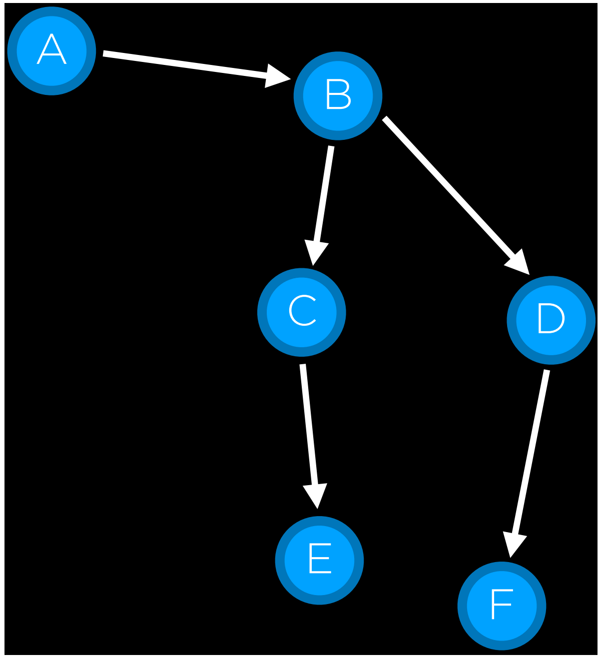


Example: Find a path from A to E

Frontier

A

Explored Set

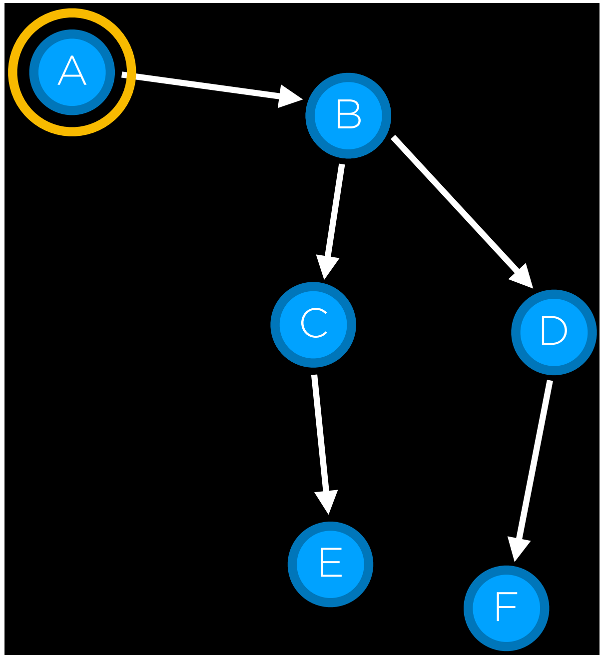


Example: Find a path from A to E

Frontier

Explored Set

A



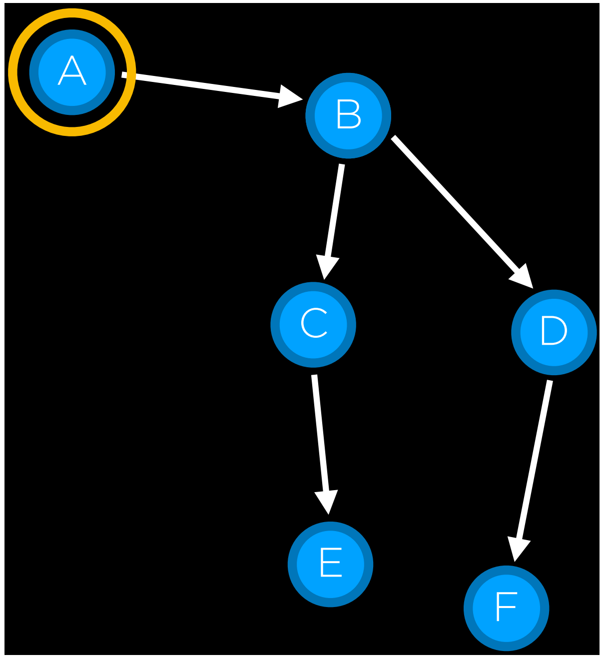
Example: Find a path from A to E

Frontier

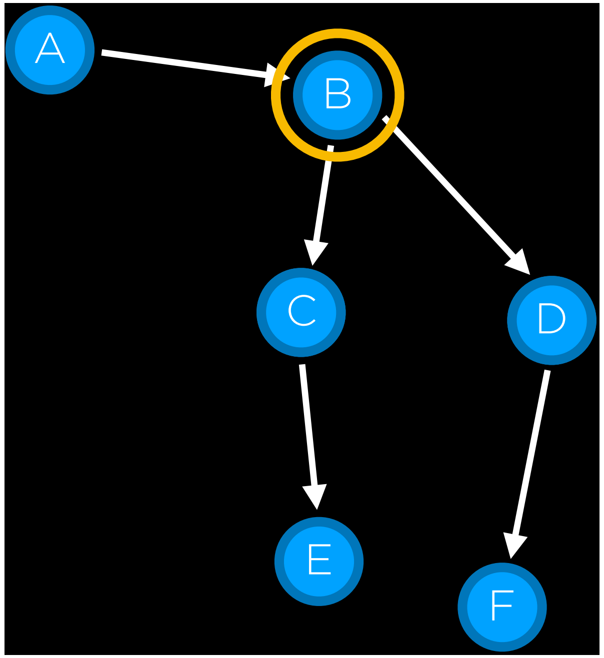
B

Explored Set

A



Example: Find a path from A to E



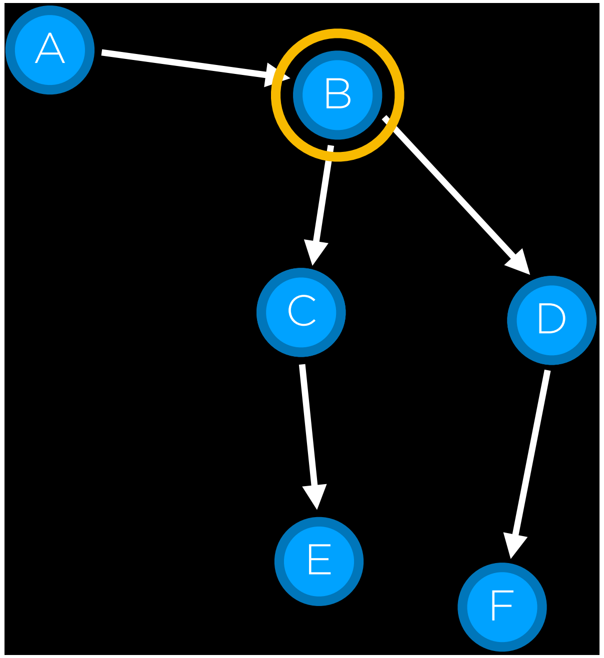
Example: Find a path from A to E

Frontier

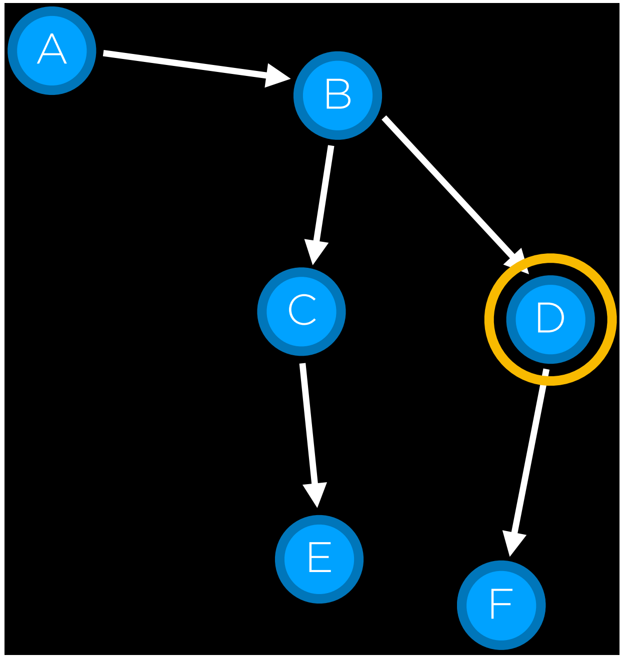
C D

Explored Set

A B



Example: Find a path from A to E



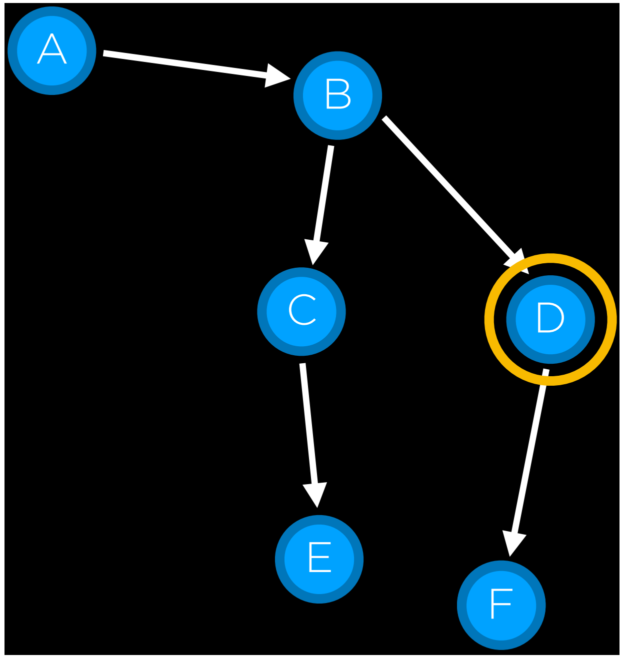
Example: Find a path from A to E

Frontier

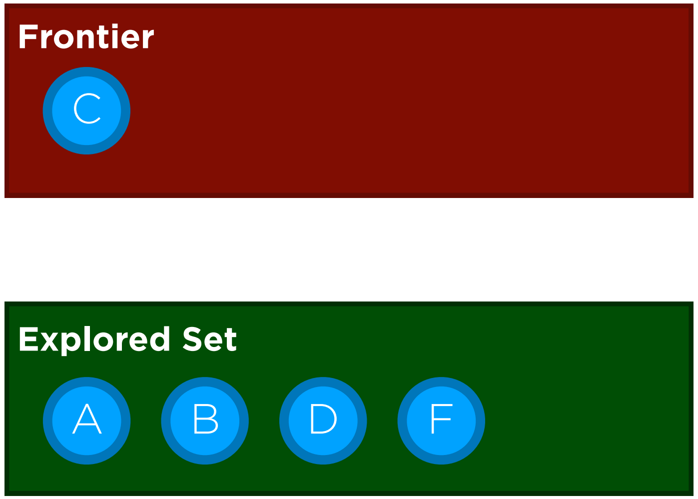
