COMP 3200
Artificial Intelligence

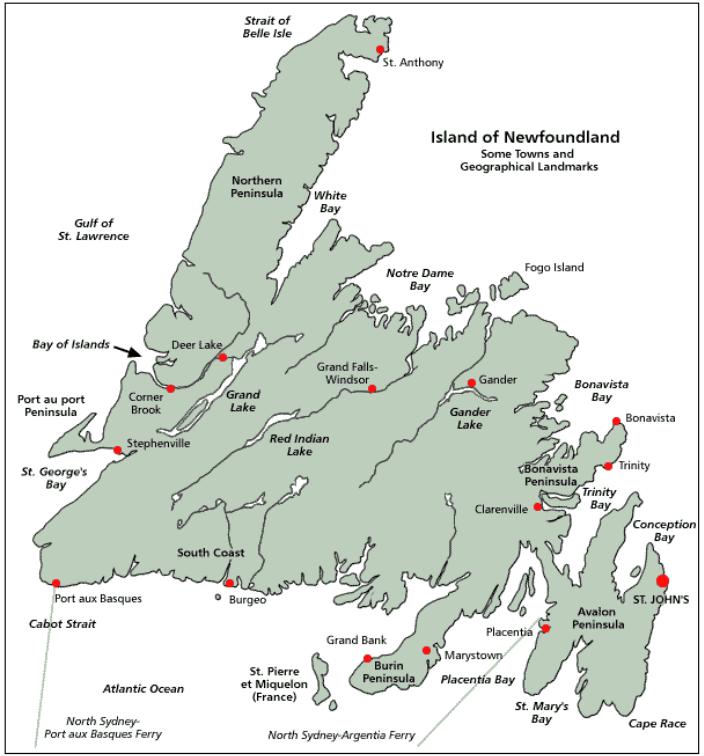
Lecture 3
Problem Solving & Search
Search Strategies

Problem Solving Agents

- Rational agents maximize performance
- Simplest Case: Satisfy a goal condition
  - Reach the end of a maze
  - Solve a puzzle
- Problem Solving Agent
  - Type of Goal-Based Agent
  - Find a sequence of actions that satisfies goal

Example Scenario

- Tourist comes to NL
  - Lands in St. John’s
- Wants to go to one of:
  - St. Anthony
  - Port aux Basques
- Extra constraints?
  - Has X hours total
  - Has $$ total for gas

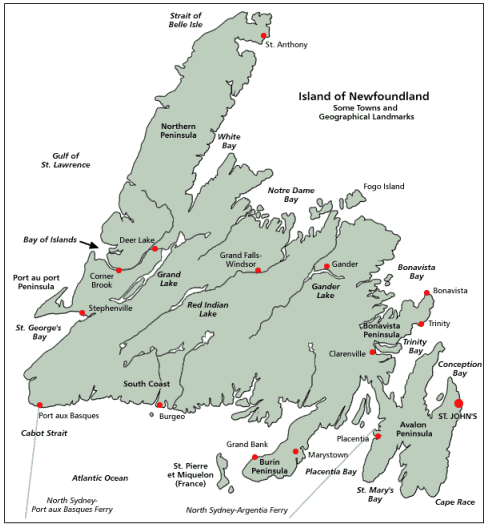


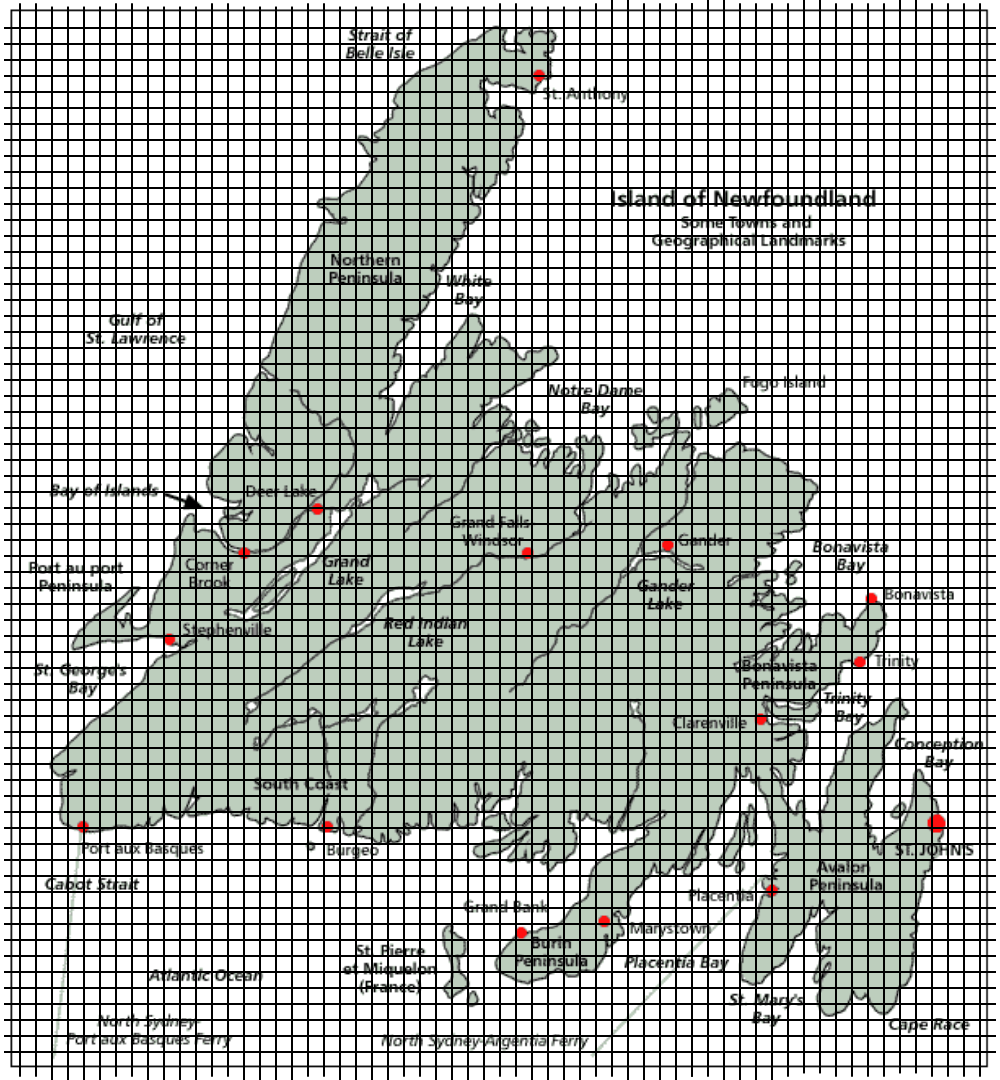
Goal Formulation

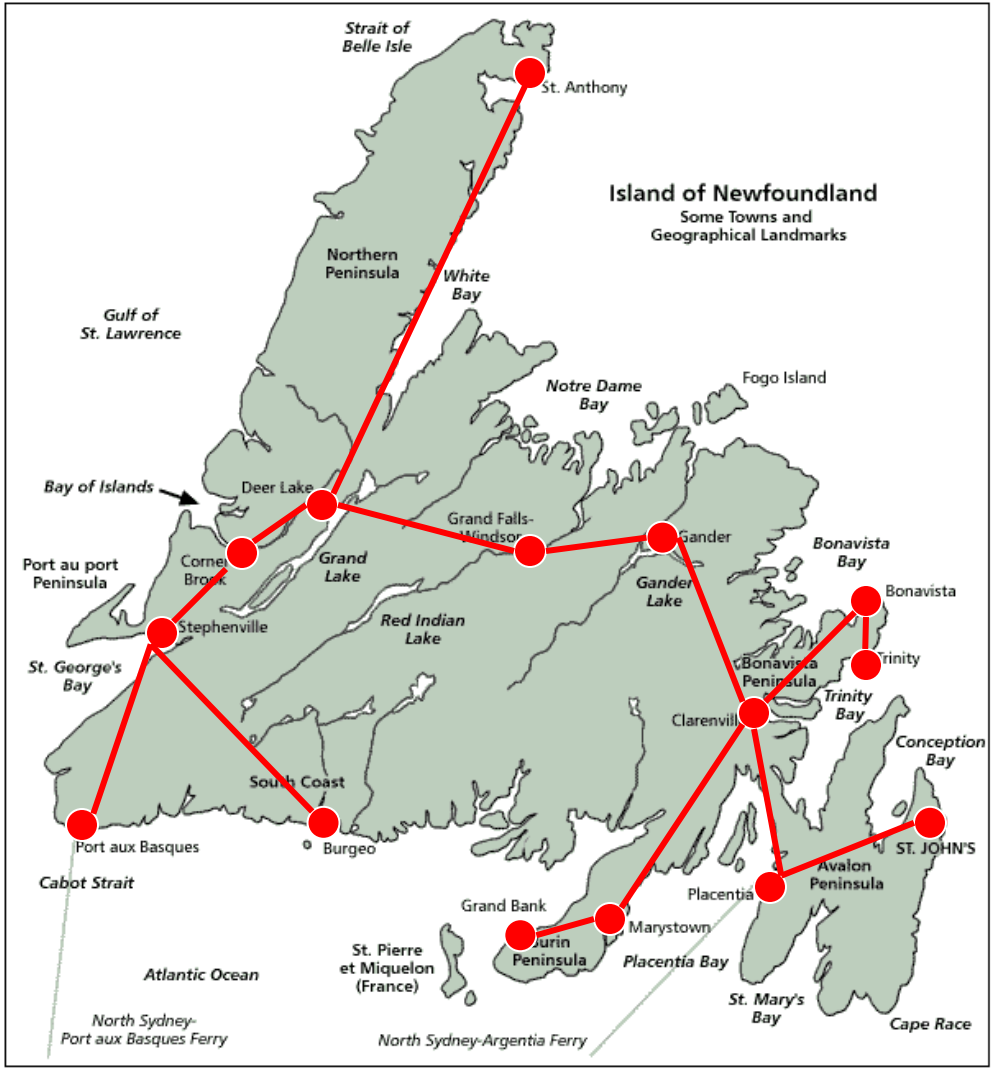
- Goal is a set of states in the environment
  - [St. Anthony, Port aux Basques]
- Goal is satisfied if one of them is reached
  - If a sequence of actions leads us to a goal state, the goal has been satisfied, and the problem is solved
- Goal can also be a function evaluation
  - If (enemy health <= 0)
  - If (enemy in checkmate)
  - If (player has all pieces of triforce AND Ganon dead)

Problem Formulation

- What states to consider?
  - Every square meter in Newfoundland
  - Towns in Newfoundland
- What actions to consider?
  - Move left by a meter
  - Move from town X to town Y
- What information is stored in a node?
  - Current town location
  - Distance / time driven so far




Island of Newfoundland
Some Towns and Geographical Landmarks
Strait of Belle Isle
St. Anthony
Northern Peninsula
White Bay
Gulf of St. Lawrence
Notre Dame Bay
Fogo Island
Bay of Islands
Deer Lake
Grand Falls-Windsor
Gander
Bonavista Bay
Corner Brook
Grand Lake
Gander Lake
Port au port Peninsula
Bonavista
Stephenville
Red Indian Lake
Trinity
St. George's Bay
Bonavista Peninsula
Trinity Bay
Clarenville
Conception Bay
South Coast
Port aux Basques
Burgeo
ST. JOHN'S
Cabot Strait
Avalon Peninsula
Placentia
Grand Bank
Marystown
Burin Peninsula
St. Pierre et Miquelon (France)
Placentia Bay
Atlantic Ocean
St. Mary's Bay
Cape Race
North Sydney Port aux Basques Ferry
North Sydney Argentia Ferry

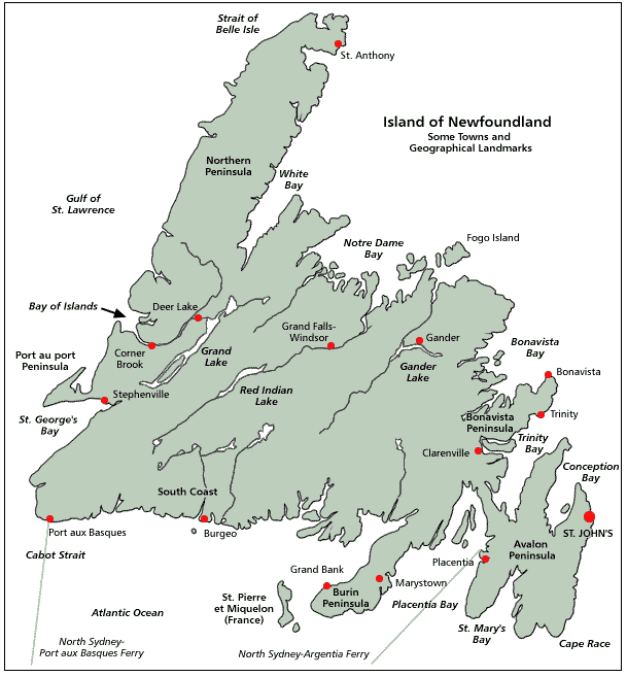

Strait of Belle Isle
St. Anthony
Island of Newfoundland
Some Towns and Geographical Landmarks
Northern Peninsula
White Bay
Gulf of St. Lawrence
Notre Dame Bay
Fogo Island
Bay of Islands
Deer Lake
Grand Falls-Windsor
Gander
Bonavista Bay
Port au port Peninsula
Corner Brook
Grand Lake
Gander Lake
Bonavista
Stephenville
Red Indian Lake
Trinity
St. George's Bay
Bonavista Peninsula
Trinity Bay
Clarenville
Conception Bay
South Coast
Port aux Basques
Burgeo
ST. JOHN'S
Cabot Strait
Avalon Peninsula
Placentia
Grand Bank
Marystown
St. Pierre et Miquelon (France)
Burin Peninsula
Placentia Bay
Atlantic Ocean
St. Mary's Bay
Cape Race
North Sydney-Port aux Basques Ferry
North Sydney-Argentia Ferry

Well-Defined Problem

1. Initial State that the agent starts in
2. Actions possible from each State
   - Transition / Successor Function
3. The set of goal states
   - Goal test function
4. Action cost function
   - Path cost = sum(cost(actions in path))

Example Problem Definition

1. Initial State
   - St. John's
2. Actions
   - Towns connected by roads
3. Goal
   - [SA, PAB]
4. Action Cost Function
   - Travel distance or time

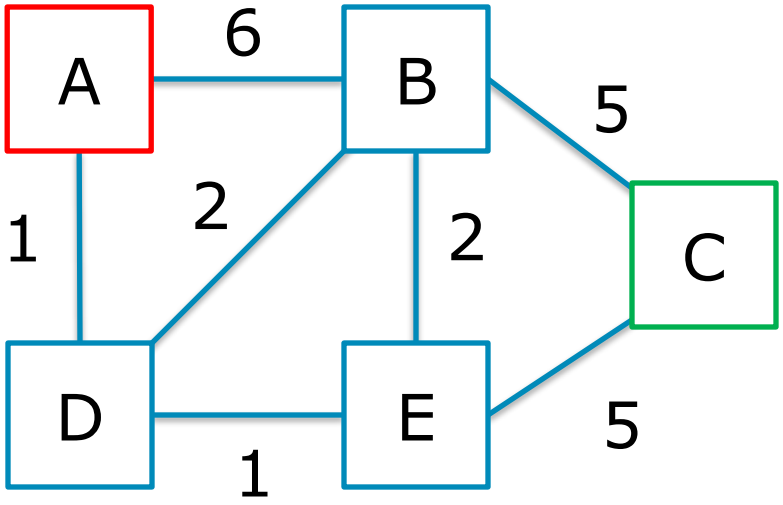


Problem Solution

- Path from start state to a goal state
  - Path = ordered sequence of actions
- Solution cost = path cost
  - Sum cost of actions (usually)
  - Most problems assume costs > 0
- Optimal solution has lowest path cost among all possible solutions

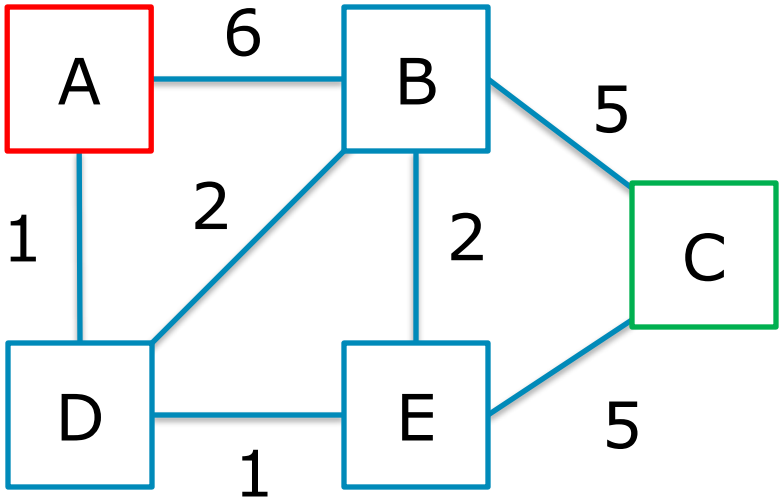
Example Graph Problem

- Initial State
  - State A
- Actions
  - (A,B) legal if edge
- Goal State
  - State C
- Action Cost Function
  - Edge label (A,B)=6



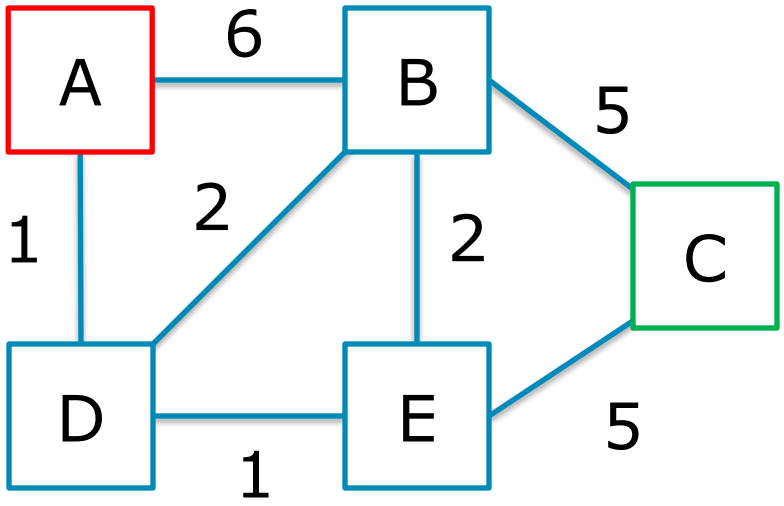
Example Graph Problem

- Fully Observable
- Static
- Discrete
- Deterministic
- Single Agent
- Sequential



Example Graph Problem

- Objective
  - Path to goal possible?
  - Shortest path to goal?
- Algorithm
  - Which to use?
- Let's try search



What is Search?

