Uninformed Search

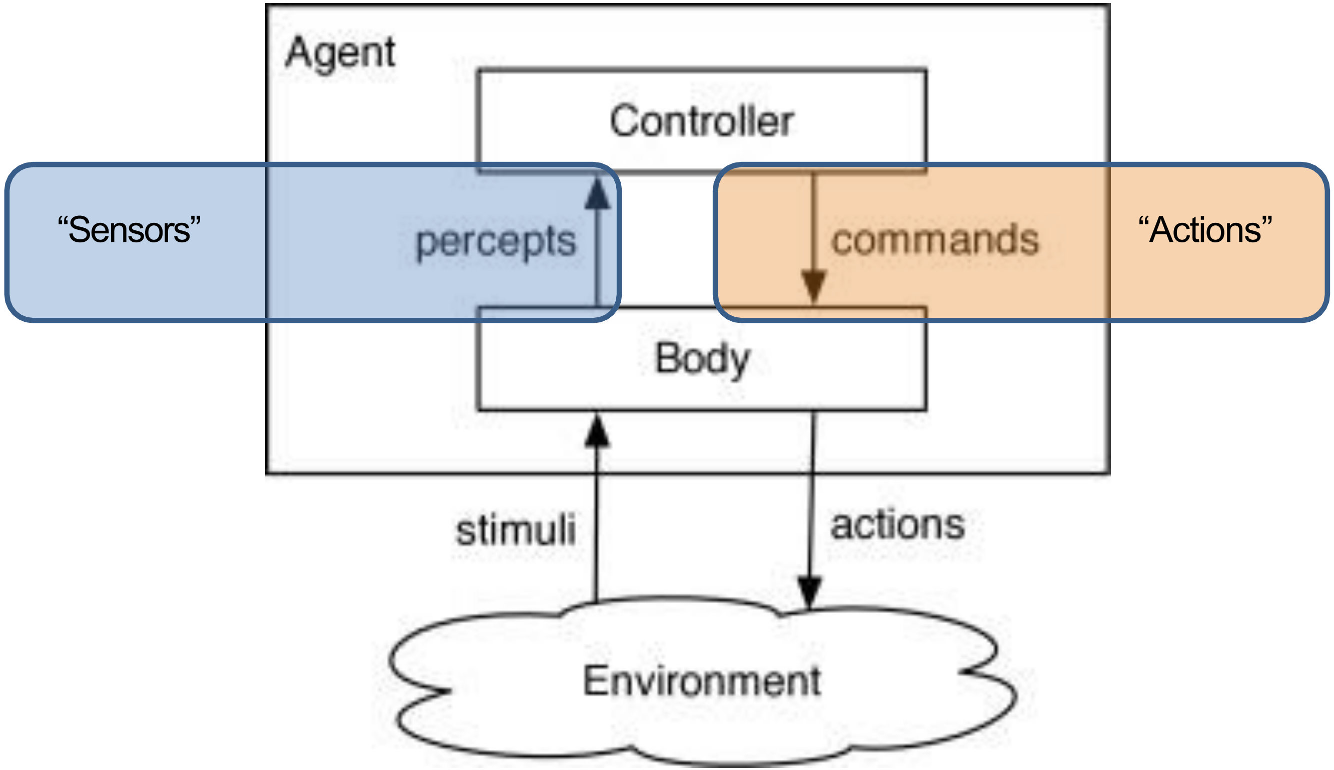
Chapter 3

Some material adopted from notes by Charles R. Dyer, University of Wisconsin-Madison

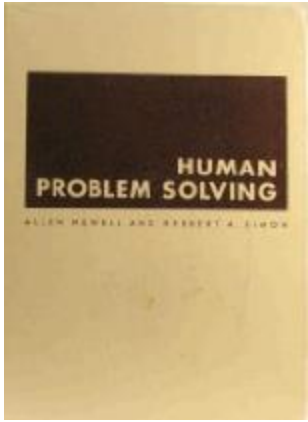
Today's topics

- Goal-based agents
- Representing states and actions
- Example problems
- Generic state-space search algorithm
- Specific algorithms
  - Breadth-first search
  - Depth-first search
  - Uniform cost search
  - Depth-first iterative deepening

