CompSci 370
Uninformed Search

Ron Parr
Department of Computer Science
Duke University

With thanks to Vince Conitzer for some slides and figures and thanks to Kris Hauser for many slides

What is Search?

- Search is a basic problem-solving method
  - We start in an initial state
  - We examine states that are (usually) connected by a sequence of actions to the initial state
- Note: Search is (usually) a thought experiment (separate topic: Real Time Search)

- We aim to find a solution, which is a sequence of actions that brings us from the initial state to the goal state, *possibly* minimizing cost

Search vs. Web Search

- When we issue a search query using Google, does Google really go poking around the web for us?

- Not in real time!
- Google spiders the web continually, caches results
- Uses page rank algorithm to find the most “popular” web pages that are consistent with your query

Overview

- Problem Formulation

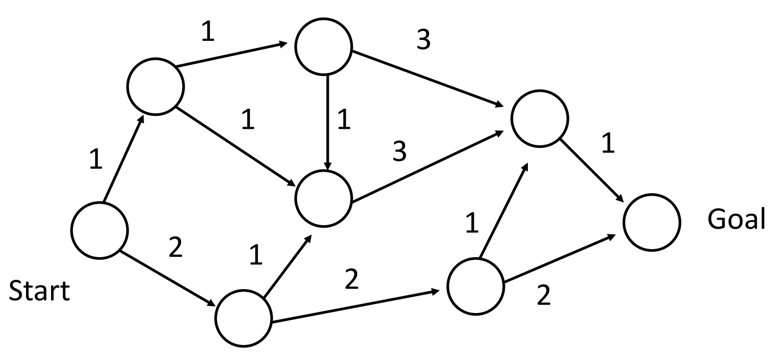
- Uninformed Search – constant cost
  - DFS, BFS, IDDFS, etc.

- Non-constant cost

Problem Formulation

- Components of a search problem
  - State space & initial state
  - Actions
  - Goal Test
  - Edge costs
    - May be constant or varying per edge (initially we assume constant)
    - Assumed to be > 0
- Optimal solution = lowest path cost to goal

Example: Path Planning, e.g. Google Maps

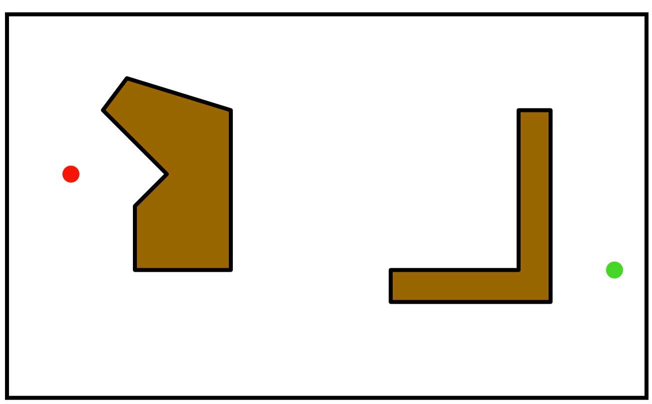


Find shortest source to destination using available roads

Other Search Problems

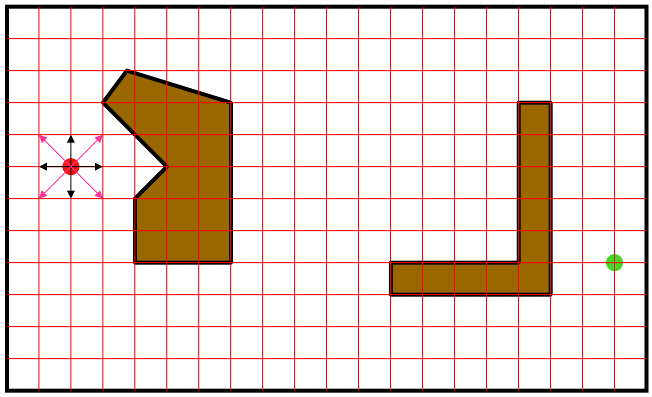
- Drug design
- Logistics
  - Route planning
  - Tour Planning
- Assembly sequencing
- Internet routing
- Robot motion/path planning

Robot Path Planning



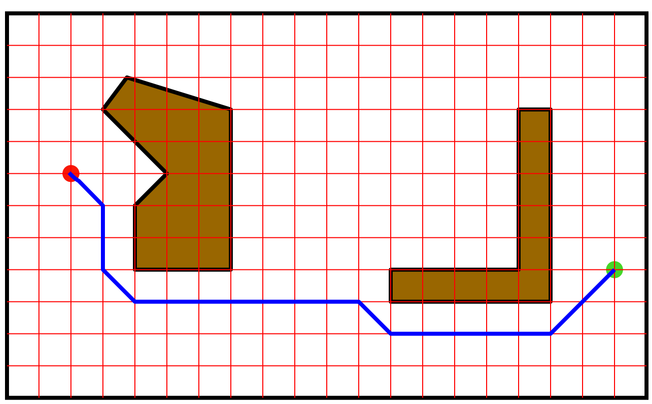
What is the state space?

Formulation #1



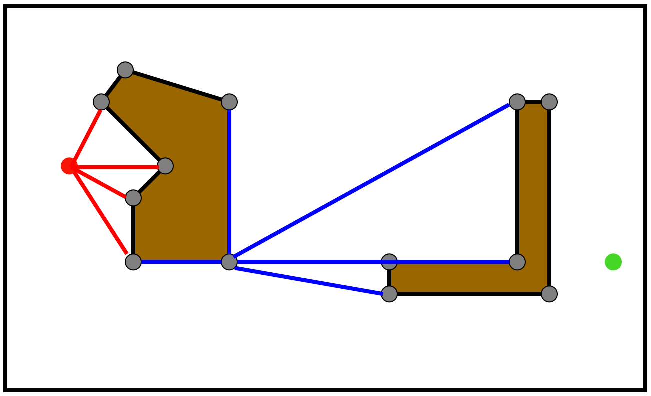
Cost of one horizontal/vertical step = 1
Cost of one diagonal step = sqrt(2)