C F

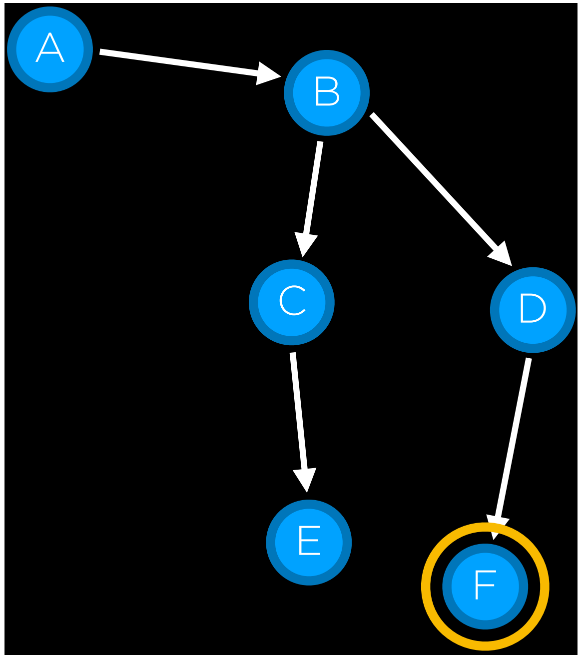
Explored Set

A B D

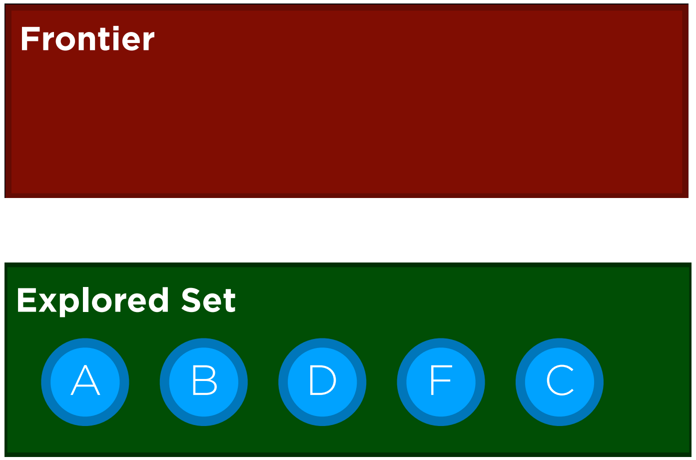


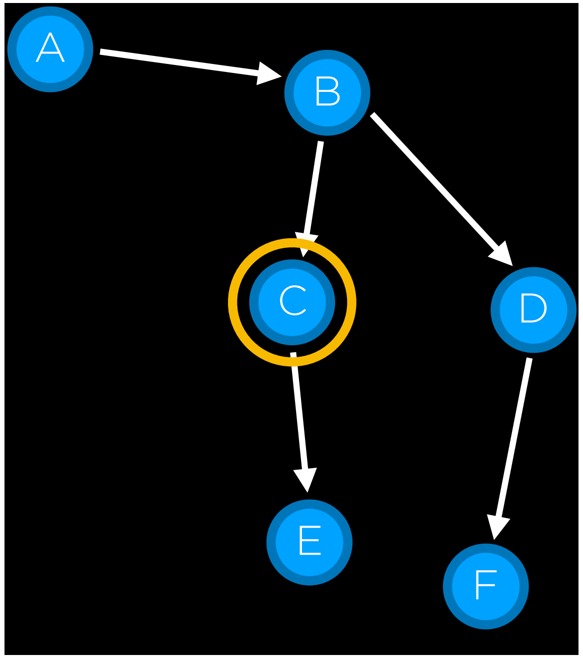
Example: Find a path from A to E





Example: Find a path from A to E





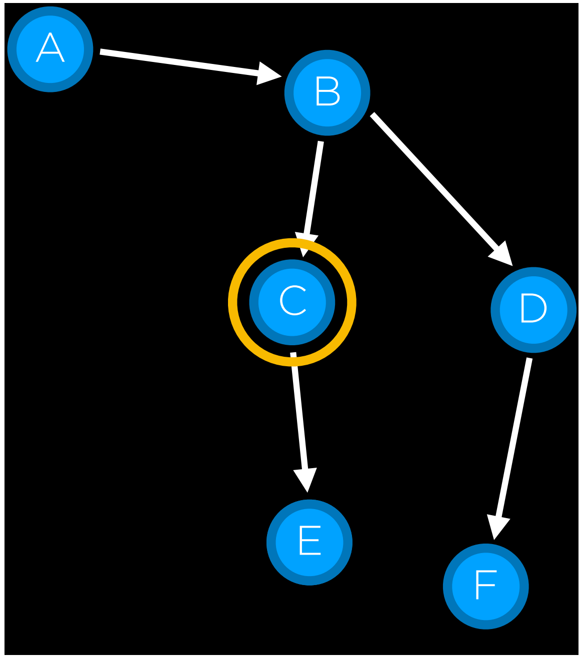
Example: Find a path from A to E

Frontier

E

Explored Set

A B D F C

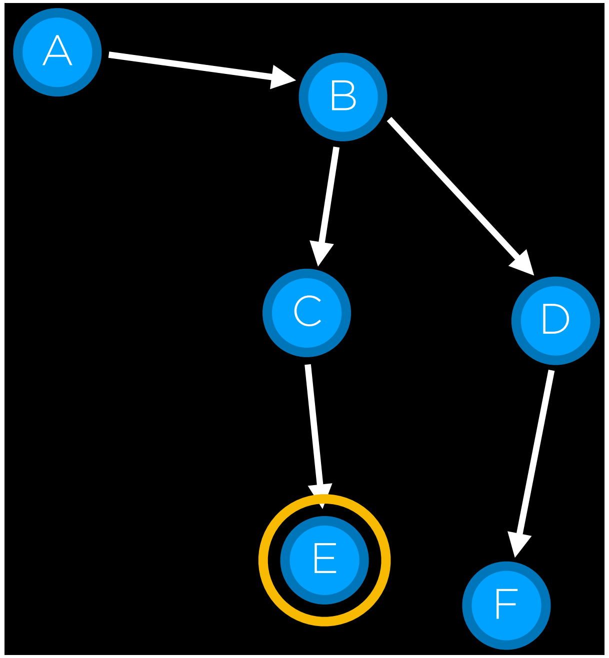


Example: Find a path from A to E

Frontier

Explored Set

A B D F C



Depth-First Search

- Pros
  - At best, this algorithm is the fastest
    - If it “lucks out” and always chooses the right path to the solution (by chance), then a *DFS* takes the least possible time to get to a solution
- Cons