Agent Design



AI as a Search Problem



Allen Newell and Herb Simon,
the *problem space principle* in AI:

"*The rational activity in which people engage to solve a problem can be described in terms of :*

*(1) a set of **states** of knowledge,*

***(2) operators** for changing one state into another,*

***(3) constraints** on applying operators, and*

***(4) control** knowledge for deciding which operator to apply next.*"

Newell A & *Simon* H A. *Human problem solving*.
Englewood Cliffs, NJ: Prentice-Hall. 1972.

Graphs

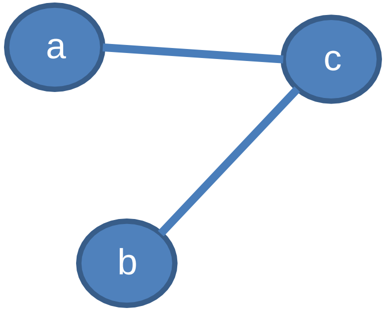
- A graph G = (E, V)
- V = set of vertices (nodes)
- E = set of edges between pairs of nodes, (x, y)

G can be:

- Undirected: order of (x, y) doesn't matter
  - These are symmetric
- Directed: order of (x, y) does matter
- Weighted: cost function $g(x, y)$

Undirected Graph

- A graph G = (E, V)
- V = set of vertices (nodes)
- E = set of edges between pairs of nodes

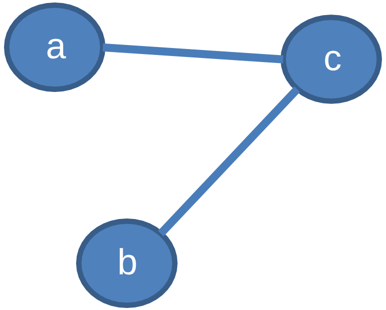


V = { ??? }

E = { ??? }

Undirected Graph

- A graph G = (E, V)
- V = set of vertices (nodes)
- E = set of edges between pairs of nodes

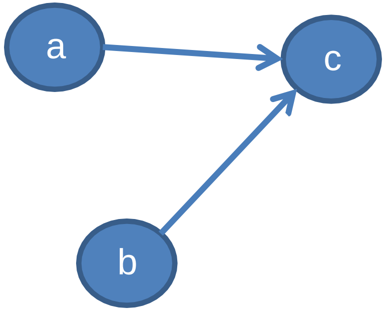


V={ a, b, c }

E={ (a, c), (c, a), (b, c) ,(c,b)}

Directed Graph

- A graph G = (E, V)
- V = set of vertices (nodes)
- E = set of edges between pairs of nodes

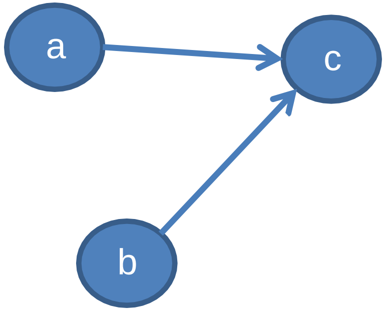


V={ ??? }

E={ ??? }

Directed Graph

- A graph G = (E, V)
- V = set of vertices (nodes)
- E = set of edges between pairs of nodes

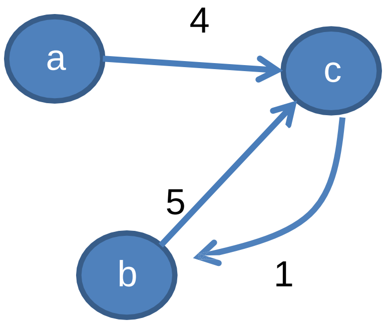


V={ a, b, c }

E={ (a, c), (b, c) }

Weighted Graph

- A graph G = (E, V)
- V = set of vertices (nodes)
- E = set of edges between pairs of nodes



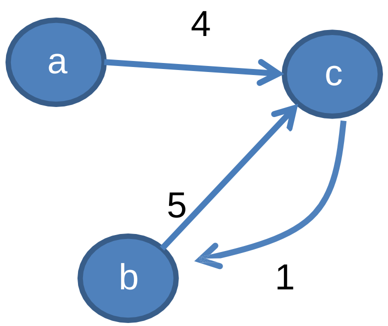
V={ ??? }

E={ ??? }

g = ???