Optimal Solution



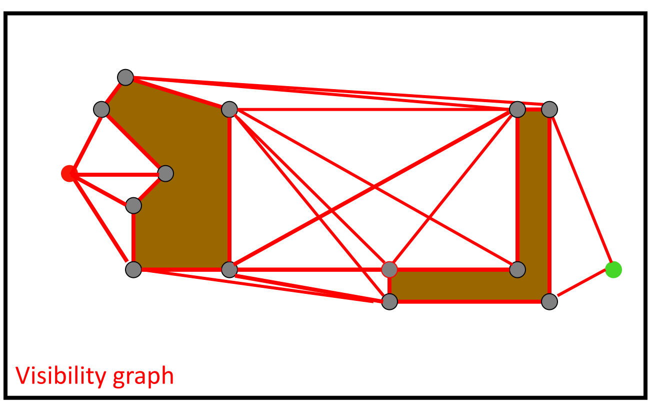
This path is the shortest in the discretized state space, but not in the original continuous space

Formulation #2



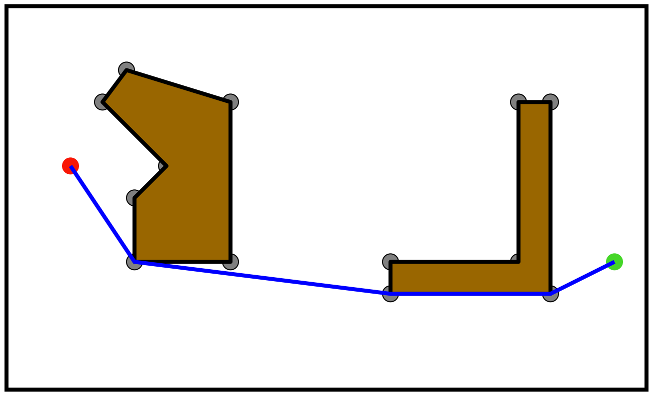
Cost of one step: length of segment

Formulation #2



Cost of one step: length of segment

Solution Path



The shortest path in this state space is also the shortest in the original continuous space

13

Take Home Points

- States = modeling choice about the world
- Trade offs often exist:
  - Example 1: Discretization is easy to work with, but optimal solution to may be suboptimal in the real world
  - Example 2: More clever representations may require ingenuity to discover, or use, but may have benefits in real world
- Always keep modeling and solving distinct in your head

Basic Search Concepts

- Search tree: Internal representation of our progress
- Nodes: Places in search tree
  (states exist in the problem space)
- Actions: Connect states to next states (nodes to nodes)
- Expansion: Generation of next states (nodes)
- Arc cost: Cost of moving from one state to another
- Frontier: Set of nodes visited, but not expanded
- Branching factor: Max no. of successors = b
- Goal depth: Depth of *shallowest* goal = d
  (root is depth 0, possibility of multiple goal states!)

Example: 8-Puzzle

8	2	
3	4	7
5	1	6

Initial state

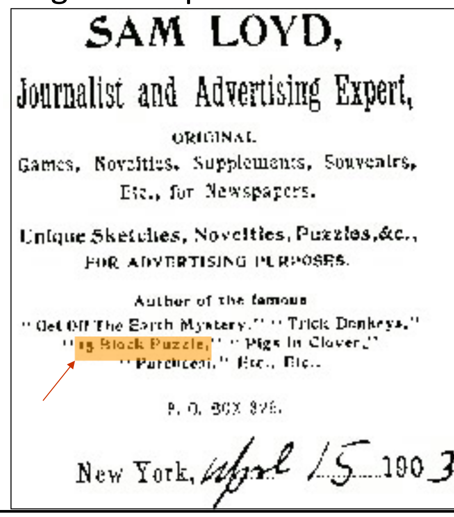
1	2	3
4	5	6
7	8	

Goal state

State: Arrangement of 8 numbered tiles & empty tile on a 3x3 board

15-Puzzle

- Introduced (?) in 1878 by Sam Loyd, who dubbed himself "America's greatest puzzle-expert"



SAM LOYD,

Journalist and Advertising Expert,

ORIGINAL

Games, Novelties, Supplements, Souvenirs, Etc., for Newspapers.

Unique Sketches, Novelties, Puzzles, &c., FOR ADVERTISING PURPOSES.

Author of the famous

"Get Off The Earth Mystery," "Trick Donkeys," "15 Block Puzzle," "Pigs in Clover," Etc., Etc.

New York, 15 1903

15-Puzzle

- Sam Loyd offered $1,000 of his own money to the first person who would solve the following problem:

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

?

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

How big is the state space of the (n^2-1)-puzzle?

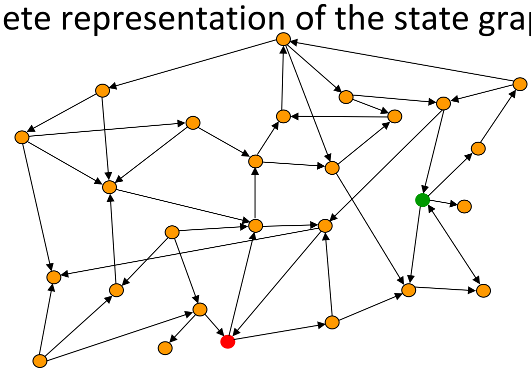
- 8-puzzle → 9! = 362,880 states
- 15-puzzle → 16! ~ 2.09 x 10^{13} states
- 24-puzzle → 25! ~ 10^{25} states

- But only half of these states are reachable from any given state (but you may not know that in advance)

- No one ever won the prize !!

Searching the State Space

- Often infeasible (or too expensive) to build complete representation of the state graph



- Key difference from algorithms class, where it is typically assumed that graph fits in memory

