CISS450 Lecture 3: Uninformed Search

Yihsiang Liow

August 30, 2020

Table of contents I

Agenda

- AI uninformed/blind/brute force search algorithms
- Readings:
  - AIMA3 Chapter 3, Sections 1-4
  - AIMA4 Chapter 3, Sections 1-4

Problem-Solving and Search I

- Recall robot vacuum problem (see Agents notes).
- Given an AI problem, try to convert into a graph search problem.

$$\text{solve AI problem} \rightarrow \text{graph search problem}$$

  - Identify states (of the environment) in your problem and action from one state to another – get the **state space graph**
  - Size of state space (if finite) will give indication of complexity of the problem.
  - Given an initial state, find a way to reach goal state(s): use graph search algorithms

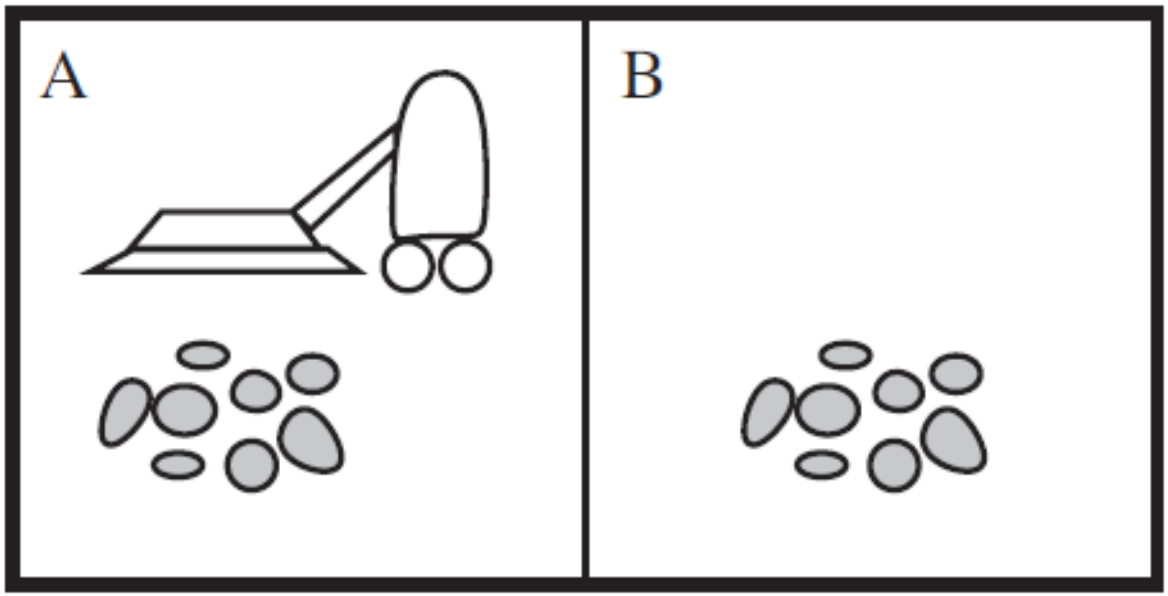
Problem-Solving and Search II

- The graph search does *not* build the whole graph before search: the search only builds what is needed. See later.
- Compute big-O of time and space of search algorithm
- Compute cost of actions from initial state to goal state. If possible, find solution with least cost – **optimal solution**
- Etc.

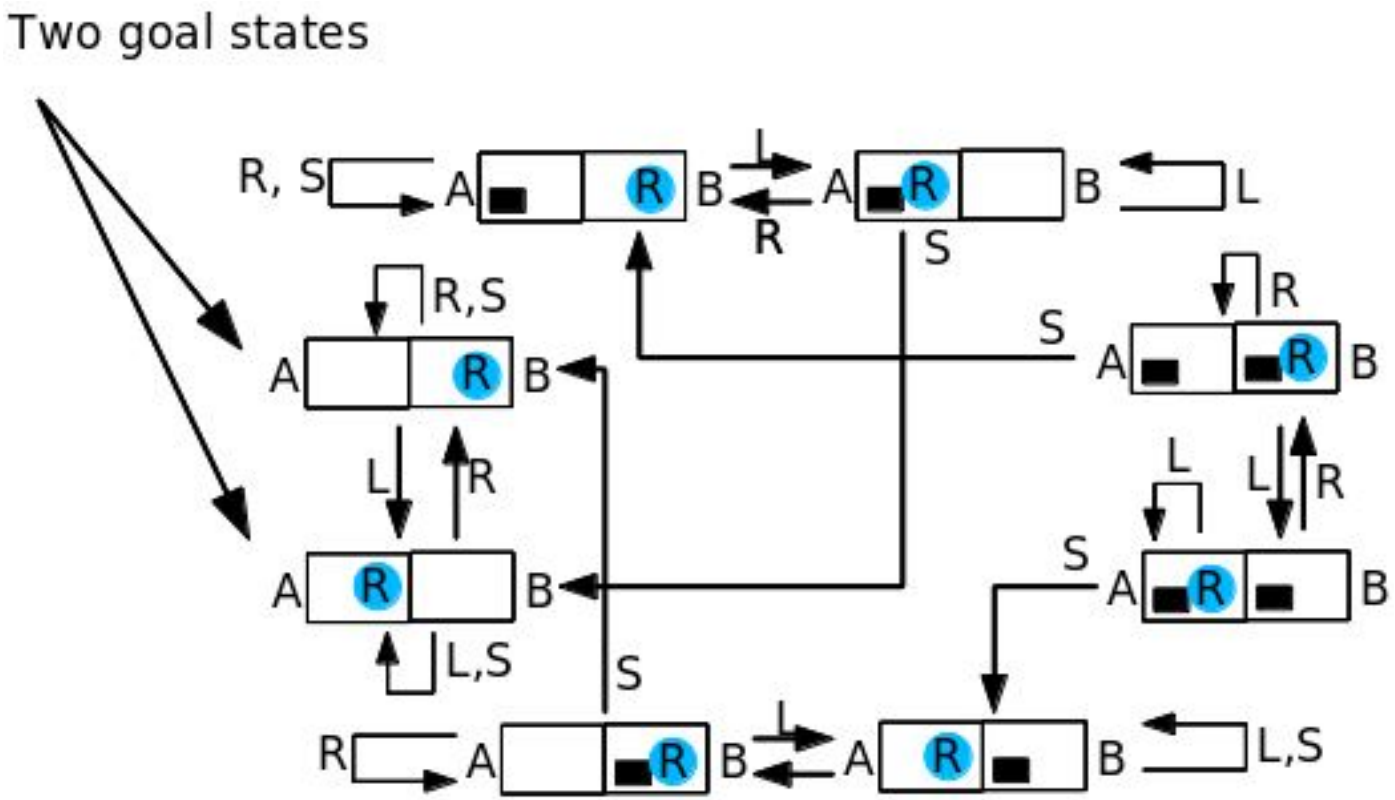
- To execute the search algorithm, we need to carry extra information for some states during the search process. Later you will see that instead of carrying the information in the state, we will create objects (one for each state) to carry the extra information. These objects are **search nodes** – each search node will contain a state (or rather pointer/reference to a state) and with some extra search information.

Example: Vacuum cleaner robot I

- Refer to vacuum cleaner robot problem from Agents notes:



Example: Vacuum cleaner robot II

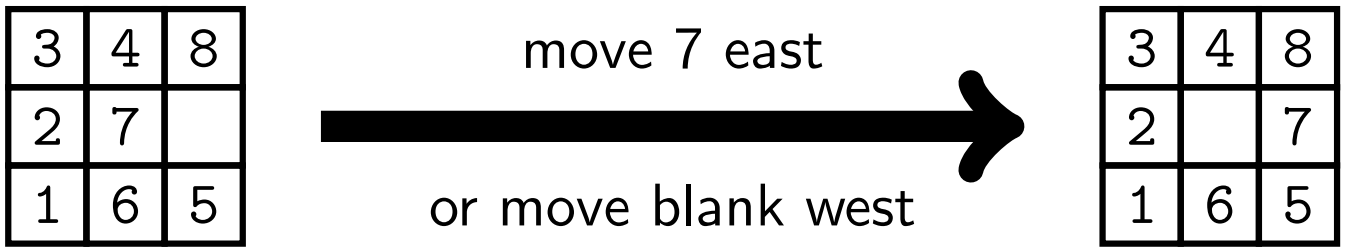


Example: Vacuum cleaner robot III

- PROBLEM. Vacuum cleaner robot problem.
  - States = (`robot_position`, `A_dirty`, `B_dirty`) where `robot_position` is A or B, the rest is 1 or 0 (for true or false).
  - Goal states: (A, 0, 0), (B, 0, 0)
  - Actions = L (move left), R (move right), S (suck)
  - State space size = 8.
- In this case, the state space is so small, that we can actually create the whole state space. Not possible for state space of chess!
- A function that tests whether a state is a goal state is called the **goal test**.