Weighted Graph

- A graph G = (E, V)
- V = set of vertices (nodes)
- E = set of edges between pairs of nodes



V = { a, b, c }

E = { (a,c), (b, c), (c, b) }

g = {(a, c): 4, (b, c): 5, (c, b): 1}

Some Key Terms: States, Goal, and Solution

State: a representation of the current world/environment (as needed for the agent)

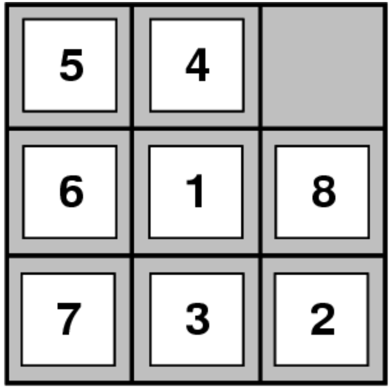
Initial State: The state the agent/problem starts in

Goal State: The desired state

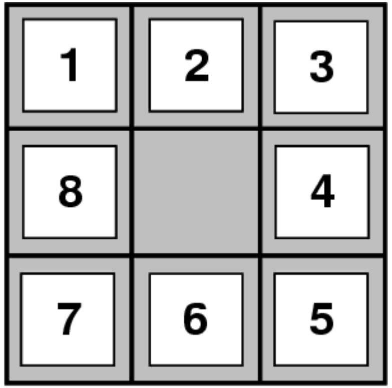
Solution: a sequence of actions that operate sequentially on states and allow the agent to achieve its goal

Example: 8-Puzzle

Given an initial configuration of 8 numbered tiles on a 3x3 board, move the tiles to produce a desired goal configuration



Start State



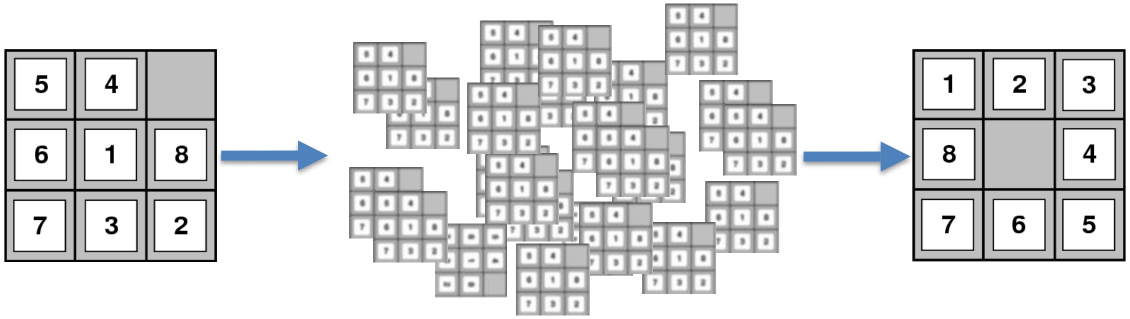
Goal State

Building goal-based agents

–How do we represent the **state** of the “world”?

–What is the **goal** and how can we recognize it?

–What are the possible **actions**?

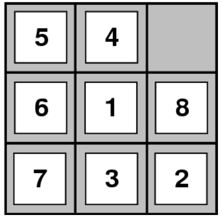


initial state

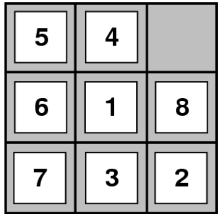
goal state

Representing states

- State of an 8-puzzle?



