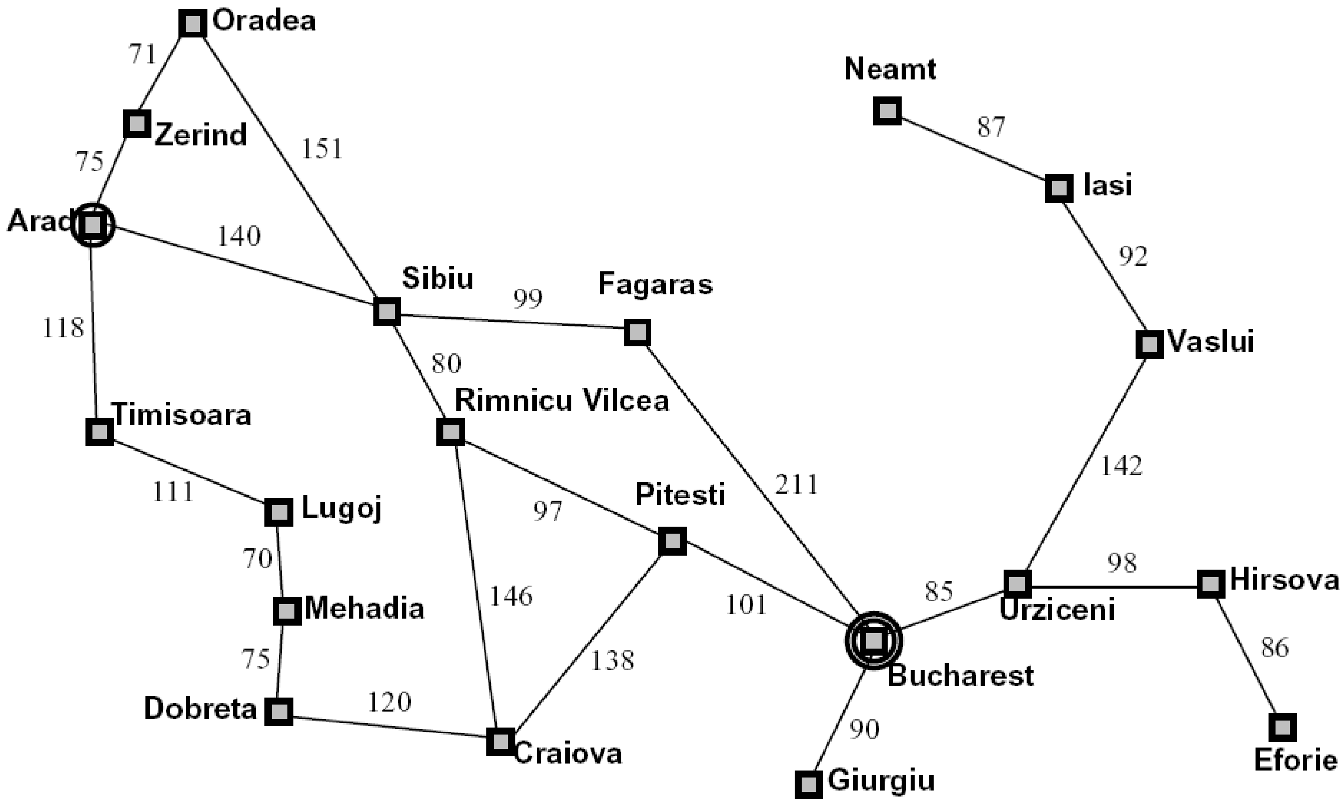
COSC76/276 Artificial Intelligence
Fall 2022
Uninformed Search (cont), Graph search and cost-sensitive search

Soroush Vosoughi
Computer Science
Dartmouth College
soroush@Dartmouth.edu

Reminders

- Thank you for completing SA-0!
  - Please read carefully the instructions
    - If in doubt, ask
- SA-1 due Sep 22nd
- PA-1 due Sep 28th
- SA-2 due Oct 1st

Recap: Search problem



- State space: cities
- Successor function: go to adjacent city
- Cost: distance between cities
- Start state: Arad
- Goal test: is state == Bucharest?