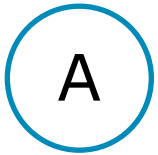
- Agent has several available actions
- Agent can explore various sequences of those actions
- Agent chooses the best sequence found

- This process in general is called search

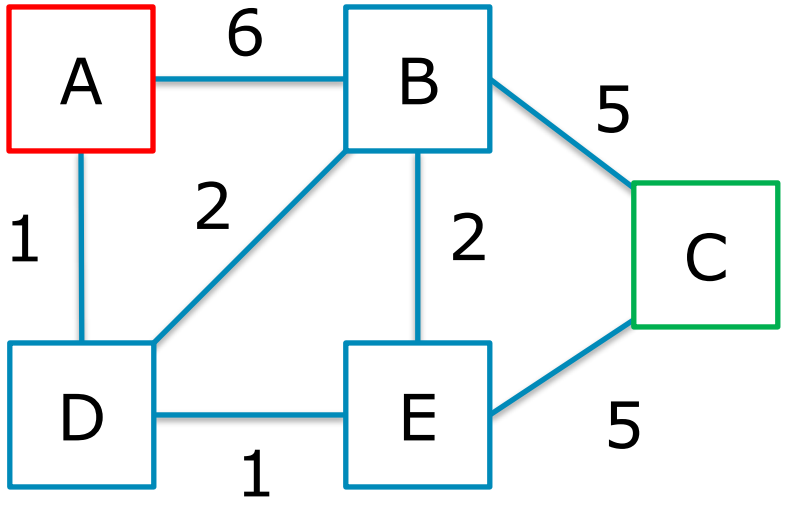
Search

- Search will explore the problem state space
- Searching this space generates a search tree
- The search tree has nodes and edges