Representing states



- State of an 8-puzzle?
- A 3x3 array of integer in {0..8}
- No integer appears twice
- 0 represents the empty space

- In Python, we might implement this using a nine-character string: `"540681732"`
- And write functions to map the 2D coordinates to an index

What's the goal to be achieved?

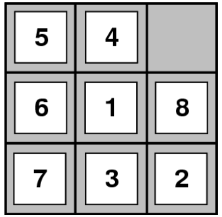
- Describe situation we want to achieve, a set of properties that we want to hold, etc.
- Defining a **goal test** function that when applied to a state returns True or False
- For our problem:

```
def isGoal(state):
    return state == "123405678"
```

What are the actions?

- **Primitive actions** for changing the state
  In a **deterministic** world: no uncertainty in an action's effects (simple model)
- Action should specify:
  - ***can*** action be applied to the current state
  - What state ***results*** after action is performed

Representing actions



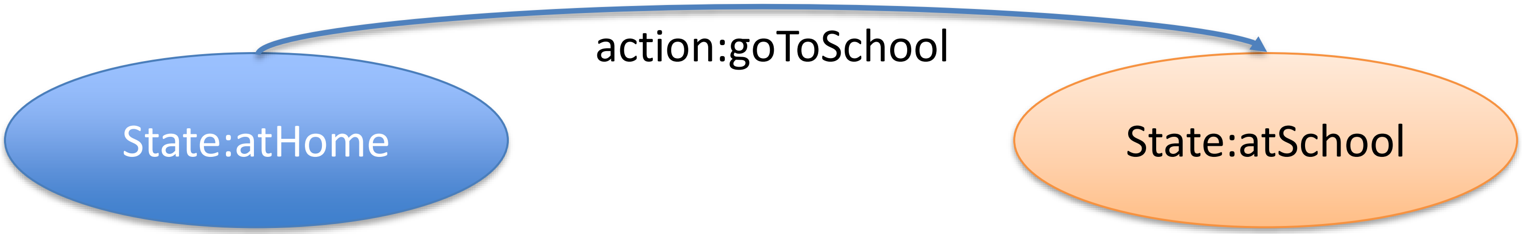
- Actions for 8-puzzle?

- Number of actions/operators depends on the **representation** used in describing a state
  - Specify 4 possible moves for each of the 8 tiles, resulting in a total of **4*8=32 operators**
  - Or: Specify four moves for “blank” square and we only need **4 operators**