State space graph

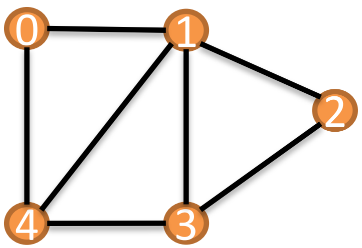
- State space graph: A mathematical representation of a search problem
  - States are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal states

- In a state space graph, each state occurs only once!

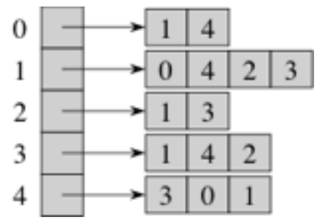
- The full graph is typically too big to store in memory

Reminder (from CS10): Graph representations

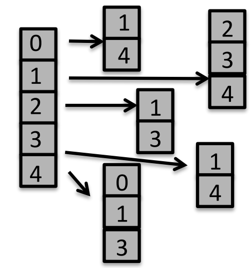
Anyone remembers?



{{0,1}, {0,4}, {1,2}, {1,3}, {1,4}, {2,3}, {3,4}}



$$\begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 2 & 0 & 1 & 0 & 1 & 0 \\ 3 & 0 & 1 & 1 & 0 & 1 \\ 4 & 1 & 1 & 0 & 1 & 0 \end{array}$$



Method	Edge List	Adjacency List	Adjacency Matrix	Adjacency Map
`in/outDegree(v)`	O(m)	O(1)	O(n)	O(1)
`in/outNeighbors(v)`	O(m)	$O(d_v)$	O(n)	$O(d_v)$
`hasEdge(u,v)`	O(m)	$O(\min(d_u,d_v))$	O(1)	O(1)
`insertVertex(v)`	O(1)	O(1)	$O(n^2)$	O(1)
`removeVertex(v)`	O(m)	$O(d_v)$	$O(n^2)$	$O(d_v)$
`insertEdge(u,v,e)`	O(1)	O(1)	O(1)	O(1)
`removeEdge(u,v)`	O(m)	O(1)	O(1)	O(1)

Best performance is shown in red

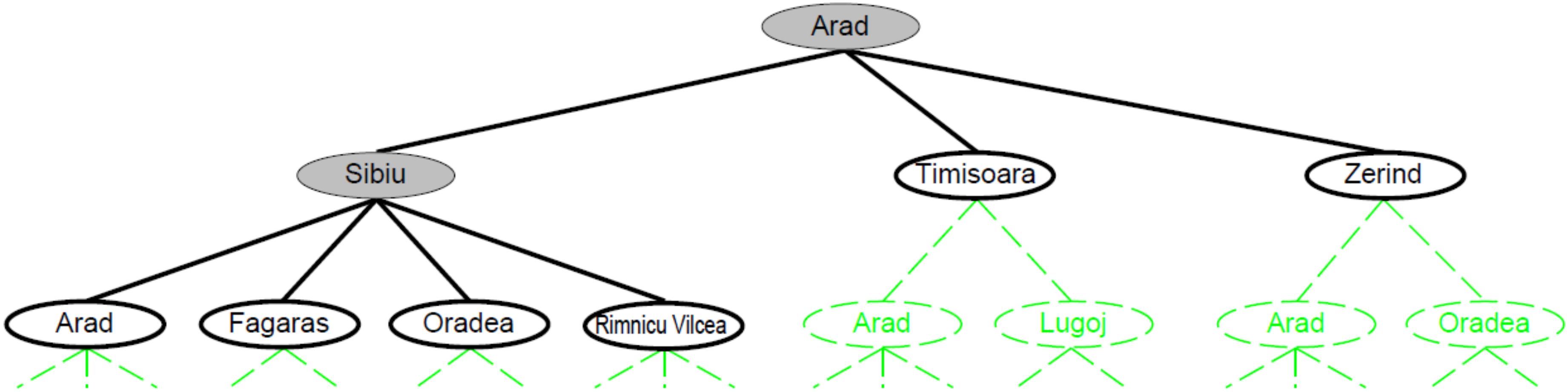
n = number of nodes (5), m = number of edges (7), d_v = degree of node v