Search Tree

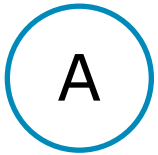


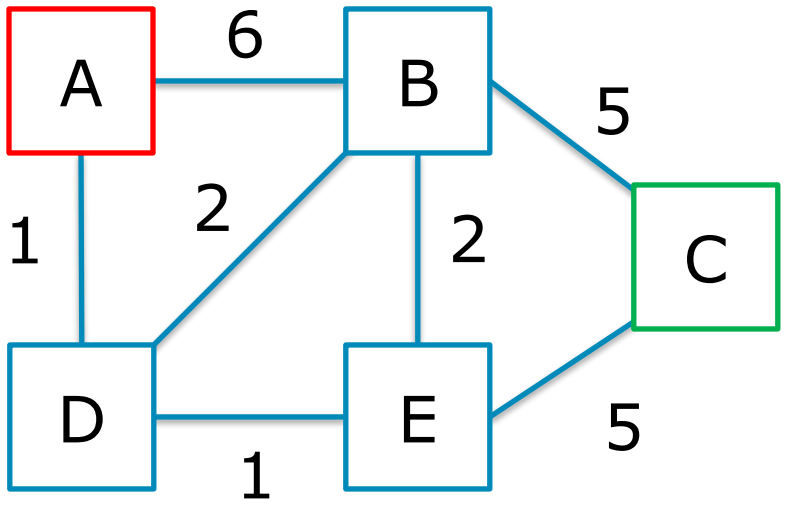
Nodes in Search Tree



States in Environment

Search Tree

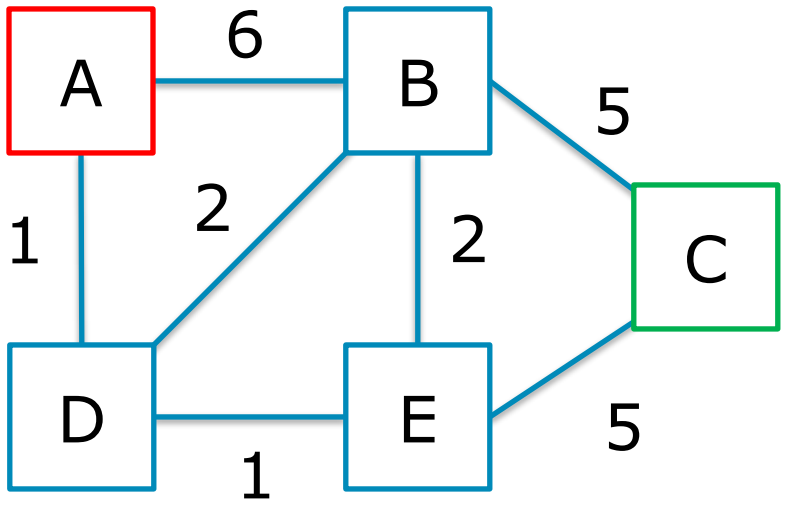




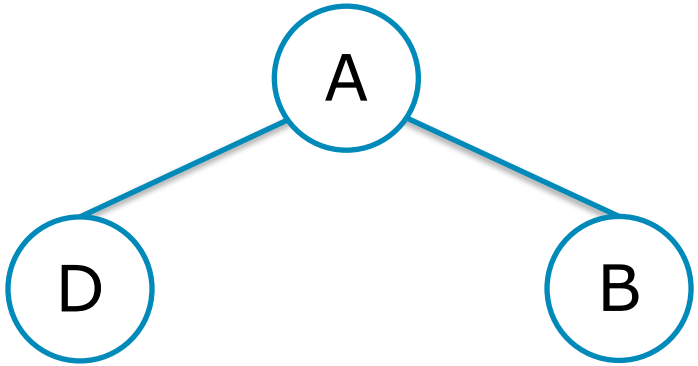
Search Tree

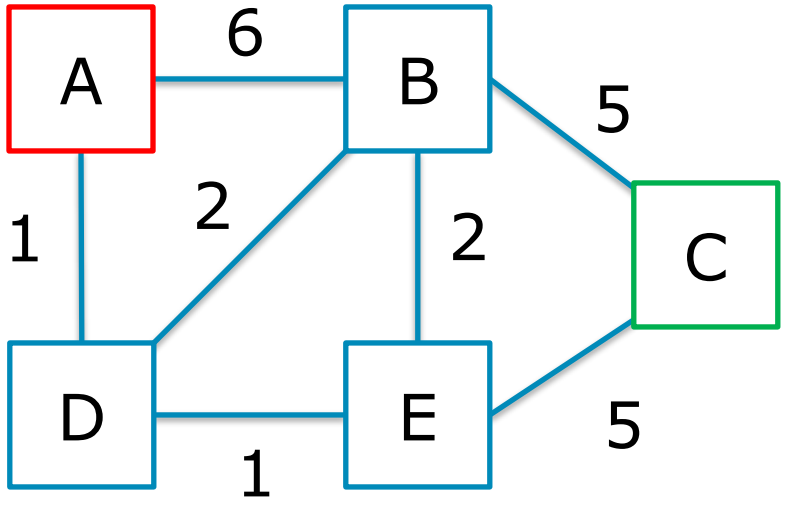
Expand

A

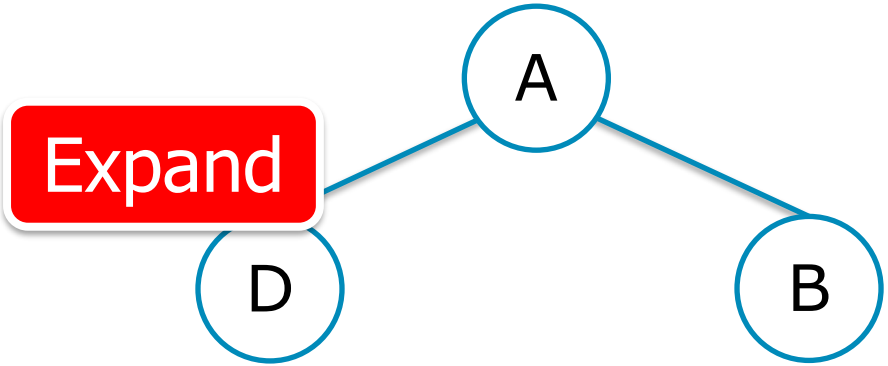


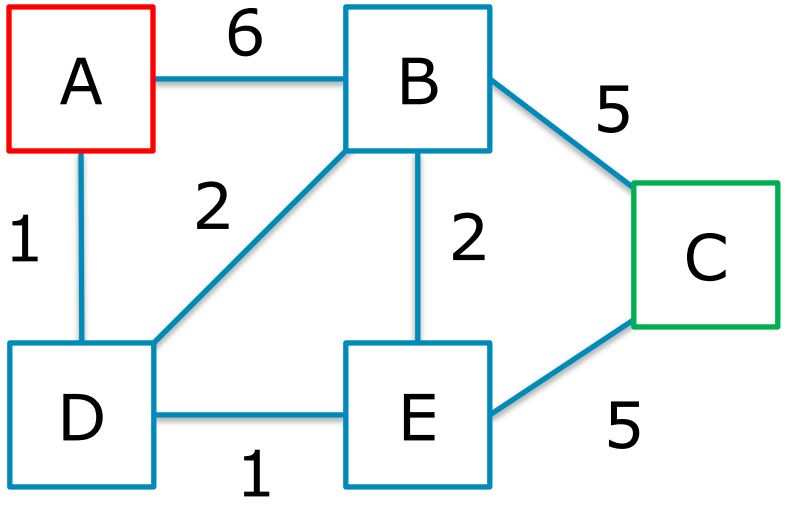
Search Tree



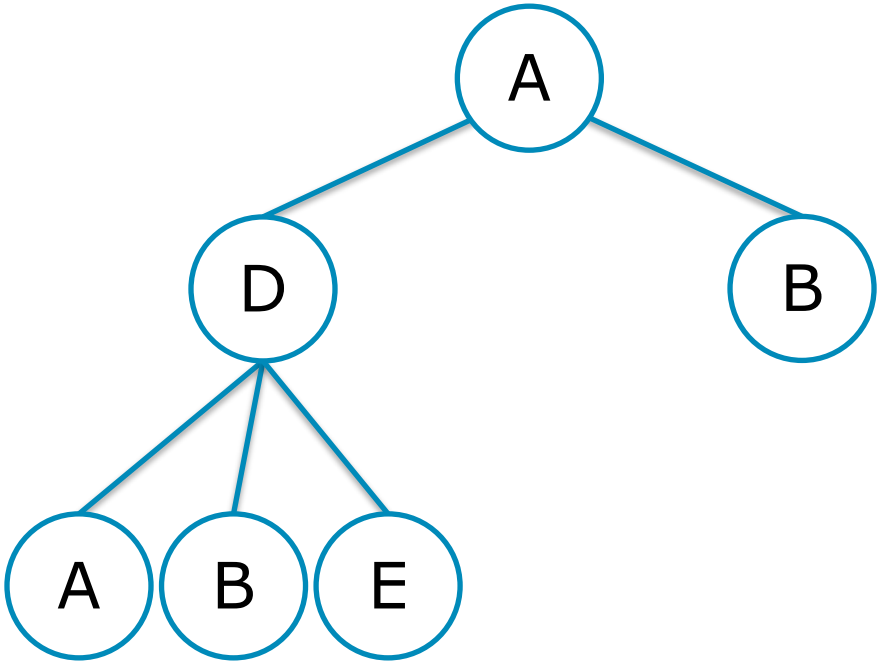


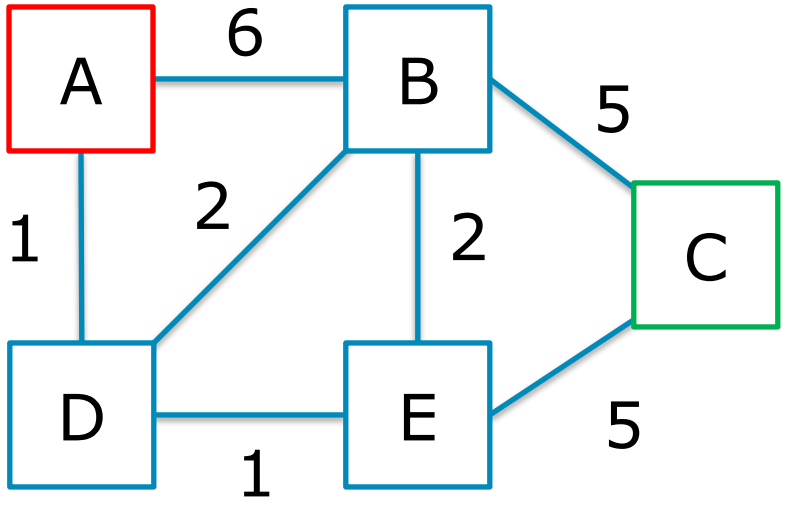
Search Tree



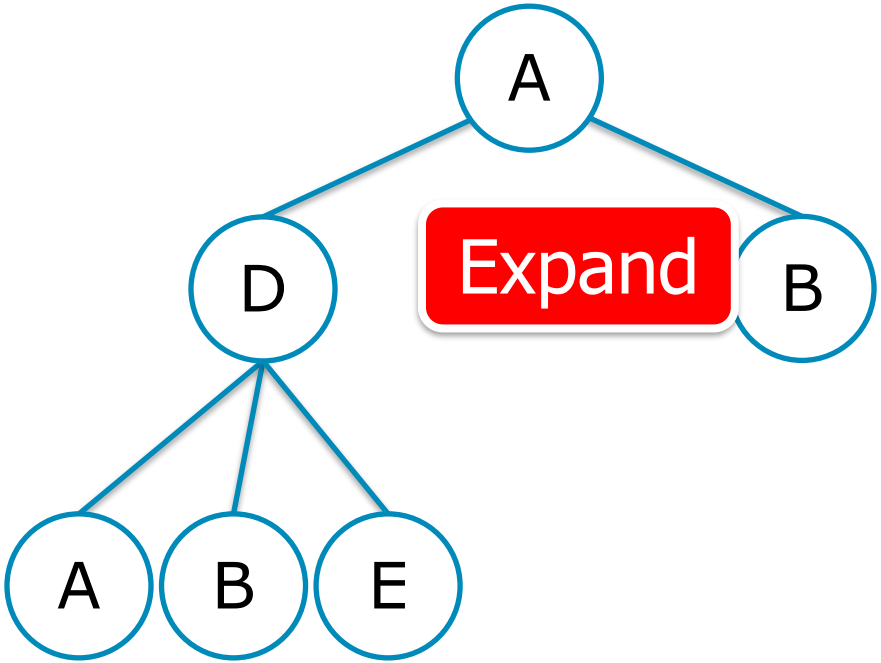


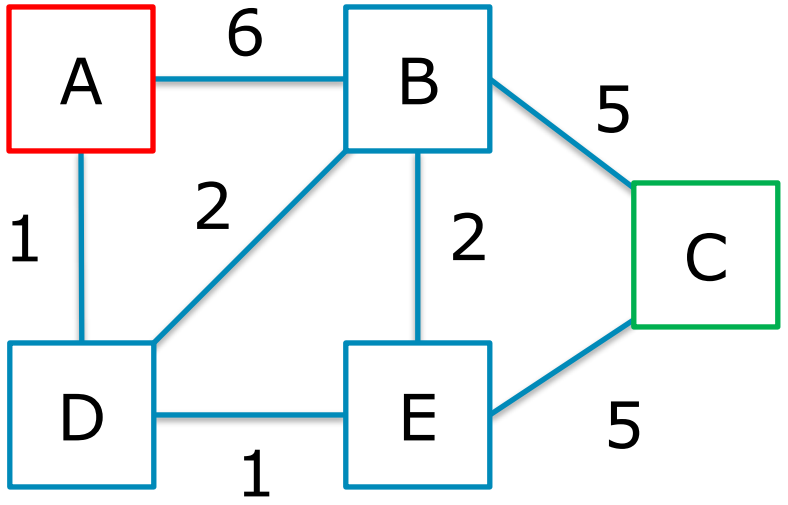
Search Tree



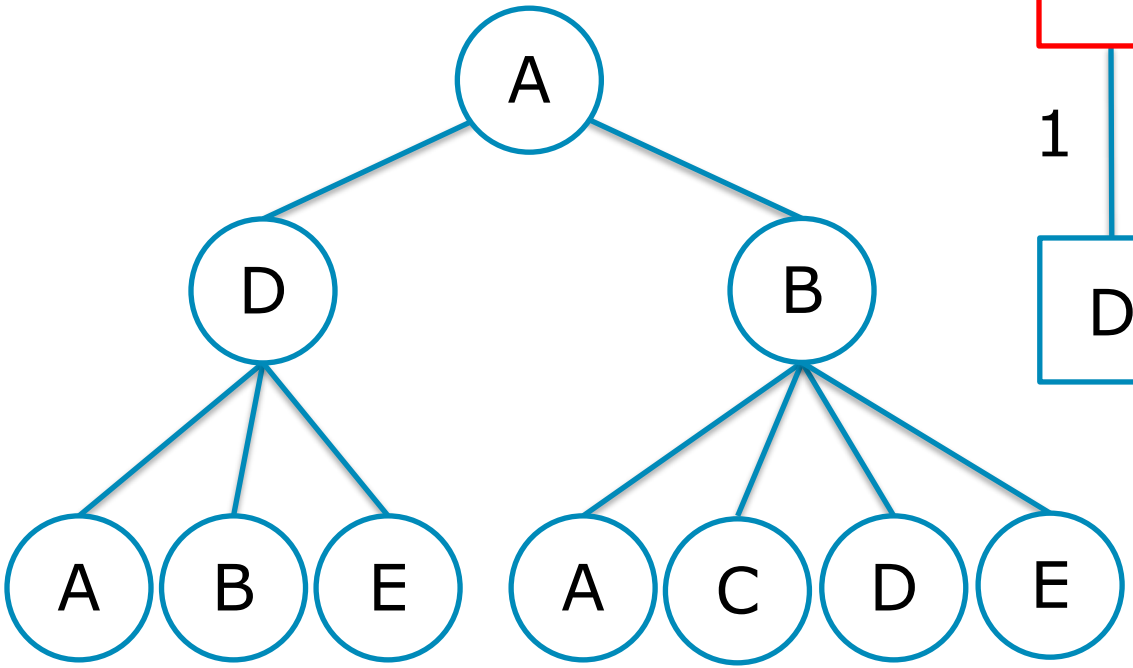


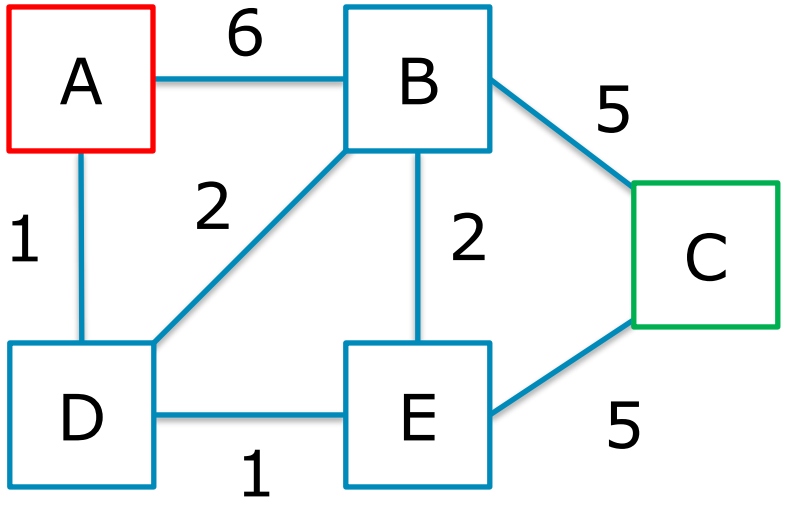
Search Tree



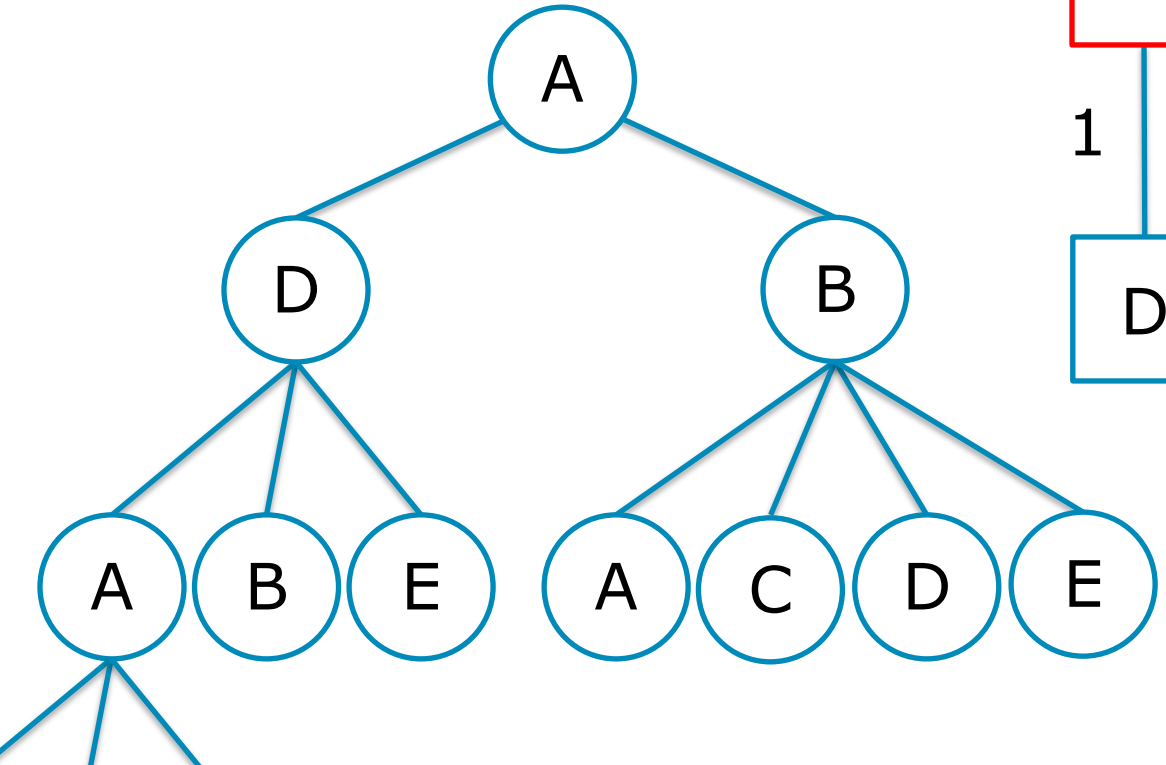


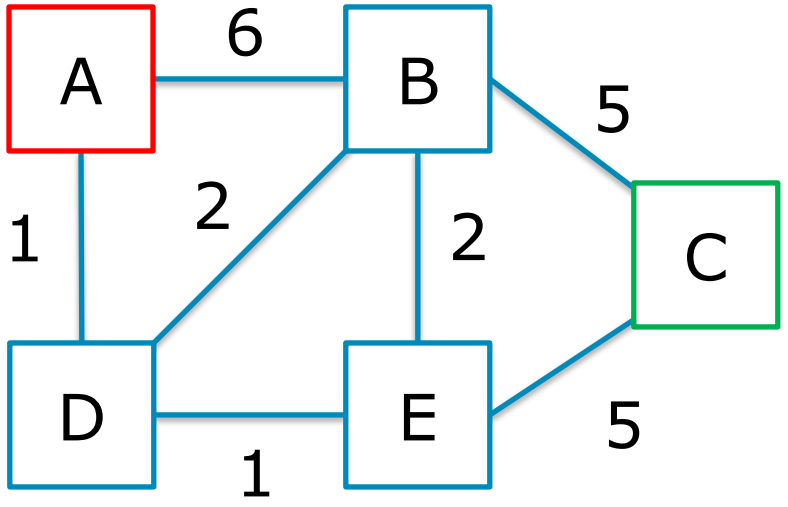
Search Tree



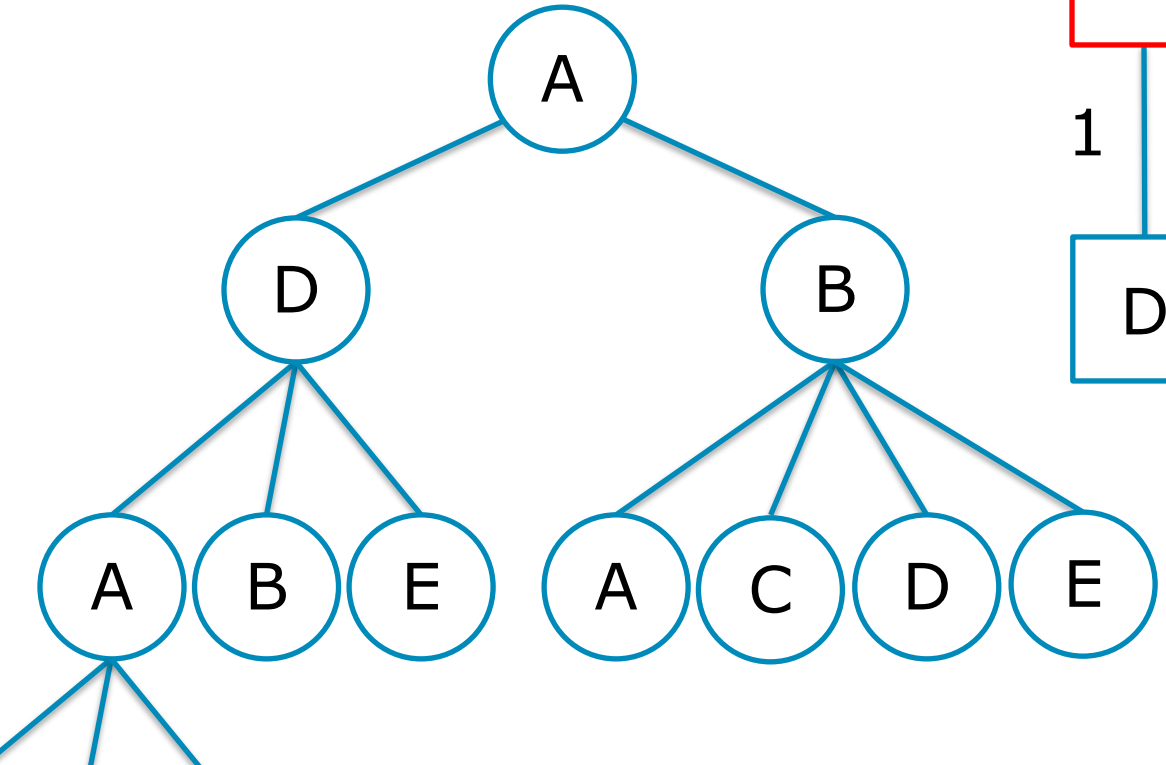


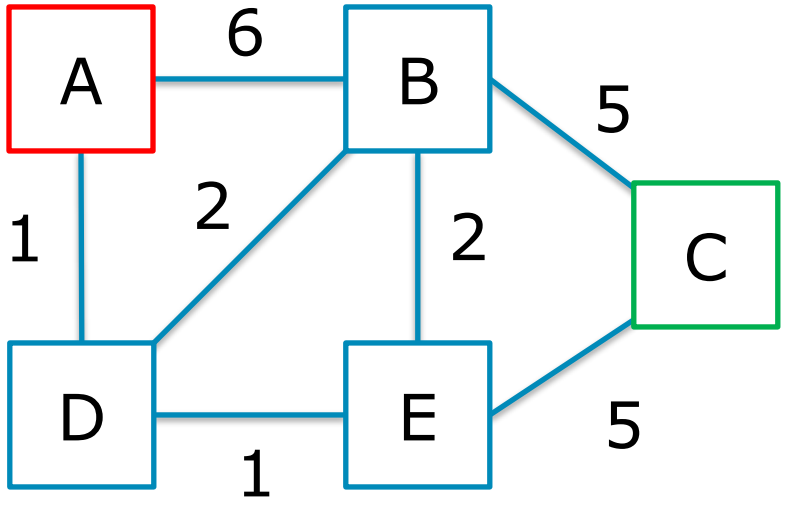
Search Tree





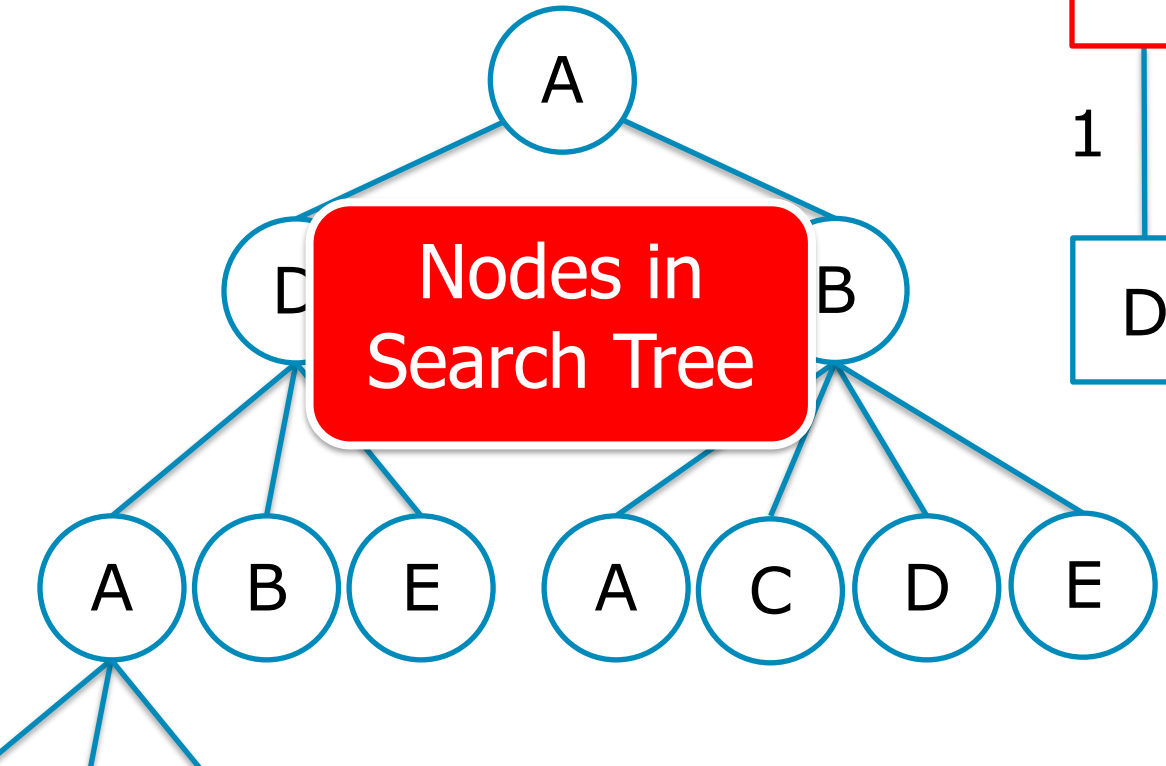