8-, 15-, 24-Puzzles

8-puzzle → 362,880 states

0.036 sec

15-puzzle → 2.09 x 10^{13} states

~ 55 hours

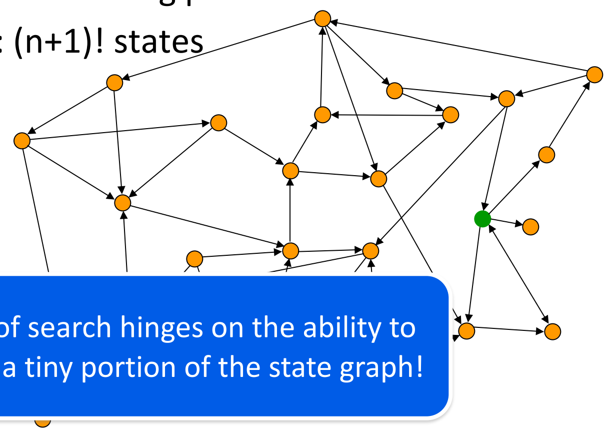
24-puzzle → 10^{25} states

> 10^{9} years

100 million states/sec

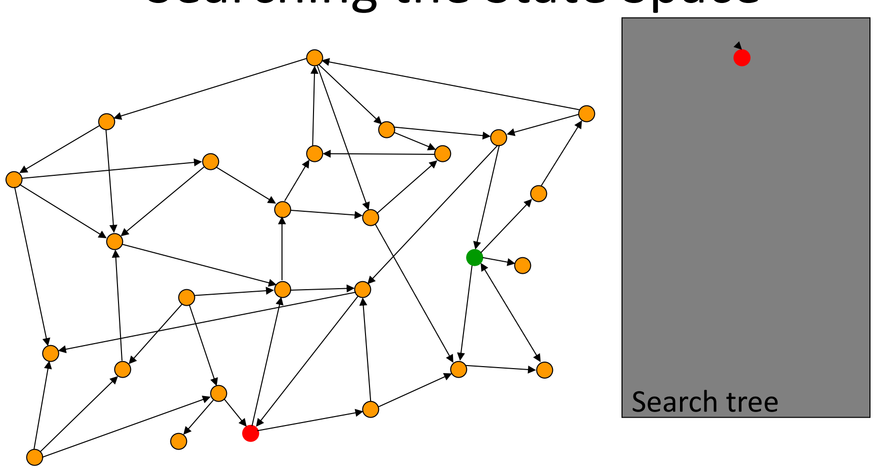
Intractability

- Constructing the full state graph is intractable for many interesting problems
- n-puzzle: (n+1)! states

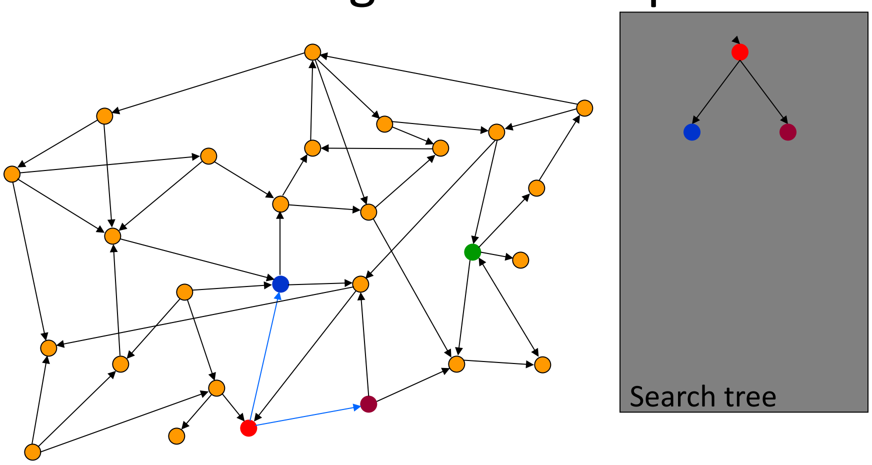


Tractability of search hinges on the ability to explore only a tiny portion of the state graph!