Example: $n^2 - 1$ Puzzle I

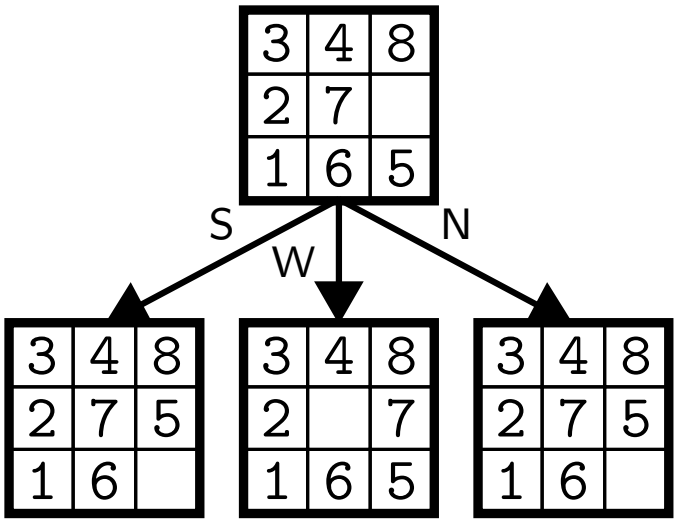
- You have $3^2 - 1 = 8$ tiles numbered 1, 2, ..., 8 on a 3×3 area. You can move one tile at a time to an empty spot:



- I'll just say the action is "move blank (or space) west", or W for short. We also have actions N, S, E for moving the blank/space north, south, east.

Example: $n^2 - 1$ Puzzle II

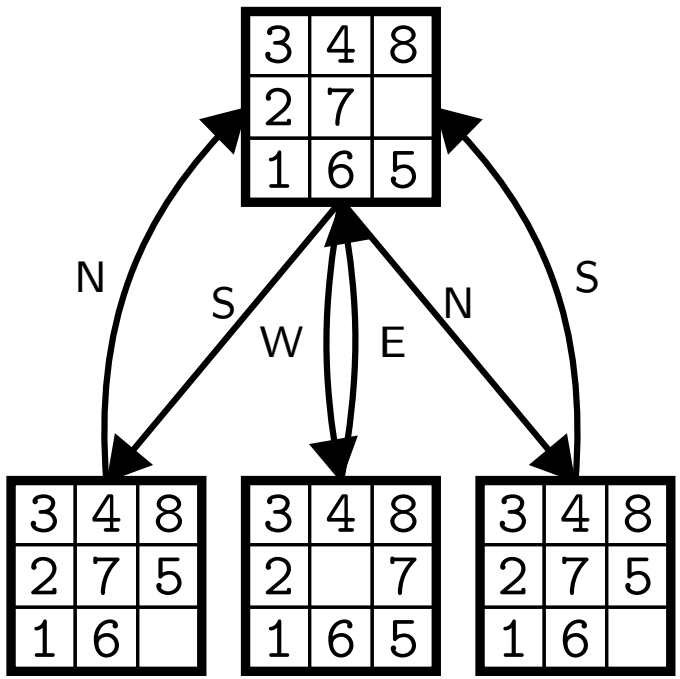
- Here is a state (the one on top) with 3 possible actions:



We say that the top state **expands** to the three states below. The three states below are the **successor** states of the top state.

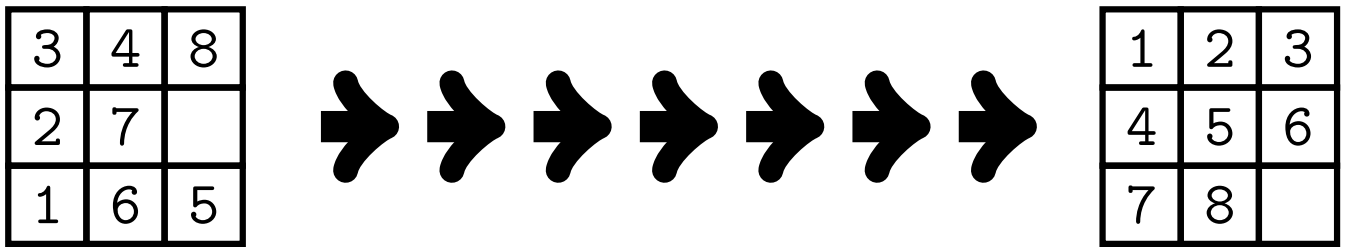
Example: $n^2 - 1$ Puzzle III

- Exercise. Of course there's a lot more. Not a trivial graph. Add all successor states of the bottom three states and all relevant actions. (You can add more if you like.)



Example: $n^2 - 1$ Puzzle IV

- Goal: Given initial state, find a sequence of moves to the following goal state:



- Generalizes easily to the $n^2 - 1$ puzzle ...

Example: $n^2 - 1$ Puzzle V

- PROBLEM. $n^2 - 1$ puzzle.
  - State = 2D array with values $1, 2, ..., n^2 - 1,$' '
  - Action = Swap space with adjacent number:
    - `N` = move space north
    - `S` = move space south
    - `E` = move space east
    - `W` = move space west
  - Goal state = 2D array with values in order of $1, 2, ..., n^2 - 1,$ ' '.
  - State space size = $(n^2)!$
- The state space size is humongus even for small n:
  - $n = 3$, state space size = $362,880$.
  - $n = 4$, state space size = $20,922,789,888,000$
  - $n = 5$, state space size $\approx 1.55 \times 10^{25}$

Example: $n^2 - 1$ Puzzle VI

- $n = 6$, state space size $\approx 3.71 \times 10^{41}$
- $n = 7$, state space size $\approx 6.08 \times 10^{62}$
- $n = 8$, state space size $\approx 1.27 \times 10^{89}$
- $n = 9$, state space size $\approx 5.80 \times 10^{120}$
- $n = 10$, state space size $\approx 9.33 \times 10^{157}$

The number of atoms in the visible universe is only 4×10^{79} !!!

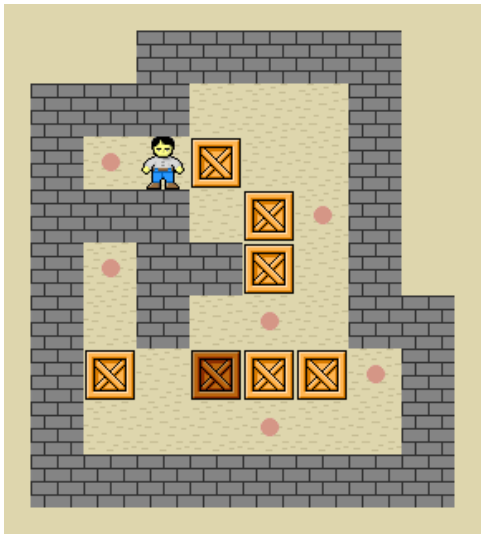
- In AI, typically the search space is astronomically large – brute force is not an option.

Example: $n^2 - 1$ Puzzle VII

- Exercise. It turns out that for each n, $1/2$ of the states can reach the goal state and $1/2$ cannot. In other words the state graph has two connected components where the goal state is in one of the connected component. Draw the complete state graph for $n = 2$ and you should see two connected components. Can you prove the general statement?
- This is an example of sliding block puzzles. Other examples:
  - Jigsaw puzzle: `https://en.wikipedia.org/wiki/Sliding_puzzle#/media/File:Batgirl.gif`
  - Sokoban (see following slides)

Example: Sokoban I

- Sokoban (warehouse keeper)



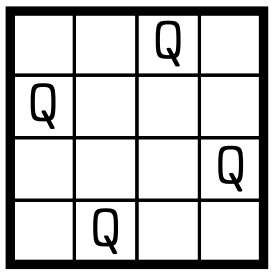
- Play: `http://sokoban.info/?1`
  - Player can push crate to adjacent empty square.
  - Goal is to place crates onto storage locations (marked with dot); crates on storage location has darker color.