Search Tree



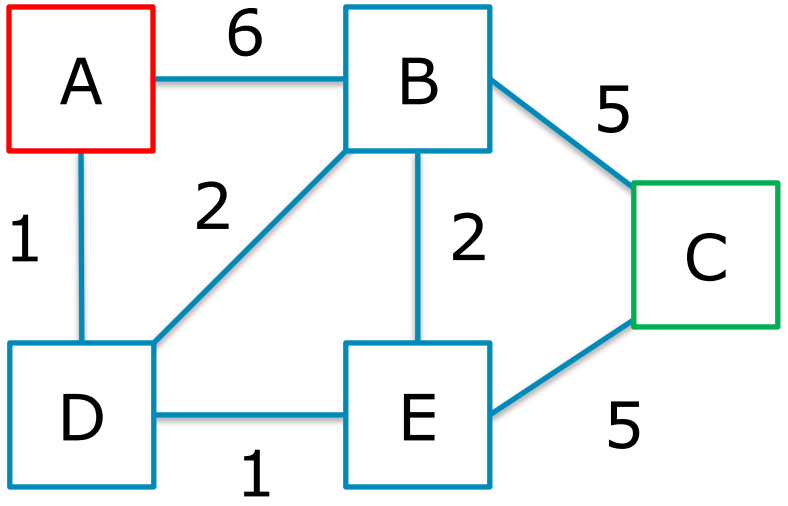


Continue Until Goal Found

Search Tree

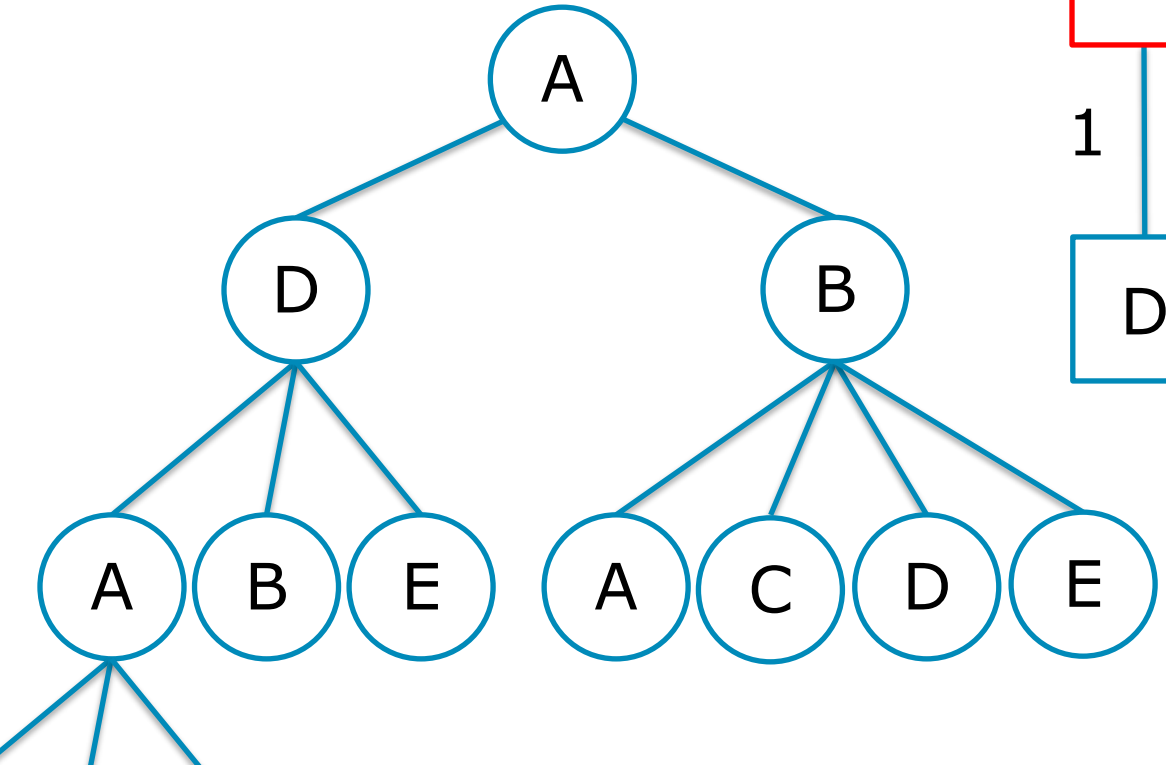


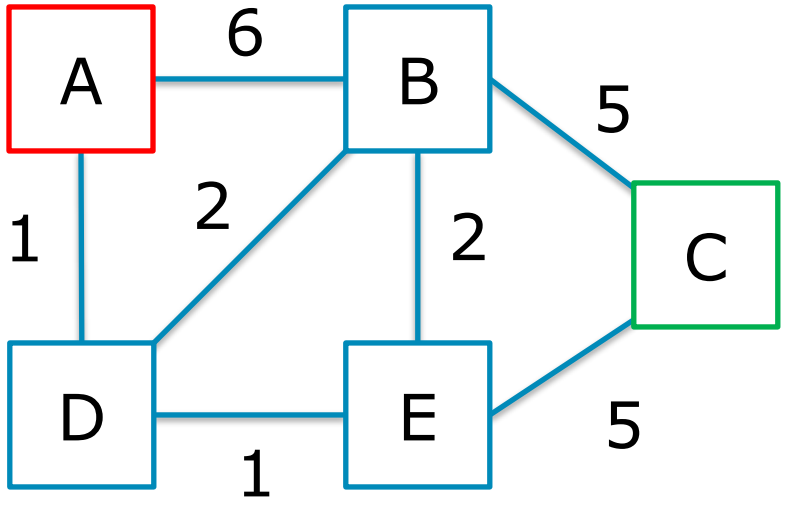
Nodes in Search Tree



States in Environment

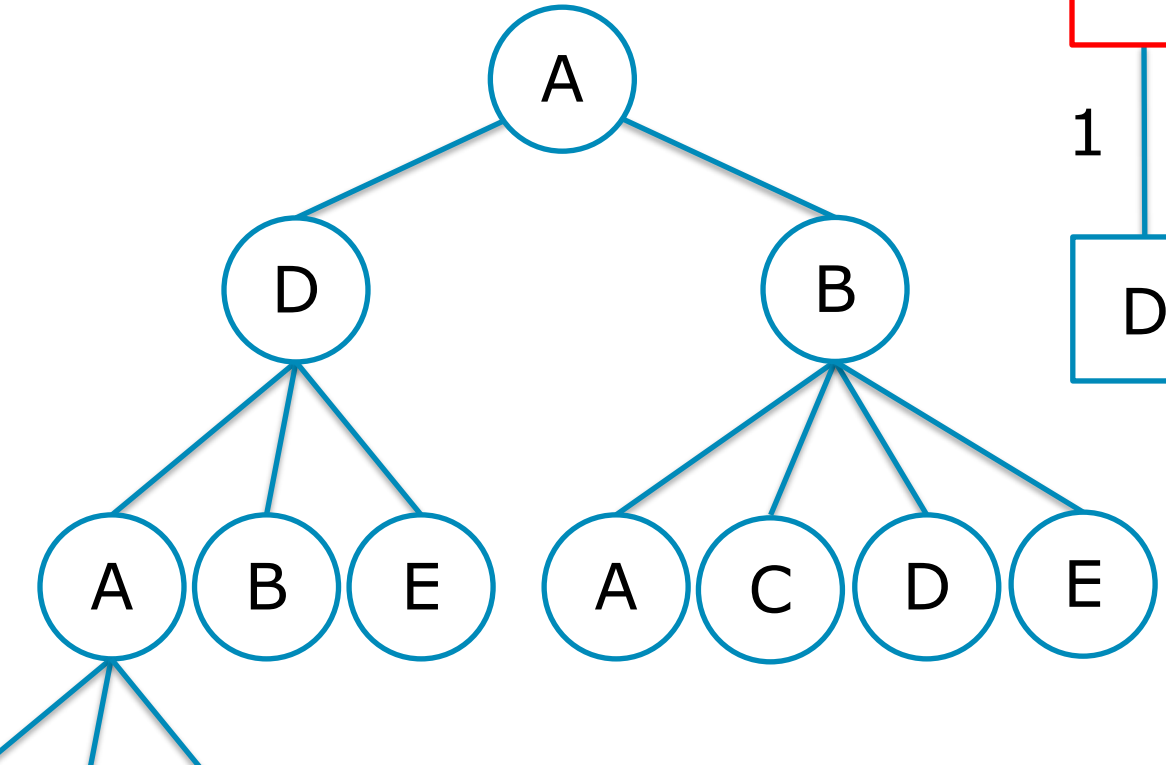
Search Tree

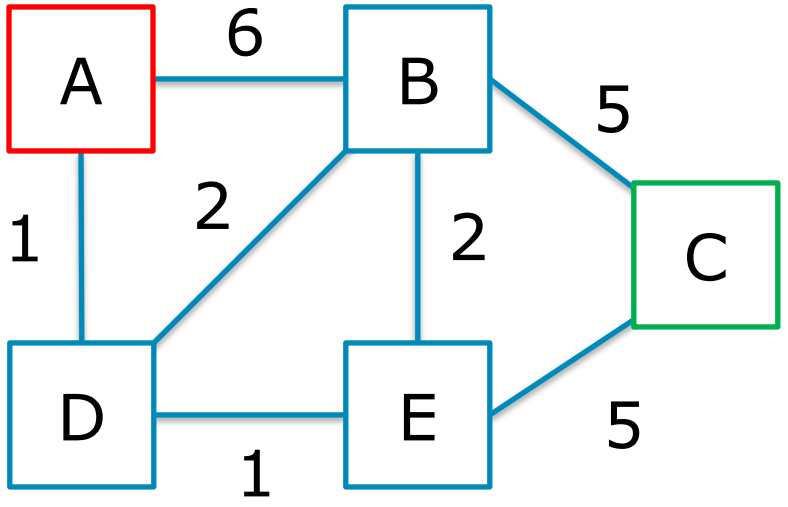




State Space: Finite

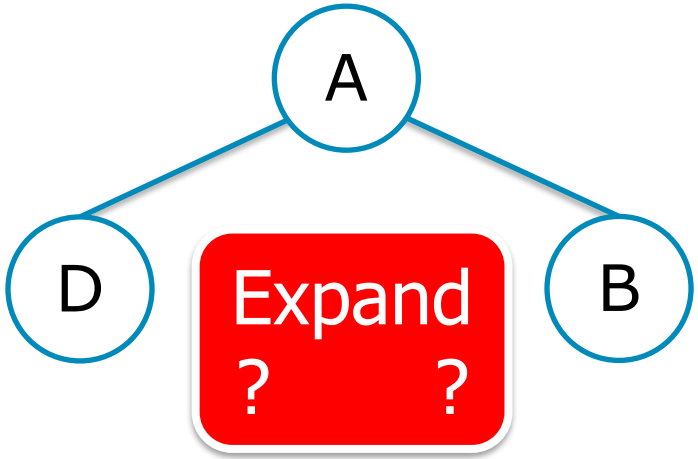
Search Tree

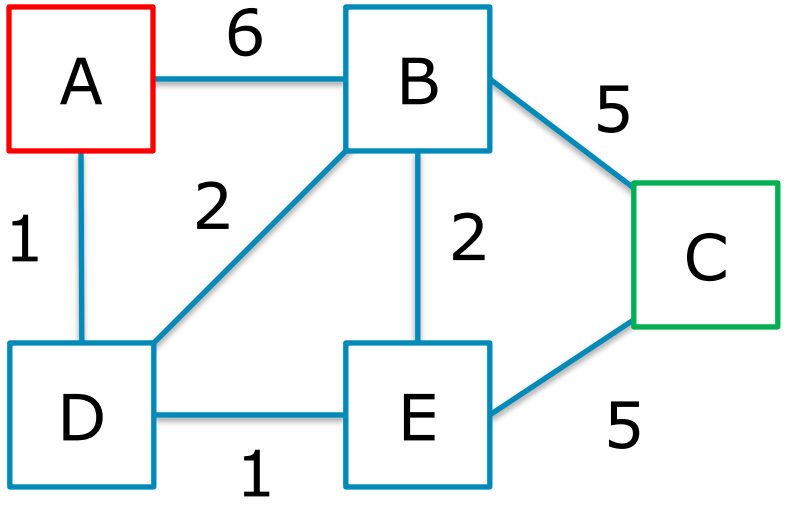




State Space: Finite
Search Tree: Infinite

Search Strategy





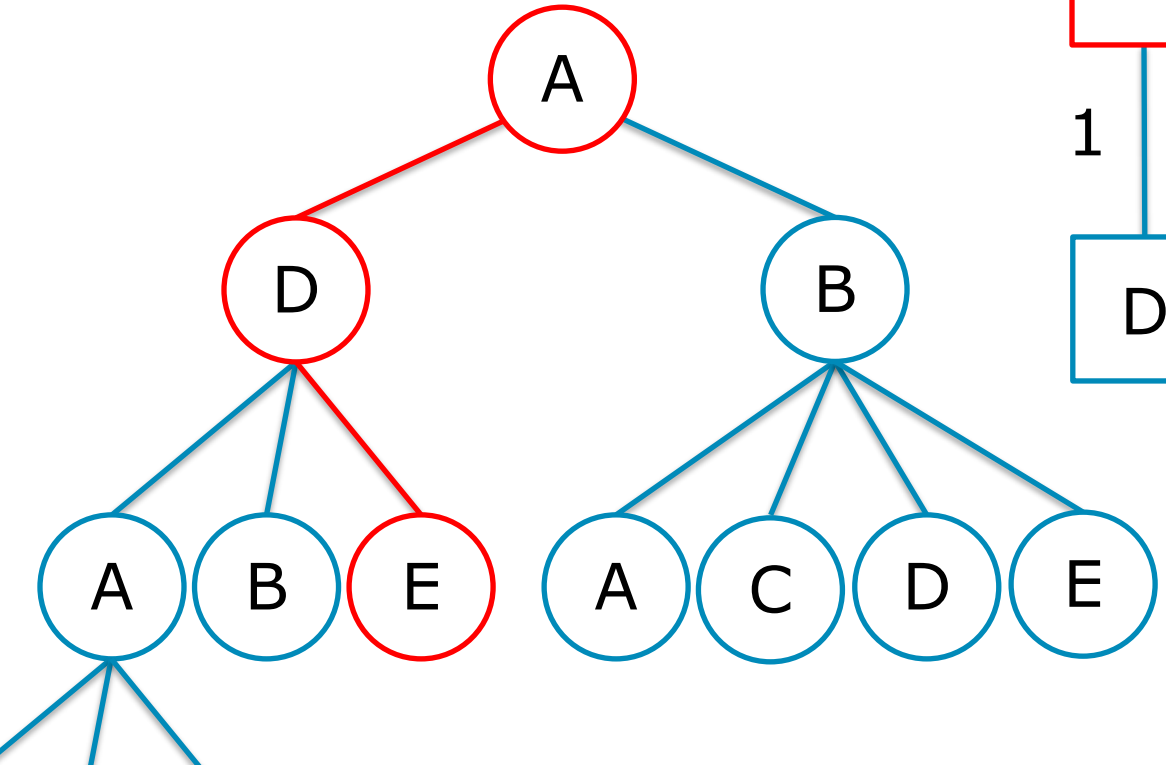
Search Tree Node Data

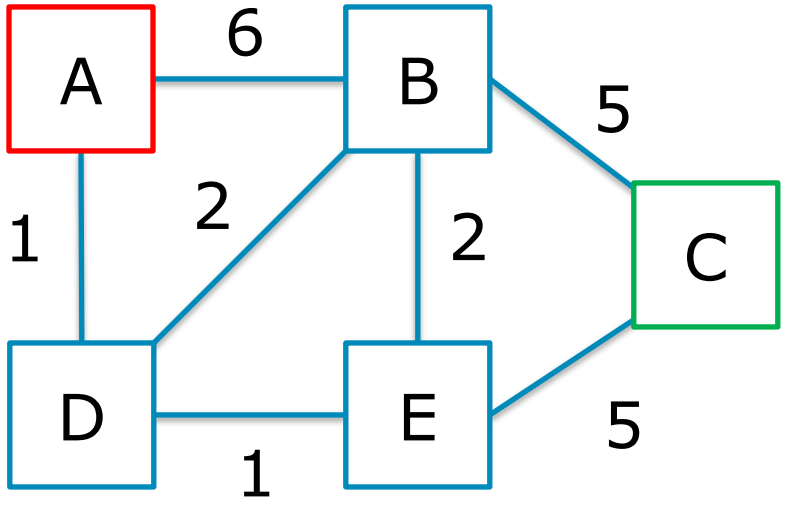
- State: The state in the graph
- Parent Node: Tree node that generated this node
- Action: Action that produced this node
- Path Cost: Cost of path so far from start
  - Traditionally denoted by g(n)
- Depth: Number of actions to this node
- Other: Can contain precomputed data (optimization)
  - Legal actions, heuristics, etc