Search problems

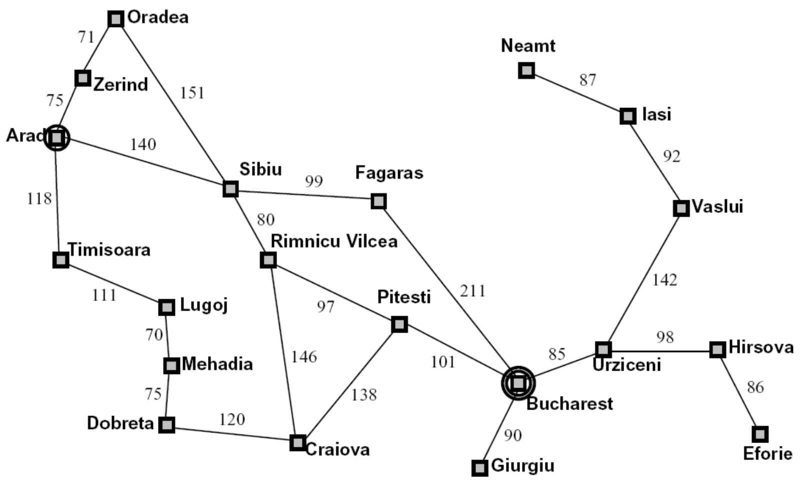
- Element of search problems
  - A start state
  - A `goal_test` function that checks if a state is a goal state
  - A `get_actions` function that finds the legal actions from some state and a `transition` function that accepts a state, an action, and returns a new state, or alternatively, a `get_successors` function that returns a list of states given a starting state
  - A path_cost function that gives the cost of a path between a pair of states.
- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

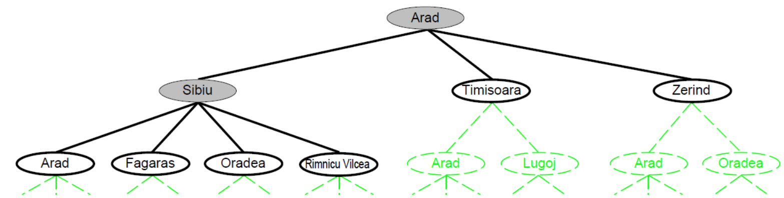
Search trees

- A search tree:
  - A "what if" tree of plans and their outcomes
  - The start state is contained in the root node
  - Children correspond to successors
  - Nodes show states and correspond to PLANS that achieve those states
  - For most problems, we can never actually build the whole tree



State space graphs vs search trees





- We construct both on demand – and we construct as little as possible

Implementation: states vs nodes

- A state is a (representation of) a physical configuration

- A node is a data structure constituting part of a search tree includes state, parent, action, depth, path cost g(x)

- Search problems
- Uninformed search algorithms (tree-search, without memory)

Searching with a search tree

- Search:
  - Expand out potential plans (tree nodes)
  - Maintain a fringe of partial plans under consideration
  - Try to expand as few tree nodes as possible

General tree search

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the 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 the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy
- Does not keep track of expanded nodes

Is there any potential inefficiency here?

Search strategies

- A strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - Time complexity – number of nodes generated/expanded
  - Space complexity – maximum number of nodes in memory
  - Completeness – does it always find a solution if one exists?
  - Optimality – does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - b – maximum branching factor of the search tree
  - d – depth of the shallowest solution
  - m – maximum depth of the state space (may be infinite)

Uninformed search strategies

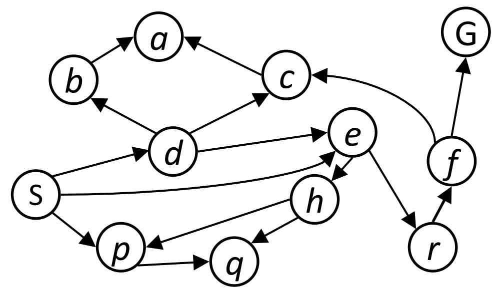
- Uninformed strategies use only information available in the problem definition

Breadth-First Search (BFS)