  - It is possible that the found solution is not optimal
  - At worst, this algorithm will explore every possible path before finding the solution, thus taking the longest possible time before reaching the solution

Depth-First Search Code Example

```
# Define the function that removes a node from the frontier and returns it.
def remove(self):
      # Terminate the search if the frontier is empty, because this means that there is no solution.
    if self.empty():
        raise Exception("empty frontier")
    else:
              # Save the last item in the list (which is the newest node added)
        node = self.frontier[-1]
        # Save all the items on the list besides the last node (i.e. removing the last node)
        self.frontier = self.frontier[:-1]
        return node
```

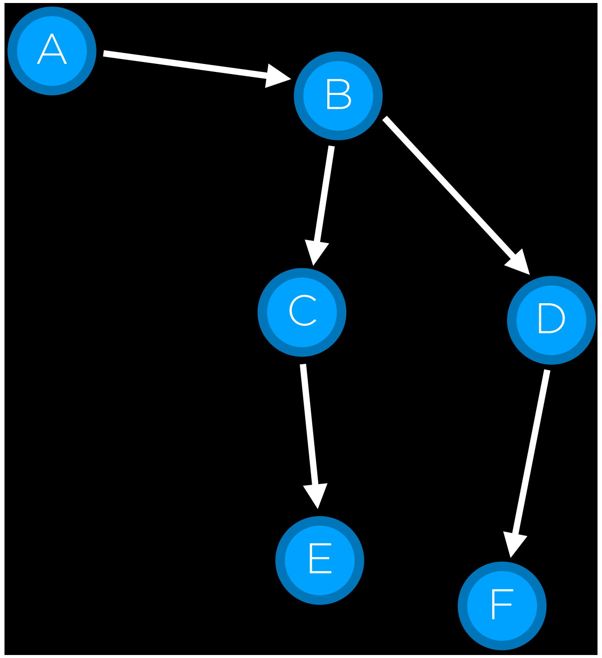
Breadth-First Search

- The opposite of DFS
- A *BFS* algorithm will follow multiple directions at the same time, taking one step in each possible direction before taking the second step in each direction
- In this case, the frontier is managed as a *queue* data structure
  - *first-in first-out* mode
- All the new nodes add up in line, and nodes are being considered based on which one was added first (first come first served!)
- This results in a search algorithm that takes one step in each possible direction before taking a second step in any one direction