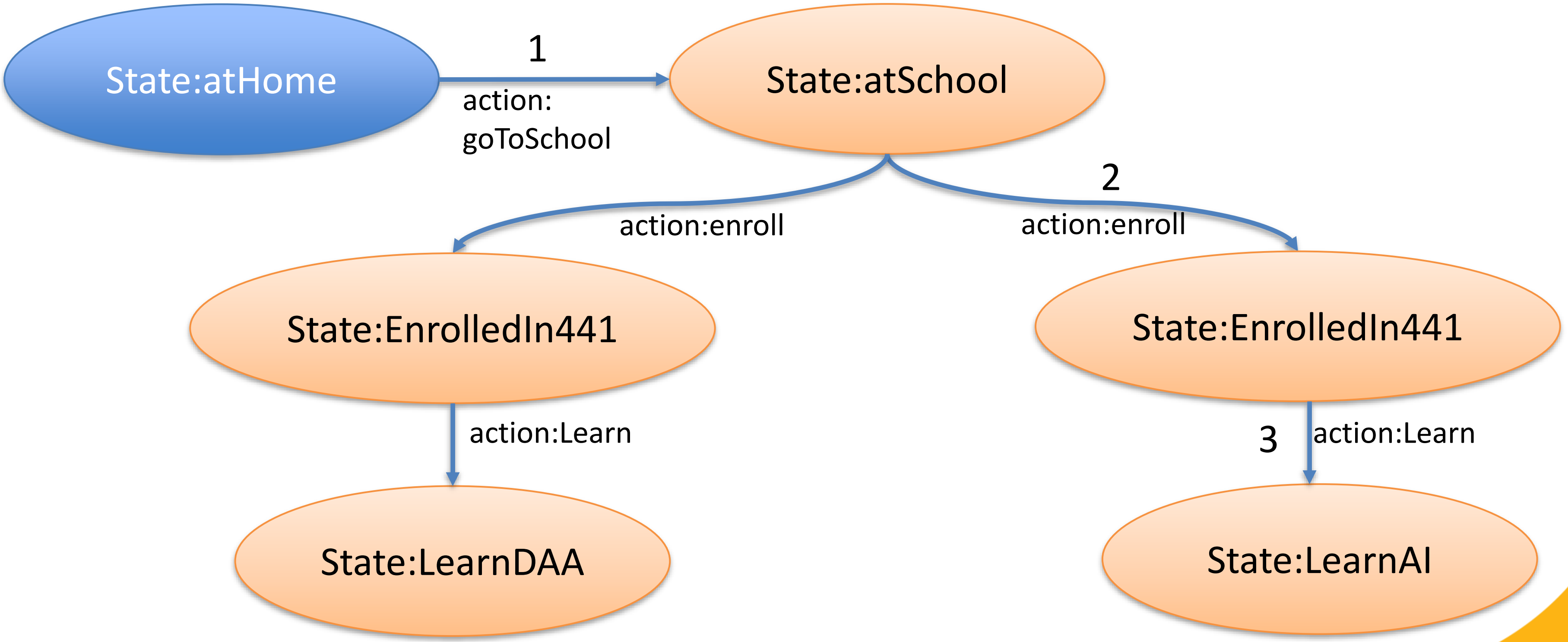
- **Representational shift can simplify a problem!**

Representing actions

- Actions ideally considered as **discrete events** that occur at an **instant of time**



Finding optimal solution = Searching in a Graph



Problem Size

- **Size of a problem** usually described in terms of possible **number of states**
  - Tic-Tac-Toe has about 3^9 states (19,683≈$2*10^4$)
  - Checkers has about 10^{40} states
  - Rubik's Cube has about 10^{19} states
  - Chess has about 10^{120} states in a typical game
  - Go has $2*10^{170}$
  - Theorem provers may deal with an infinite space

- State space size ≈ solution difficulty

Water Jug Problem

- Problem Space:
  - Two jugs J1 & J2
  - Capacity C1 & C2 respectively
- Initial State:
  - J1 has W1 water and J2 has W2 water
- Actions:
  - Pour from jug X to jug Y until X empty or Y full
  - Empty jug X onto the floor
- Goal State:
  - J1 has G1 water and J2 G2

Search in a state space

- Basic idea:
  - Create representation of initial state
  - Try all possible actions & connect states that result
  - Recursively apply process to the new states until we find a solution or dead ends
- We need to keep track of the connections between states and might use a
  - Tree data structure or
  - Graph data structure
- A graph structure is best in general…

Formalizing state space search

- A state space is a **graph** (V, E) where V is a set of **nodes** and E is a set of **arcs**, and each arc is directed from a node to another node
- **Nodes:** data structures with state description and other info, e.g., node's parent, name of action that generated it from parent, etc.
- **Arcs:** instances of actions, head is a state, tail is the state that results from action

Formalizing search in a state space

- Each arc has fixed, positive **cost** associated with it corresponding to the action cost
- Each node has a set of **successor nodes** corresponding to all legal actions that can be applied at node's state
- One or more nodes are marked as **start nodes**
- A **goal test** predicate is applied to a state to determine if its associated node is a goal node

Water Jug Problem

- Problem Space:
  - Two jugs J1 & J2
  - Capacity C1 & C2 respectively
- Initial State:
  - J1 has W1 water and J2 has W2 water
- Actions:
  - Pour from jug X to jug Y until X empty or Y full
  - Empty jug X onto the floor
- Goal State:
  - J1 has G1 water and J2 G2

