CSC361
Lecture

SOLVING PROBLEMS BY SEARCHING

Outline

- Problem-Solving Agents
  - Search problems and solutions
  - Formulating problems
- Example Problems
  - Standardized problems
  - Real-world problems
- Search Algorithms
  - Best-first search
  - Search data structures
  - Redundant paths
  - Measuring problem-solving performance
- Uninformed Search Strategies
  - Breadth-first search
  - Dijkstra's algorithm or uniform-cost search
  - Depth-first search and the problem of memory

Problem-Solving Agents

- Problem-solving agent: An agent that
  - plans ahead: to consider a sequence of actions that form a path to a goal state
  - when correct action to take is not immediately obvious
  - computational process it undertakes is called search
- For sake of simplicity, environment is:
  - episodic, single agent, fully observable, deterministic, static, discrete, and known
- Two types of algorithms:
  - informed algorithms: in which agent can estimate how far it is from the goal
  - uninformed algorithms: where no such estimate is available.

Problem-Solving Agents

- Agent is currently in city A, trying to reach destination Z
  - Observes that there are three roads leading out of A: one toward B, one to C, and one to D
  - None of these are the goal
  - Unless agent is familiar w/ geography of its world, it will not know which road to follow
  - If no additional info. (environment is unknown) agent can do no better than random actions
- Assume agents always have access to info. about world (map)
  → follow four-phase problem-solving process:
  - **Goal formulation:** agent adopts goal of reaching Z.
    - Goals organize behavior by limiting objectives → actions to be considered
  - **Problem formulation:** agent devises a description of states and actions necessary to reach goal
    - abstract model of relevant part of world
    - one good model: consider actions of traveling from one city to an adjacent city → current city
  - **Search:** before taking any action in real world, agent simulates sequences of actions in its model
    - searching until it finds solution: sequence of actions that reaches goal.
    - might have to simulate multiple sequences that do not reach goal, or find that no solution is possible
  - **Execution:** agent can now execute actions in solution, one at a time

Problem-Solving Agents

- In fully observable/deterministic/known environment: solution is fixed actions sequence
  - If model is correct
    → once agent has found a solution, it can ignore its percepts while executing actions
    - Open-loop: because solution is guaranteed to lead to goal
  - If there is chance model is incorrect, or environment is nondeterministic
    → agent would be safer using:
    - Closed-loop approach that monitors the percepts.
- In partially observable/non-deterministic environments:
  - solution would be branching strategy that recommends different future actions depending on percepts
  - Ex. agent might plan to drive from A to B but might need contingency plan in case:
    - it arrives in Z by accident
    - or finds a sign saying “Road Closed”

Problem-Solving Agents
Search Problems and Solutions

- Search Problem:
  - set of possible **states** of environment: **state space**, can be represented as graph vertices
  - **initial state** that agent starts in
  - set of **goal states**
    - can be empty, multiple, property that applies to many/infinite states
      - Ex. in vacuum-cleaner world, goal might be to have no dirt in any location
    - specify IS-GOAL method for problem
  - **actions** available to agent, can be the graph directed edges
    - Given state s, ACTIONS(s) returns a finite set of actions that can be executed in s.
    - We say that each of these actions is **applicable** in s
    - **path**: sequence of actions
    - **solution**: path from initial state to a goal state.
  - **transition model**: describes what each action does
    - RESULT(s, a) returns state that results from doing action a in state s
  - **action cost function**: ACTION-COST(s, a, s0) in programming or c(s, a, s_0) in math
    - gives numeric cost of applying action a in state s to reach state s_o
    - should use cost function that reflects its own performance measure
      - Ex. in route-finding agents, cost of action might be distance in miles, or time it takes to complete action.
    - **optimal solution** has lowest path cost among all solutions

Problem-Solving Agents
Search problems and solutions

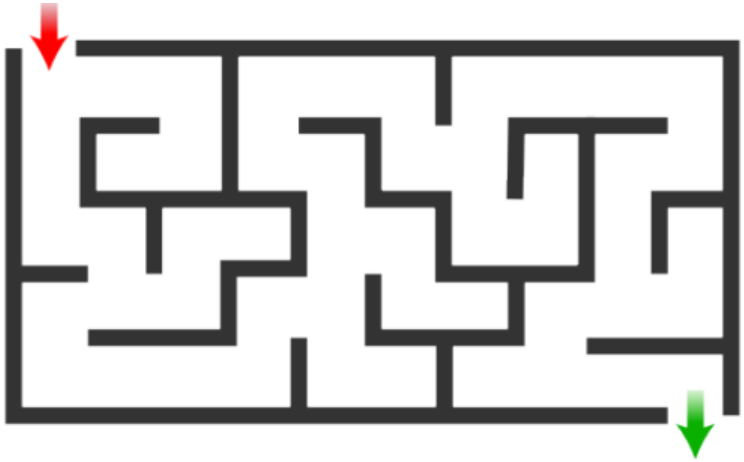
- Notes:
  - Real world is absurdly complex → state space must be abstracted for problem solving
    - (Abstract) state = set of real states
    - (Abstract) action = complex combination of real actions
    - (Abstract) solution = set of real paths that are solutions in the real world
    - Each abstract action should be “easier" than the original problem!

Example Problems

- Standardized problem:
  - intended to illustrate/exercise various problem-solving methods
  - It can be given a concise, exact description
    → suitable as benchmark for Real-world problem researchers to compare performance of algorithms.
- Real-world problem:
  - one whose solutions people actually use,
  - whose formulation is idiosyncratic, not standardized
    because, for example, each robot has different sensors that produce different data.

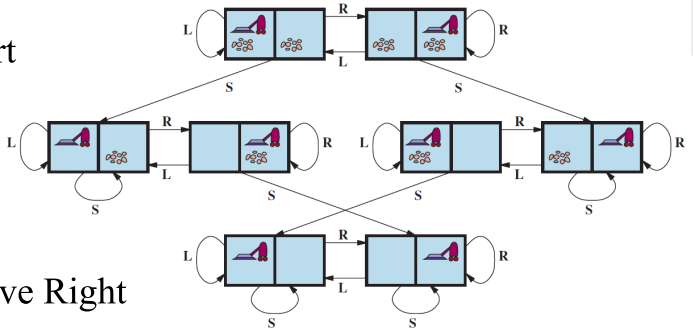
Example Problems
Standardized problems

- **Grid world problem**
  - 2D/3D rectangular array of square cells in which agents can move from cell to cell
  - agent can move to any obstacle-free adjacent cell horizontally or vertically, in some problems diagonally.
  - cells can contain:
    - objects, which agent can pick up/push, or otherwise act upon
    - wall or other impassible obstacle in a cell prevents an agent from moving into that cell.



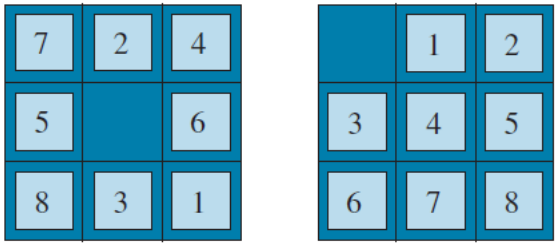
Example Problems
Standardized problems: Grid world problem

- **Vacuum world as a grid world problem**:
  - **States**: which objects are in which cells: objects are agent and dirt
    - simple two-cell version, agent can be in either
    - each cell can either contain dirt or not
    - → there are 2×2×2=8
    - In general, a vacuum environment with n cells has $n.2^n$ states.
  - **Initial state**: Any state can be designated as the initial state.
  - **Actions**: In two-cell world we defined: Suck, move Left, and move Right
    - In 2D multi-cell world we could add Upward and Downward
    - or switch to egocentric relative actions: Forward, Backward, TurnRight, and TurnLeft
  - **Transition model**:
    - Suck removes any dirt from agent's cell: ([X,Y], Suck) → [X, CLEAN]
    - Forward moves agent ahead 1 cell in direction it is facing, unless it hits a wall, in which case action has no effect
    - Backward moves agent in opposite direction
    - TurnRight and TurnLeft change direction it is facing by 90
  - **Goal states**: states in which every cell is clean.
  - **Action cost**: Each action costs 1.



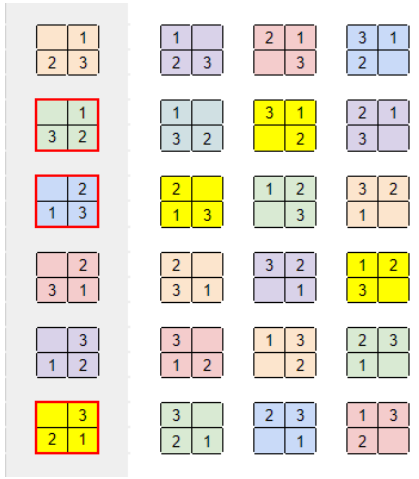
Example Problems
Standardized problems: Grid world problem

- **Sliding-tile puzzle as a grid world problem**:



Start State Goal State

- **States**: state description specifies location of each tile
- **Initial state**: Any state can be designated as initial state
  - parity property partitions state space
    → any given goal can be reached from exactly half of possible initial states
- **Actions**: tile slides (physical world), simpler to move blank Left,Right,Up,Down
  - If blank is at an edge or corner then not all actions will be applicable.
- **Transition model**: Maps state/action to resulting state
- **Goal state**: although any state could be goal, typically state w/ numbers in order
- **Action cost**: Each action costs 1.