Node vs. State

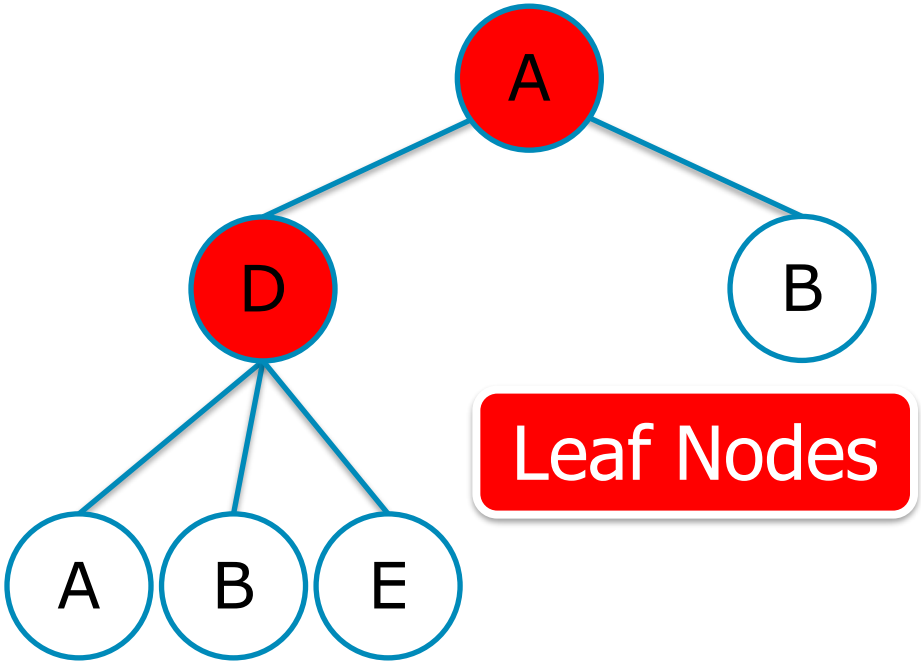
- Important distinction
- State
  - Configuration of the environment
- Node
  - Bookkeeping data structure
  - Exists only within the search tree
  - Nodes are ‘on paths’ in the search tree

Node vs. State

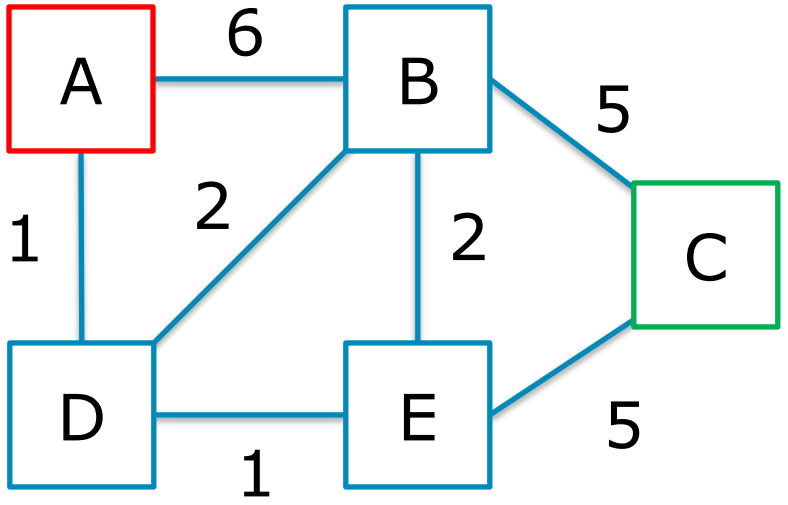




The Fringe



Leaf Nodes



Nodes that have been generated but not yet expanded

General Uninformed Tree Search

```
1. Function Tree-Search(problem, strategy)
2.   fringe = {Node(problem.initial_state)}
3.   while (true)
4.     if (fringe.empty) return fail
5.     node = strategy.select_node(fringe)
6.     if (node.state is goal) return solution
7.     else fringe.add(Expand(node, problem))
```

Node Expansion

```
1. Function Expand(node, problem)
2.   successors = {}
3.   for a : problem.actions(node)
4.     child = Node(node.state.do_action(a))
5.     child.parent = node
6.     child.g = node.g + a.cost
7.     child.action = a
8.     child.depth = node.depth + 1
9.     succesors.add(child)
10.  return successors
```

Problem Solving Performance

- Completeness
  - Is it guaranteed to find a solution if it exists?
- Optimality
  - Does it find the optimal solution?
- Time Complexity
  - How long does it take to find a solution?
- Space Complexity
  - How much memory is needed to run the search?

Time + Space Complexity

- Measured on size of input problem / graph
- Time measures number of nodes generated
- Space measured by how much memory is needed to store the maximum number of nodes (worst case) needed at any given time
- Search / AI traditionally measures:
  - Branching factor b (max successors at any node)
  - Depth d (of the shallowest goal node)
  - Tree size ~ b^d

Search Strategies

Search Strategies

- Uninformed (Blind) Search
  - No info about states beyond problem desc.
  - Limited to successor generation, goal test

- Informed (Heuristic) Search
  - Can guess which states are ‘more promising’
  - Hopefully leads to faster search episodes

Breadth-First Search (BFS)

- Root node expanded first
- Root successors expanded next
- Their successors next… etc
- BFS in general
  - All nodes at a given depth are expanded before any nodes in the next level
- Use a Queue for fringe

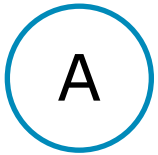
General Uninformed Tree Search

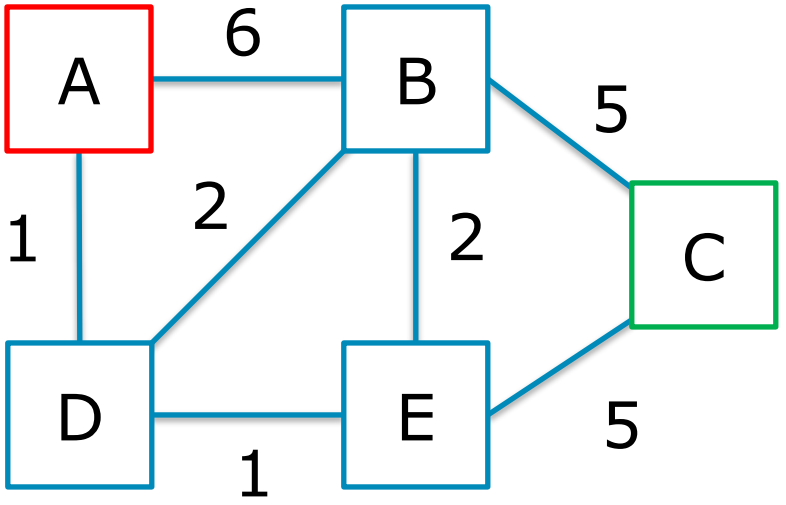
```
1. Function Tree-Search(problem, strategy)
2.   fringe = {Node(problem.initial_state)}
3.   while (true)
4.     if (fringe.empty) return fail
5.     node = strategy.select_node(fringe)
6.     if (node.state is goal) return solution
7.     else fringe.add(Expand(node, problem))
```

Breadth-First Search

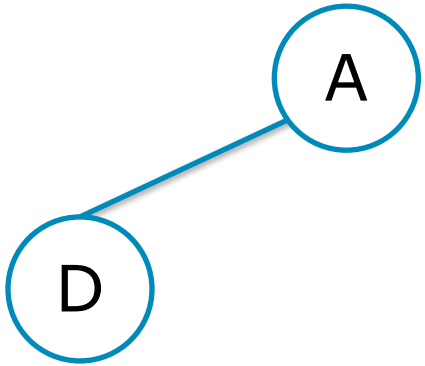
```
1. Function BFS(problem)
2.   fringe = Queue{Node(problem.initial_state)}
3.   while (true)
4.     if (fringe.empty) return fail
5.     node = fringe.pop()
6.     if (node.state is goal) return solution
7.     else fringe.add(Expand(node, problem))
```