Example: Water Jug Problem

Given full 5-gal. jug and empty 2-gal. jug, fill 2-gal jug with one gallon

- State = (x,y), where x is water in jug 1; y is water in jug 2
- Initial State = (5,0)
- Goal State = (-1,1), where -1 means any amount

Action table

Name	Cond.	Transition	Effect

Example: Water Jug Problem

Given full 5-gal. jug and empty 2-gal. jug, fill 2-gal jug with one gallon

- State = (x,y), where x is water in jug 1; y is water in jug 2
- Initial State = (5,0)
- Goal State = (-1,1), where -1 means any amount

Action table

Name	Cond.	Transition	Effect
dump1	x>0	(x,y)→(0,y)	Empty Jug 1
dump2	y>0	(x,y)→(x,0)	Empty Jug 2
pour_1_2	x>0 & y<C2	(x,y)→(x-D,y+D) *D = min(x,C2-y)*	Pour from Jug 1 to Jug 2
pour_2_1	y>0 & X<C1	(x,y)→(x+D,y-D) *D = min(y,C1-x)*	Pour from Jug 2 to Jug 1

Formalizing search

- **Solution:** sequence of actions associated with a path from a start node to a goal node
- **Solution cost:** sum of the arc costs on the solution path
  - If all arcs have same (unit) cost, then solution cost is length of solution (number of steps)

State-space search algorithm

;; problem describes the start state, operators, goal test, and operator costs
;; queueing-function is a comparator function that ranks two states
;; general-search returns either a goal node or failure

```
function general-search (problem, QUEUEING-FUNCTION)
  nodes = MAKE-QUEUE(MAKE-NODE(problem.INITIAL-STATE))
  loop
     if EMPTY(nodes) then return "failure"
     node = REMOVE-FRONT(nodes)
     if problem.GOAL-TEST(node.STATE) succeeds
        then return node
     nodes = QUEUEING-FUNCTION(nodes, EXPAND(node,
                 problem.OPERATORS))
 end
```

;; Note: The goal test is NOT done when nodes are generated
;; Note: This algorithm does not detect loops

Key procedures to be defined

- EXPAND
  - Generate a node's successor nodes, adding them to the graph if not already there
- GOAL-TEST
  - Test if state satisfies all goal conditions
- QUEUEING-FUNCTION
  - Maintain ranked list of nodes that are candidates for expansion
  - Changing definition of the QUEUEING-FUNCTION leads to different search strategies

Informed vs. uninformed search

Uninformed search strategies (blind search)

- Use no information about likely *direction* of a goal
- Methods: breadth-first, depth-first, depth-limited, uniform-cost, depth-first iterative deepening, bidirectional

Informed search strategies (heuristic search)

- Use information about domain to (try to) (usually) head in the general direction of goal node(s)
- Methods: hill climbing, best-first, greedy search, beam search, algorithm A, algorithm A*

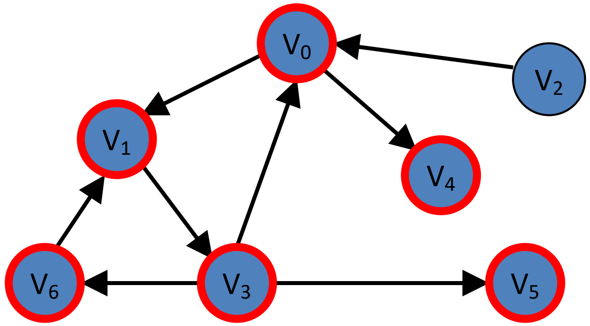
Evaluating search strategies

- **Completeness**
  - Guarantees finding a solution whenever one exists
- **Time complexity** (worst or average case)
  - Usually measured by *number of nodes expanded*
- **Space complexity**
  - Usually measured by maximum size of graph/tree during the search
- **Optimality/Admissibility**
  - If a solution is found, is it **guaranteed** to be an optimal one, i.e., one with minimum cost

Classic uninformed search methods

- The four classic uninformed search methods
  - Breadth first search (BFS)
  - Depth first search (DFS)
  - Uniform cost search *(generalization of BFS)*
  - Iterative deepening *(blend of DFS and BFS)*
- To which we can add another technique
  - Bi-directional search *(hack on BFS)*