Example: Find a path from A to E

Frontier

Explored Set

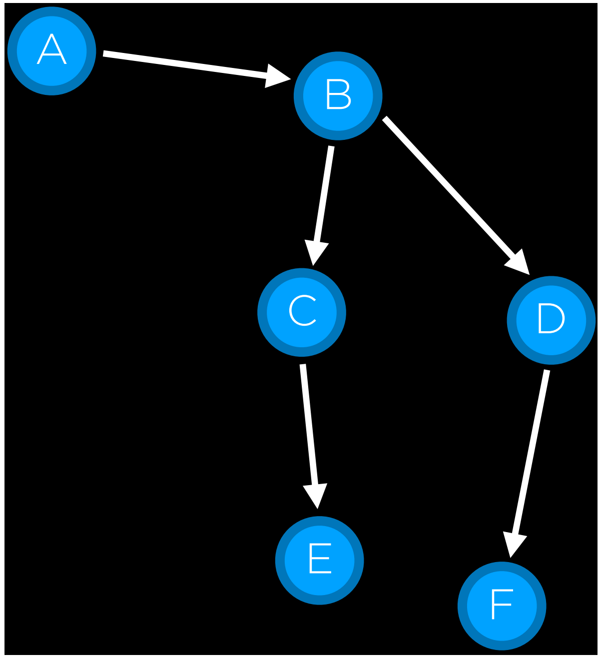


Example: Find a path from A to E

Frontier

A

Explored Set

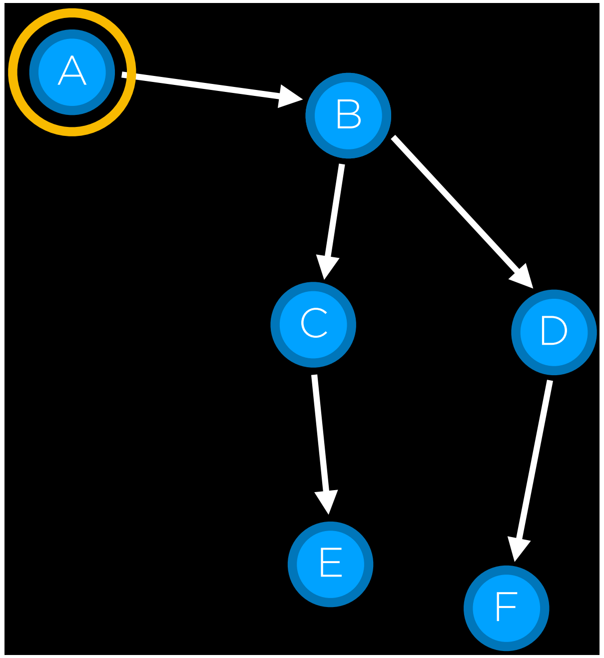


Example: Find a path from A to E

Frontier

Explored Set

A



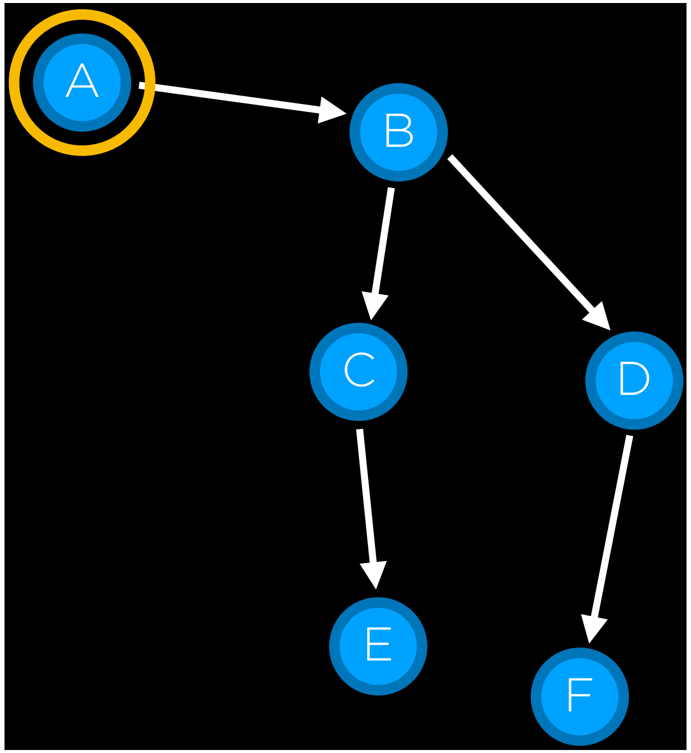
Example: Find a path from A to E

Frontier

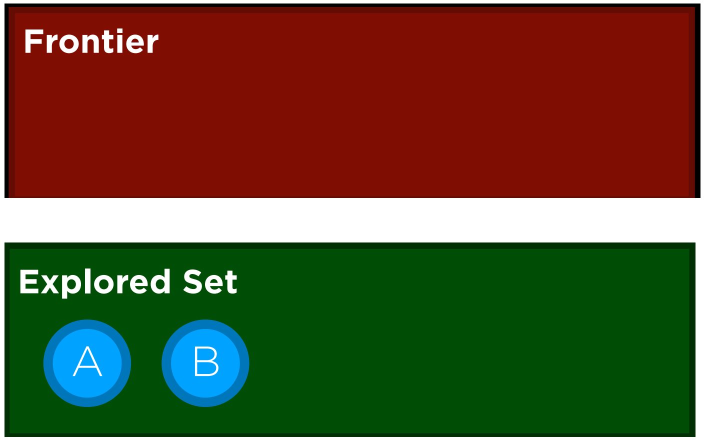
B

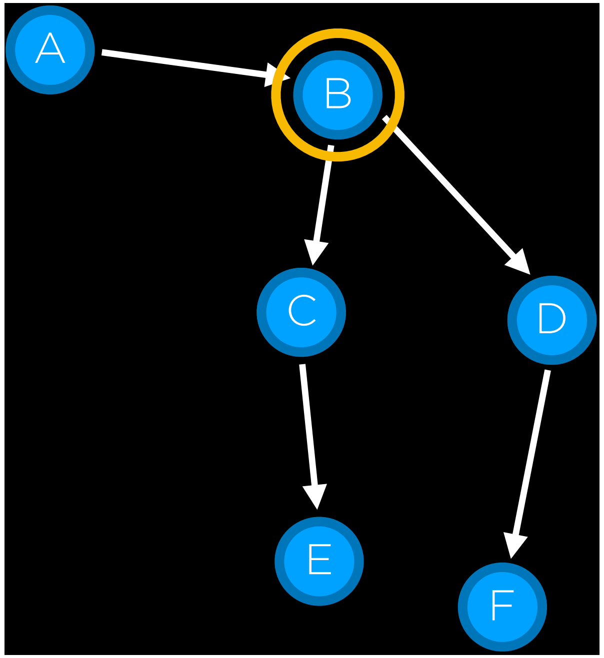
Explored Set

A



Example: Find a path from A to E