- Configurations w/ same color obtained by rotation
- States w/ red border are unsolvable

Example Problems
Standardized problems

❑ Knuth conjecture problem

- Starting with number 4, sequence of square root/floor/factorial operations can reach any positive integer
  - Ex. we can reach 5 from 4 as follows:
- state space is infinite: for any integer >2 factorial operator will always yield a larger integer
- problem is interesting because it explores very large numbers
  - shortest path to 5 goes through (4!)! = 620,448,401,733,239,439,360,000

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}} \right\rfloor = 5$$

- **States**: Positive real numbers.
- **Initial state**: 4.
- **Actions**: Apply square root, floor, or factorial operation (factorial for integers only).
- **Transition model**: As given by mathematical definitions of operations.
- **Goal state**: desired positive integer.
- **Action cost**: Each action costs 1.

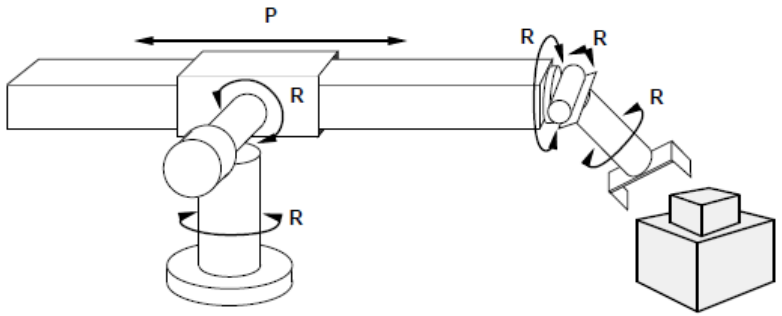
Example Problems
Real-world problems

- **Route-finding problem**: Ex. airline travel problems
  - **States**: Each state includes a location , current time.
    - cost of action (flight segment) depends on: previous segments, their fare bases, domestic/international
    - → state must record extra information about these “historical” aspects.
  - **Initial state**: user’s home airport.
  - **Actions**: Take any flight from current location, in any seat class, leaving after current time, leaving enough time for within-airport transfer if needed.
  - **Transition model**: state resulting from taking flight will have flight’s destination as new location and flight’s arrival time as new time.
  - **Goal state**: destination city. Sometimes goal can be more complex, such as “arrive at destination on a nonstop flight.”
  - **Action cost**: combination of monetary cost, waiting time, flight time, customs/immigration procedures, seat quality, time of day, type of airplane, frequent-flyer reward points, and so on.

Example Problems
Real-world problems

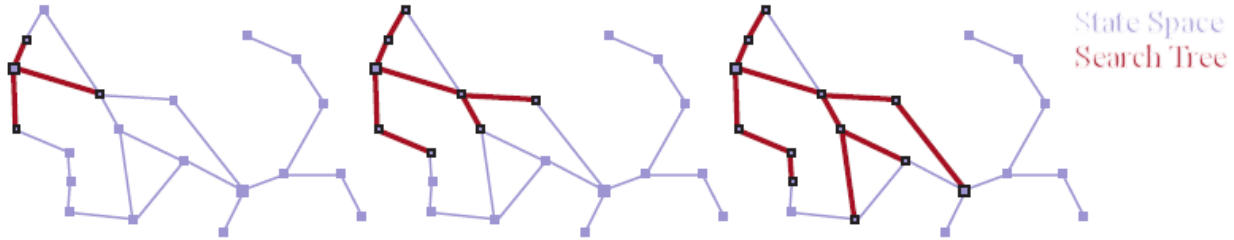
- **Robot navigation problem**: generalization of route-finding problem
  - roam around making its own paths vs following distinct paths
    - circular robot moving on flat surface → space is essentially 2D
    - robot has controlled arms/legs
    - search space becomes many-dimensional—one dimension for each joint angle
  - Advanced techniques are required just to make essentially continuous search space finite
  - complexity + errors in sensor readings/motor controls, partial observability, other agents alter environment

  - **States**: real-valued coordinates of robot joint angles , parts of the object to be assembled
  - **Actions**: continuous motions of robot joints.
  - **Goal state**: complete assembly with no robot included!
  - **Action cost**: time to execute



Search Algorithms

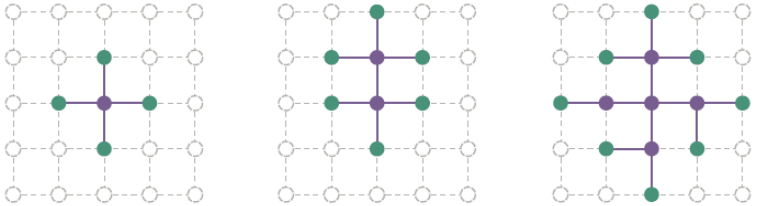
- Search algorithm takes search problem as input , returns solution / failure indication
- Algorithms that superimpose search tree over state space graph
  - forming various paths from initial state, trying to find path that reaches a goal state
    - each node in search tree corresponds to state in state space
    - edges in search tree correspond to actions
    - root of tree corresponds to initial state of problem
- Distinction between state space and search tree
  - state space describes set of states in world, and actions that allow transitions from state to another
  - search tree describes paths between these states, reaching towards goal
  - search tree may have multiple paths to (and thus multiple nodes for) any given state, but each node in tree has a unique path back to root



Search Algorithms

- Starting from root, expand node, by considering available ACTIONS for that state
  - use RESULT function to see where those actions lead to, generate child/successor node for each state
  - choose which of child nodes to consider next
  - search: following up one option now and putting others aside for later
- Reached states: states that have had nodes generated for them
- Frontier: children of child node that have not been expanded yet
  - frontier separates two regions of state-space graph:
    - interior region where every state has been expanded
    - exterior region of states that have not yet been reached

- Ex: exterior/unreached frontier/fringe interior/expanded /reached
  1. just root has been expanded
  2. top frontier node is expanded
  3. remaining successors of root are expanded in clockwise order



Search Algorithms
Best-first search

- Choose node n w/ min. value of some evaluation function $f(n)$
  - child node is added to frontier if:
    - it has not been reached before
    - re-added if it is now being reached w/ path that has lower path cost than any previous path
  - returns either failure indication or node that represents path to goal

```
function BEST-FIRST-SEARCH(problem, f) returns solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← priority queue ordered by f , with node as an element
  reached ← lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] <- child
        add child to frontier
return failure
```

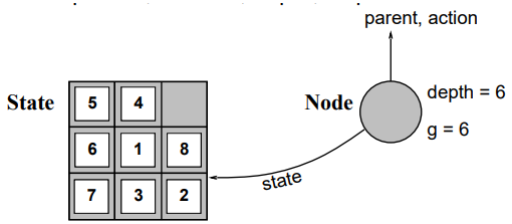
Search Algorithms
Search data structures

- data structure to store search tree → node represented by data structure with:
  - node.STATE: state to which node corresponds
  - node.PARENT: node in tree that generated this node
  - node.ACTION: action that was applied to parent's state to generate this node
  - node.PATH-COST: total cost of path from initial state to this node, g(node)
- following PARENT pointers back from node → recover states/actions along path
  - Doing this from goal node gives us solution
- data structure to store frontier → queue of some kind, operations are:
  - IS-EMPTY(frontier) returns true only if there are no nodes in frontier.
  - POP(frontier) removes top node from frontier and returns it.
  - TOP(frontier) returns (but does not remove) top node of frontier.
  - ADD(node, frontier) inserts node into its proper place in queue.
- candidates: priority queue (best-first), FIFO queue (breadth-first), LIFO queue (depth-first)
- reached states can be stored as lookup/hash table: key=state , value=node

Search Algorithms
Search data structures

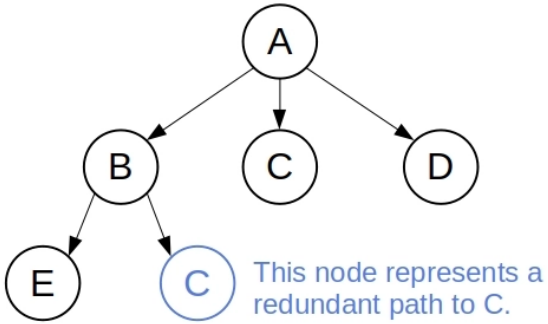
- Implementation note:
  - states vs. nodes:
    - **node:** is data structure constituting part of search tree: includes parent, children, depth, path cost *g(x)*
    - **state:** is a (representation of) a physical configuration: do not have parents, children, depth, or path cost!
  - `Expand` function:
    - creates new nodes
    - filling in various fields
    - use `SuccessorFn` of problem to create corresponding states

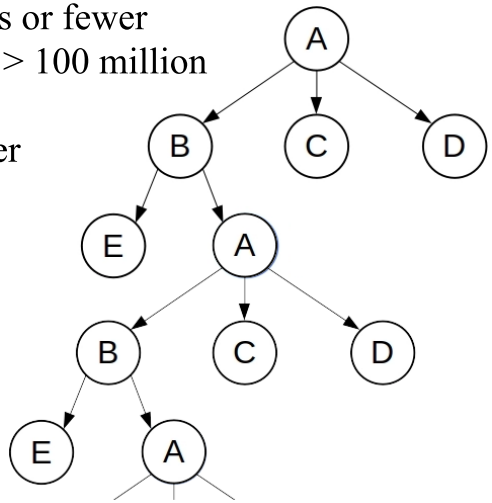
```
function Tree-Search(problem,strategy) returns a solution, or failure
  initialize search tree using initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if node contains a goal state then return corresponding solution
    else expand node and add resulting nodes to search tree
end
```