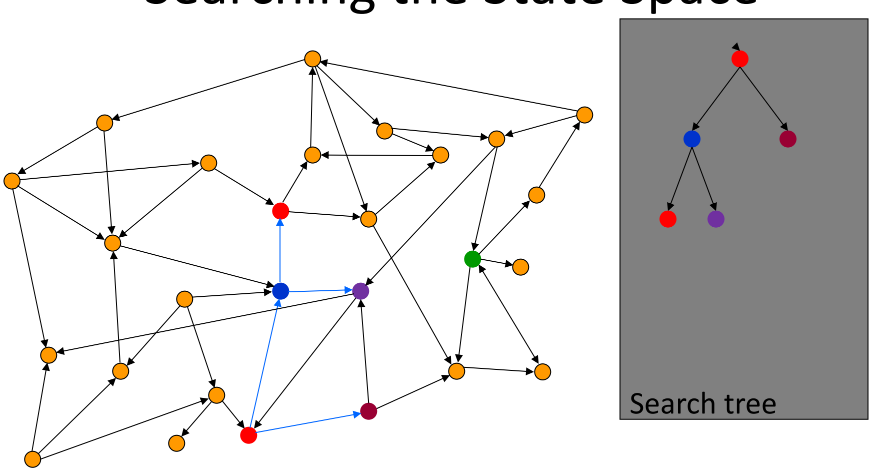
Searching the State Space



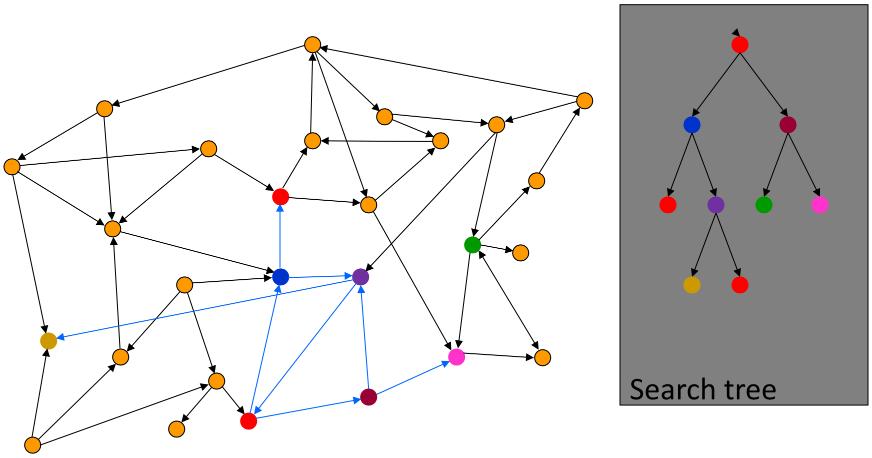
Searching the State Space



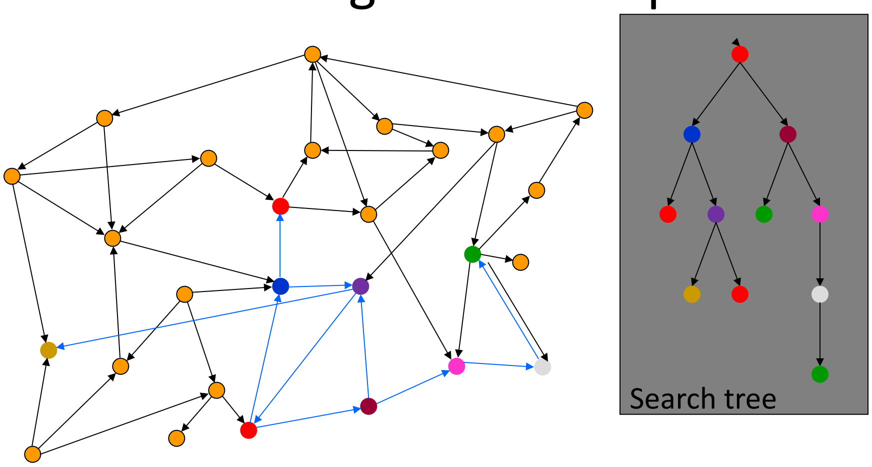
Searching the State Space



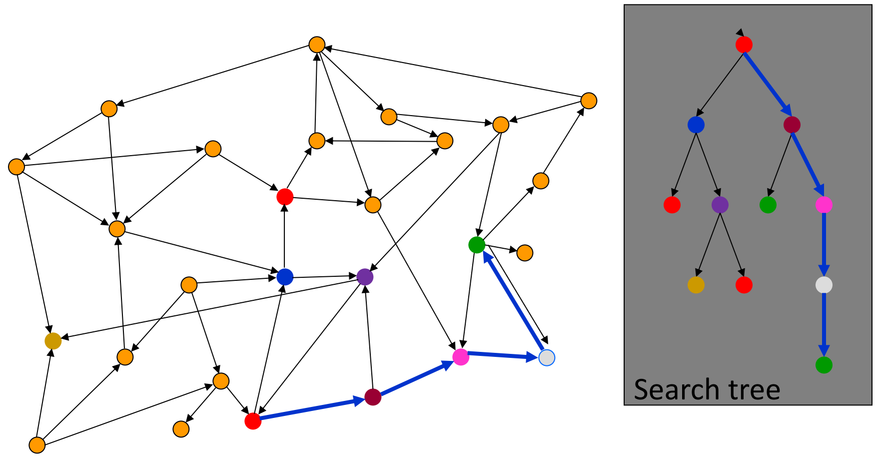
Searching the State Space



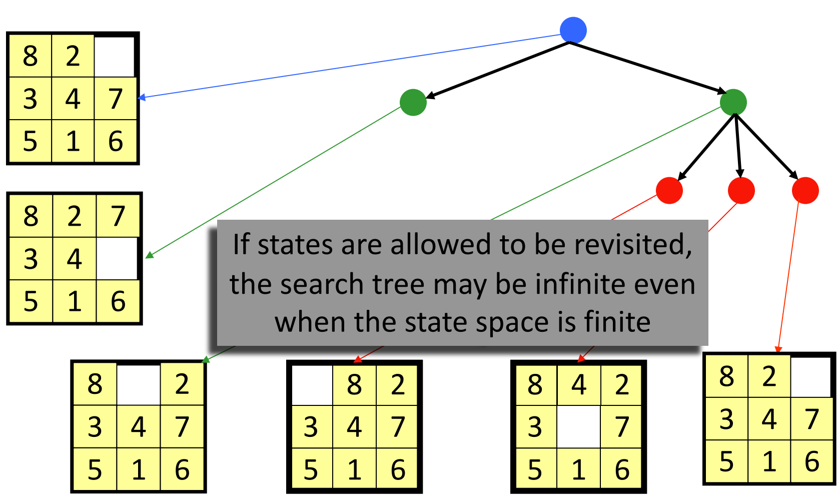
Searching the State Space



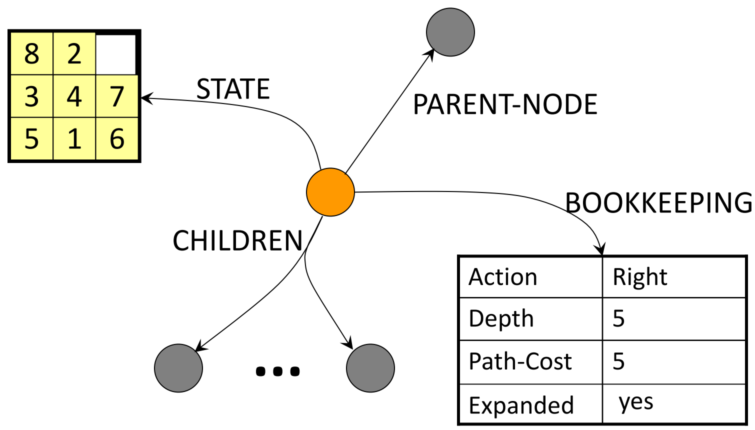
Searching the State Space



Search Nodes and States



Data Structure of a Node

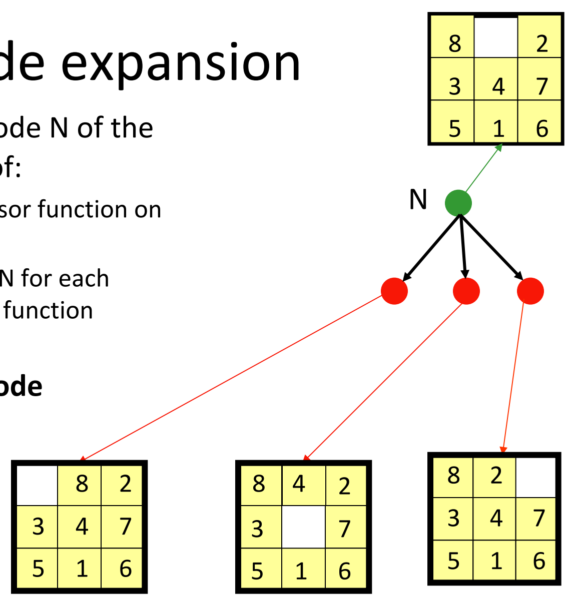


Depth of a node N
= length of path from root to N

(depth of the root = 0)

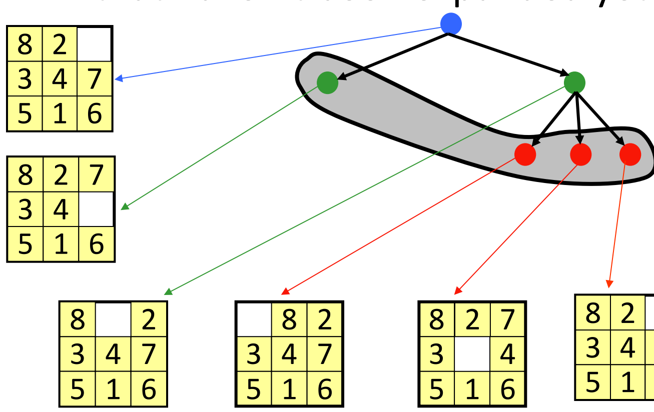
Node expansion

- The expansion of a node N of the search tree consists of:
  - Evaluating the successor function on STATE(N)
  - Generating a child of N for each state returned by the function
- **node generation ≠ node expansion**