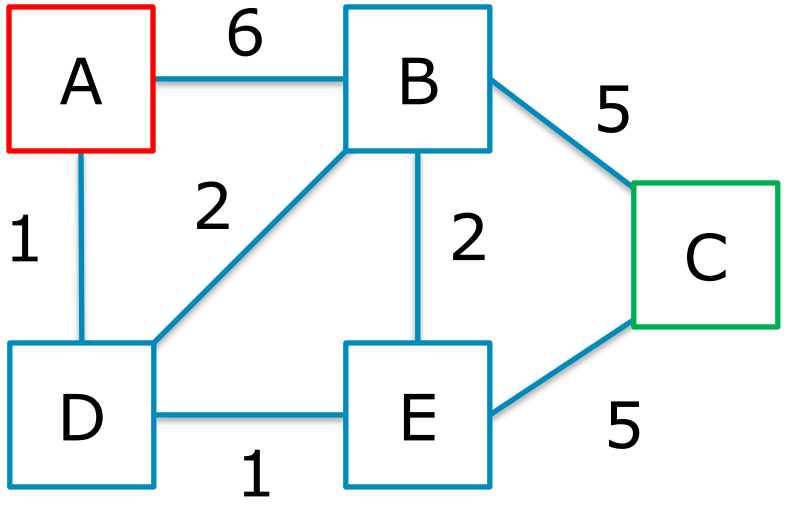
BFS



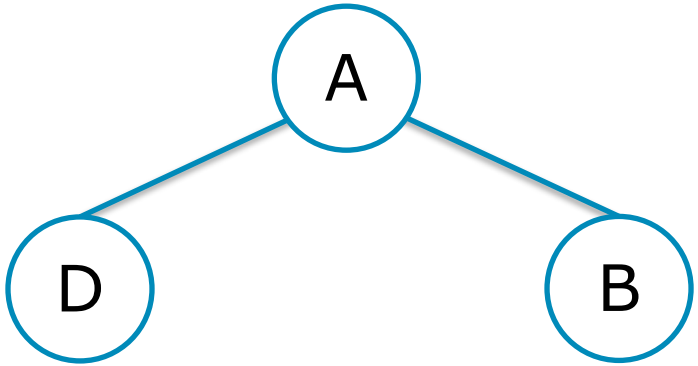


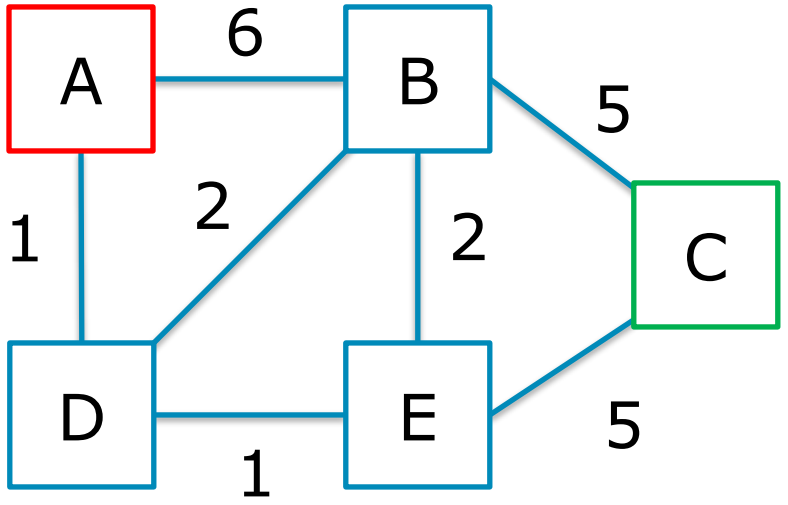
BFS



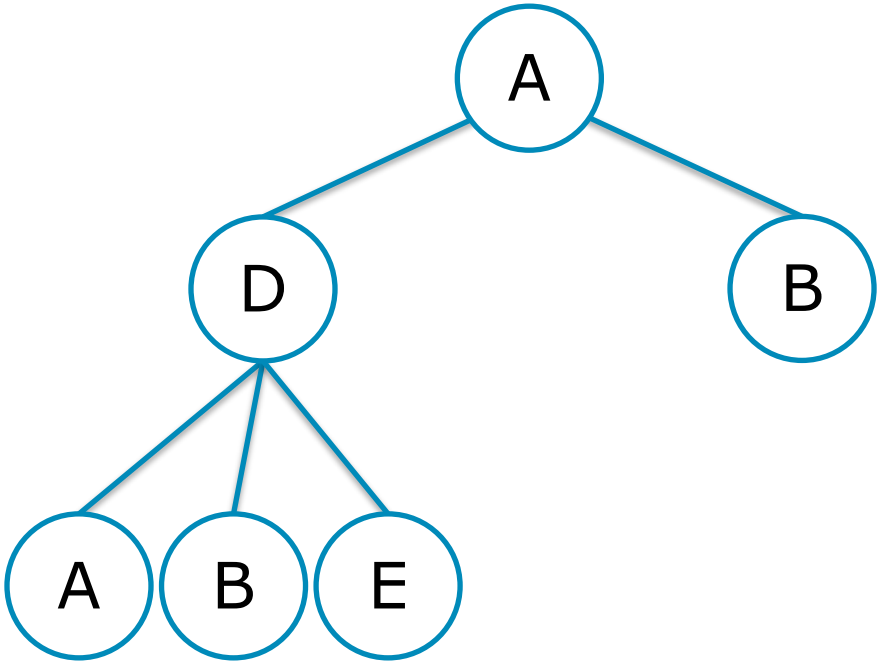


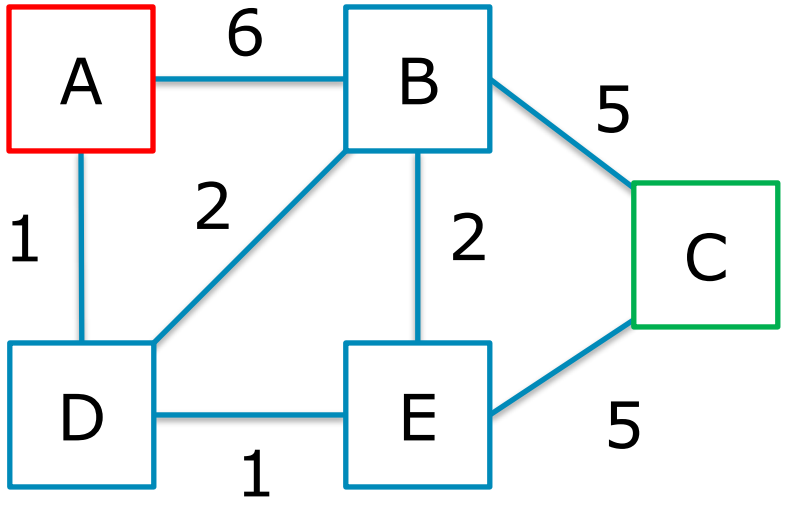
BFS



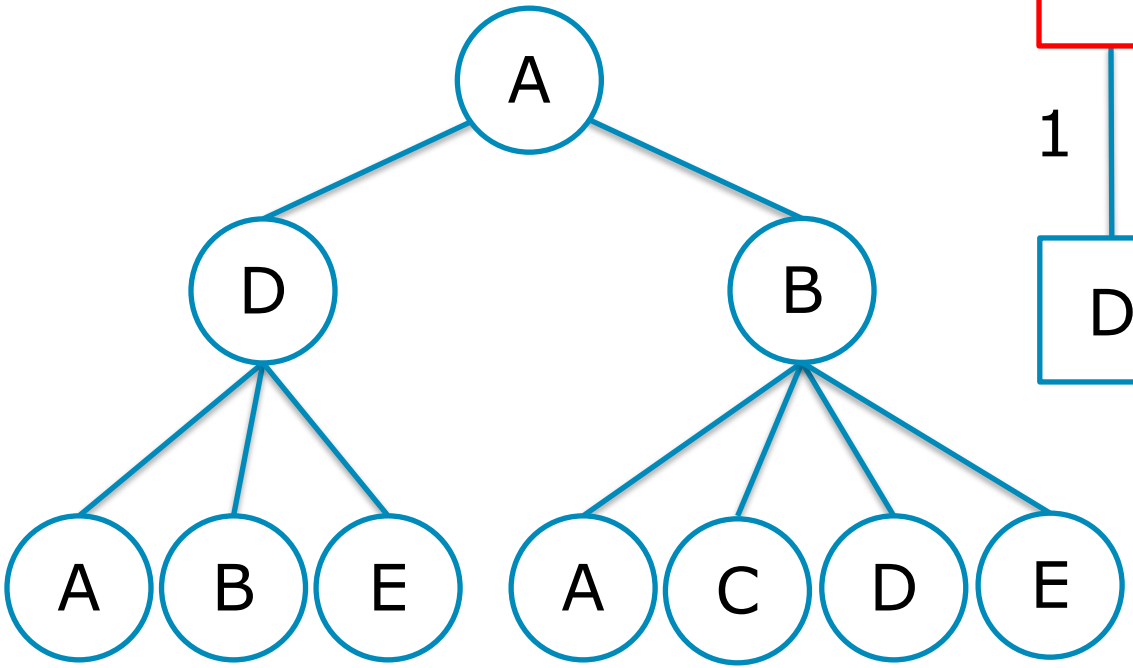


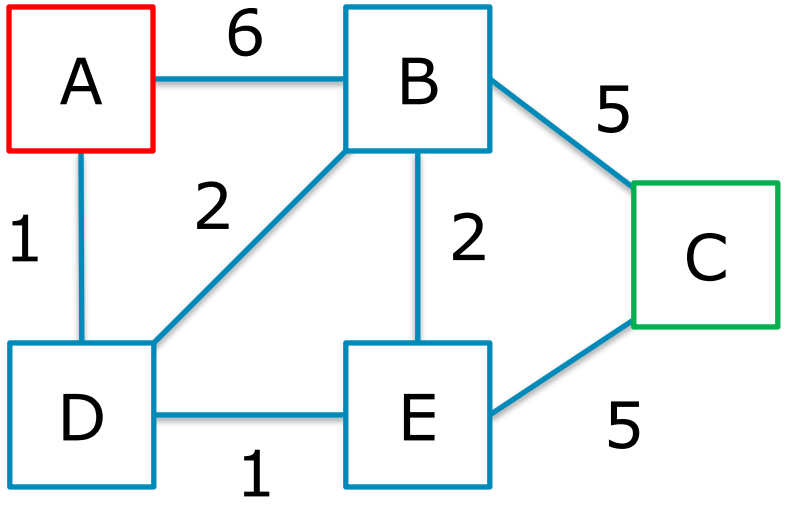
BFS

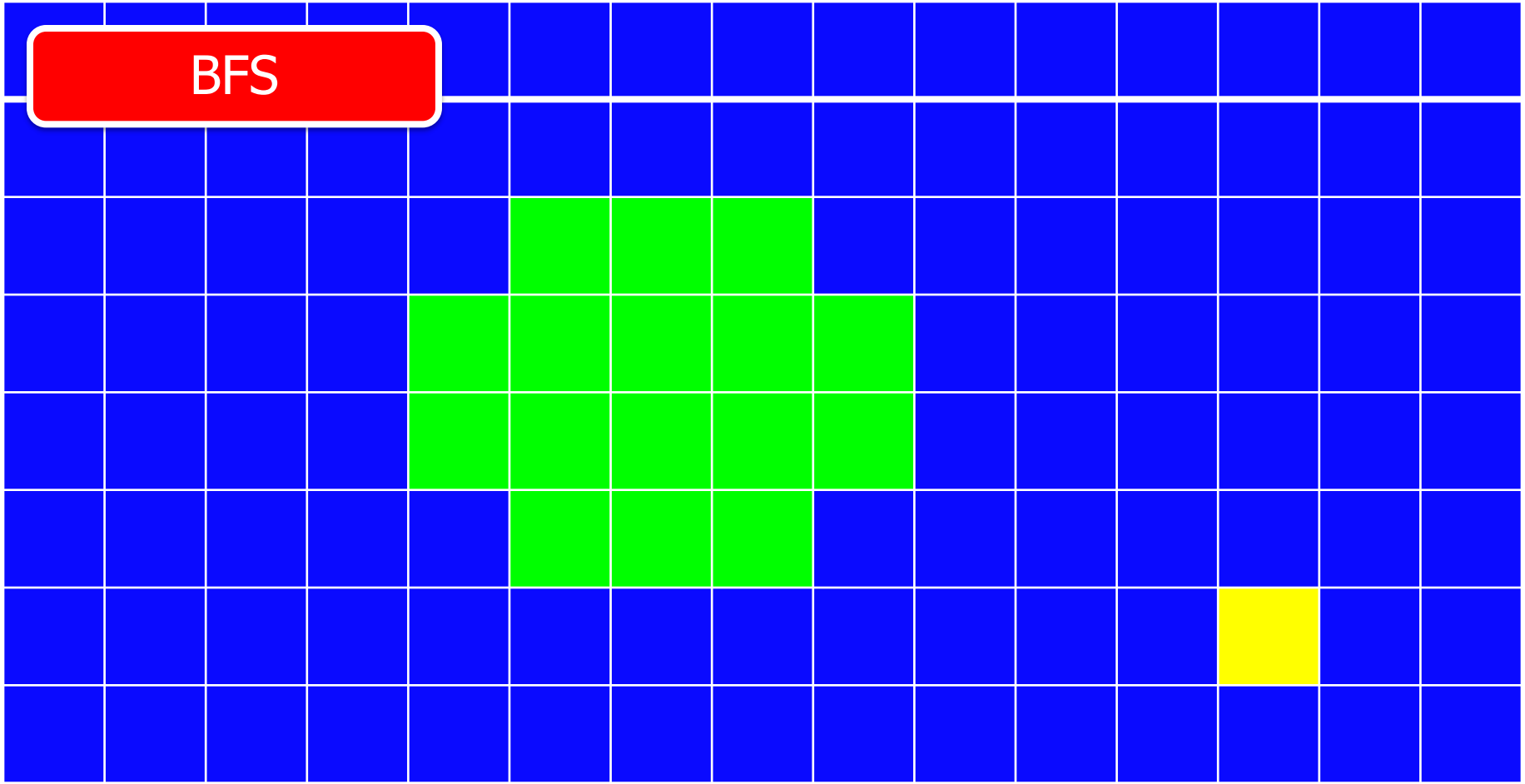


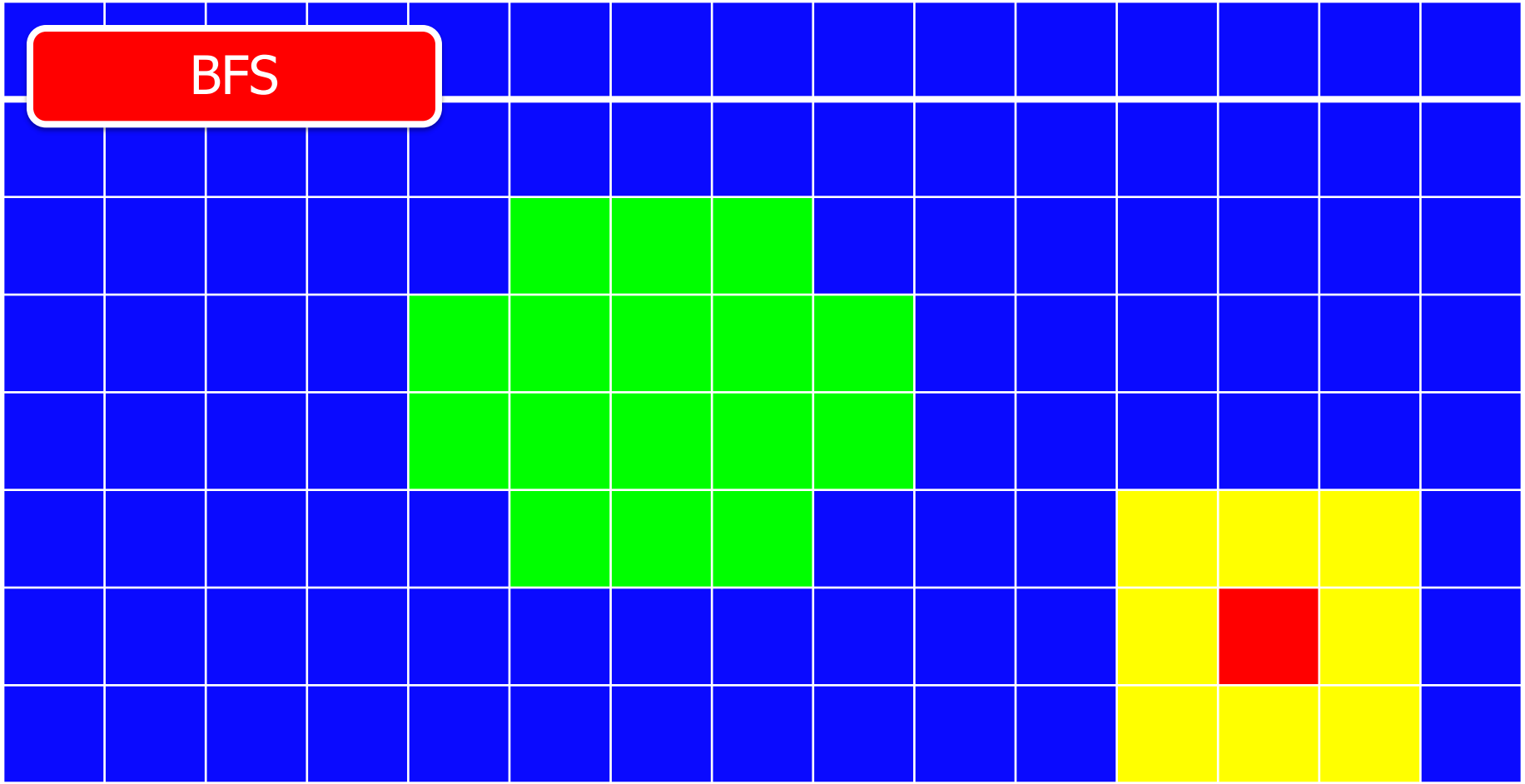


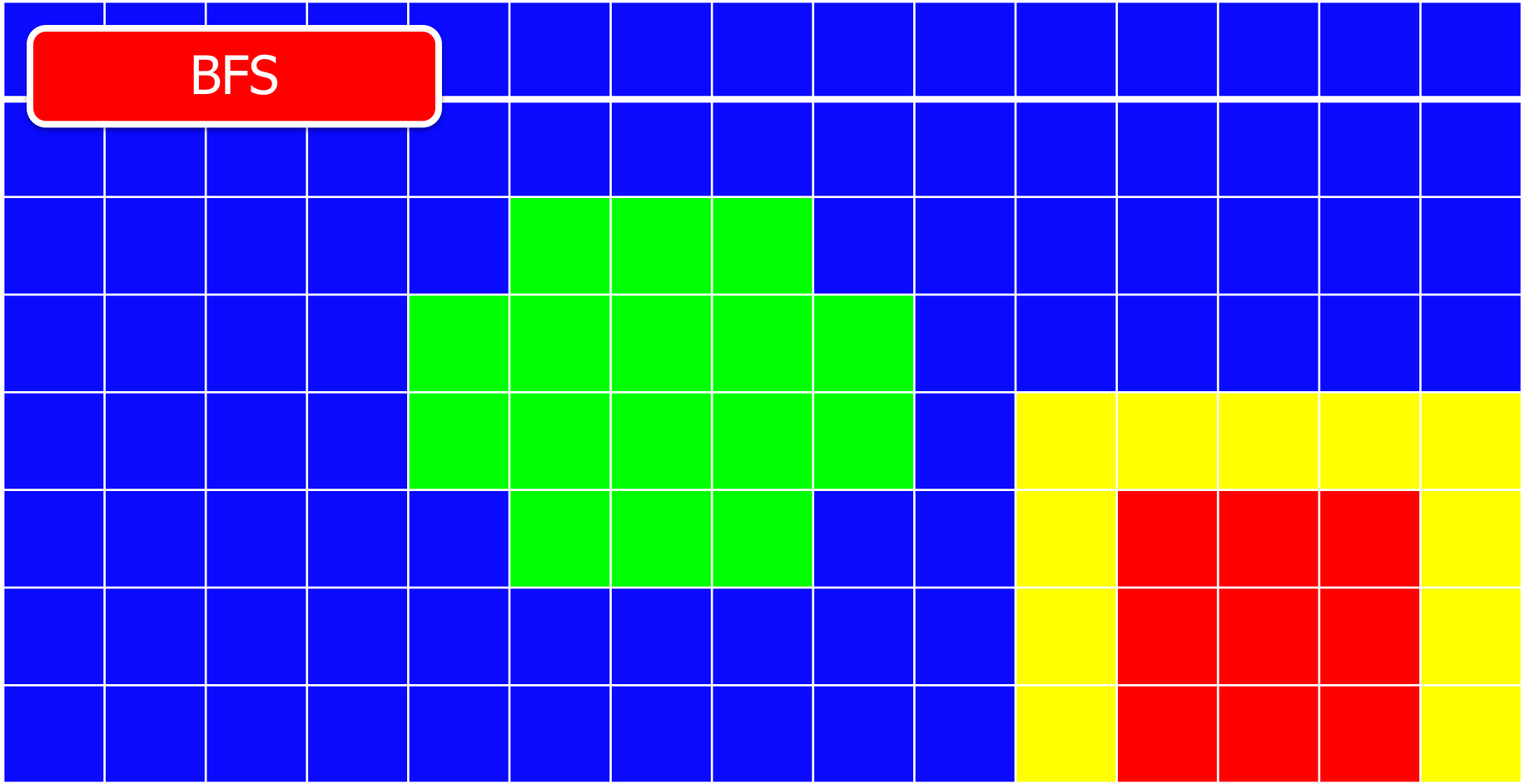
BFS

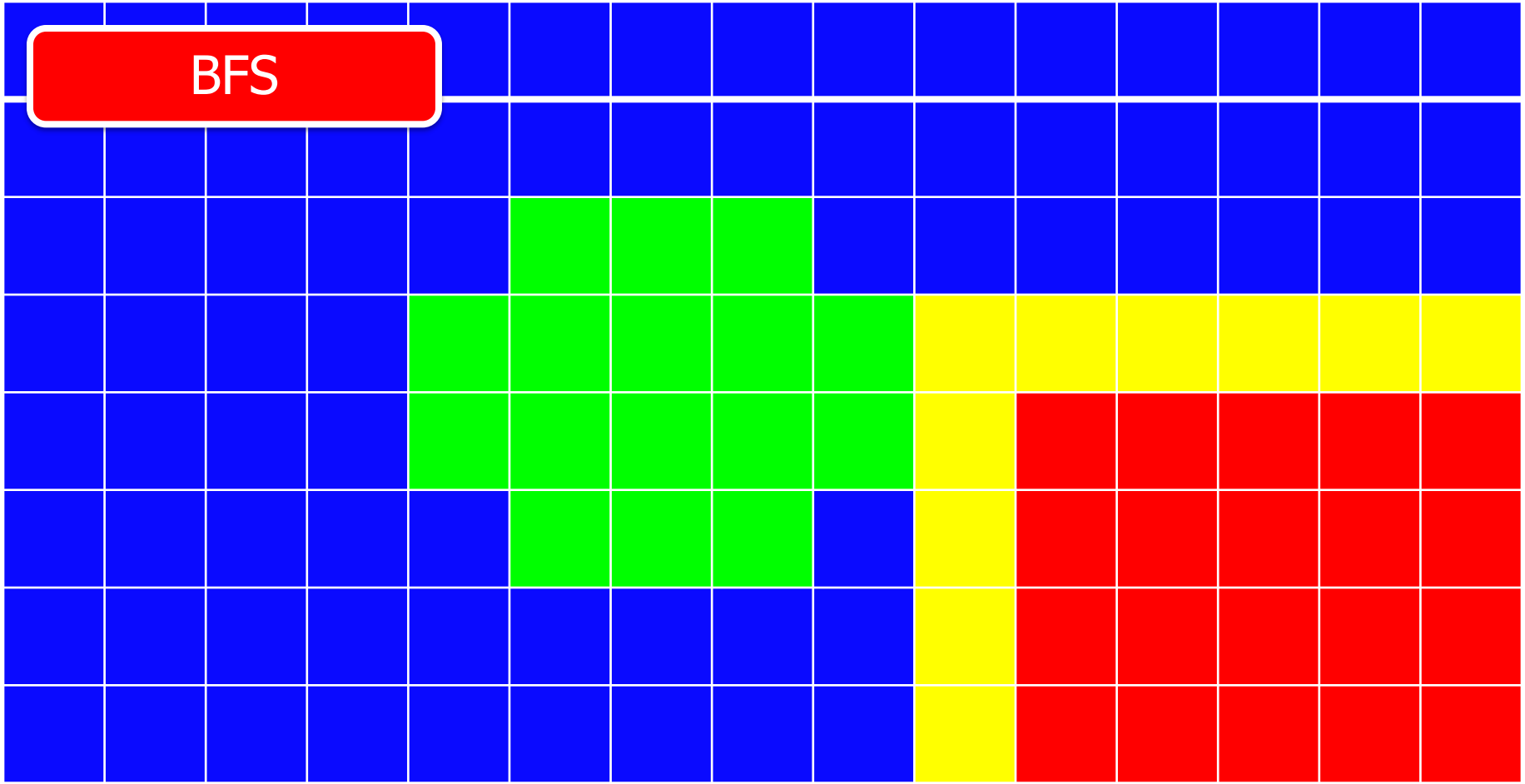


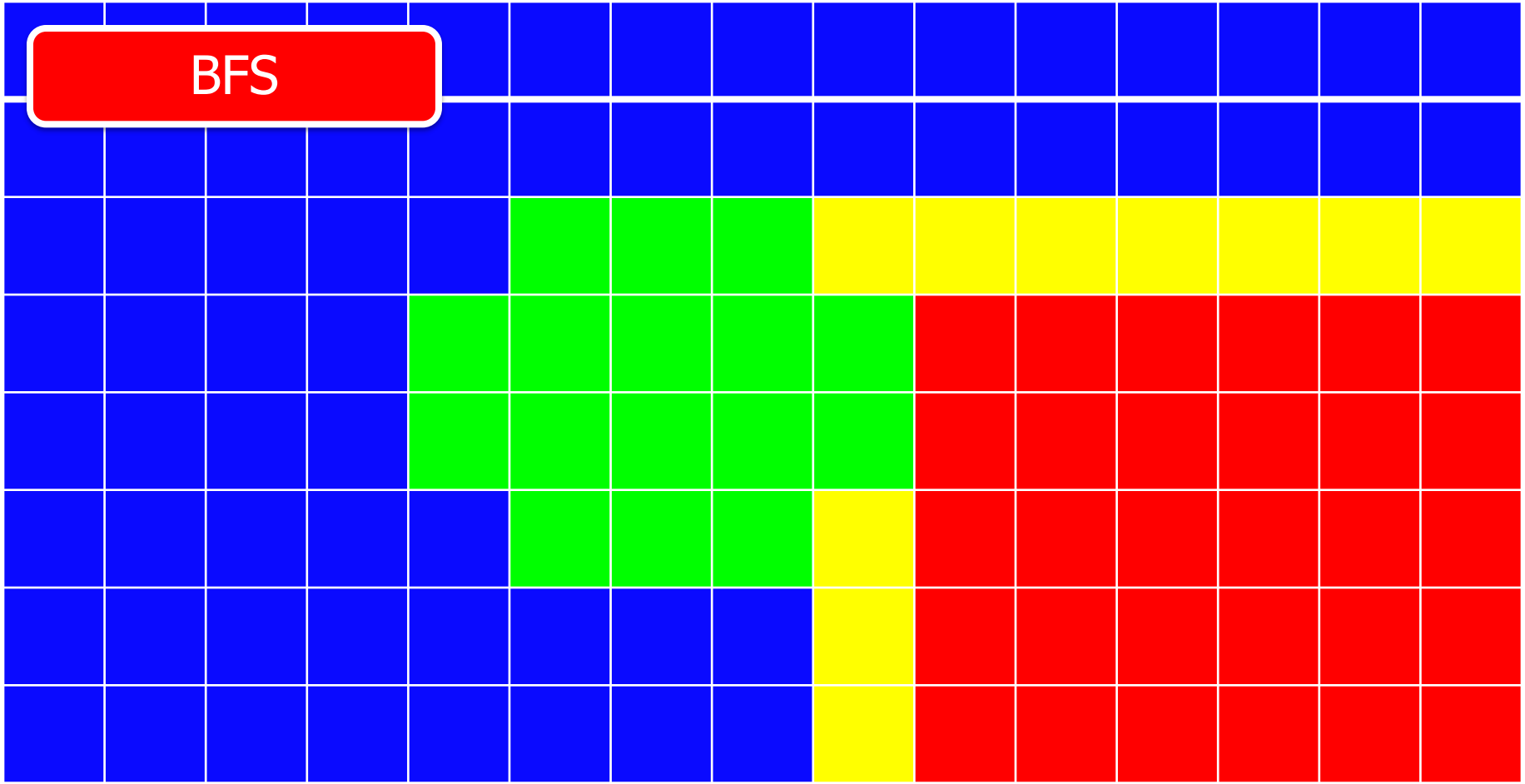


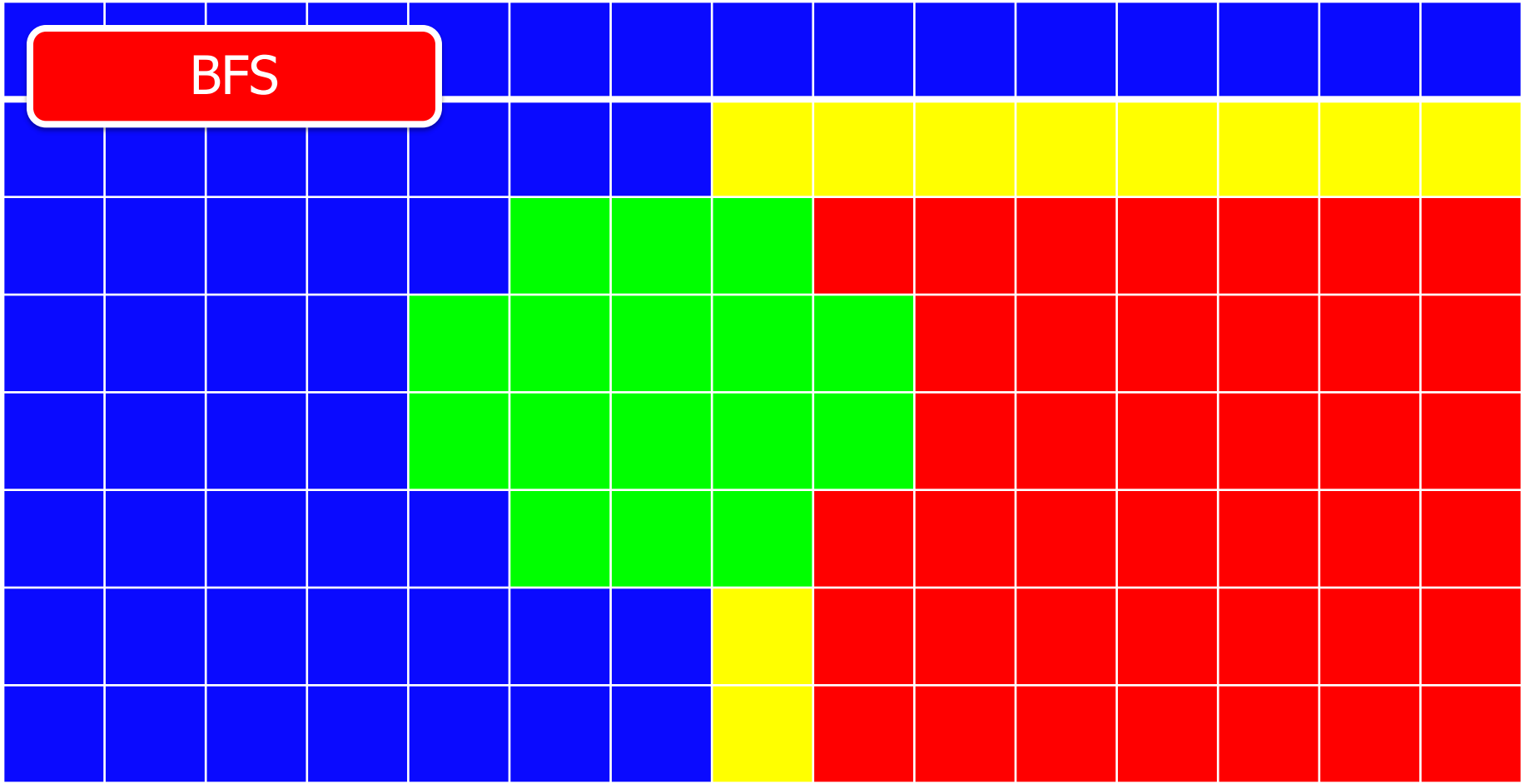

BFS


BFS


BFS


BFS


BFS


BFS

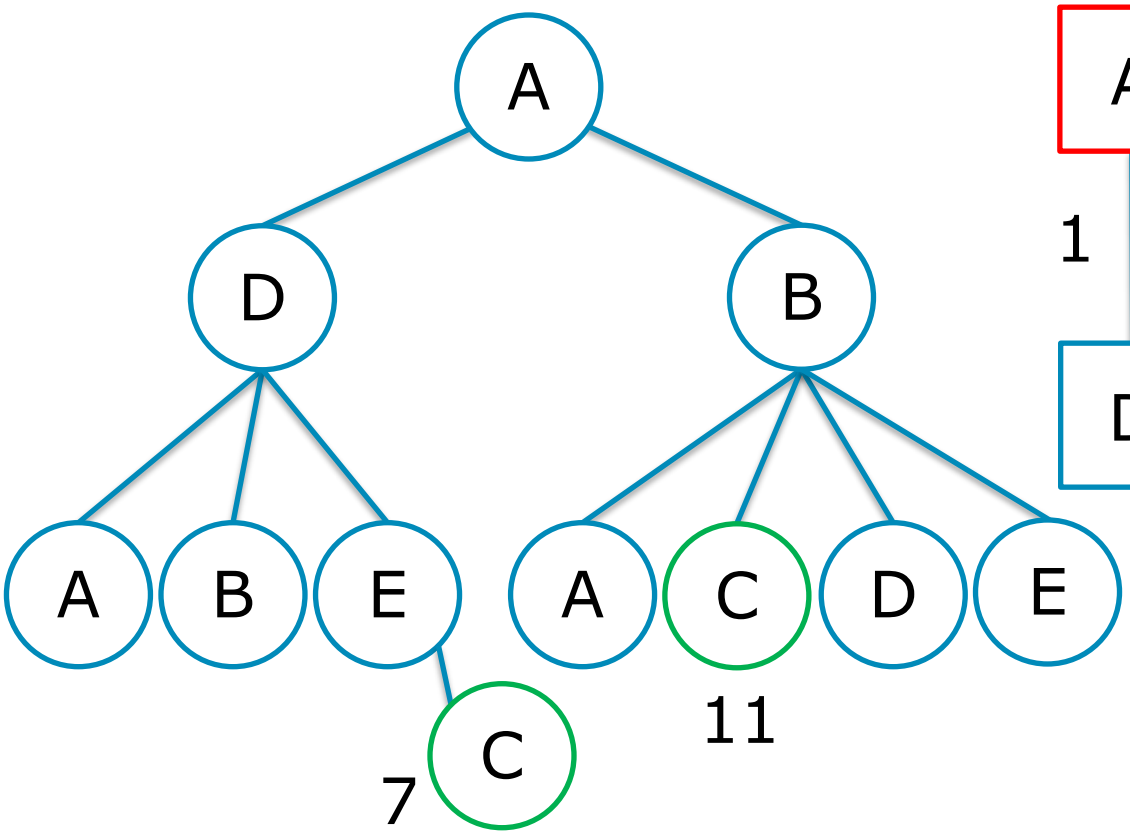
Problem Solving Performance

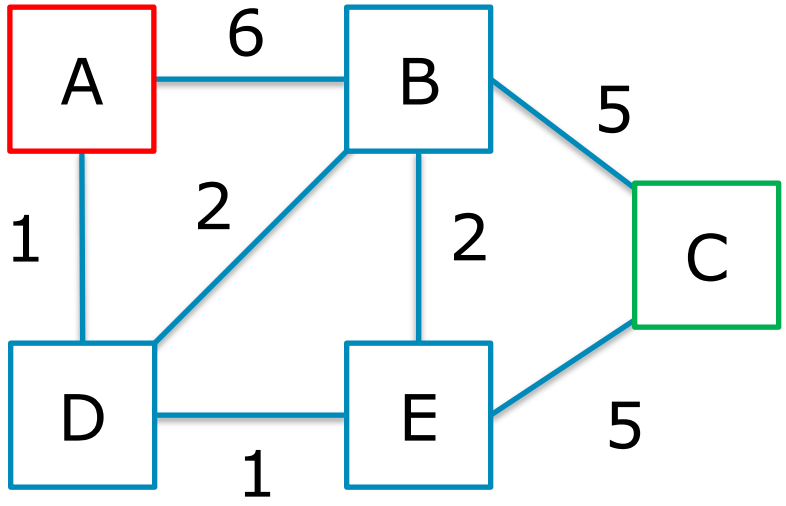
- Completeness
  - Is it guaranteed to find a solution if it exists
- Optimality
  - Does it find the optimal solution
- Time Complexity
  - How many nodes must it generate to find a soluton
- Space Complexity
  - How much memory is needed to run the search

BFS Performance

- BFS is Complete
  - BFS explores entire tree, so it will find a goal node
  - BFS will find the shallowest goal node, since it explores all nodes at a given depth before the next depth
- BFS is not Optimal
  - In general BFS does not produce optimal solutions, since the shallowest node is not necessarily optimal
  - BFS optimal if path cost is non-decreasing function on depth
  - BFS is optimal if action costs are all the same

Example: BFS Not Optimal





BFS Performance

- Time Complexity $O(b^d)$
  - Each state has b children
  - Consider goal at depth d
  - b * b * b…. = $O(b^d)$ nodes generated
- Space Complexity $O(b^d)$
  - Must store entire search tree in memory
  - Space = nodes generated

BFS Performance Table

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 MB
4	111,100	11 seconds	106 MB
6	10^7	19 minutes	10 GB
8	10^9	31 hours	1 TB
10	10^{11}	129 days	101 TB
12	10^{13}	35 years	10 PB
14	10^{15}	3523 years	1 EB