Search Algorithms
Redundant paths

- Repeated state in search tree can be generated by cycles (loopy path)
  - even though state space has finite states, loopy path complete search tree is infinite
  - no limit to how often one can traverse a loop
- cycle is special case of redundant path
  - Consider agent in 10x10 grid world, with ability to move to any of 8 adjacent squares
  - If there are no obstacles, agent can reach any of the 100 squares in 9 moves or fewer
  - But number of paths of length 9 is almost 8^9 (because of edges of grid), or > 100 million
  - average cell can be reached by over million redundant paths of length 9
  - eliminate redundant paths → complete search roughly a million times faster





Search Algorithms
Redundant paths

- First, remember all previously reached states → detect all redundant paths
  - **Graph search**: keep only best path to each state, appropriate where many redundant paths
  - preferred choice when table of reached states will fit in memory

```
def depth_first_search(self, value):
  if self.value == value:
    return True

  for child in self.children:
    if child.depth_first_search(value):
      return True
  return False
```

```
def depth_first_search(node, value, visited):
  if node.value == value:
    return True
  visited.add(node)
  for neighbor in node.neighbors:
    if neighbor not in visited:
      if depth_first_search(neighbor, value, visited):
        return True
  return False
```

- Second, not worry about repeating past: rare/impossible for two paths to reach same state
  - Ex. assembly problem where each action adds part to an evolving assemblage, in some order
  - save memory space if we don't track reached states → don't check for redundant paths
- Third, compromise and check for cycles, but not for redundant paths in general

Search Algorithms
Measuring problem-solving performance

- Search strategy: picking order of node expansion
  - evaluated along following dimensions:
    - Completeness: does it always find a solution if one exists?
    - Cost optimality: does it always find a least-cost solution?
    - Time complexity: number of nodes generated/expanded
    - Space complexity: maximum number of nodes in memory

  - Time and space complexity are measured in terms of
    - b: maximum branching factor of the search tree
      - Eight puzzle has effective branching factor of 2.13
      - Rubik's cube has effective branching factor of 13.34
      - Chess has a branching factor of about 35
    - d: depth of the least-cost solution
    - m: maximum depth of the state space (may be ∞)

Search Algorithms
Uninformed search strategies

- Uninformed strategies use only information available in problem definition
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search

Search Algorithms
Uninformed search: Breadth-first search

Search Algorithms
Uninformed search: Breadth-first search

Search Algorithms
Uninformed search: Breadth-first search

- systematic search strategy → complete even on infinite state spaces

Search Algorithms
Uninformed search: Breadth-first search

- systematic search strategy → complete even on infinite state spaces

  1-root is expanded 2-all root successors are expanded next 3-then their successors and so on

Search Algorithms
Uninformed search: Breadth-first search

- systematic search strategy → complete even on infinite state spaces

  1-root is expanded 2-all root successors are expanded next 3-then their successors and so on

- BEST-FIRST-SEARCH where evaluation function $f(n)$ is node depth

  e.g., # of actions it takes to reach node

Search Algorithms
Uninformed search: Breadth-first search

- systematic search strategy → complete even on infinite state spaces
  1-root is expanded 2-all root successors are expanded next 3-then their successors and so on
- BEST-FIRST-SEARCH where evaluation function $f(n)$ is node depth
  - # of actions it takes to reach node

Search Algorithms
Uninformed search: Breadth-first search

- systematic search strategy → complete even on infinite state spaces
  1-root is expanded 2-all root successors are expanded next 3-then their successors and so on
- BEST-FIRST-SEARCH where evaluation function $f(n)$ is node depth
  - # of actions it takes to reach node
- can get additional efficiency with couple of tricks

Search Algorithms
Uninformed search: Breadth-first search

- systematic search strategy → complete even on infinite state spaces
  1-root is expanded 2-all root successors are expanded next 3-then their successors and so on
- BEST-FIRST-SEARCH where evaluation function $f(n)$ is node depth
  - # of actions it takes to reach node
- can get additional efficiency with couple of tricks
  - FIFO queue faster than priority queue
    → give correct order of nodes

Search Algorithms
Uninformed search: Breadth-first search

- systematic search strategy → complete even on infinite state spaces
  1-root is expanded 2-all root successors are expanded next 3-then their successors and so on
- BEST-FIRST-SEARCH where evaluation function $f(n)$ is node depth
  - # of actions it takes to reach node
- can get additional efficiency with couple of tricks
  - FIFO queue faster than priority queue
    → give correct order of nodes
    - new nodes are always deeper than their parents
      → go to back of queue

Search Algorithms
Uninformed search: Breadth-first search

- systematic search strategy → complete even on infinite state spaces
  1-root is expanded 2-all root successors are expanded next 3-then their successors and so on
- BEST-FIRST-SEARCH where evaluation function $f(n)$ is node depth
  - # of actions it takes to reach node
- can get additional efficiency with couple of tricks
  - FIFO queue faster than priority queue
    → give correct order of nodes
    - new nodes are always deeper than their parents
      → go to back of queue
    - old nodes which are shallower than new nodes
      → get expanded first
  - Reached can be states set, rather than state-node map

Search Algorithms
Uninformed search: Breadth-first search

- systematic search strategy → complete even on infinite state spaces
  1-root is expanded 2-all root successors are expanded next 3-then their successors and so on
- BEST-FIRST-SEARCH where evaluation function $f(n)$ is node depth
  - # of actions it takes to reach node
- can get additional efficiency with couple of tricks
  - FIFO queue faster than priority queue
    → give correct order of nodes
    - new nodes are always deeper than their parents
      → go to back of queue
    - old nodes which are shallower than new nodes
      → get expanded first
  - Reached can be states set, rather than state-node map

Search Algorithms
Uninformed search: Breadth-first search

- systematic search strategy → complete even on infinite state spaces
  1-root is expanded 2-all root successors are expanded next 3-then their successors and so on
- BEST-FIRST-SEARCH where evaluation function $f(n)$ is node depth
  - # of actions it takes to reach node
- can get additional efficiency with couple of tricks
  - FIFO queue faster than priority queue
    → give correct order of nodes
    - new nodes are always deeper than their parents
      → go to back of queue
    - old nodes which are shallower than new nodes
      → get expanded first
  - Reached can be states set, rather than state-node map
    - once a state is reached → never find better path to it
      → do early goal test