Frontier of Search Tree

- The frontier is the set of all search nodes that haven't been expanded yet



Search Strategy

- The frontier is the set of all search nodes that haven't been expanded yet
- Implemented as a priority queue FRONTIER
  - INSERT(node, FRONTIER)
  - REMOVE(FRONTIER)
- The ordering of the nodes in FRONTIER defines the search strategy

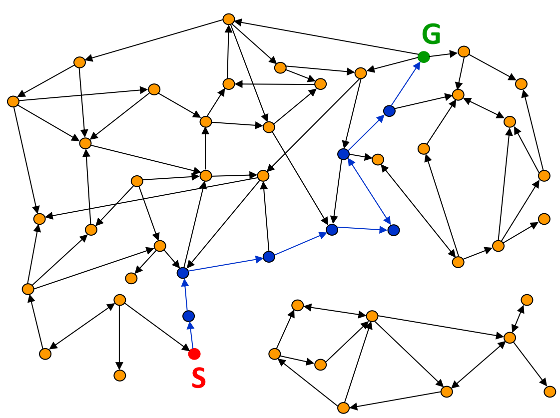
Generic Tree Search (*assumes* state space is a tree)

TREE-SEARCH(initial-state)

```
1.  If GOAL?(initial-state) then return initial-state
2.  INSERT(initial-node, FRONTIER)
3.  Repeat:
4.   If empty(FRONTIER) then return failure
5.      N ← REMOVE(FRONTIER)
6.      s ← STATE(N)                          Expansion of N
7.      For every state s' in SUCCESSORS(s)
8.         Create a new node N' as a child of N
9.         If GOAL?(s') then return path or goal state
10.        INSERT(N', FRONTIER)
```

Solution to the Search Problem

- A solution is a path connecting the initial node to a goal node (any goal)
- The cost of a path is the sum of the arc costs along this path
- An optimal solution is a solution path of minimum cost
- There might be no solution !



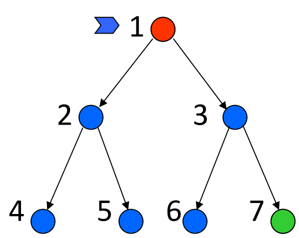
Recall: Typically assume costs > 0

Algorithm Performance Measures

- Completeness:
  - Does it find a solution when one exists?
- Optimality:
  - Does it return a min cost path whenever solution exists?
- Complexity (space or time):
  - Resources required by the algorithm

Breadth-First Search

- FRONTIER is a FIFO Queue

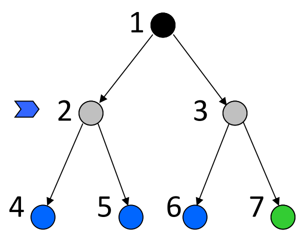


FRONTIER = (1)

Note: Typically assume that nodes are generated in left-to right order.

Breadth-First Search

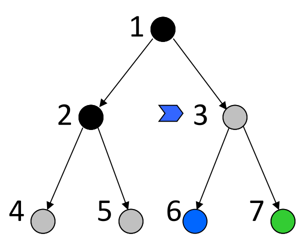
- FRONTIER is a FIFO Queue



FRONTIER = (2, 3)

Breadth-First Search

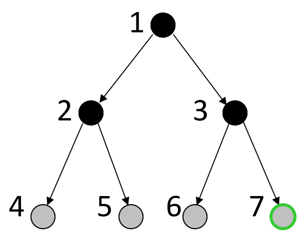
- FRONTIER is a FIFO Queue



FRONTIER = (3, 4, 5)

Breadth-First Search

- FRONTIER is a FIFO Queue



FRONTIER = (4, 5, 6, 7)

BFS Properties

- Completeness: Y
- Optimality: (Y for constant cost, N for arbitrary cost)
- Time complexity (nodes generated): $O(b^{d+1})$
- Space complexity: $O(b^d)$

Note: We are counting nodes generated in time complexity; textbook counts nodes expanded (so exponent can be 1 less).

How bad is exponential in d?

d	# Nodes	Time	Memory
2	111	.01 msec	11 Kbytes
4	11,111	1 msec	1 Mbyte
6	~10^6	1 sec	100 Mb
8	~10^8	100 sec	10 Gbytes
10	~10^{10}	2.8 hours	1 Tbyte
12	~10^{12}	11.6 days	100 Tbytes
14	~10^{14}	3.2 years	10,000 Tbytes

Assumptions: b = 10; 1,000,000 nodes/sec; 100bytes/node

Bi-directional Search

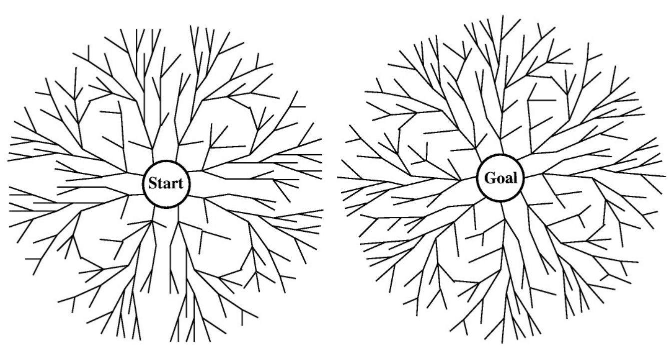


image from cs-alb-pc3.massey.ac.nz/notes/59302/fig03.17.gif

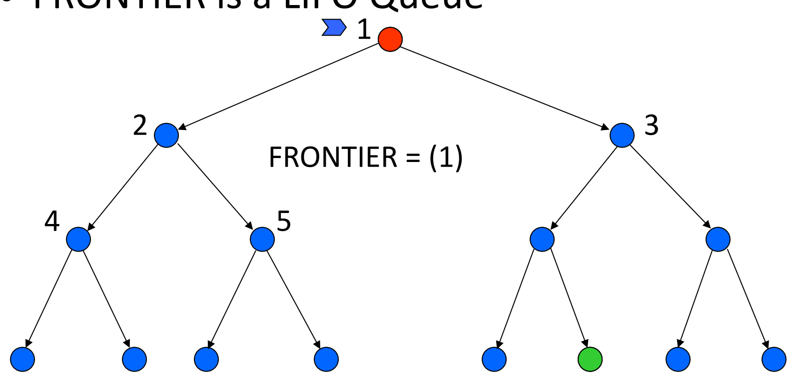
$$b^{d/2} + b^{d/2} << b^d$$

Issues with Bi-directional Search

- Uniqueness of goal
  - Suppose goal is parking your car at airport
  - Huge no. of possible goal states
    - Configurations of other vehicles
    - Which space you use
- Invertability of actions