b = 10, 10000 nodes/s, 1000bytes / node

Uniform-Cost Search

- BFS optimal when all action costs equal because it expands the shallowest node
- Uniform cost search optimal for any cost
- UCS expands node with lowest path cost
- If all costs are equal, equivalent to BFS
- Works only if all costs > 0
  - Can infinite loop if 0 or negative costs
- UCS is Dijsktra's Algorithm for a single goal

Uniform Cost Search

```
1. Function Tree-Search(problem)
2.   fringe = {Node(problem.initial_state)}
3.   while (true)
4.     if (fringe.empty) return fail
5.     node = min_g_value(fringe)
6.     if (node.state is goal) return solution
7.     else fringe.add(Expand(node, problem))
```

Uniform Cost Search Complexity

- UCS is Complete if b is finite
- UCS is Optimal if action costs > 0 (ε)
- Time Complexity
  - Measured by path costs, not depth
  - Can’t use b and d for complexity
  - Let C* be the cost of optimal solution
  - Complexity = $O(b^{1+ \lfloor C^*/ \varepsilon \rfloor})$
  - Worst case this can be much bigger than b^d

Depth-First Search (DFS)

- Expands the deepest node on fringe
- Search goes immediately to the deepest level of the search tree to a leaf node
  - Leaf = Node has no successors
- If a leaf is reached, it is discarded and the search 'backs up' to previous depth
- Implementation
  - Use a Stack for fringe
  - More often implemented as recursive function

Depth-First Search

```
1. Function Tree-Search(problem)
2.   fringe = Stack{Node(problem.initial_state)}
3.   while (true)
4.     if (fringe.empty) return fail
5.     node = fringe.pop()
6.     if (node.state is goal) return solution
7.     else fringe.add(Expand(node, problem))
```

Breadth-First Search

```
1. Function Tree-Search(problem)
2.   fringe = Queue{Node(problem.initial_state)}
3.   while (true)
4.     if (fringe.empty) return fail
5.     node = fringe.pop()
6.     if (node.state is goal) return solution
7.     else fringe.add(Expand(node, problem))
```