Search Algorithms
Uninformed search: Breadth-first search

- systematic search strategy → complete even on infinite state spaces
  1-root is expanded 2-all root successors are expanded next 3-then their successors and so on
- BEST-FIRST-SEARCH where evaluation function $f(n)$ is node depth
  - # of actions it takes to reach node
- can get additional efficiency with couple of tricks
  - FIFO queue faster than priority queue
    → give correct order of nodes
    - new nodes are always deeper than their parents
      → go to back of queue
    - old nodes which are shallower than new nodes
      → get expanded first
  - Reached can be states set, rather than state-node map
    - once a state is reached → never find better path to it
      → do early goal test

```
function BREADTH-FIRST-SEARCH(problem) returns solution or
failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← FIFO queue, with node as element
  reached ← problem.INITIALg
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

Search Algorithms
Uninformed search: Breadth-first search

- Properties
  - Complete? Yes
  - Time? $1+b+ b^2+b^3+. . .+b^d = O(b^d)$: exp. in d
  - Space? $O(b^d)$ (keeps every node in memory)
  - Optimal? Yes (if cost = 1/step), not in general
- Space is big problem
  - typical real-world problem: b = 10
    processing speed 10^6 nodes/sec
    memory requirements of 1 Kbyte/node
  - depth d = 10 → take less than 3 hours
    10 terabytes
  - depth d = 14, even with infinite memory
    takes 3.5 years

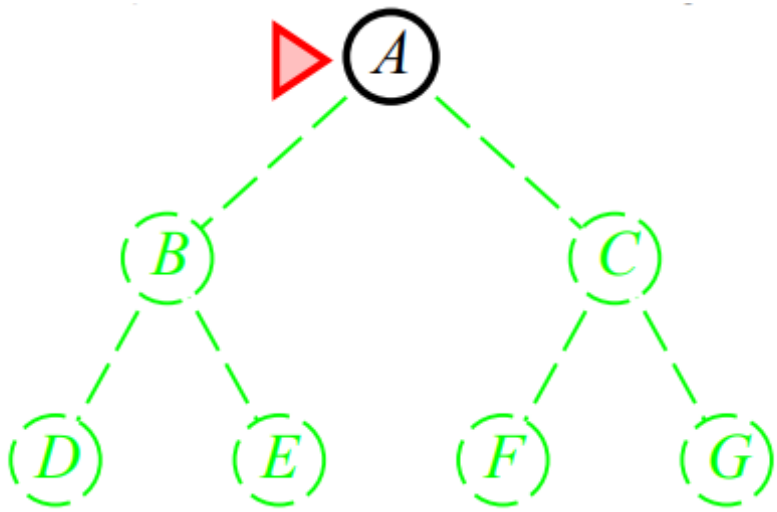
Search Algorithms
Uninformed search: Breadth-first search

- Properties
  - Complete? Yes
  - Time? $1+b+ b^2+b^3+. . .+b^d = O(b^d)$: exp. in d
  - Space? $O(b^d)$ (keeps every node in memory)
  - Optimal? Yes (if cost = 1/step), not in general
- Space is big problem
  - typical real-world problem: b = 10
    processing speed 10^6 nodes/sec
    memory requirements of 1 Kbyte/node
  - depth d = 10 → take less than 3 hours
    10 terabytes
  - depth d = 14, even with infinite memory
    takes 3.5 years

Search Algorithms
Uninformed search: Breadth-first search

- Properties
  - Complete? Yes
  - Time? $1+b+b^2+b^3+\ldots+b^d = O(b^d)$: exp. in d
  - Space? $O(b^d)$ (keeps every node in memory)
  - Optimal? Yes (if cost = 1/step), not in general

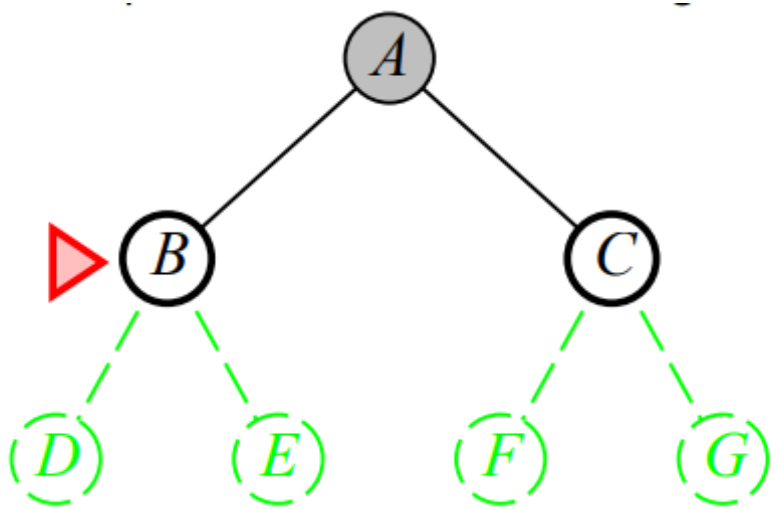
- Space is big problem
  - typical real-world problem: b = 10
    processing speed 10^6 nodes/sec
    memory requirements of 1 Kbyte/node
  - depth d = 10 → take less than 3 hours
    10 terabytes
  - depth d = 14, even with infinite memory
    takes 3.5 years



Search Algorithms
Uninformed search: Breadth-first search

- Properties
  - Complete? Yes
  - Time? $1+b+b^2+b^3+\ldots+b^d = O(b^d)$: exp. in d
  - Space? $O(b^d)$ (keeps every node in memory)
  - Optimal? Yes (if cost = 1/step), not in general

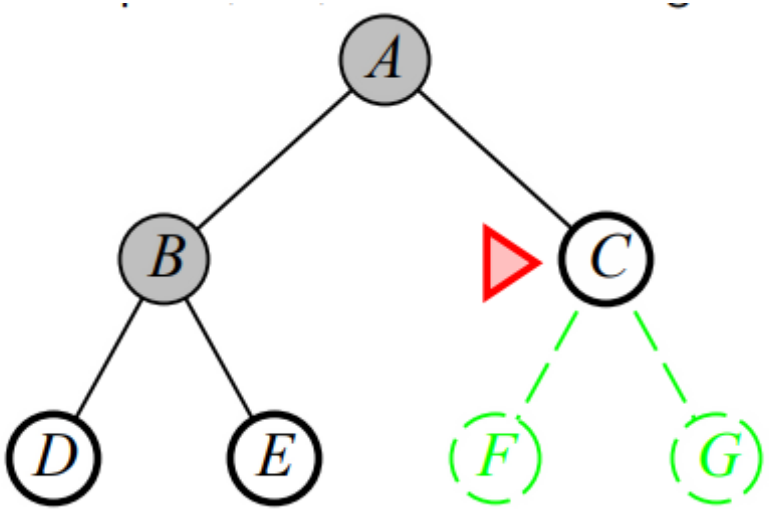
- Space is big problem
  - typical real-world problem: b = 10
    processing speed 10^6 nodes/sec
    memory requirements of 1 Kbyte/node
  - depth d = 10 → take less than 3 hours
    10 terabytes
  - depth d = 14, even with infinite memory
    takes 3.5 years



Search Algorithms
Uninformed search: Breadth-first search

- Properties
  - Complete? Yes
  - Time? $1+b+b^2+b^3+\ldots+b^d = O(b^d)$: exp. in d
  - Space? $O(b^d)$ (keeps every node in memory)
  - Optimal? Yes (if cost = 1/step), not in general

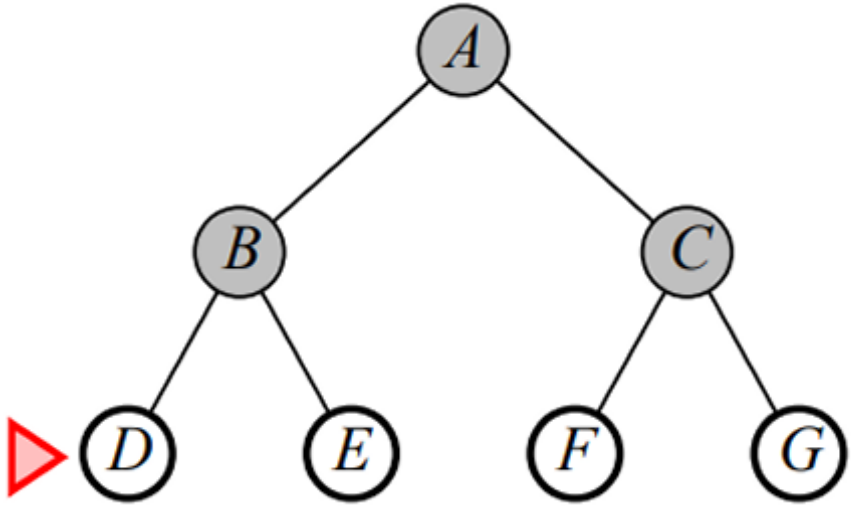
- Space is big problem
  - typical real-world problem: b = 10
    processing speed 10^6 nodes/sec
    memory requirements of 1 Kbyte/node
  - depth d = 10 → take less than 3 hours
    10 terabytes
  - depth d = 14, even with infinite memory
    takes 3.5 years



Search Algorithms
Uninformed search: Breadth-first search

- Properties
  - Complete? Yes
  - Time? $1+b+b^2+b^3+\ldots+b^d = O(b^d)$: exp. in d
  - Space? $O(b^d)$ (keeps every node in memory)
  - Optimal? Yes (if cost = 1/step), not in general

- Space is big problem
  - typical real-world problem: b = 10
    processing speed 10^6 nodes/sec
    memory requirements of 1 Kbyte/node
  - depth d = 10 → take less than 3 hours
    10 terabytes
  - depth d = 14, even with infinite memory
    takes 3.5 years



Search Algorithms
Uninformed search: Uniform-cost search

Search Algorithms
Uninformed search: Uniform-cost search

Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs

Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node

Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost

Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function

Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node

Search Algorithms
Uninformed search: Uniform-cost search

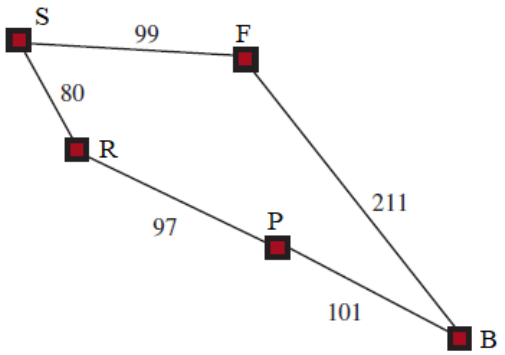
- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node