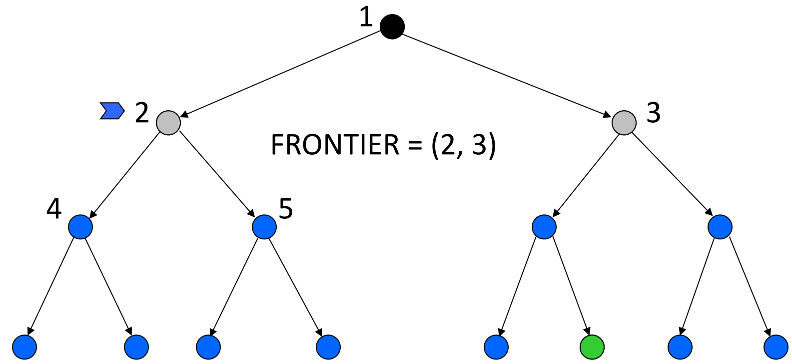
Depth-First Search

- FRONTIER is a LIFO Queue



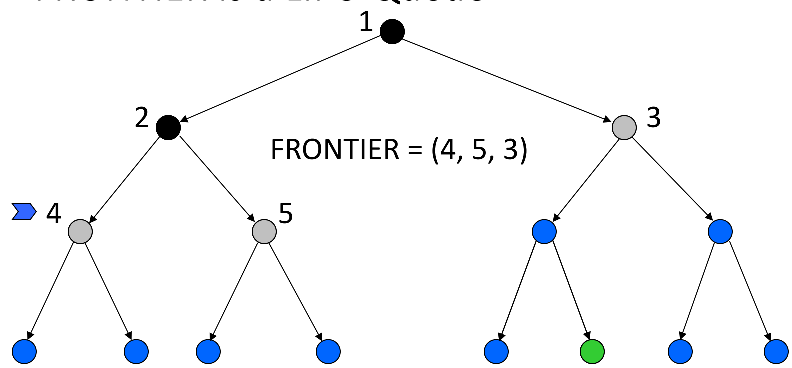
Depth-First Search

- FRONTIER is a LIFO Queue



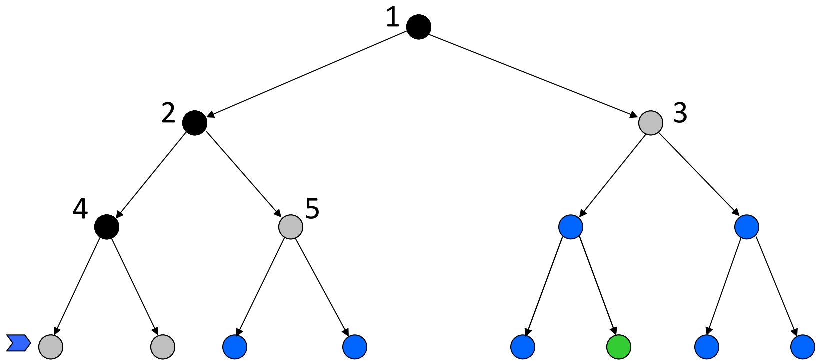
Depth-First Search

- FRONTIER is a LIFO Queue



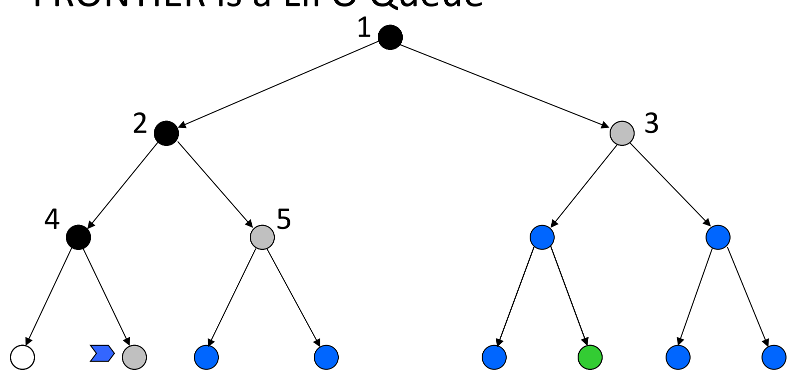
Depth-First Search

- FRONTIER is a LIFO Queue



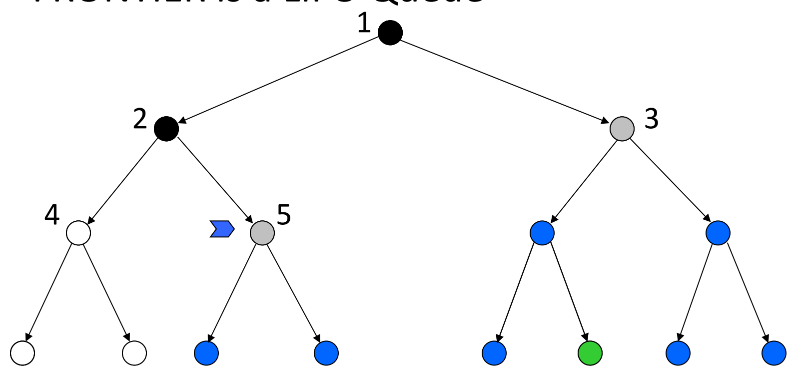
Depth-First Search

- FRONTIER is a LIFO Queue



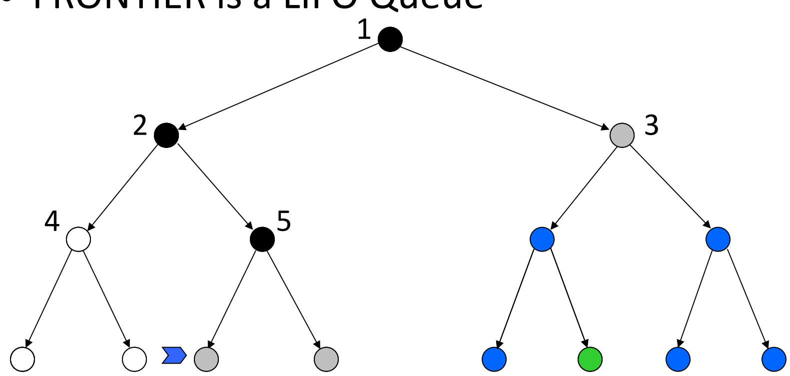
Depth-First Search

- FRONTIER is a LIFO Queue



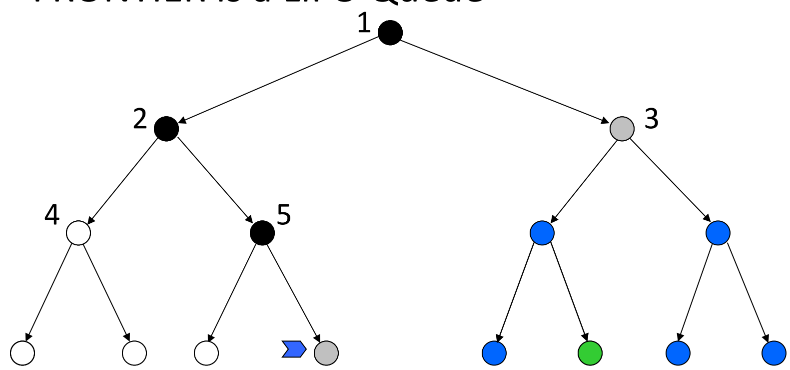
Depth-First Search

- FRONTIER is a LIFO Queue



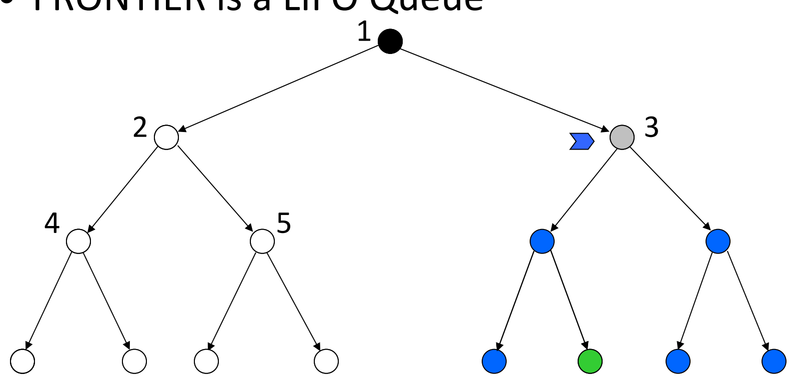
Depth-First Search