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue

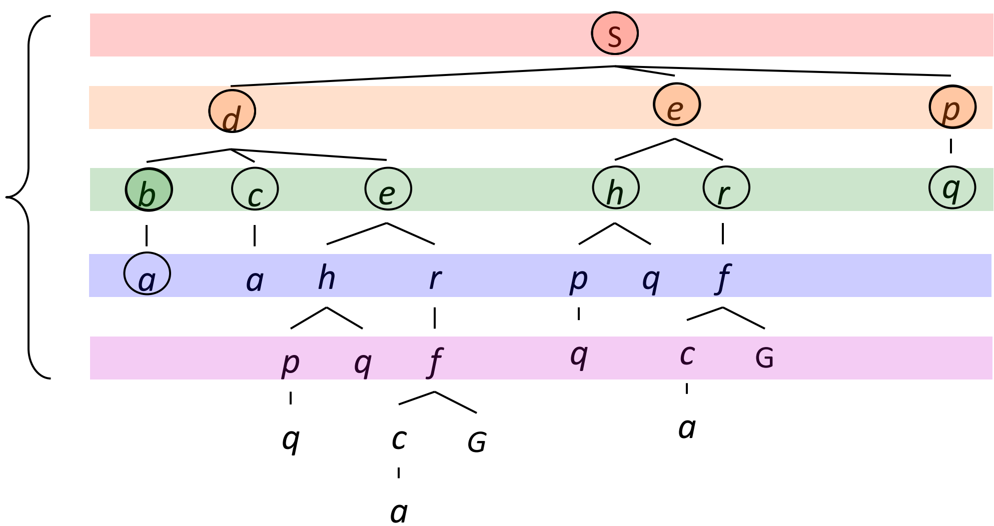
Example

Example: Breadth-First Search (tree-search)



Search

Tiers



BFS pseudocode (tree-search)

```
frontier = new queue
pack start state into a node
add node to frontier

while frontier is not empty:
  get current_node from the frontier
  get current_state from current_node

  if current_state is the goal:
    backchain from current_node and return solution

  for each child of current_state:
    pack child state into a node, with backpointer to current_node
    add the node to the frontier

return failure
```

Properties of BFS

- Time:
  - $O(b^d)$
- Space:
  - $O(b^d)$
- Complete:
  - Yes if b is finite
- Optimal:
  - Yes, only if costs are all identical

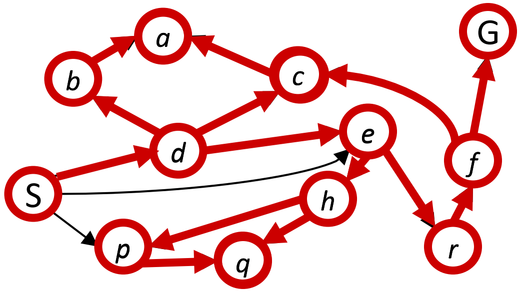
Discussion

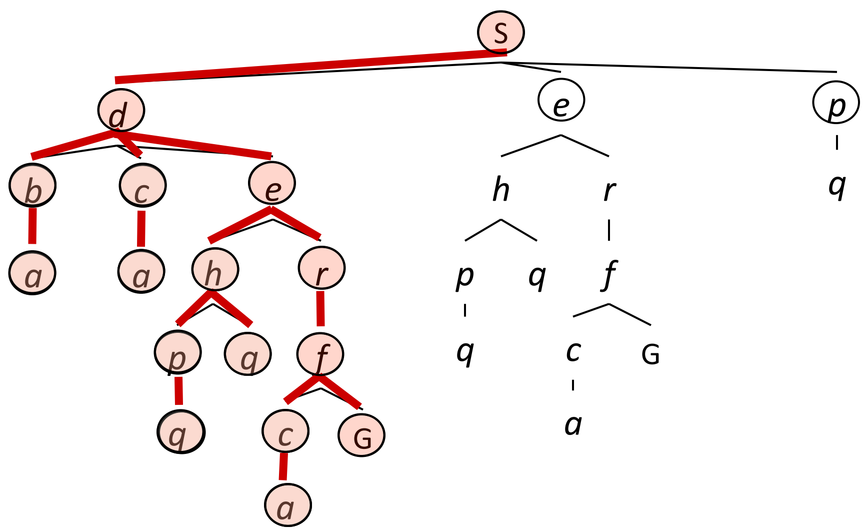
Depth-First Search (DFS)

- Expand deepest unexpanded nodes
- Implementation:
  - *fringe* is a LIFO stack

Example

Example: Depth-First Search (tree-search)