```
function UNIFORM-COST-SEARCH(problem)
returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem,
PATH-COST)
```

Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node

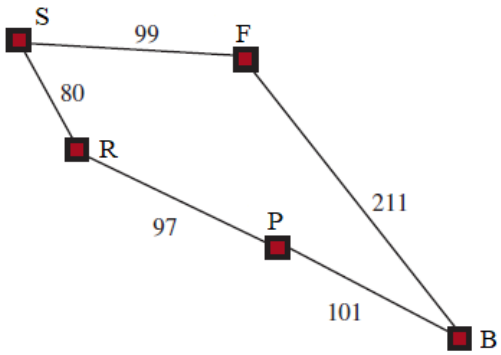
```
function UNIFORM-COST-SEARCH(problem)
returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem,
PATH-COST)
```



Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node
- Ex. To reach B from S
  1. POP (S,0), Expand

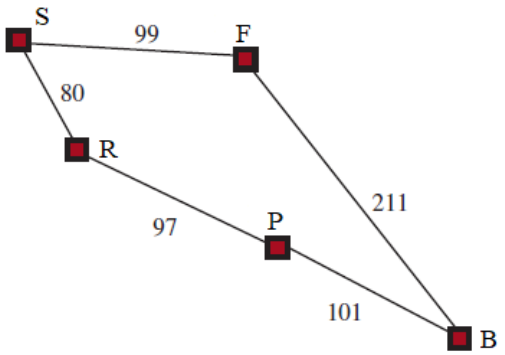
```
function UNIFORM-COST-SEARCH(problem)
returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem,
PATH-COST)
```



Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node
- Ex. To reach B from S
  1. POP (S/0), Expand:

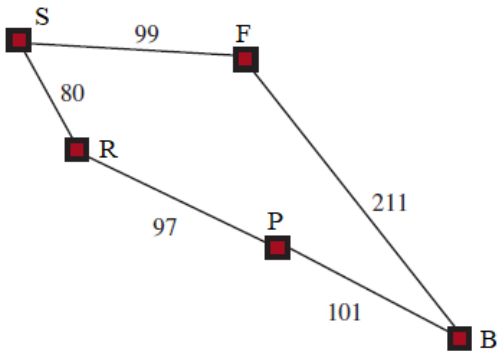
```
function UNIFORM-COST-SEARCH(problem)
returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem,
PATH-COST)
```



Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node
- Ex. To reach B from S
  1. POP (S/0), Expand:
     - Successors of (S): (R/80) , (F/99)

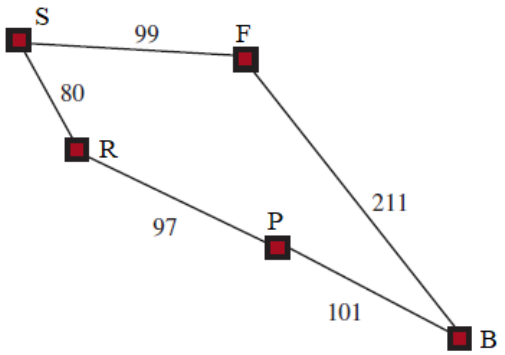
```
function UNIFORM-COST-SEARCH(problem)
returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem,
PATH-COST)
```



Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node
- Ex. To reach B from S
  1. POP (S/0), Expand:
     - Successors of (S): (R/80) , (F/99)
     - Added to priority queue (prioritize lowest cost)

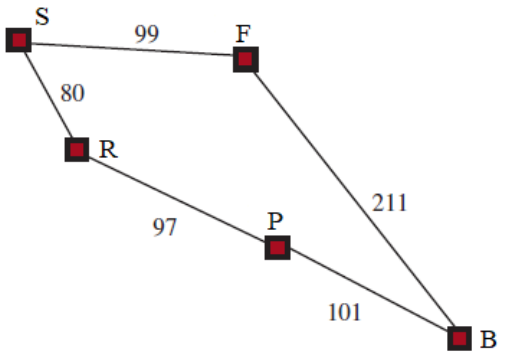
```
function UNIFORM-COST-SEARCH(problem)
returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem,
PATH-COST)
```



Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node
- Ex. To reach B from S
  1. POP (S/0), Expand:
     - Successors of (S): (R/80) , (F/99)
     - Added to priority queue (prioritize lowest cost)
  2. POP (R/80), Expand:

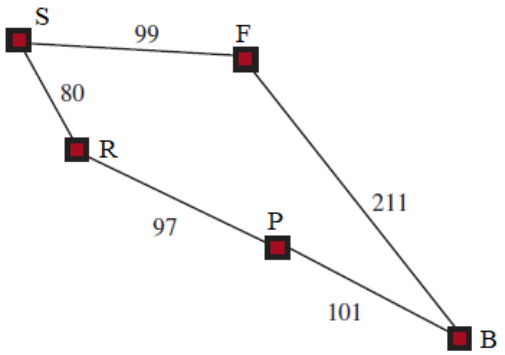
```
function UNIFORM-COST-SEARCH(problem)
returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem,
PATH-COST)
```



Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node
- Ex. To reach B from S
  1. POP (S/0), Expand:
     - Successors of (S): (R/80) , (F/99)
     - Added to priority queue (prioritize lowest cost)
  2. POP (R/80), Expand:
     - Successor(s): (P/80+97=177)

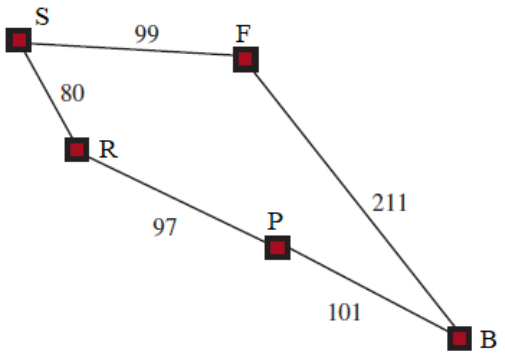
```
function UNIFORM-COST-SEARCH(problem)
returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem,
PATH-COST)
```



Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node
- Ex. To reach B from S
  1. POP (S/0), Expand:
     - Successors of (S): (R/80) , (F/99)
     - Added to priority queue (prioritize lowest cost)
  2. POP (R/80), Expand:
     - Successor(s): (P/80+97=177)
     - Added to priority queue

```
function UNIFORM-COST-SEARCH(problem)
returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem,
PATH-COST)
```

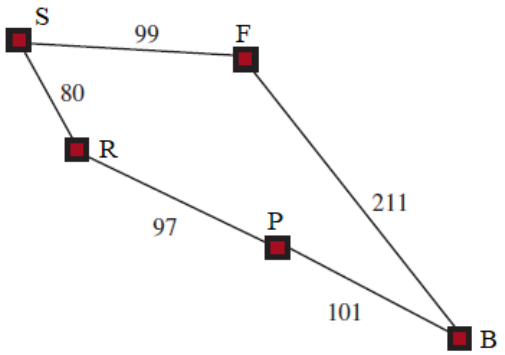


Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node
- Ex. To reach B from S

1. POP (S/0), Expand:
   - Successors of (S): (R/80) , (F/99)
   - Added to priority queue (prioritize lowest cost)
2. POP (R/80), Expand:
   - Successor(s): (P/80+97=177)
   - Added to priority queue
3. POP (F/99), Expand:

```
function UNIFORM-COST-SEARCH(problem)
returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem,
PATH-COST)
```

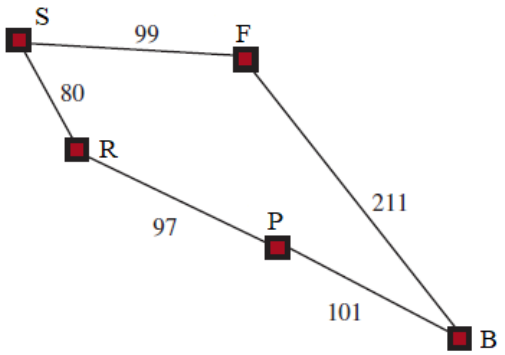


Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node
- Ex. To reach B from S

1. POP (S/0), Expand:
   - Successors of (S): (R/80) , (F/99)
   - Added to priority queue (prioritize lowest cost)
2. POP (R/80), Expand:
   - Successor(s): (P/80+97=177)
   - Added to priority queue
3. POP (F/99), Expand:
   - Successor(s): (B/99+211=310)

```
function UNIFORM-COST-SEARCH(problem)
returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem,
PATH-COST)
```

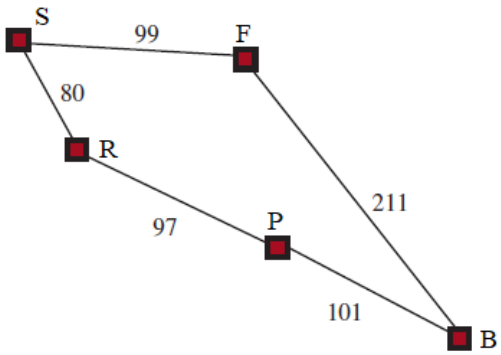


Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node
- Ex. To reach B from S

1. POP (S/0), Expand:
   - Successors of (S): (R/80) , (F/99)
   - Added to priority queue (prioritize lowest cost)
2. POP (R/80), Expand:
   - Successor(s): (P/80+97=177)
   - Added to priority queue
3. POP (F/99), Expand:
   - Successor(s): (B/99+211=310)
   - Added to priority queue , Note not detected as goal

```
function UNIFORM-COST-SEARCH(problem)
returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem,
PATH-COST)
```

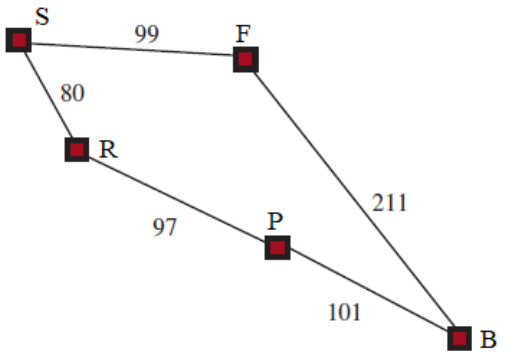


Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node
- Ex. To reach B from S

```
function UNIFORM-COST-SEARCH(problem)
returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem,
PATH-COST)
```

1. POP (S/0), Expand:
   - Successors of (S): (R/80) , (F/99)
   - Added to priority queue (prioritize lowest cost)
2. POP (R/80), Expand:
   - Successor(s): (P/80+97=177)
   - Added to priority queue
3. POP (F/99), Expand:
   - Successor(s): (B/99+211=310)
   - Added to priority queue , Note not detected as goal
4. POP (P/177), Expand:

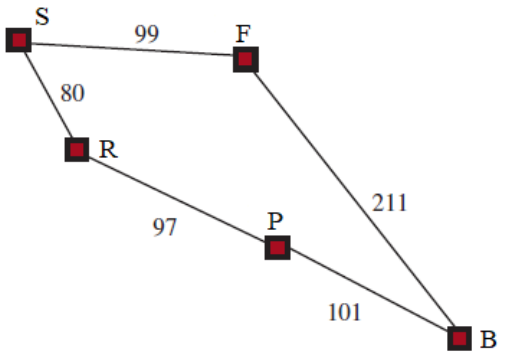


Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node
- Ex. To reach B from S

1. POP (S/0), Expand:
   - Successors of (S): (R/80) , (F/99)
   - Added to priority queue (prioritize lowest cost)
2. POP (R/80), Expand:
   - Successor(s): (P/80+97=177)
   - Added to priority queue
3. POP (F/99), Expand:
   - Successor(s): (B/99+211=310)
   - Added to priority queue , Note not detected as goal
4. POP (P/177), Expand:
   - Successor(s): (B/177+101=278)
   - Added to priority queue , Note not detected as goal