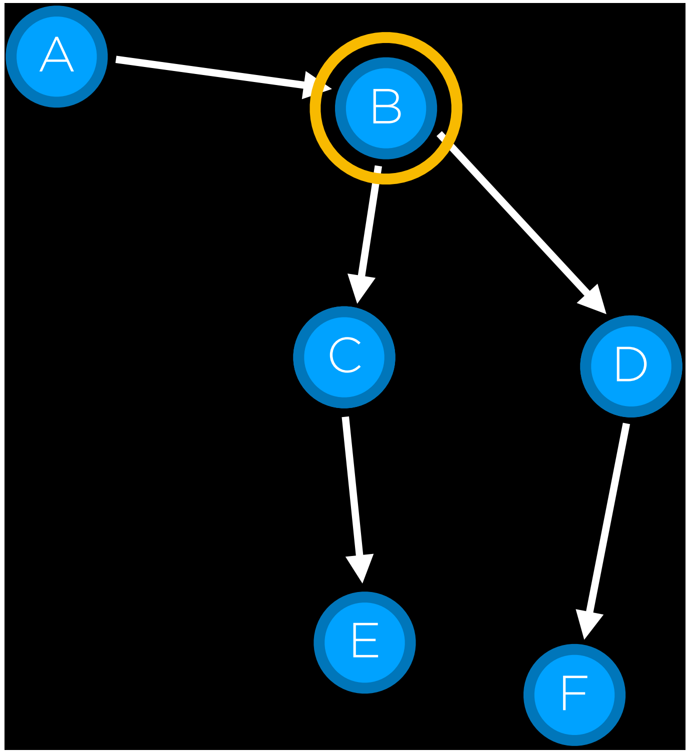
Example: Find a path from A to E

Frontier

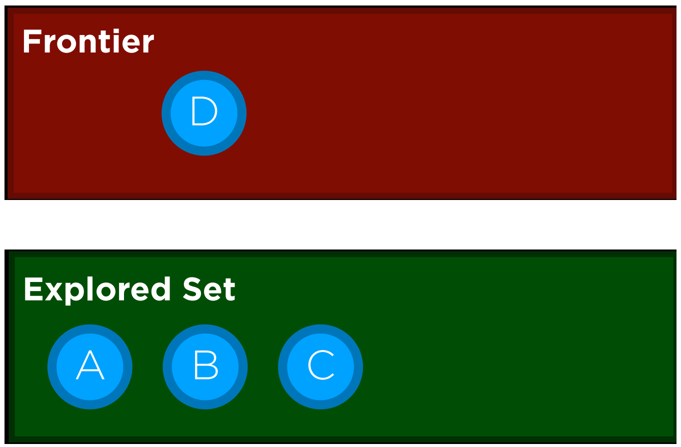
C D

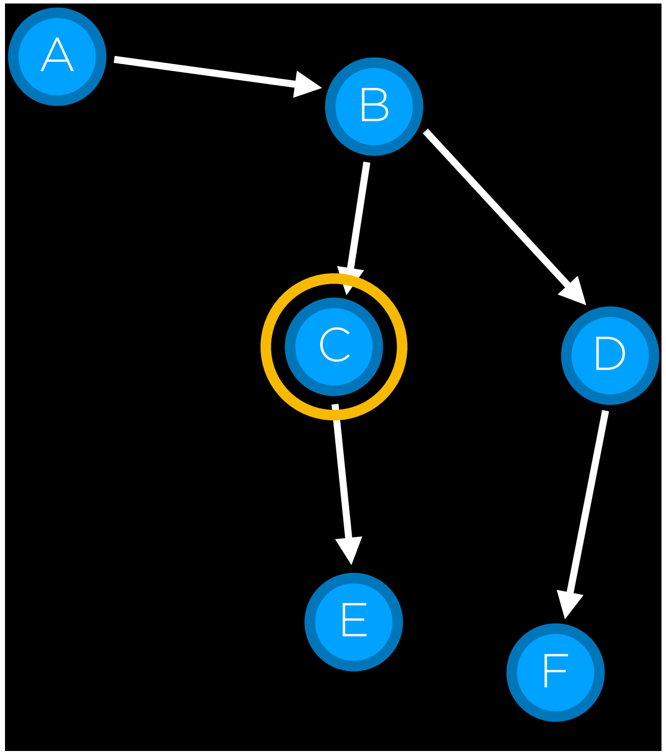
Explored Set

A B

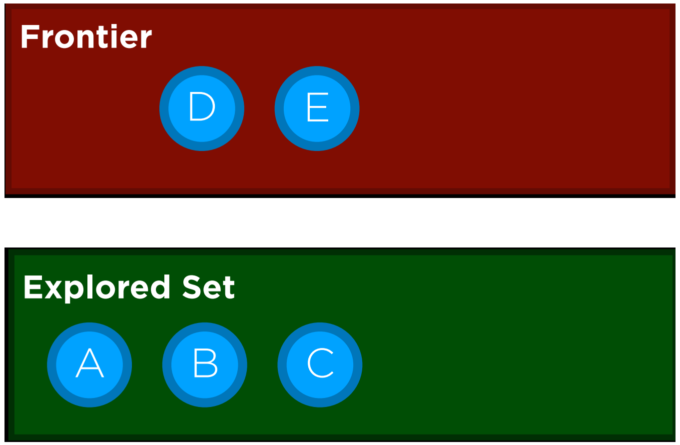


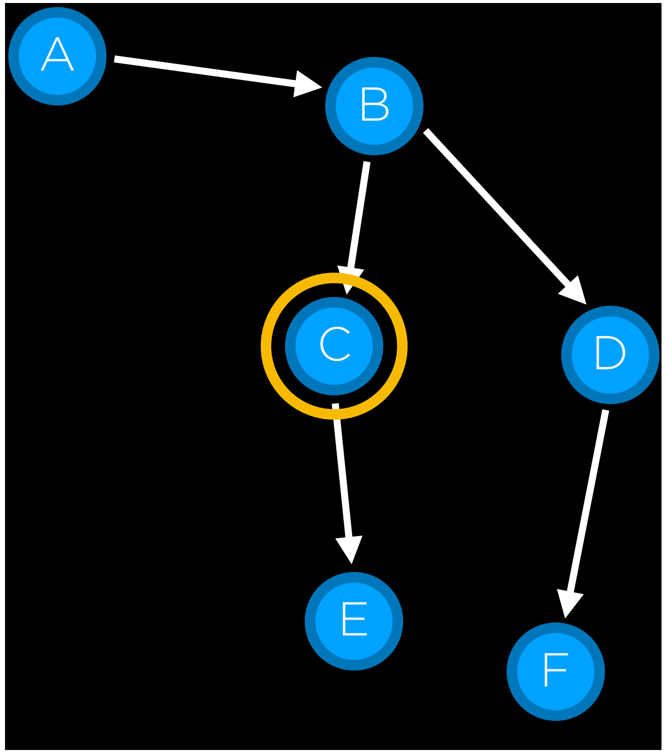
Example: Find a path from A to E



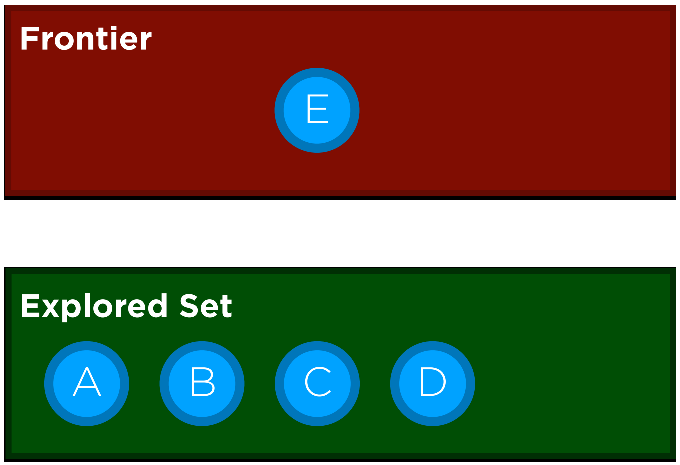


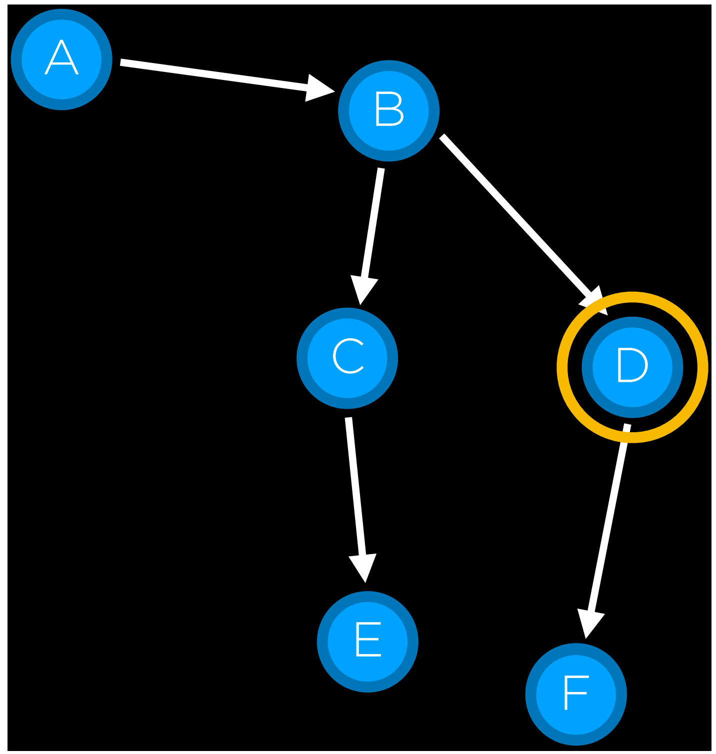
Example: Find a path from A to E



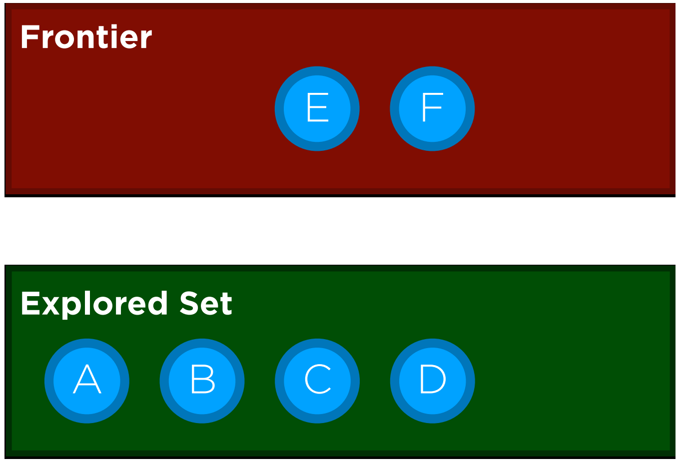


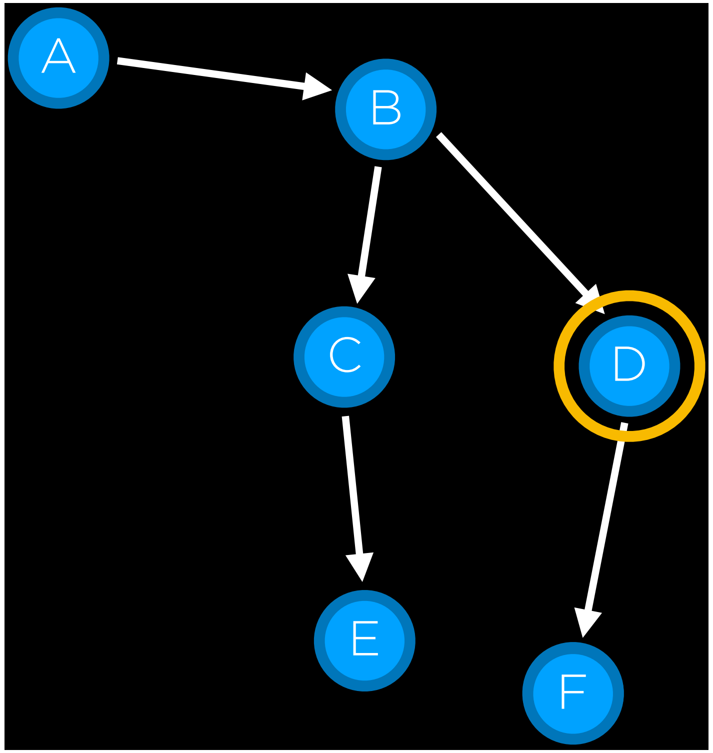