DFS Pseudocode (tree-search)

```
frontier = new stack
pack start_state into a node
add node to frontier

while frontier is not empty:
  get current_node from the frontier
  get current_state from current_node

  if current_state is the goal:
    backchain from current_node and return solution

  for each child of current_state:
    pack child state into node, add backpointer
    add the node to the frontier

return failure
```

Properties of DFS (tree-search)

- Time:
  - $O(b^m)$
- Space:
  - $O(bm)$
- Complete:
  - No
- Optimal:
  - No it finds the "leftmost" solution, regardless of depth or cost

Discussion

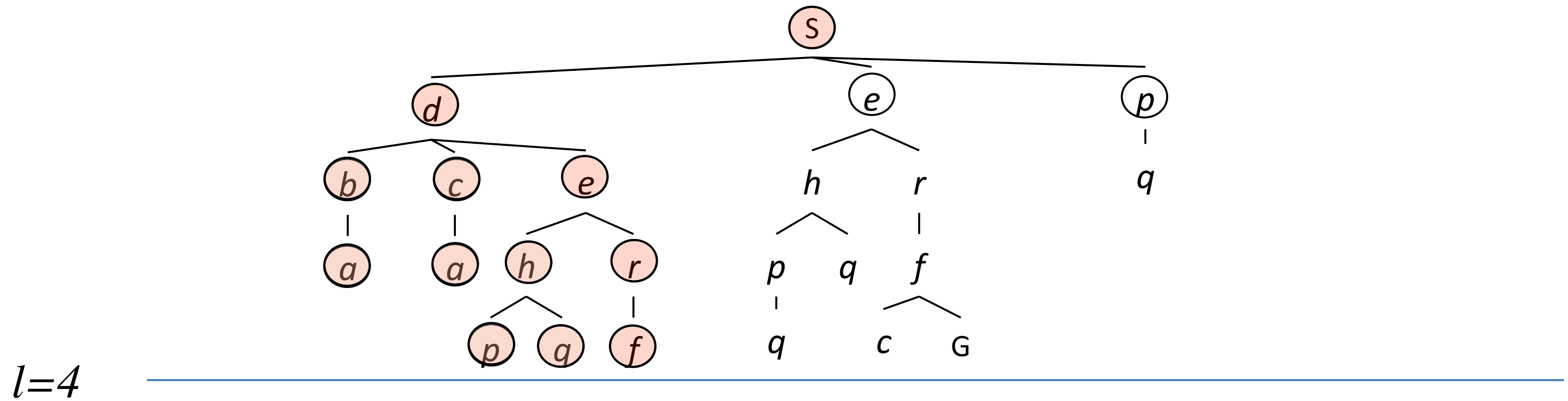
BFS vs DFS

- When will BFS outperform DFS?
  - Solutions not too far down
- When will DFS outperform BFS?
  - Solutions far at the bottom and memory constrained

Discussion

Depth-limited search

- DFS with depth limit l

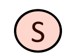


Properties of Depth limited

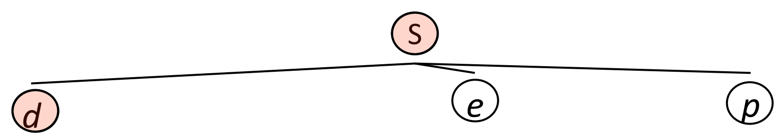
- Complete: No
- Time: $O(b^l)$
- Space: $O(bl)$
- Optimal: No

Iterative deepening

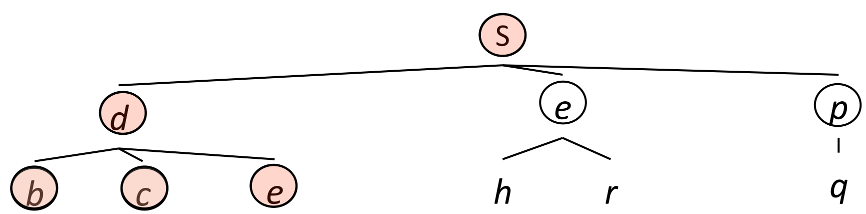
- Run Depth-limited search with increasing depth limit, i.e.,
  - $l=0$



  - $l=1$



  - $l=2$



  - …

Properties of Iterative deepening

- Complete: Yes if b is finite
- Time: $O(b^d)$
- Space: $O(bd)$
- Optimal: only if costs are all identical