Recursive DFS Implementation

```
1.  Function DFS(node, problem)
2.      if (node.state is goal) return solution
3.      successors = Expand(node, problem)
4.      for s in successors:
5.          DFS(s, problem)

Call DFS(root_node, problem)
```

DFS Performance

- DFS is not Complete
  - In general, DFS is not complete
  - For example, can enter infinite loop
  - (Enhancements can make it sort of complete)

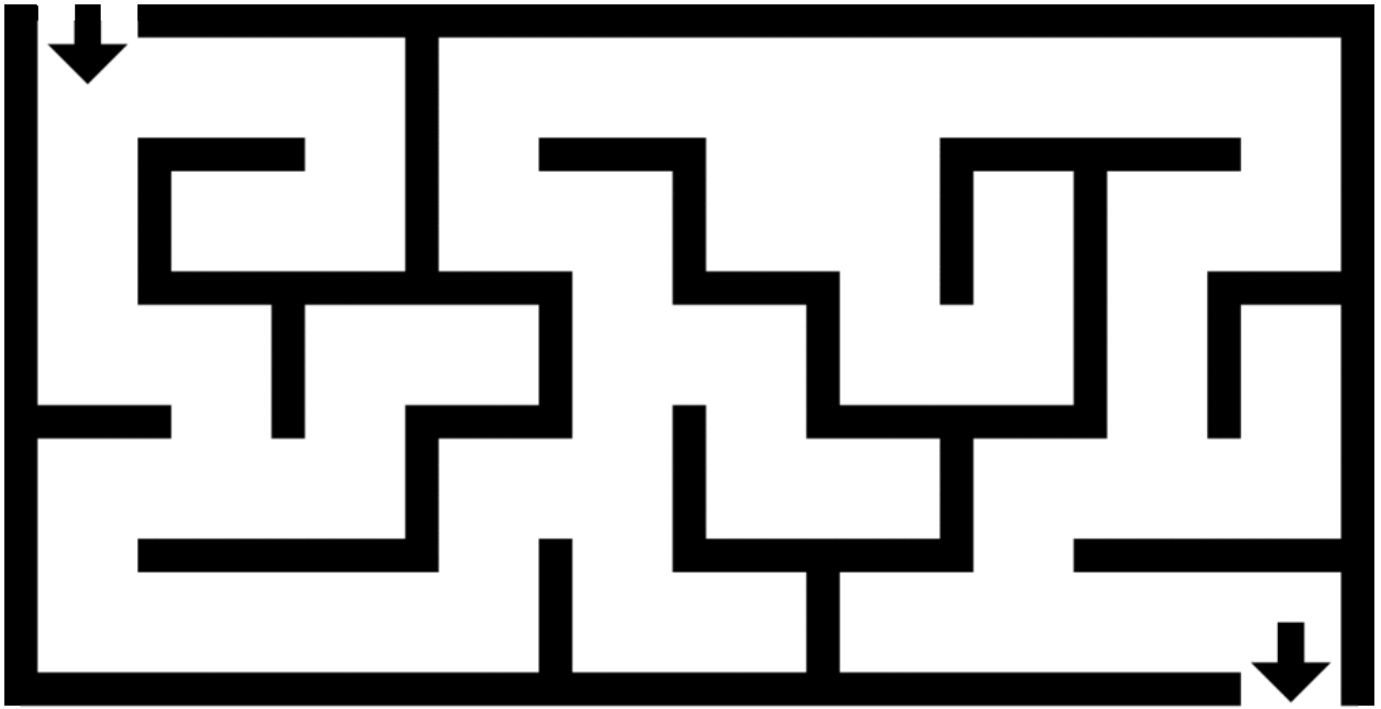
- DFS is not Optimal
  - In general, not optimal
  - Returns the first goal found, could be anywhere in the tree, not guaranteed to be optimal

DFS Performance

- Time Complexity $O(b^m)$
  - May go down long possibly infinite paths
  - May have to generate the entire tree
  - Generates b^m nodes where m is max depth of a node

- Space Complexity O(bm)
  - Only has to store the current path at any time
  - Uses O(bm) memory where m is max node depth
  - Stores successors of nodes in the path (b factor)

Depth-First Search



Depth-Limited Search (DLS)

- How to prevent DFS from 'getting lost'
- Apply a depth limit to the search
  - Choose a depth limit L
  - Enforce nodes at depth L have no successors
- Solves the infinite path problem
  - Will search entire tree up to depth L
- Introduces a new problem
  - Incomplete if solution depth d > L
- Time Complexity $O(b^L)$
- Space Complexity O(bL)

Depth-First Search

```
1. Function Tree-Search(problem)
2.   fringe = Stack{Node(problem.initial_state)}
3.   while (true)
4.     if (fringe.empty) return fail
5.     node = fringe.pop()
6.     if (node.state is goal) return solution
7.
8.     else fringe.add(Expand(node, problem))
```

Depth-Limited Search

```
1. Function Tree-Search(problem)
2.   fringe = Stack{Node(problem.initial_state)}
3.   while (true)
4.     if (fringe.empty) return fail
5.     node = fringe.pop()
6.     if (node.state is goal) return solution
7.     if (node.depth > L) continue
8.     else fringe.add(Expand(node, problem))
```

Recursive DLS

```
1.  Function DLS(node, L, problem)
2.      if (node.state is goal) return solution
3.      if (node.depth > L) return
4.      successors = Expand(node, problem)
5.      for s in successors:
6.          DLS(s, problem)
Call DLS(root_node, L, problem)
```

Choosing a Maximum Depth

- Analysis of maximum path length
- Known ‘diameter’ of a search space
- Number of states in problem

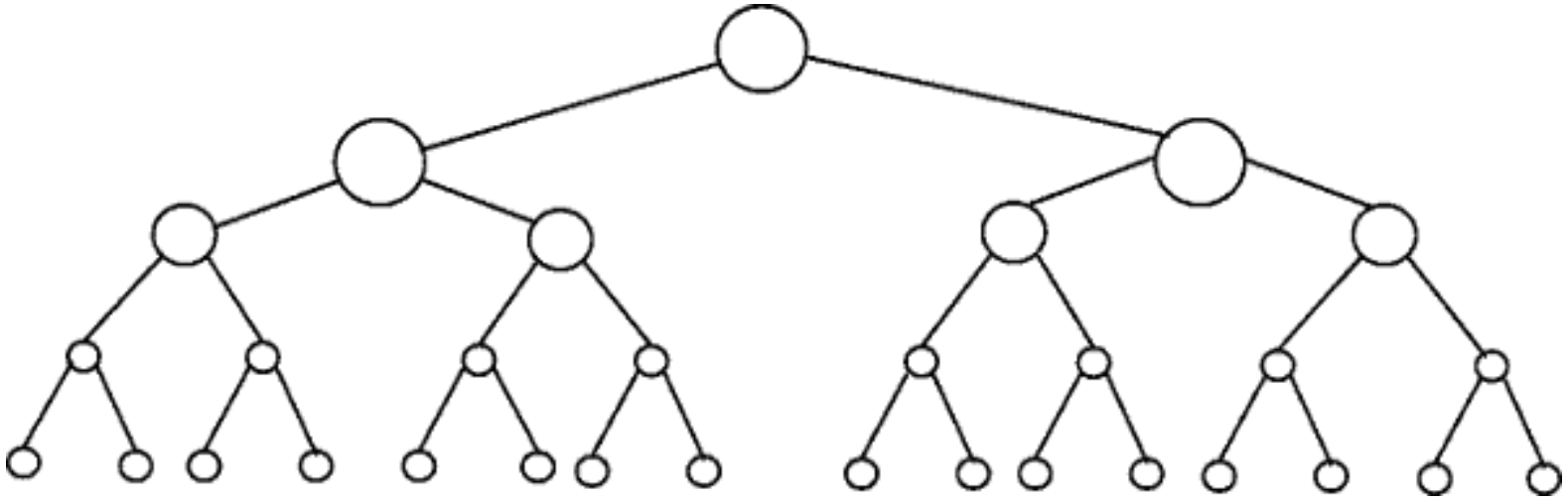
- Let the search figure it out!

Iterative Deepening DFS

- Main Idea: Gradually increase depth limit
  - Try max depth 1, 2, 3, …. M
- Goal will be found at shallowest depth d
  - No solution found at 1… d-1, d = shallowest
- Completeness + Optimality of BFS
  - Guaranteed to find a solution
  - Optimal iff action costs are the same
- Space Complexity of DFS O(bd)

ID-DFS Time Complexity

- May seem wasteful, some re-computation



ID-DFS Time Complexity

- Number of nodes searched
  - $(d)b + (d-1)b^2 + \ldots + 1(b^d)$
  - Time Complexity $O(b^d)$
- Same time complexity as BFS
- Same space complexity as DFS O(bd)
- In practice, generates ~ twice the nodes
- In general, ID-DFS is preferable to BFS

Iterative Deepening DFS

```
1.  Function ID-DFS(node, problem)
2.    for d in (1, infty)
3.      result = DLS(node, d, problem)
        // how to stop the search?
```

Iterative Deepening DFS

```
1.  Function ID-DFS(node, problem)
2.    for d in (1, infty)
3.      result = DLS(node, d, problem)
4.      if (result != cutoff) return result

// cutoff = some node reached depth limit d
//          instead of failure or solution
```

Depth-Limited Search

```
1. Function Tree-Search(problem)
2.   fringe = Stack{Node(problem.initial_state)}
3.   while (true)
4.     if (fringe.empty) return fail
5.     node = fringe.pop()
6.     if (node.state is goal) return solution
7.     if (node.depth > L) continue
8.     else fringe.add(Expand(node, problem))
```

Depth-Limited Search (for ID-DFS)

```
1.  Function Tree-Search(problem)
2.    fringe = Stack{Node(problem.initial_state)}
3.
4.    while (true)
5.      if (fringe.empty) return fail
6.      node = fringe.pop()
7.      if (node.state is goal) return solution
8.      if (node.depth > L) continue
9.      else fringe.add(Expand(node, problem))
```

Depth-Limited Search (for ID-DFS)

```
1.  Function Tree-Search(problem)
2.    fringe = Stack{Node(problem.initial_state)}
3.    had_cutoff = false
4.    while (true)
5.      if (fringe.empty) return had_cutoff ? cutoff : fail
6.      node = fringe.pop()
7.      if (node.state is goal) return solution
8.      if (node.depth > L) had_cutoff = true; continue
9.      else fringe.add(Expand(node, problem))
```

Recursive DLS (for ID-DFS)

```
1. Function DLS(node, L, problem)
2.       had_cutoff = false
3.       if (node.state is goal) return solution
4.       if (node.depth >= L) return cutoff
5.       for s in Expand(node, problem):
6.              result = DLS(s, L, problem)
7.              if (result == cutoff) had_cutoff = true
8.              else if (result != fail) return result
9.       return (had_cutoff ? cutoff : failure)

Call DLS(root_node, L, problem)
```

Iterative Deepening DFS

```
1.  Function ID-DFS(node, problem)
2.    for d in (1, infty)
3.      result = DLS(node, d, problem)
4.      if (result != cutoff) return result

// cutoff = some node reached depth limit d
//          instead of failure or solution
```

Comparing Search Strategies

Criterion	BFS	UCS	DFS	DLS	ID-DFS
Complete	YES [a]	YES [ab]	NO	NO	YES [a]
Optimal	YES [c]	YES	NO	NO	YES [c]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^L)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(bm)$	$O(bL)$	$O(bd)$

(a) if b is finite (b) if action costs > 0 (c) if all costs equal

Comparing Search Strategies

Criterion	BFS	UCS	DFS	DLS	ID-DFS
Complete	YES [a]	YES [ab]	NO	NO	YES [a]
Optimal	YES [c]	YES	NO	NO	YES [c]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^L)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	O(bm)	O(bL)	O(bd)

(a) if b is finite (b) if action costs > 0 (c) if all costs equal

Note: Terminology

- Algorithm terminology / name from

- Artificial Intelligence: A Modern Approach
  - Stuart Russel and Peter Norvig
  - Not required for course, but amazing book

- Names not important, concepts are

Avoiding Repeated States

- One of the most important search ideas
- Especially important with reversible actions
- Most infinite loops / wasted time can be avoided by not returning to identical states
- We can remember nodes expanded
  - Don’t re-expand them
- Can lead to exponential savings in the number of nodes generated

‘Closed List’

- Store states we have already expanded
- Closed List stores expanded states
  - Often implemented as a hash table / dictionary for efficient lookup
- Check if a node is closed before expanding
- Open List = fringe of unexpanded nodes
  - Often implemented as a priority queue

General Tree-Search (recall)

```
1. Function Tree-Search(problem, strategy)

2.      open = {Node(problem.initial_state)}
3.      while (true)
4.          if (open.empty) return fail
5.          node = strategy.select_node(open)
6.          if (node.state is goal) return solution


7.          open.add(Expand(node, problem))
```

General Graph-Search

```
1.  Function Graph-Search(problem, strategy)
2.      closed = {} // add closed list
3.      open = {Node(problem.initial_state)}
4.      while (true)
5.          if (open.empty) return fail
6.          node = strategy.select_node(open)
7.          if (node.state is goal) return solution
8.          if (node.state in closed) continue // check if closed
9.          closed.add(node.state) // mark node closed
10.         open.add(Expand(node, problem))
```

General Graph-Search

```
1.  Function Graph-Search(problem, strategy)
2.      closed = {}
3.      open = {Node(problem.initial_state)}
4.      while (true)
5.          if (open.empty) return fail
6.          node = strategy.select_node(open)
7.          if (node.state is goal) return solution
8.          if (node.state in closed) continue
9.          closed.add(node.state)
10.         open.add(Expand(node, problem))
```

General Graph-Search (Note)

- Only called ‘Graph-Search’ because that is how it is described in the AI text book
- It is still an algorithm that searches a ‘search tree’, it just uses the closed list to avoid repeated searching of nodes that represent the same environment state

Assignment 1: BFS

```
1.  Function Graph-Search(problem, BFS)
2.      closed = {}
3.      open = Queue{Node(problem.initial_state)}
4.      while (true)
5.          if (open.empty) return fail
6.          node = open.pop() // get from front of queue
7.          if (node.state is goal) return solution
8.          if (node.state in closed) continue
9.          closed.add(node.state)
10.         open.add(Expand(node, problem))
```

Assignment 1: DFS

```
1.  Function Graph-Search(problem, DFS)
2.      closed = {}
3.      open = Stack{Node(problem.initial_state)}
4.      while (true)
5.          if (open.empty) return fail
6.          node = open.pop() // get from top of stack
7.          if (node.state is goal) return solution
8.          if (node.state in closed) continue
9.          closed.add(node.state)
10.         open.add(Expand(node, problem))
```

Tree-Search vs Graph-Search

- Tree Search remembers nothing
  - Will re-visit states multiple times
- Graph-Search remembers states visited
  - Will never re-visit states a second time
- Tree/Graph Search = Just a name

Graph-Search Complexity

- Time Complexity O(states)
  - Each state visited at most once
  - Worst case, expand every state in environment
  - In most cases, far lower than $O(b^d)$
  - Same upper bound no matter which search strategy
- Space Complexity O(states)
  - Must store entire search space in memory
  - Space = nodes generated = # states
  - Better, but often times still too much
  - Previously linear space methods may now use more memory

Graph-Search Optimality

- Graph-Search may no longer be Optimal
  - Tricky issue, since it depends on underlying strategy
  - We a repeated state is detected, the algorithm has found another path to the same state
  - Graph-Search (as shown) keeps the original path, and discards new paths found, which may be optimal
  - Only optimal if we can guarantee that the first solution found is the optimal one (BFS, UCS, with same costs)

Recap / Exam Questions

- Problems are defined by
  - Initial + Goal States, Actions, Cost Function
- Optimal solution
  - Has lowest cost function among all solutions
- Search
  - Finds solution by exploring the search tree
  - Search tree consists of Nodes, Edges
  - Nodes in search tree != states in problem
- Problem solving performance measured by
  - Completeness, Optimality, Space + Time Complexity

Recap / Exam Questions

- What is a search strategy
- BFS / UCS / DFS / ID-DFS
  - Algorithm Pseudocode
  - Completeness, Optimality
  - Time / Memory Complexity
- Open List / Closed List
  - Differences / Usage of Both
  - Node vs. State difference
  - Data stored in Node structure (and why)