Example: Sokoban II

- Info:
  `https://en.wikipedia.org/wiki/Sokoban#/media/File:Sokoban_ani.gif`
- Similar to $n^2 - 1$.
- Exercise. Using the above picture, design the states and the actions. Write a goal test function. For the time being, let the player move with the keyboard. By the end of this set of notes, you will be able to write a program to compute a solution. (See assignment.) What is an upper bound on the number of states for the above scenario (see picture above)?

n-Queens Problem I

- PROBLEM. n-queens problem: Given an n-by-n chess board, how can you place n queens so that none of them are attacking each other?
- Note that the above problem does not come with an initial state. You have to choose one.
- Example: For $n = 4$, the following is a solution:



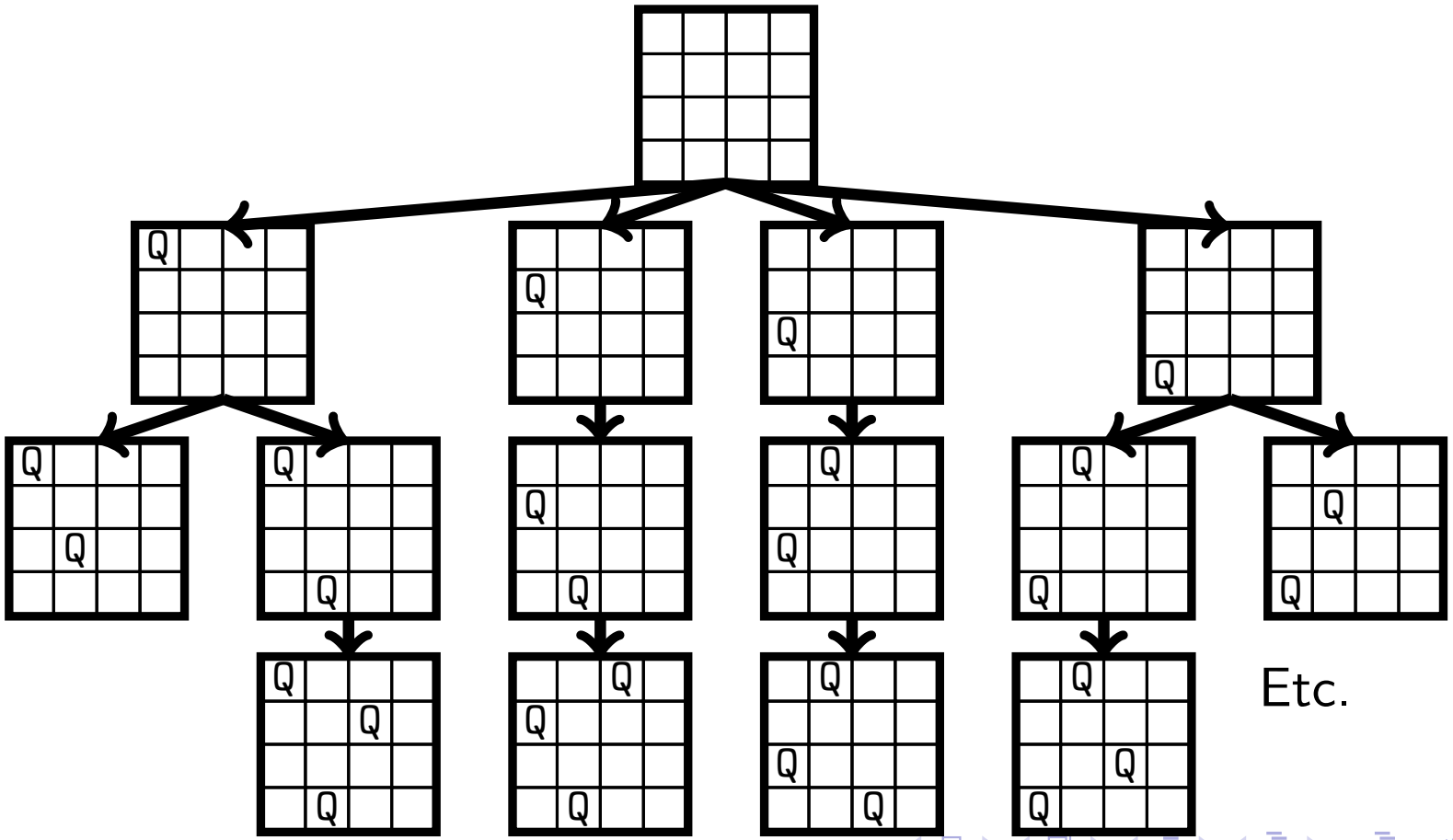
n-Queens Problem II

- I will show you
  - two different state graphs for finding solutions, and also briefly mention
  - solving this problem by viewing it as a **local search problem**. (See Local Search later.)

n–Queens Problem: First Solution I

- First method: Fill the board column-by-column
- Initial state: an empty n-by-n grid.
- You look at all the options for filling in the first column.
- For each option from above, look at all the options to fill the second column.
- For each option from above, look at all the options to fill the third column.
- Etc.