```
function UNIFORM-COST-SEARCH(problem)
returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem,
PATH-COST)
```

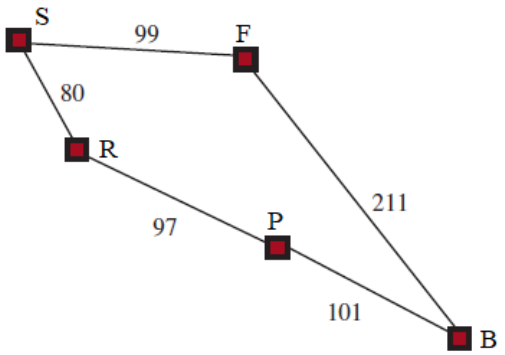


Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node
- Ex. To reach B from S

1. POP (S/0), Expand:
   - Successors of (S): (R/80) , (F/99)
   - Added to priority queue (prioritize lowest cost)
2. POP (R/80), Expand:
   - Successor(s): (P/80+97=177)
   - Added to priority queue
3. POP (F/99), Expand:
   - Successor(s): (B/99+211=310)
   - Added to priority queue , Note not detected as goal
4. POP (P/177), Expand:
   - Successor(s): (B/177+101=278)
   - Added to priority queue , Note not detected as goal

```
function UNIFORM-COST-SEARCH(problem)
returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem,
PATH-COST)
```

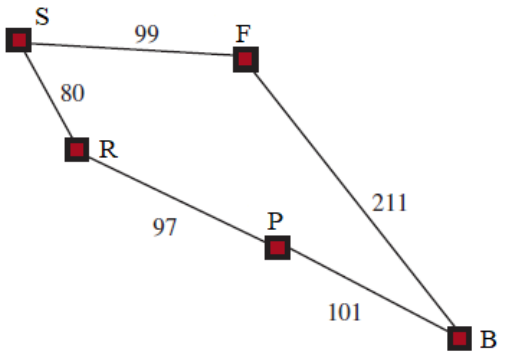


Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node
- Ex. To reach B from S

1. POP (S/0), Expand:
   - Successors of (S): (R/80) , (F/99)
   - Added to priority queue (prioritize lowest cost)
2. POP (R/80), Expand:
   - Successor(s): (P/80+97=177)
   - Added to priority queue
3. POP (F/99), Expand:
   - Successor(s): (B/99+211=310)
   - Added to priority queue , Note not detected as goal
4. POP (P/177), Expand:
   - Successor(s): (B/177+101=278)
   - Added to priority queue , Note not detected as goal
5. POP (B/278)

```
function UNIFORM-COST-SEARCH(problem)
returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem,
PATH-COST)
```

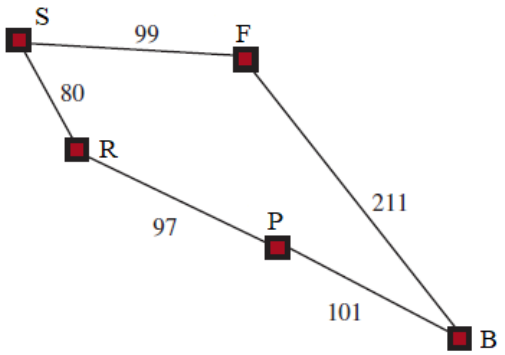


Search Algorithms
Uninformed search: Uniform-cost search

- When actions have different costs
  - obvious choice is use best-first search where evaluation function is cost of path from root to current node
  - idea is search spreads out in waves of uniform path-cost
  - can be implemented as call to BEST-FIRST-SEARCH with PATH-COST as evaluation function
- Expand least-cost unexpanded node
- Ex. To reach B from S

1. POP (S/0), Expand:
   - Successors of (S): (R/80) , (F/99)
   - Added to priority queue (prioritize lowest cost)
2. POP (R/80), Expand:
   - Successor(s): (P/80+97=177)
   - Added to priority queue
3. POP (F/99), Expand:
   - Successor(s): (B/99+211=310)
   - Added to priority queue , Note not detected as goal
4. POP (P/177), Expand:
   - Successor(s): (B/177+101=278)
   - Added to priority queue , Note not detected as goal
5. POP (B/278)
   - Detect Goal!