- FRONTIER is a LIFO Queue



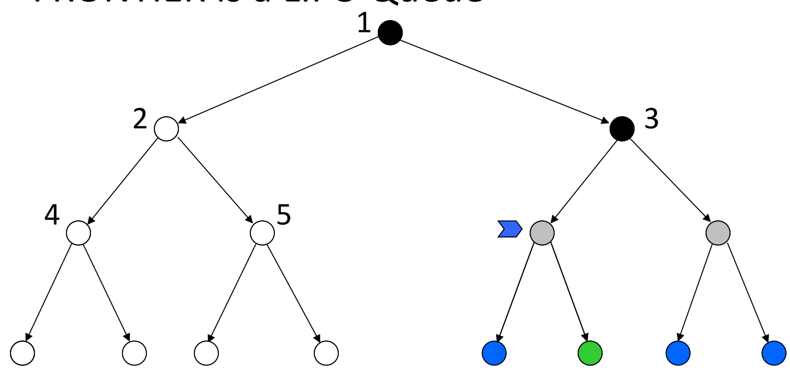
Depth-First Search

- FRONTIER is a LIFO Queue



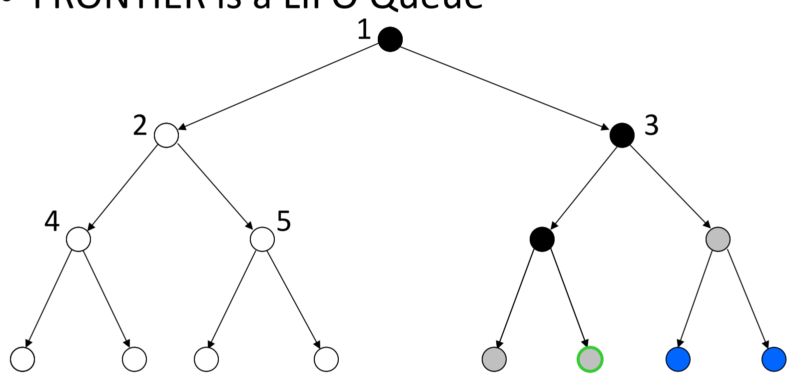
Depth-First Search

- FRONTIER is a LIFO Queue



Depth-First Search

- FRONTIER is a LIFO Queue



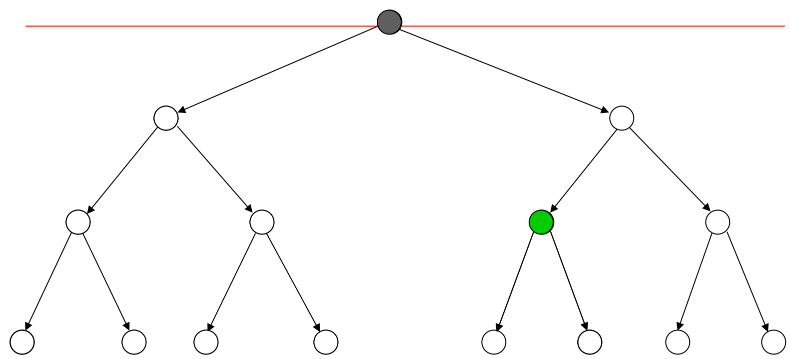
DFS Properties

- Completeness: (Y for finite trees, N for infinite trees)
- Optimality: N
- Time complexity: $O(b^{m+1})$ (m = depth we hit, m>d?)
- Space complexity: O(bm) (bounded for trees)

Iterative Deepening

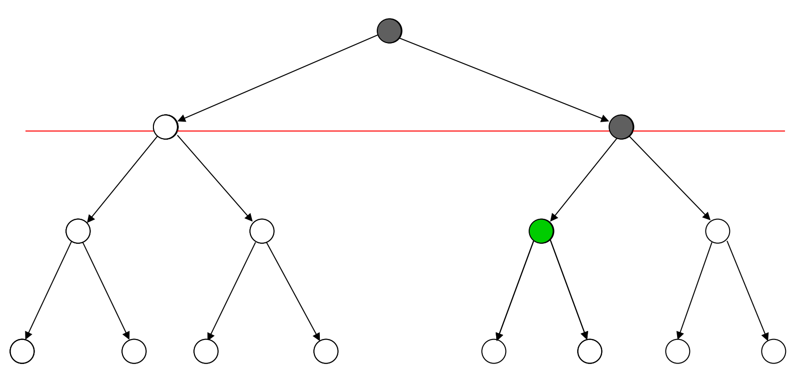
- Want:
  - DFS memory requirements
  - BFS optimality, completeness
- Idea:
  - Do a depth-limited DFS for depth m
  - Iterate over m