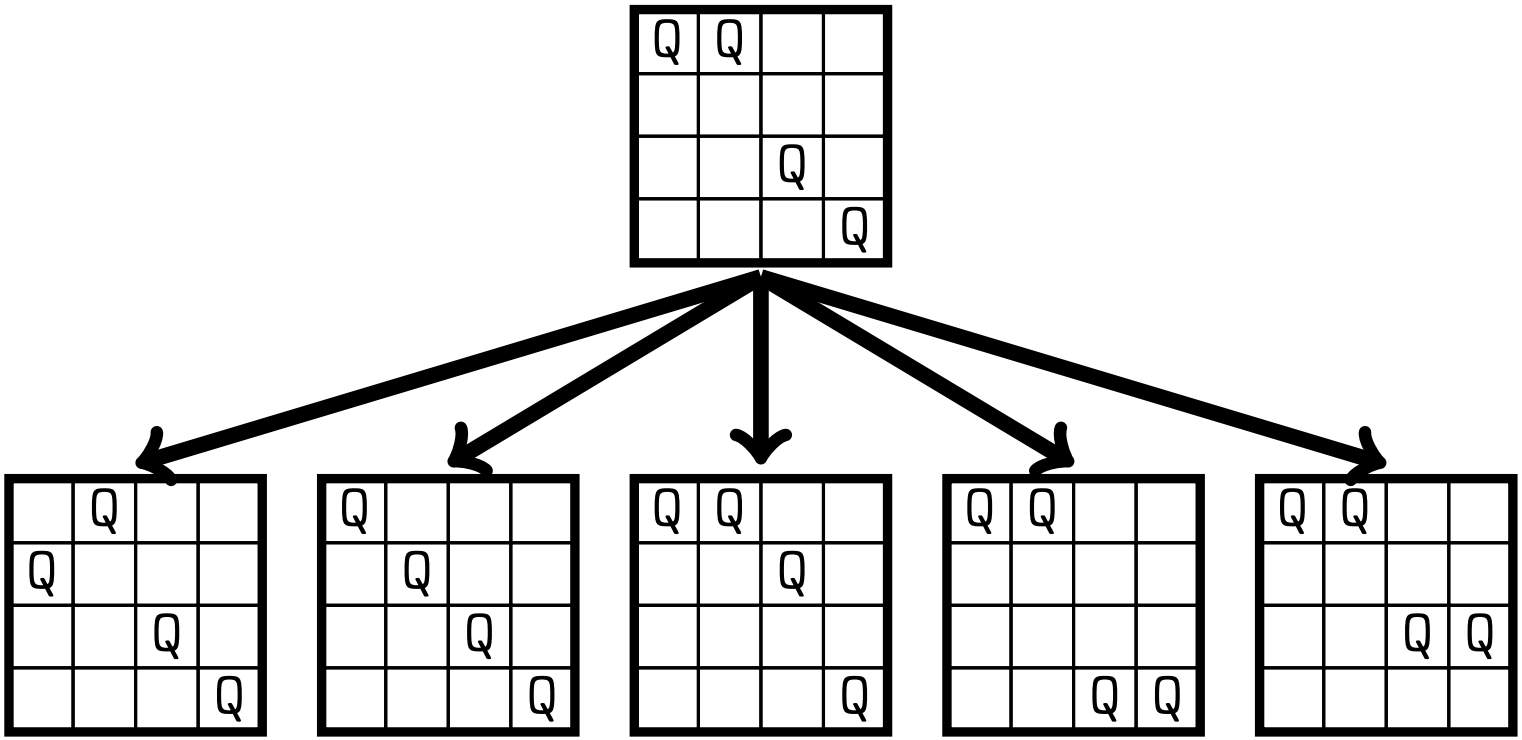
n–Queens Problem: First Solution II



n–Queens Problem: Second Solution I

- Second method: Make small adjustments for a queen
- Initial state: choose an n–by–n grid where for each column, a queen is placed randomly in any of the n rows. (You can for instance fill top row with queens.)
- Given a state, the actions are: choose a column and move the queen in that column up or down by one square.

n–Queens Problem: Second Solution II



- Perform search on the above state graph.

n–Queens Problem: Second Solution III

- POINT: It is possible to have two totally different state graphs for the same problem.
- Note: This state graph is *not* a directed acyclic graph.
- Exercise. Compare the state space size of the above two methods.
- Exercise. Find one solution to the n-queens problem for $n = 4$ using this method.

n-Queens Problem: Third Solution (local search) I

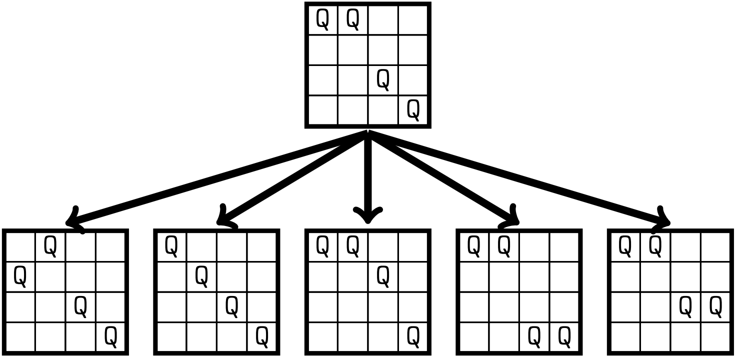
- For some n, it is *impossible* to solve the n–queens problem. (Why?) You can change the problem to ...
- PROBLEM: Find an n–queens board with *least* number of attacking pairs.
- This becomes an **optimization problem**: Given a state space, find state(s) that minimizes or maximizes a particular function of the states.
- This type of discrete optimization problem can be (more or less) solved using local search.

n-Queens Problem: Third Solution (local search) II

- One local search solution:
  - Use the same graph as the second solution above.
  - Start with randomly placed queens one queen per column and compute number of attacking pairs. Let s be this state.
  - For the above state, look at all resulting states and compute their number of attacking pairs.
  - Choose the resulting state with the best (i.e., least) number of attacking pairs and call it s.
  - Repeat until number of attacking pairs cannot be decreased.
  - If there are two resulting states with the same number of attacking pairs – randomly pick between the two.

n-Queens Problem: Third Solution (local search) III

- In earlier example, how many attacking pairs are there at the top state? Which state should you go to?



n-Queens Problem: Third Solution (local search) IV

- Exercise. Find at least one solution to the n queens problem for $n = 5$. Start with a random queens placement. If you're stuck (i.e., you cannot move to a board with fewer attacking pairs, restart the algorithm with another random queens placement.)
- There are many other local search techniques. See notes on local search (later).
- Local search is a very important area of study in AI with lots of applications:
  - Job scheduling in automated manufacturing (x machines, y jobs)
  - Traffic routing
  - Telecommunication

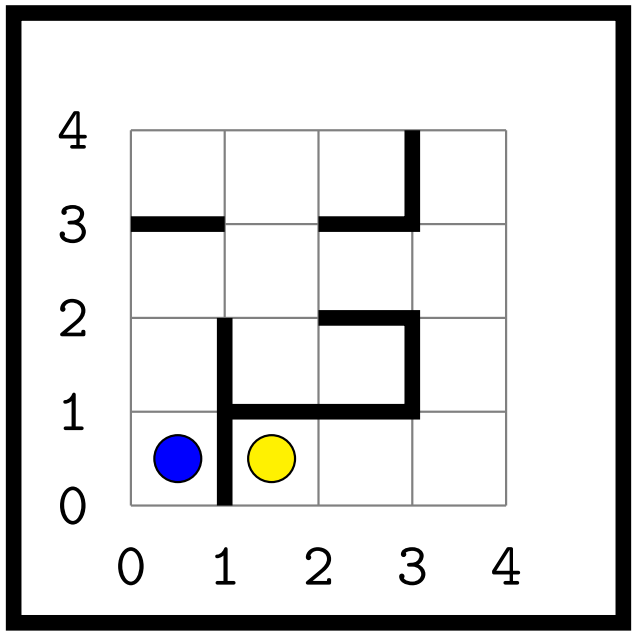
n-Queens Problem: Third Solution (local search) V

- VLSI layout
- Etc.

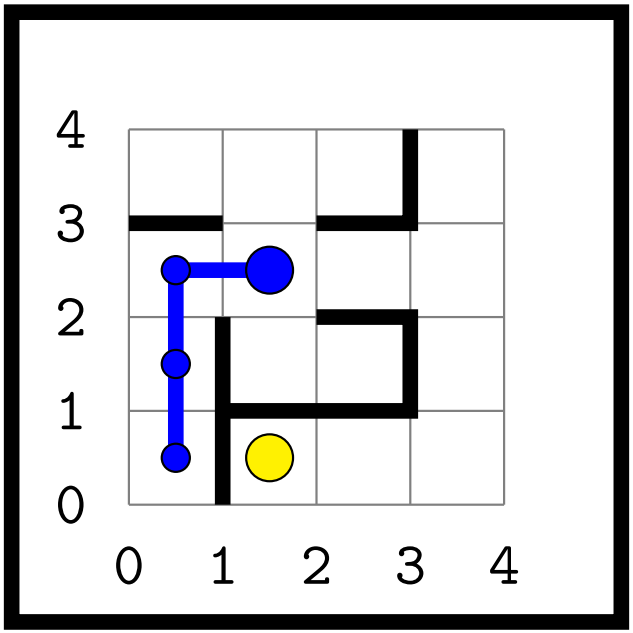
- There are many variations of the n–queens problem: n–bishops, n–knights, etc.

Example: Maze I

- Robot starts at blue dot. Goal = reach gold dot.

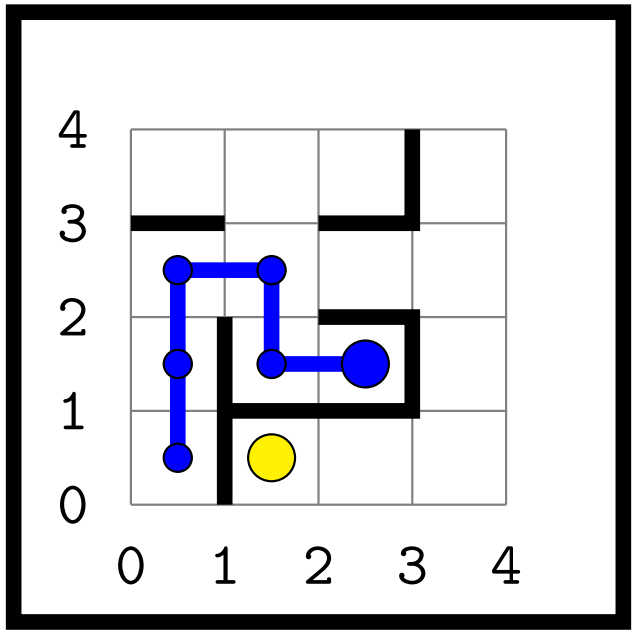


Example: Maze II



Path: (0.5,0.5), (0.5,1.5),(0.5,2.5),(1.5,2.5)