Breadth-First Search



Queue Contents:

V0

V1, V4

V4, V3

V3

V6, V5

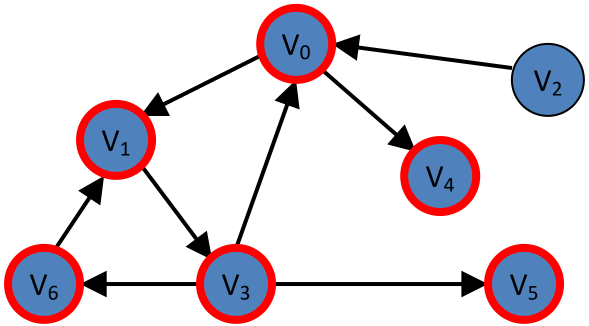
V5

Empty

Breadth-First Search

- Enqueue nodes in **FIFO** (first-in, first-out) order
- **Complete**
- **Optimal** : finds shortest path, which is optimal if all operators have same cost
- **Exponential time and space complexity**, $O(b^d)$, where d is depth of solution; b is branching factor (i.e., # of children)
- Takes a **long time to find solutions** with large number of steps because must explore all shorter length possibilities first

Depth-First Search



Stack Contents:

V0

V4, V1

V4, V3

V4, V6, V5

V4, V6

V4

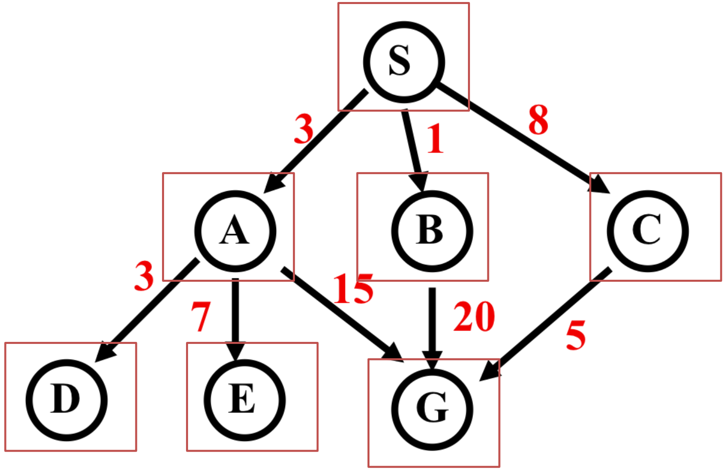
Empty

A traversal processes only those vertices that can be reached from the start vertex.

Depth-First (DFS)

- Enqueue nodes on nodes in **LIFO** (last-in, first-out) order, i.e., use stack data structure to order nodes
- **May not terminate** w/o *depth bound*
- **Not complete** (with or w/o cycle detection, with or w/o a cutoff depth)
- **Exponential time**, $O(b^d)$, but **linear space**, $O(bd)$
- Can find **long solutions quickly** if lucky (and **short solutions slowly** if unlucky!)

Uniform-Cost Search



Expanded node	Nodes list
	{ S^0 }
S^0	{ B^1 A^3 C^8 }
B^1	{ A^3 C^8 G^{21} }
A^3	{ D^6 C^8 E^{10} G^{18} G^{21} }
D^6	{ C^8 E^{10} G^{18} G^{21} }
C^8	{ E^{10} G^{13} G^{18} G^{21} }
E^{10}	{ G^{13} G^{18} G^{21} }
G^{13}	{ G^{18} G^{21} }

Solution path found is S C G, cost 13

Uniform-Cost Search (UCS)

- Enqueue nodes by **path cost**. i.e., let g(n) = cost of path from *start* to current node *n*. Sort nodes by increasing value of g(n).
- Also called *Dijkstra's Algorithm*
- **Complete**
- **Optimal/Admissible**
- **Exponential time and space complexity**, $O(b^d)$

Depth-First Iterative Deepening (DFID)