PA-1 - First programming assignment

- Includes modeling of real problem as search problem
- Apply uninformed search to find a solution to the problem
- You will find it on Canvas soon

Summary

- Modeling a real-world problem as a search problem to abstract away real-world details
  - State and action space, transition function
  - Planning is all "in simulation"
  - Model is a simplification of the world
- Search tree built on the fly to find a solution
  - Does not keep track of expanded nodes
- Variety of uninformed search (tree-search version) with different time and space complexity
  - BFS: expands shallowest node first
  - DFS: expands deepest node first
  - Limited DFS: DFS up to a given depth
  - Iterative DFS: run limited DFS with increasing depth limit until solution found

Additional readings

- AIMA book: chapters 3.1-3.4
- (Reported on Canvas too in the calendar)

Next

- Keeping history to avoid repetitions
- Can we do any better when searching for a solution than the algorithms we have seen so far?

- Implement graph search methods (keeping track of history) for BFS and DFS

Outline

- Graph-search (memoizing)
  - BFS
  - DFS
    - Path-checking DFS
- Bi-directional search

Graph Search (memoizing)

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

- Tree-search does not keep track of the states already visited
- Graph-search does: memoizing – i.e., keeping track of the states already visited

BFS (graph) - pseudocode

```
frontier = new queue
pack start_state into a node
add node to frontier

explored = new set
add start_state to explored

while frontier is not empty:
  get current_node from the frontier
  get current_state from current_node

  if current_state is the goal:
    backchain from current_node and return solution

  for each child of current_state:
    if child not in explored:
      add child to explored
      pack child state into a node, with backpointer to current_node
      add the node to the frontier

return failure
```

DFS (graph) - pseudocode

```
frontier = new stack
pack start_state into a node
add node to frontier

explored = new set
add start_state to explored

while frontier is not empty:
    get current_node from the frontier
    get current_state from current_node

    if current_state is the goal:
      backchain from current_node and return solution

    for each child of current_state:
      if child not in explored:
        add child to explored
        pack child state into node, add backpointer
        add the node to the frontier

return failure
```

Is memoizing memory cost good for BFS and DFS?

- For BFS, memoizing memory cost is not so bad
  - Frontier is already big: $O(b^d)$
- For DFS, memoizing seems expensive
  - Frontier is tiny: $O(bm)$
- *Can we avoid building complete explored set for DFS?*

Discussion. Write down on paper first

Path-checking DFS

- Path-checking DFS keeps track of states on the current path only, and doesn't loop
- Does not eliminate redundant paths

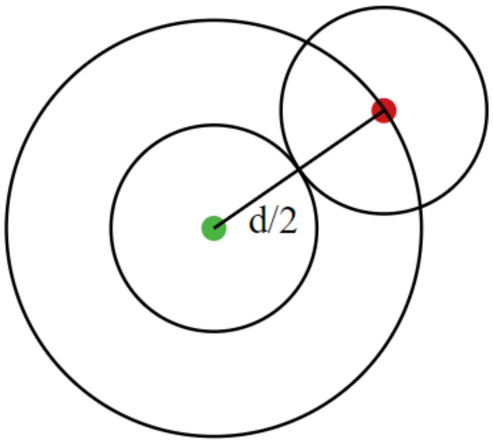
Comparing uninformed graph search

Algorithm	Time	Memory	Complete	Optimal
BFS (graph)	$O(min(n,b^d))$	$O(min(n,b^d))$	y	y*
DFS (memoizing)	$O(min(n,b^m))$	$O(min(n,b^m))$	y	n
DFS (path-checking)	$O(min(m^n,mb^m))$	$O(m)$	y	n
Iterative deepening (path-checking)	$O(min(d^n,db^d))$	$O(d)$	y	y*

With state space size n

Bi-directional search

- Sometimes you can search backwards:
  - a single identifiable goal
  - **inverse transition function** available
- Bi-directional search
  – Time and space $b^{d/2} + b^{d/2} < b^d$
  – Complete and Optimal: y if BFS (same caveats)



Summary

- Graph search to avoid repetitions
  - BFS, DFS (memoizing or path checking)
  - Trade-offs with memory use
- Bi-directional search: apply search from start and goal

Next

- Can we use cost and information about the goal to guide the search?
  - Uniform cost search
  - Informed search