Example: Maze III

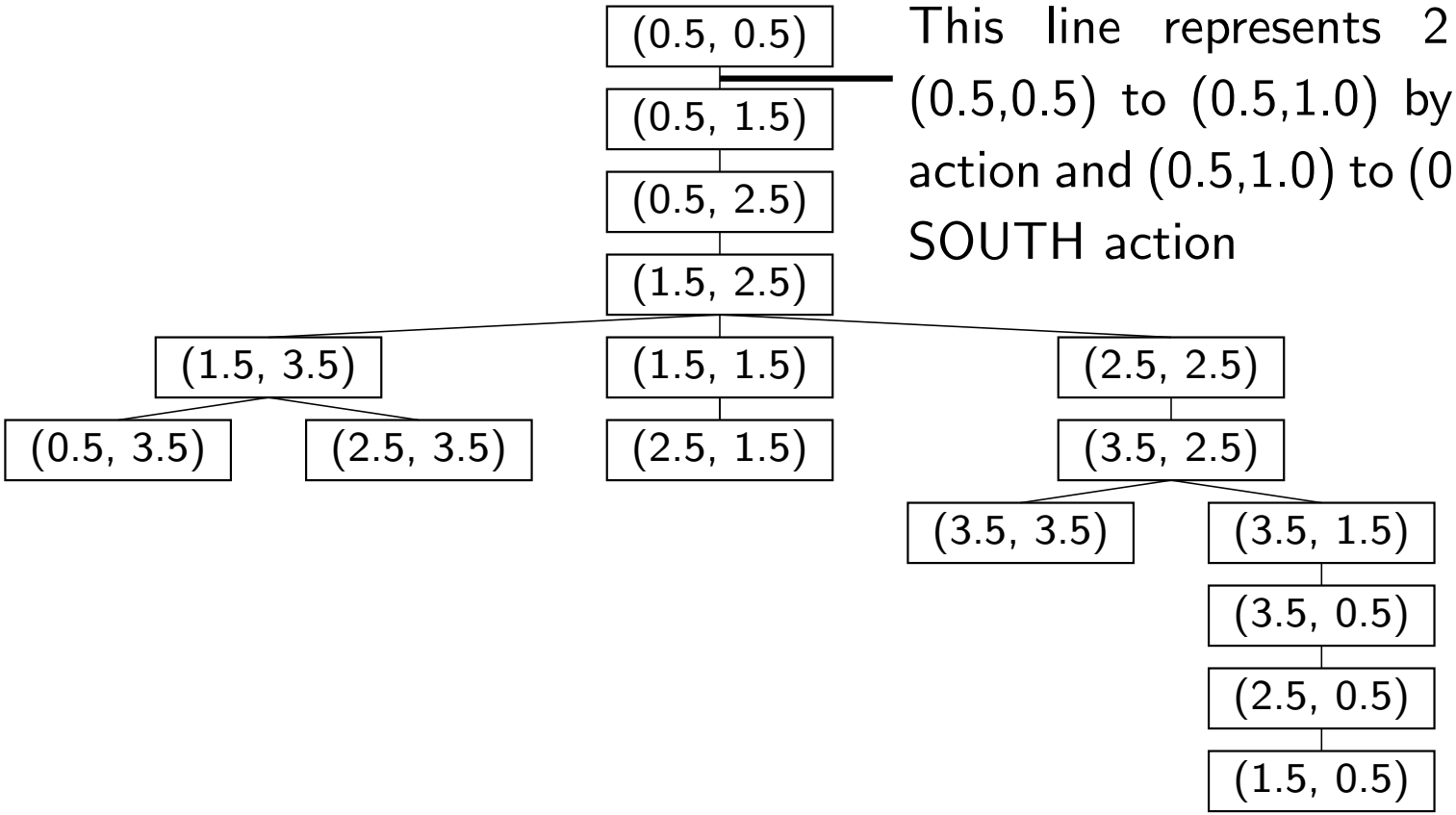


We might have to go back and try another option.

Example: Maze IV

- The path from the starting point (blue dot) to a square with yellow dot, is a path in the following graph of states where each state is the location of the robot.

Example: Maze V

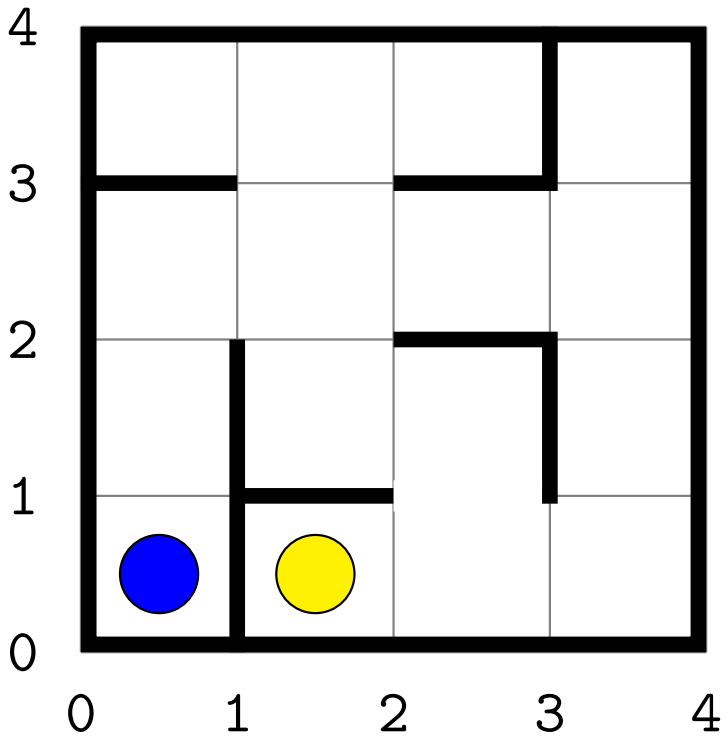


This line represents 2 actions: (0.5,0.5) to (0.5,1.0) by NORTH action and (0.5,1.0) to (0.5,0.5) by SOUTH action