Example: Find a path from A to E



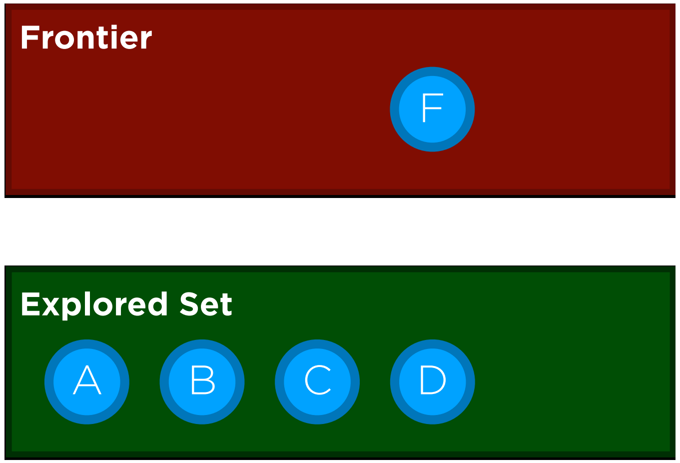


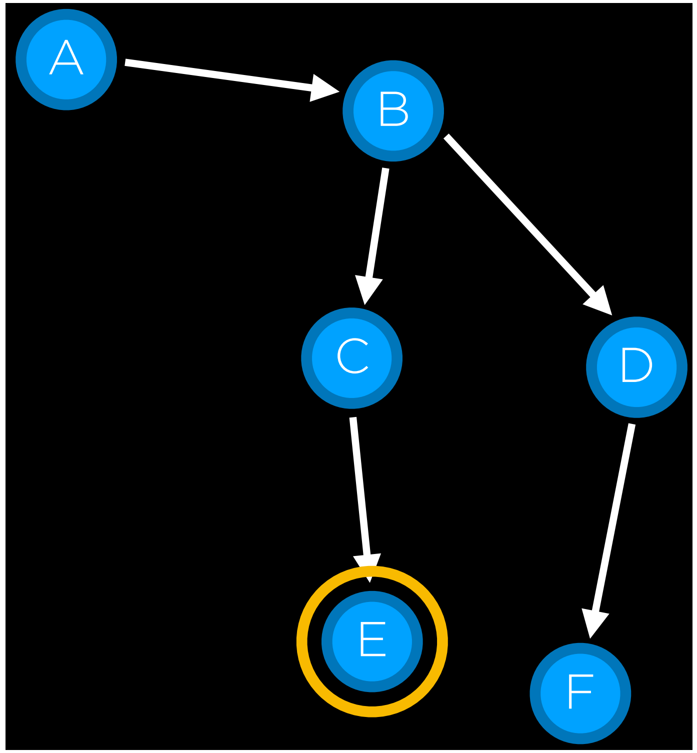
Example: Find a path from A to E





Example: Find a path from A to E





BFS

- Pros
  - This algorithm is guaranteed to find the optimal solution.
- Cons
  - This algorithm is almost guaranteed to take longer than the minimal time to run
  - At worst, this algorithm takes the longest possible time to run

Breadth-First Search Code Example

```
# Define the function that removes a node from the frontier and returns it.
def remove(self):
       # Terminate the search if the frontier is empty, because this means that there is no solution.
    if self.empty():
        raise Exception("empty frontier")
    else:
        # Save the oldest item on the list (which was the first one to be added)
        node = self.frontier[0]
        # Save all the items on the list besides the first one (i.e. removing the first node)
        self.frontier = self.frontier[1:]
        return node
```