```
function UNIFORM-COST-SEARCH(problem)
returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem,
PATH-COST)
```



Search Algorithms
Uninformed search: Uniform-cost search

- Implementation:
  - fringe = priority queue ordered by path cost, lowest first
  - Equivalent to breadth-first if step costs all equal
- Properties
  - Complete? Yes, if action cost $\geq \varepsilon > 0$
    - where ε is lower bound on cost of each action
  - Time? in worst case $O(b^{\lceil C^*/\varepsilon \rceil})$
    - where C^* is cost of optimal solution
  - Space? in worst case, $O(b^{\lceil C^*/\varepsilon \rceil})$
  - Optimal? Yes, nodes expanded in increasing order of cost

Search Algorithms
Uninformed search: Depth-first search

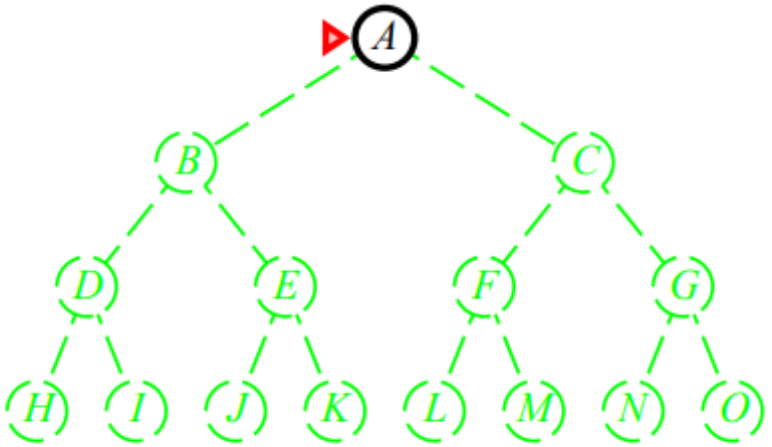
- Expand deepest unexpanded node
- Implementation:
  - fringe = LIFO queue, i.e., put successors at front
- Properties
  - Complete? No: fails in infinite-depth spaces
    - spaces with loops
    - Modify to avoid repeated states along path
      → complete in finite spaces
  - Time? $O(b^m)$: terrible if m >> d
    - but if solutions are dense →
      may be much faster than breadth-first
  - Space? O(b.m), i.e., linear space!
  - Optimal? No

Search Algorithms
Uninformed search: Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - fringe = LIFO queue, i.e., put successors at front
- Properties
  - Complete? No: fails in infinite-depth spaces
    - spaces with loops
    - Modify to avoid repeated states along path
      → complete in finite spaces
  - Time? $O(b^m)$: terrible if m >> d
    - but if solutions are dense →
      may be much faster than breadth-first
  - Space? O(b.m), i.e., linear space!
  - Optimal? No

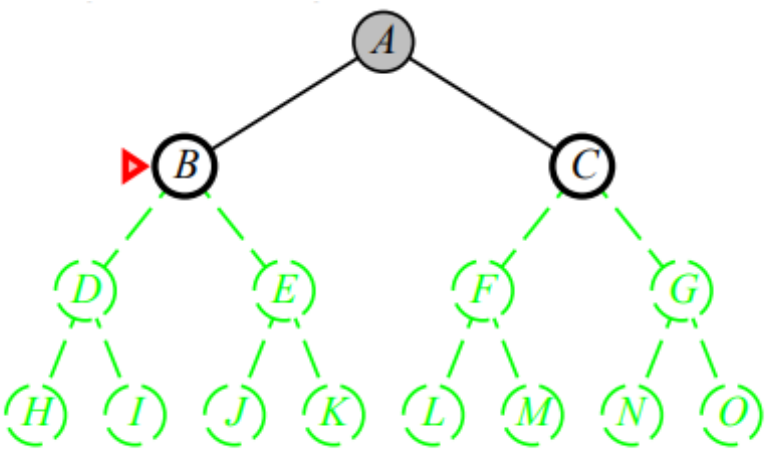
Search Algorithms
Uninformed search: Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - fringe = LIFO queue, i.e., put successors at front
- Properties
  - Complete? No: fails in infinite-depth spaces
    - spaces with loops
    - Modify to avoid repeated states along path
      → complete in finite spaces
  - Time? $O(b^m)$: terrible if m >> d
    - but if solutions are dense →
      may be much faster than breadth-first
  - Space? O(b.m), i.e., linear space!
  - Optimal? No



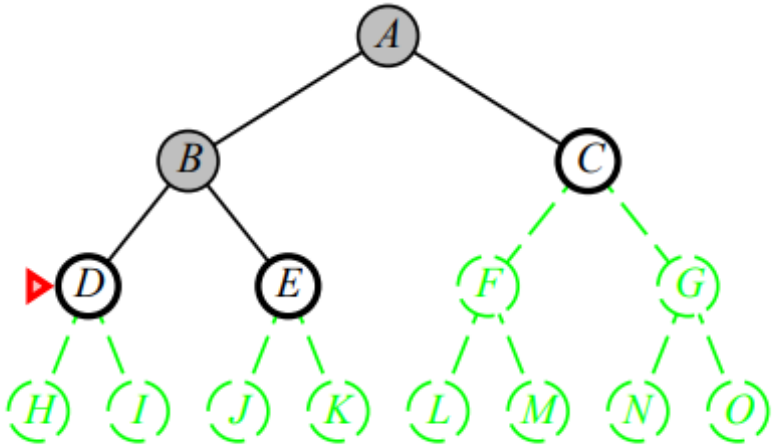
Search Algorithms
Uninformed search: Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - fringe = LIFO queue, i.e., put successors at front
- Properties
  - Complete? No: fails in infinite-depth spaces
    - spaces with loops
    - Modify to avoid repeated states along path
      → complete in finite spaces
  - Time? $O(b^m)$: terrible if m >> d
    - but if solutions are dense →
      may be much faster than breadth-first
  - Space? O(b.m), i.e., linear space!
  - Optimal? No



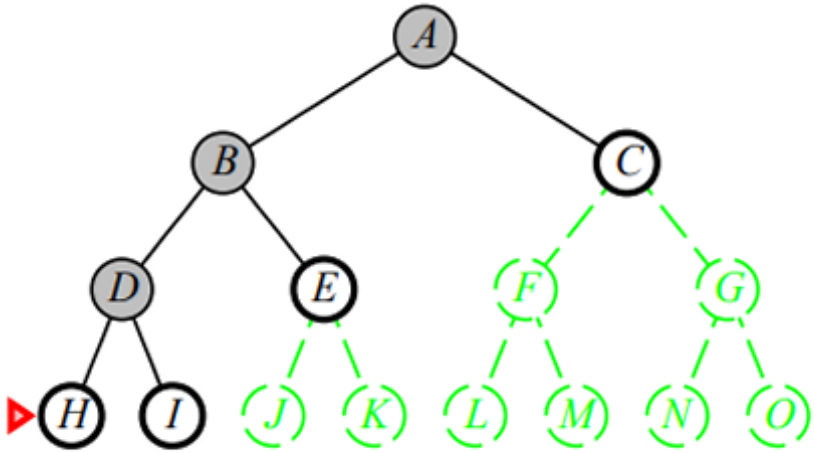
Search Algorithms
Uninformed search: Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - fringe = LIFO queue, i.e., put successors at front
- Properties
  - Complete? No: fails in infinite-depth spaces
    - spaces with loops
    - Modify to avoid repeated states along path
      → complete in finite spaces
  - Time? $O(b^m)$: terrible if m >> d
    - but if solutions are dense →
      may be much faster than breadth-first
  - Space? O(b.m), i.e., linear space!
  - Optimal? No



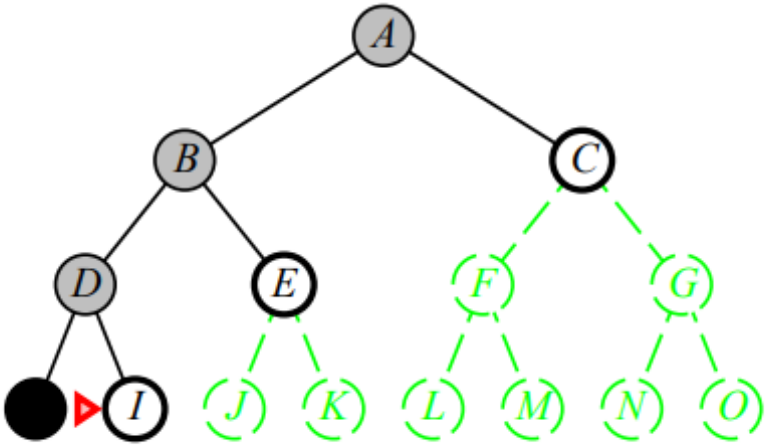
Search Algorithms
Uninformed search: Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - fringe = LIFO queue, i.e., put successors at front
- Properties
  - Complete? No: fails in infinite-depth spaces
    - spaces with loops
    - Modify to avoid repeated states along path
      → complete in finite spaces
  - Time? $O(b^m)$: terrible if m >> d
    - but if solutions are dense →
      may be much faster than breadth-first
  - Space? O(b.m), i.e., linear space!
  - Optimal? No



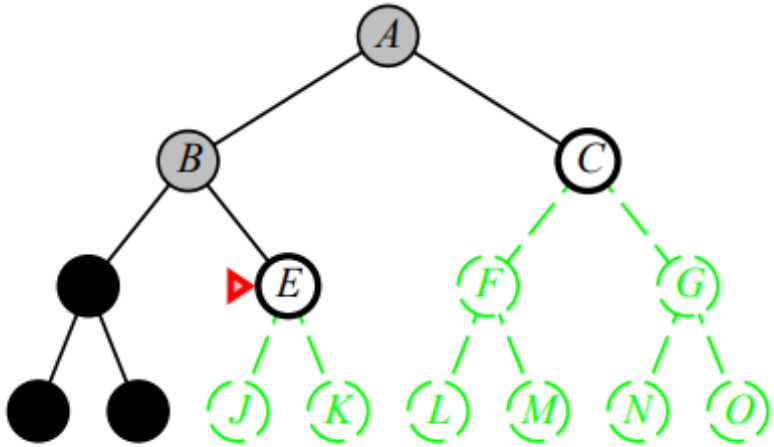
Search Algorithms
Uninformed search: Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - fringe = LIFO queue, i.e., put successors at front
- Properties
  - Complete? No: fails in infinite-depth spaces
    - spaces with loops
    - Modify to avoid repeated states along path
      → complete in finite spaces
  - Time? $O(b^m)$: terrible if m >> d
    - but if solutions are dense →
      may be much faster than breadth-first
  - Space? O(b.m), i.e., linear space!
  - Optimal? No



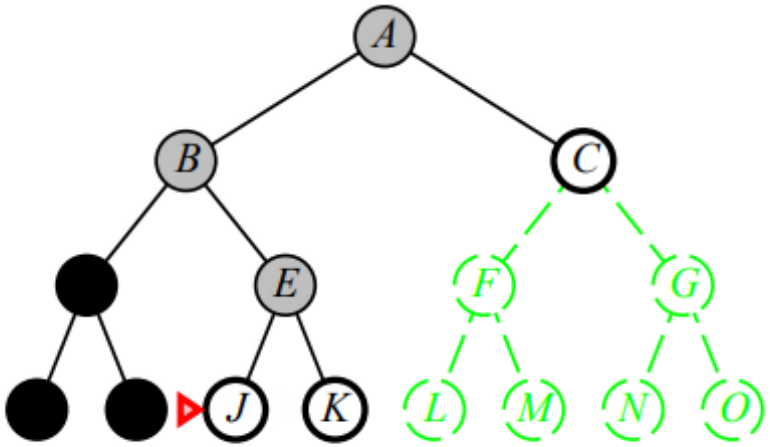
Search Algorithms
Uninformed search: Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - fringe = LIFO queue, i.e., put successors at front
- Properties
  - Complete? No: fails in infinite-depth spaces
    - spaces with loops
    - Modify to avoid repeated states along path
      → complete in finite spaces
  - Time? $O(b^m)$: terrible if m >> d
    - but if solutions are dense →
      may be much faster than breadth-first
  - Space? O(b.m), i.e., linear space!
  - Optimal? No



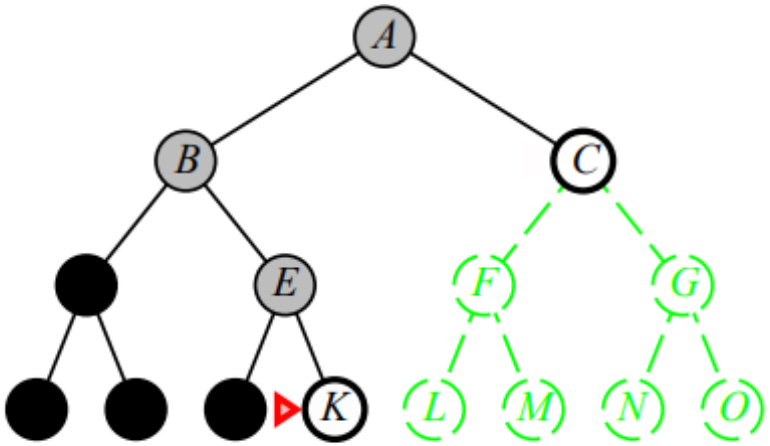
Search Algorithms
Uninformed search: Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - fringe = LIFO queue, i.e., put successors at front
- Properties
  - Complete? No: fails in infinite-depth spaces
    - spaces with loops
    - Modify to avoid repeated states along path
      → complete in finite spaces
  - Time? $O(b^m)$: terrible if m >> d
    - but if solutions are dense →
      may be much faster than breadth-first
  - Space? O(b.m), i.e., linear space!
  - Optimal? No



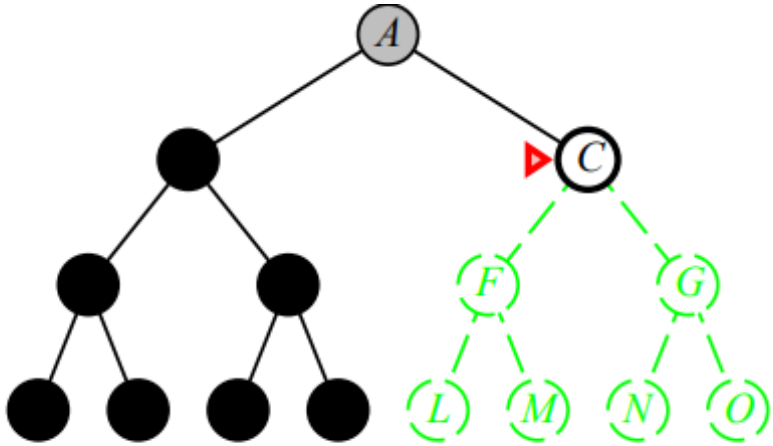
Search Algorithms
Uninformed search: Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - fringe = LIFO queue, i.e., put successors at front
- Properties
  - Complete? No: fails in infinite-depth spaces
    - spaces with loops
    - Modify to avoid repeated states along path
      → complete in finite spaces
  - Time? $O(b^m)$: terrible if m >> d
    - but if solutions are dense →
      may be much faster than breadth-first
  - Space? O(b.m), i.e., linear space!
  - Optimal? No



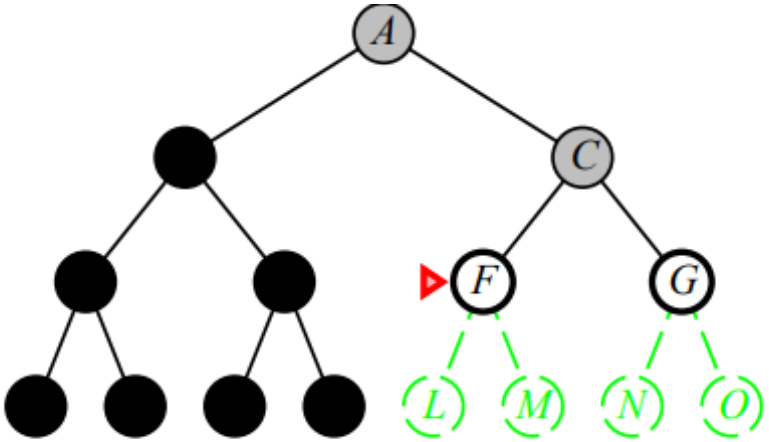
Search Algorithms
Uninformed search: Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - fringe = LIFO queue, i.e., put successors at front
- Properties
  - Complete? No: fails in infinite-depth spaces
    - spaces with loops
    - Modify to avoid repeated states along path
      → complete in finite spaces
  - Time? $O(b^m)$: terrible if m >> d
    - but if solutions are dense →
      may be much faster than breadth-first
  - Space? O(b.m), i.e., linear space!
  - Optimal? No



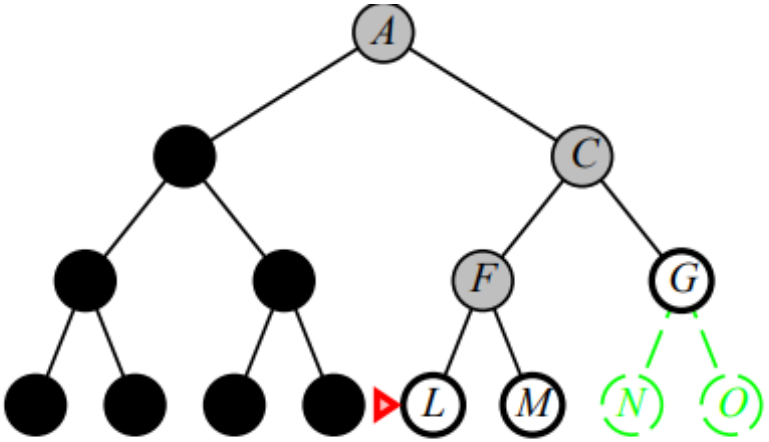
Search Algorithms
Uninformed search: Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - fringe = LIFO queue, i.e., put successors at front
- Properties
  - Complete? No: fails in infinite-depth spaces
    - spaces with loops
    - Modify to avoid repeated states along path
      → complete in finite spaces
  - Time? $O(b^m)$: terrible if m >> d
    - but if solutions are dense →
      may be much faster than breadth-first
  - Space? O(b.m), i.e., linear space!
  - Optimal? No



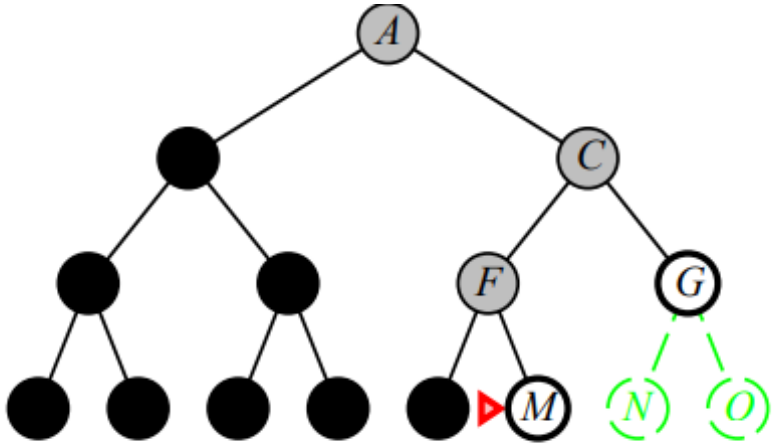
Search Algorithms
Uninformed search: Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - fringe = LIFO queue, i.e., put successors at front
- Properties
  - Complete? No: fails in infinite-depth spaces
    - spaces with loops
    - Modify to avoid repeated states along path
      → complete in finite spaces
  - Time? $O(b^m)$: terrible if m >> d
    - but if solutions are dense →
      may be much faster than breadth-first
  - Space? O(b.m), i.e., linear space!
  - Optimal? No



Search Algorithms
Uninformed search: Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - fringe = LIFO queue, i.e., put successors at front
- Properties
  - Complete? No: fails in infinite-depth spaces
    - spaces with loops
    - Modify to avoid repeated states along path
      → complete in finite spaces
  - Time? $O(b^m)$: terrible if m >> d
    - but if solutions are dense →
      may be much faster than breadth-first
  - Space? O(b.m), i.e., linear space!
  - Optimal? No



Search Algorithms
Uninformed search: Depth-limited search

- Same as depth-first search with depth limit *l*
  - nodes at depth *l* have no successors
  - can eliminate cycles at cost of some computation time.
    - If we look only a few links up in parent chain we can catch most cycles
  - good depth limit can be chosen based on knowledge of the problem
    - Diameter of the state-space graph → better depth limit → more efficient depth-limited search

- Implementation:
  - Recursive implementation