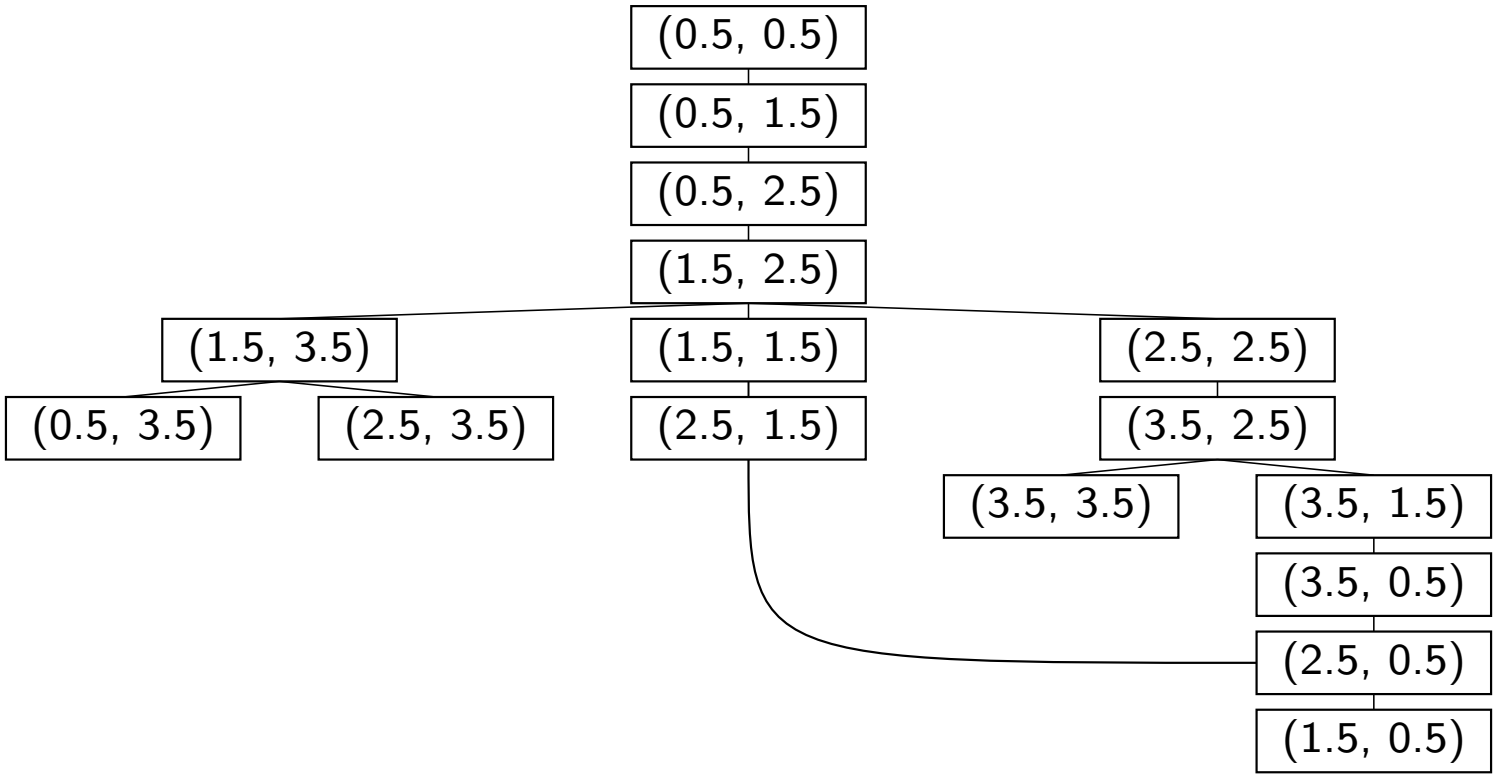
Example: Maze VI

- Things can be a lot more complicated ...

Example: Maze VII



Example: Maze VIII

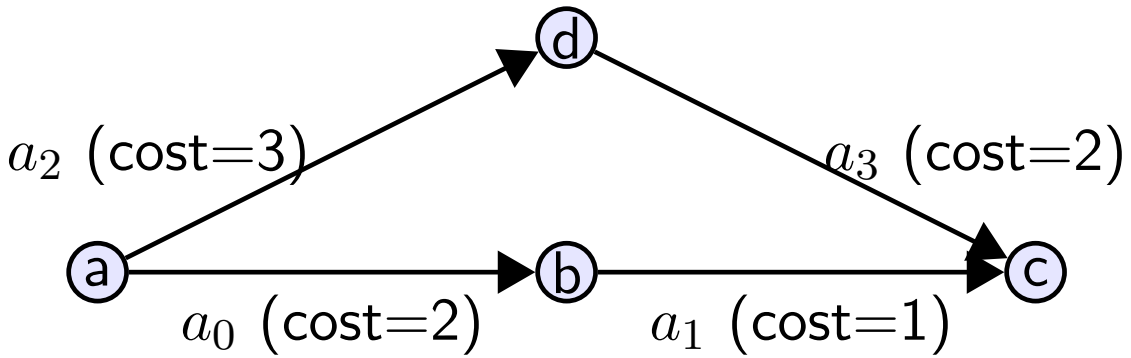


Example: Maze IX

- Note that in the above case where there's a cycle, there are two possible paths: one requires 7 steps, another requires 9 steps.
- We can also talk about **path cost**
- Example: In a different maze,
  - The distance between each cell of the maze is not always 1. Therefore the action of going from one cell to another has a different cost.
  - One of the rooms has a spell. The cost of entering this room is higher than entering regular rooms.
  - Etc.
- The cost of a single edge/action is also called **step cost** or **action cost** or **arc cost** or **edge cost**

Example: Maze X

- **path cost** of a path = the sum of costs of actions along the given path.
- A solution is **optimal** if the path cost is lowest along all solutions. Example:



You can get from state a to c in 2 ways: using actions a_0, a_1 or actions a_2, a_3. First path is cheaper.

Example: Maze XI

- In the above maze, we assume each step has a cost of 1. In general each action might have different cost. General practice: if step cost is not stated, assume it is 1.

Example: Maze XII

- PROBLEM. Maze problem (m-by-n).
  - States = (row, column) where 0 $\leq$ row $\leq m - 1$, 0 $\leq$ column $\leq n - 1$,
  - Action = Move to unobstructed adjacent cell
  - Goal state: a fixed target (row, column) for robot to reach
  - State space size = mn (assuming all cells are reachable from one another)

Real-world problems: airline travel problem I

- Airline travel planning problem:
  - User go to a travel website, key in source airport, destination airport, and date. System finds a travel plan and cost.
  - SABRE (Semi-automated Business Research Environment) `https://en.wikipedia.org/wiki/Sabre_(computer_system)`
  - An airplane flies through a number of airports (flight segments).
  - An airplane has a waiting time at each airport.
  - Buying adjacent flight segments is cheaper.
  - Etc.! Etc.! Etc.!
  - The goal is to find a flight plan that is cheapest, minimal but sufficient waiting time between flight segments, etc.

Real-world problems: airline travel problem II

- Very complex problem.
- Design a state space graph for this problem.

Real-world problems: touring problem I

- Touring problems.
  - Given the map of USA, find the shortest route from Columbia,MO to San Francisco,CA that goes through
    - Lava Hot Springs, ID
    - Denver,CO
    - Yuma,CO
  - Given the map of USA, find the shortest route that goes through ALL cities (!!!)
  - Given a map, find the shortest route that visits every city exactly once. (Traveling Salesman Problem - TSP)
  - Many applications of the above route planning problems.
  - We will come back to these problems in local search.

Real-world problems: robot motion planning I

- Robot motion planning.
  - Not just get from point A to point B
    - arm and gripper in a specific position to grab something
    - parts (arms, legs, head/camera, etc.) of the robot negotiate obstacles
    - on uneven ground, robot maintains stability
    - if robot has legs (instead of tracks), need to handle balance
    - errors: moving left and right tracks by the same distance does not mean the direction of motion is straight ahead because of slips, friction, etc.
  - Note that the 3D physical space is *continuous*.
  - Lots of geometry.
  - Also used in self-driving cars

Real-world problems: robot motion planning II

- Gazebo Simulator for DARPA Virtual Robotics Challenge: `https://www.youtube.com/watch?v=yVICMC_BAiU`
- Robot Motion Planning for Learned Tasks: `https://www.youtube.com/watch?v=eUf1JUAQ074`

Problem-Solving Agents I

- In the above examples, we assume that there's a well-defined problem with state space, actions, goal test, etc. and how solving the problem is the same as searching in the state space.
- What is the relationship between these set of notes and the previous set of notes on agents? What is the big picture?

```
ALGORITHM:  SIMPLE-PROBLEM-SOLVING-AGENT:
INPUT:      percept
OUTPUT:     action
PERSISTENT: seq:     an action sequence, initially empty
            state:   current world state
            goal:    agent's goal, initially null
            problem: a problem formulation

state = UPDATE-STATE(state, percept)
if seq is empty:
    goal = FORMULATE-GOAL(state)
    problem = FORMULATE-PROBLEM(state, goal)
    seq = SEARCH(problem)
    if seq == FAILURE:
        return null action
action = first of actions in seq
seq = seq with first removed
return action
```