Iterative Deepening

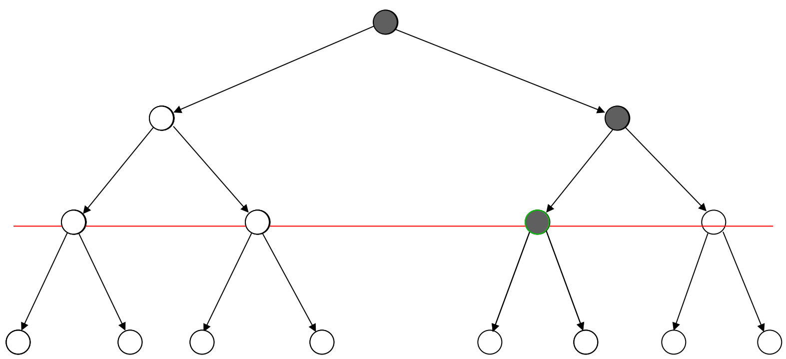


Note: The IDDFS slides are animated, showing DFS running down to the red line on each slide.

Iterative Deepening



Iterative Deepening



IDDFS Properties

- Completeness: Y
- Optimality: (whenever BFS is optimal)
- Time complexity: $O(b^{d+2})$
- Space complexity: $O(bd)$

IDDFS vs. BFS

Theorem: IDDFS generates no more than twice as many nodes for a binary tree as BFS.

Proof: Assume the tree bottoms out at depth d, BFS generates:

$$2^{d+1} - 1$$

In the worst case, IDDFS does no more than:

$$\sum_{i=0}^{d}(2^{i+1}-1) = \sum_{i=0}^{d}2^{i+1} - \sum_{i=0}^{d}1 = 2(2^{d+1}-1)-(d+1) < 2(2^{d+1}-1) = 2 \times BFS(d)$$

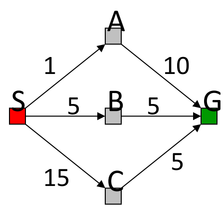
What about b-ary trees? IDDFS relative cost is lower!

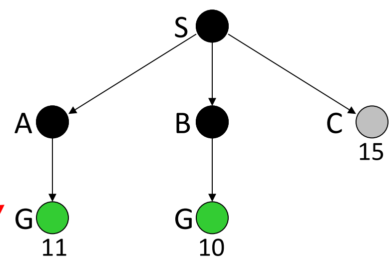
Non-constant Costs

- Arcs between states can have variable costs
- The cost of the path to each node N is g(N) = Σ costs of arcs from root to N
- Breadth-first is no longer optimal with variable arc costs!

Uniform-Cost Search (UCS)

- Expand node in FRONTIER with the cheapest path so far, i.e., frontier is a priority queue prioritized on g(N)





Suboptimal path!

(how to fix this?)

Search Algorithm #2

TREE-SEARCH2(initial-state)

```
If GOAL?(initial-state) then return initial-state
INSERT(initial-node,FRONTIER)
Repeat:
 If empty(FRONTIER) then return failure
    N ← REMOVE(FRONTIER)
    s ← STATE(N)
    If GOAL?(s) then return path or goal state
    For every state s' in SUCCESSORS( )
       Create a new node N' as a child of N
       INSERT(N',FRONTIER)
```

The goal test is applied to a node when this node is **expanded**, not when it is generated.

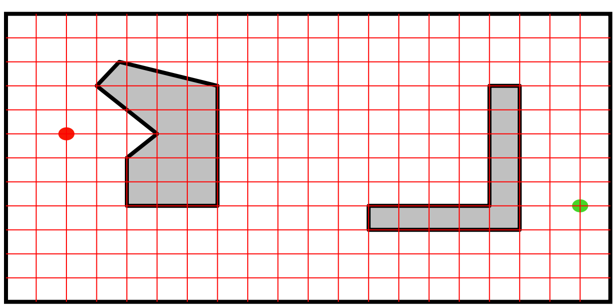
Now, UCS is optimal!

Avoiding Revisited States

- Requires comparing state descriptions
- Breadth-first search:
  - Store all states associated with generated nodes in VISITED
  - If the state of a new node is in VISITED, then discard the node

Implemented as hash-table (e.g. python dictionary) or as explicit data structure with flags

Explicit Data Structures



- Robot navigation
- VISITED: array initialized to 0, matching grid
- When grid position (x,y) is visited, mark corresponding position in VISITED as 1
- **Size of the entire state space!**

Avoiding Revisited States in DFS

- Depth-first search:
  - Solution 1:
    - Store all states in current path in VISITED
    - If the state of a new node is in VISITED, then discard the node
  - Only avoids loops
  - Solution 2:
    - Store all generated states in VISITED
    - If the state of a new node is in VISITED, then discard the node
  - Same space complexity as breadth-first !

Avoiding Revisited States in Uniform-Cost Search

- For any state S, when the first node N such that STATE(N) = S is expanded, the path to N is the best path from the initial state to S
- So:
  - When a node is **expanded**, store its state into VISITED
  - When a new node N is generated:
    - If STATE(N) is in VISITED, discard N
    - If there exists a node N' in the frontier such that STATE(N') = STATE(N), discard the node -- N or N' – w/highest cost

Search Algorithm #3

```
GRAPH-SEARCH(initial-state)
1. If GOAL?(initial-state) then return initial-state
2. INSERT(initial-node,FRONTIER)
3. Repeat:
4.   If empty(FRONTIER) then return failure
5.        N ← REMOVE(FRONTIER)
6.        s ← STATE(N)
7.        Add s to VISITED
7.        If GOAL?(s) then return path or goal state
8.        For every state s' in SUCCESSORS( )
9.        Create a new node N' as a child of N
10.       If s' is in VISITED then discard N'
11.       If there is N'' in FRONTIER with STATE(N')=STATE(N'')
12.          If g(N'') is lower than g(N') then discard N'
13.          Else discard N''
14.       INSERT(N',FRONTIER) (if not discarded above)
```

Uninformed Search Summary

- Many variations on same basic algorithm
- Key differences:
  - How frontier is implemented (FIFO, LIFO, priority queue)
  - When goal test is applied
  - Whether visited list is maintained
- Big impact on:
  - Completeness
  - Optimality
  - Complexity