```
function Depth-Limited-Search(problem,limit) returns soln/fail/cutoff
  Recursive-DLS(Make-Node(Initial-State[problem]),problem,limit)

function Recursive-DLS(node,problem,limit) returns soln/fail/cutoff
  cutoff-occurred? ←false
  if Goal-Test(problem,State[node]) then return node
  else if Depth[node] = limit then return cutoff
  else for each successor in Expand(node,problem) do
    result ← Recursive-DLS(successor,problem,limit)
    if result = cutoff then cutoff-occurred? ←true
    else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

Search Algorithms
Uninformed search: Iterative deepening search

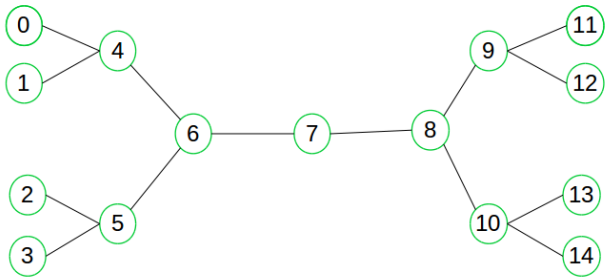
- Solves problem of picking a good value for *l*
  - by trying all values: 0, 1, 2, and so on
  - until either solution is found, or depth limited search returns failure value rather than the cutoff value
- Combines many of the benefits of depth-first and breadth-first search
  - Like DFS: its memory requirements are modest
    - O(bd) when there is solution
    - O(bm) on finite state spaces with no solution
  - Like BFS:
    - optimal for problems where all actions have same cost
    - complete on finite acyclic state spaces, or on any finite state space (check nodes for cycles all way up path)

```
function Iterative-Deepening-Search(problem) returns a solution
  inputs: problem, a problem
  for depth ← 0 to ∞ do
    result ← Depth-Limited-Search( problem, depth)
    if result ≠ cutoff then return result
  end
```

Search Algorithms
Uninformed search: Bi-directional Search (BDS)

- Main idea:
  - start searching from both initial state and goal state
    - Forward search from source state toward goal state
    - Backward search from goal state toward source state
  - meet in middle.

- Properties
  - Complete? Yes
  - Optimal? Yes
  - Time Complexity: $O(b^{d/2})$
  - Space Complexity: $O(b^{d/2})$



Search Algorithms
Uninformed search: Comparison

	BFS	UCS	DFS	DLS	IDS	BDS*
Complete?	Yes	Yes	No	If l >= d	Yes	Yes
Optimal?	Yes	Yes	No	No	Yes	Yes
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	$b.m$	$b.l$	$b.d$	$b^{d/2}$

* if applicable, with BDS
d : branching factor
b : depth of solution
m : depth of problem
l : depth limit