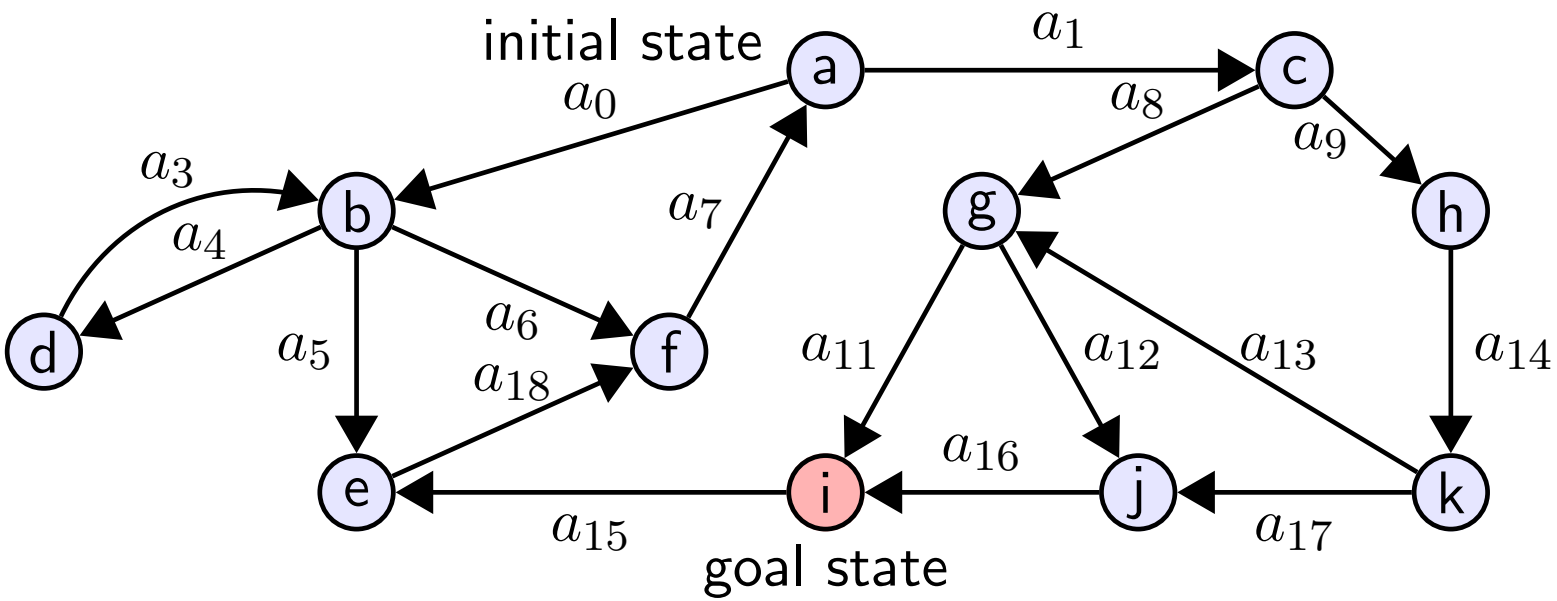
Problem-Solving Agents I

- The focus of next few sets of notes is on finding a solution (a sequence of actions).
- You will have to identify the problem which includes design of state space.
- In AI, at the next level, it is possible for the agent to understand the current state of the world, formulate a goal, a problem, and then search for actions to move toward the goal.

General idea I

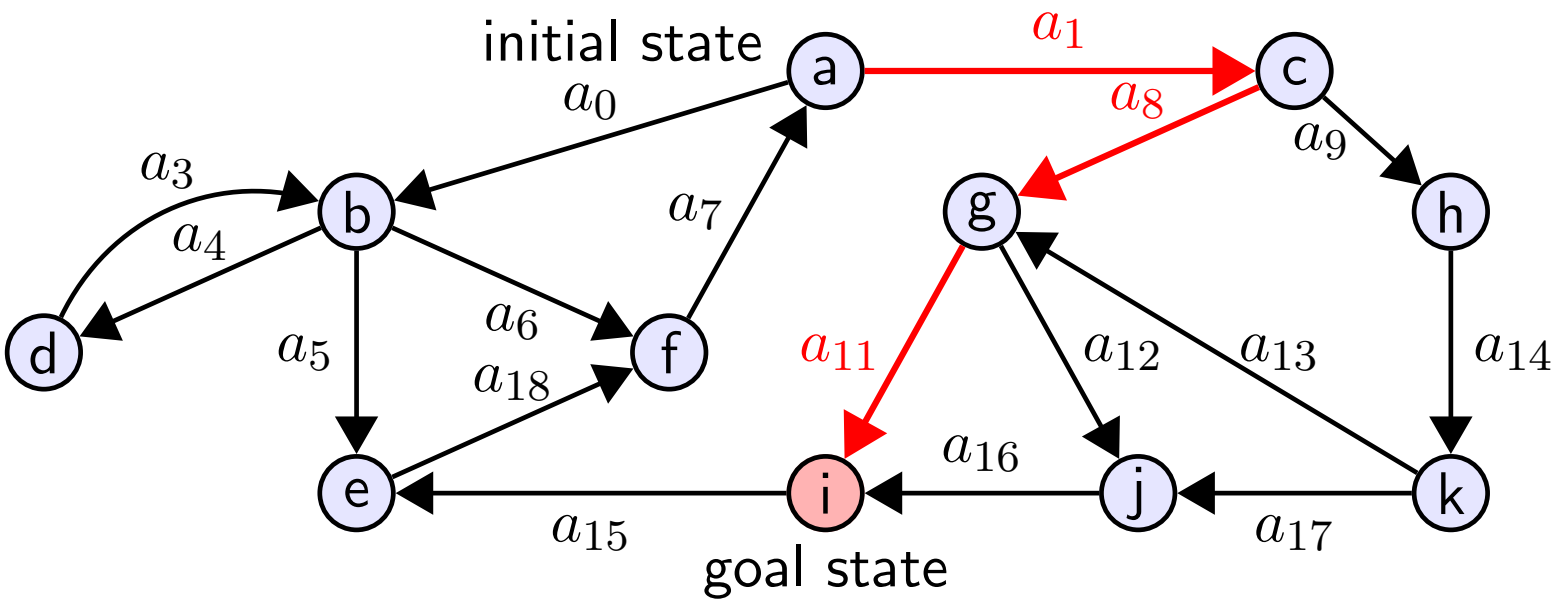
- What's common among the problems above?
- Fully observable, deterministic, goal-based.
- Clearly this is similar to search for a path in the graph of states-and-actions from the initial state to some goal state(s).
- Here's a picture for now:

General idea II



At state (of the environment) a, if the agent applies action a_1, the state becomes c.

General idea III



One solution from a to i: $[a_1,a_8,a_{11}]$. I.e., starting at a, apply actions a_1 then a_8 then a_{11} to arrive at i.

General idea IV

- Here's the general idea. Using the maze as an example:

```
You start at one state (the given initial state)
You look around you and see where you can go.
You make a move.
You look around you and see where you can go.
You make a move.
Etc.
```

- The above is too vague and incomplete to be an algorithm.
  - At a fork where you can act to go to two different states, what if you choose the wrong action?
  - What if you go into an infinite loop of repeating states?
  - Etc.

General idea V

- Here are some factors to consider. Right now focus on general ideas – big picture first. Details later.

General idea VI

- **Fringe**: A list of options to try.
  - At a state in the state graph, you might have more than one action available. You might choose the wrong action, ending in a dead-end. At that point, you need to go back and try an earlier option. The fringe records options to try out.

  Also called **open list**, **frontier**.
- **Closed list**: A list of states that you do not want to look at (example: you have already looked at them earlier).
  - If you're not careful, you might go to a state that you have looked at earlier. You might end up going into an infinite loop. The closed list prevents this from happening.

  Also called **explored set**.

General idea VII

- **Parent state** and **parent action** of each state: To remember how you arrived at this state
  - When you arrive at a goal state, you need to write down the actions from the initial state to this goal state. One way is to note down how to get to the parent state and the action from that parent state to the current state.
- **Path cost** of each state:
  - Suppose each action has a cost involved. Then we want to find a solution with the least cost – the **optimal solution**. the cost of a solution is the sum of the cost of all actions on the path. In that case, for each state, we will also need to record the cost from the initial state to that state. If the goal is just to find any solution, then this is not necessary.