- Iteratively increase depth cutoff:
  - Do DFS to depth 0, then (if no solution) DFS to depth 1, etc.
- **Complete**
- **Optimal/Admissible** if all operators have unit cost, else finds shortest solution (like BFS)
- Time complexity a bit worse than BFS or DFS

  Nodes near top of search tree generated many times, but since almost all nodes are near tree bottom, worst case time complexity still exponential, $O(b^d)$

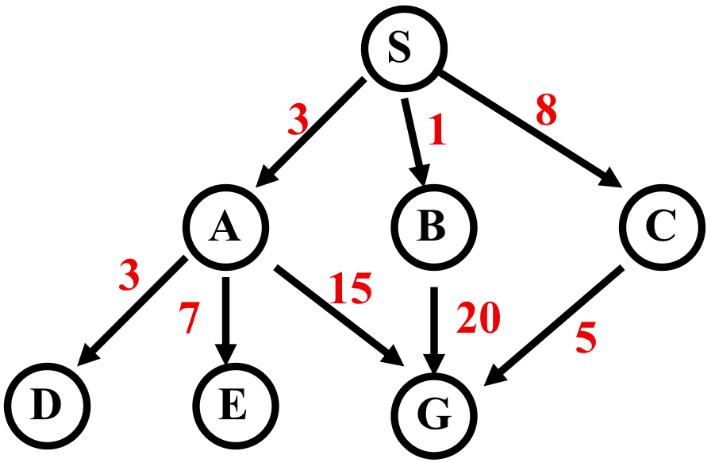
Depth-First Iterative Deepening (DFID)

- **Linear space complexity**, O(bd), like DFS
- Has advantages of BFS (completeness) and DFS (i.e., limited space, finds longer paths quickly)
- Preferred for **large state spaces** where **solution depth is unknown**

How they perform

- **Depth-First Search:**
  - 4 Expanded nodes: S A D E G
  - Solution found: S A G (cost 18)
- **Breadth-First Search**:
  - 7 Expanded nodes: S A B C D E G
  - Solution found: S A G (cost 18)
- **Uniform-Cost Search**:
  - 7 Expanded nodes: S A D B C E G
  - Solution found: S C G (cost 13)

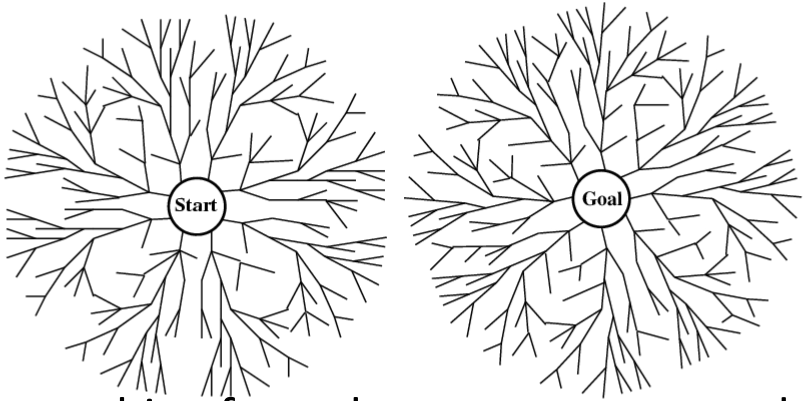
  Only uninformed search that worries about costs
- **Iterative-Deepening Search**:
  - 10 nodes expanded: S S A B C S A D E G
  - Solution found: S A G (cost 18)



Searching Backward from Goal

- Usually a successor function is reversible
  - i.e., can generate a node’s predecessors in graph
- If we know a single goal (rather than a goal’s properties), we could search backward to the initial state
- It might be more efficient
  - Depends on whether the graph fans in or out

Bi-directional search



- Alternate searching from the start state toward the goal and from the goal state toward the start
- Stop when the frontiers intersect
- Works well only when there are unique start & goal states
- Requires ability to generate "predecessor" states
- Can (sometimes) lead to finding a solution more quickly