General idea VIII

- **Generate-and-test**: The agent should only generate states (in its memory) when needed.
  - For instance, for the graph diagram above, if the agent sees that the state (of the environment) is a, it seems that there are two applicable actions a_0 and a_1 that result in states b and c, the agent only need to know about three states at this point in time.
  - This is different from searching for a value within a graph where all nodes are present. For instance in CISS350 you can traverse (using BFS, DFS, ...) a tree to find a value where the tree already has all the nodes.

```
INFORMAL GRAPH SEARCH (EARLY GOAL TEST VERSION)
if initial state is goal state: return NO ACTION NEEDED!

let closed list be empty
let fringe have only the initial state

while fringe is not empty:
    take an option, s, out of fringe
    put s into closed list
    expand s to state s0 with action a0, s1 with a1, ...
    if s0 not in fringe and not in closed list:
        if s0 is a goal state:
            walk parent pointers and record parent actions
            return the reverse of the record
        point s0 back to parent s and mark with parent action a0
        put s0 into the fringe
    repeat above for s1,a1 and s2,a2, ...
return ERROR - NO SOLUTION
```

General idea I

- The above version is called the **early goal test version**: you goal test a state as soon as possible – goal test the initial right away and goal test a state once it appears in an expansion, i.e., before you put it into the fringe.
- The next version is called the **late goal test version**. You goal test a state after you fetch it from the fringe.
- Why do goal test late?
  - An option added to the fringe might allow the discovery of a cheaper solution (i.e., lower path cost). If you goal test too soon, you might miss the better solution.
  - Goal test might be expensive

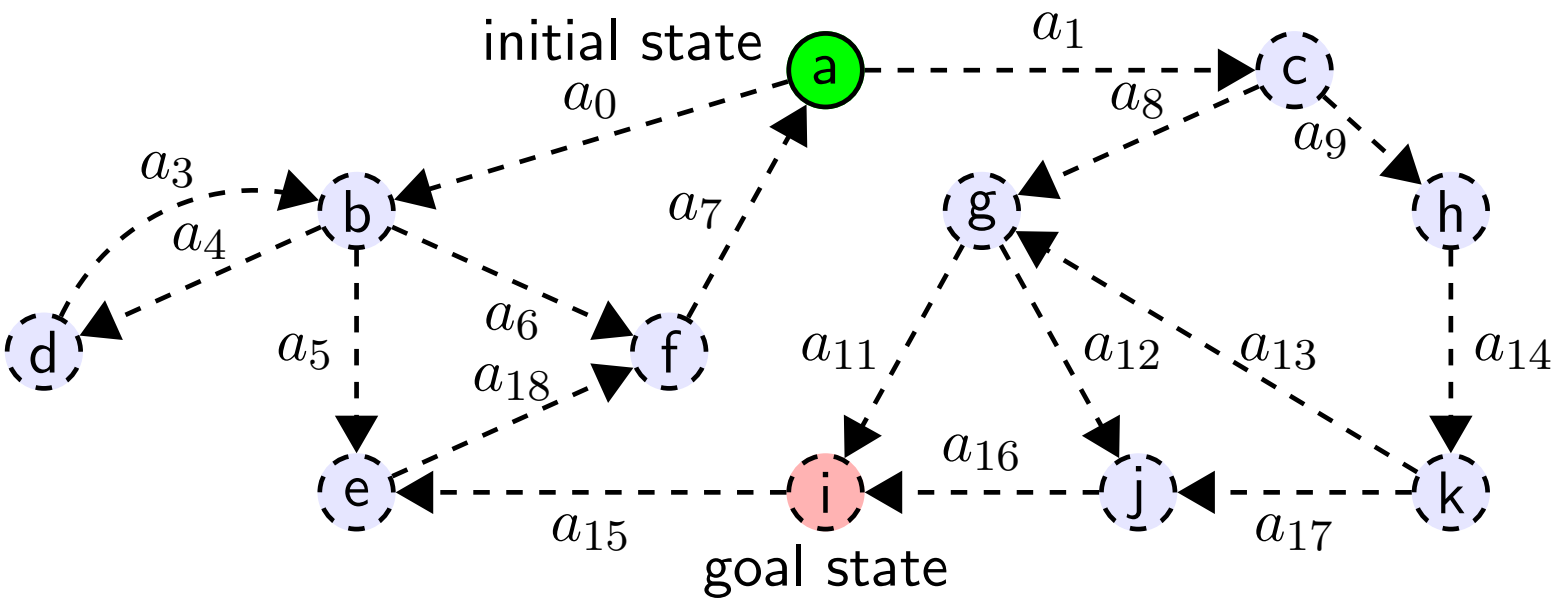
```
INFORMAL GRAPH SEARCH (LATE GOAL TEST VERSION)
let closed list be empty
let fringe have only the initial state

while fringe is not empty:
    take an option, s, out of fringe
    if s is a goal state:
        walk parent pointers and record parent actions
        return the reverse of the record
    else:
        add s to closed list
        expand s to state s0 with action a0, s1 with a1, ...
        if s0 is not in fringe and not in closed list:
            point s0 back to parent s and mark with parent
              action a0
            put s0 into fringe
        repeat above for s1,a1 and s2,a2, ...
print ERROR - NO SOLUTION
```

Visual trace (conceptual view) I

- The following is a trace of the early goal test version:
  - States: $a, b, c, ..., k$
  - Actions: $a_0, ..., a_{17}$
  - Blue arcs point from s to it's parent state. The action from the parent state to s is also noted (in blue).
  - Fringe: states that can be considered in the search are in green.
  - Closed list: states that should not be explored are in red.

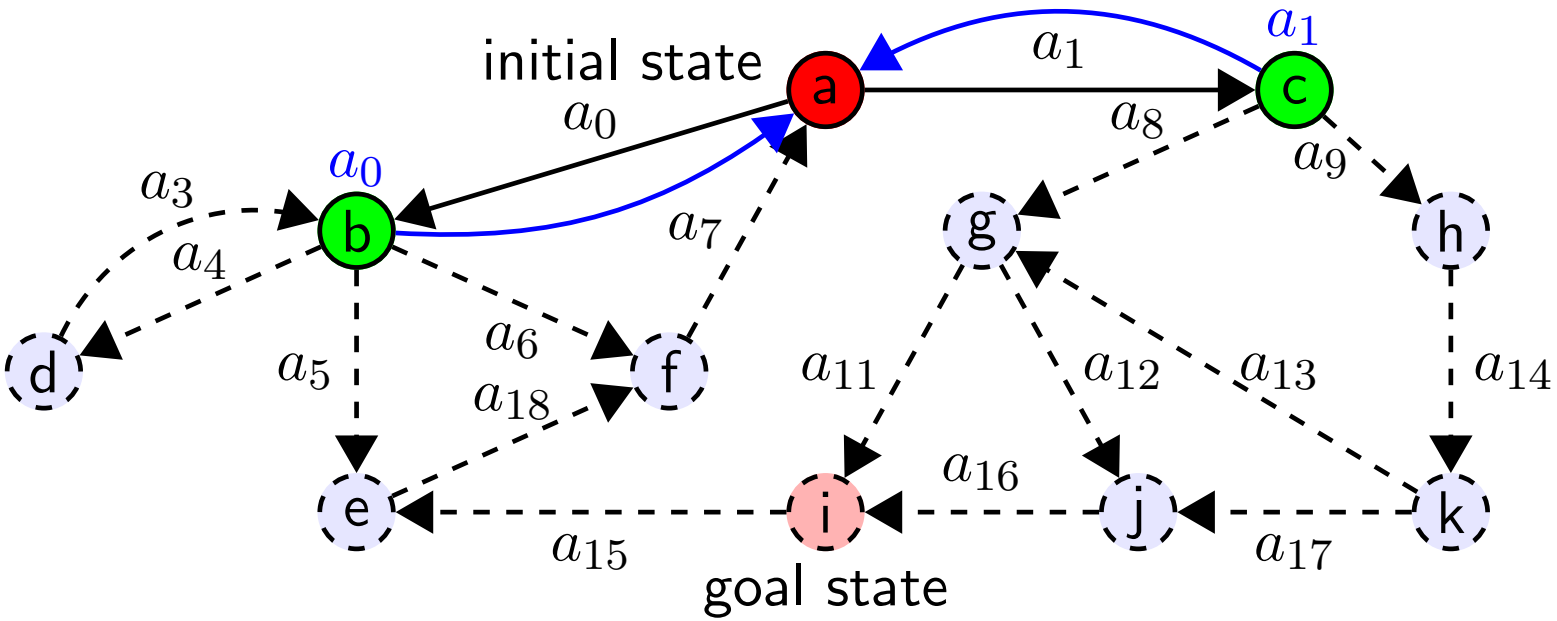
Visual trace (conceptual view) II



Initialization: a is in the fringe. The other states are not generated (into memory) yet.

fringe $= a$; closed $=$

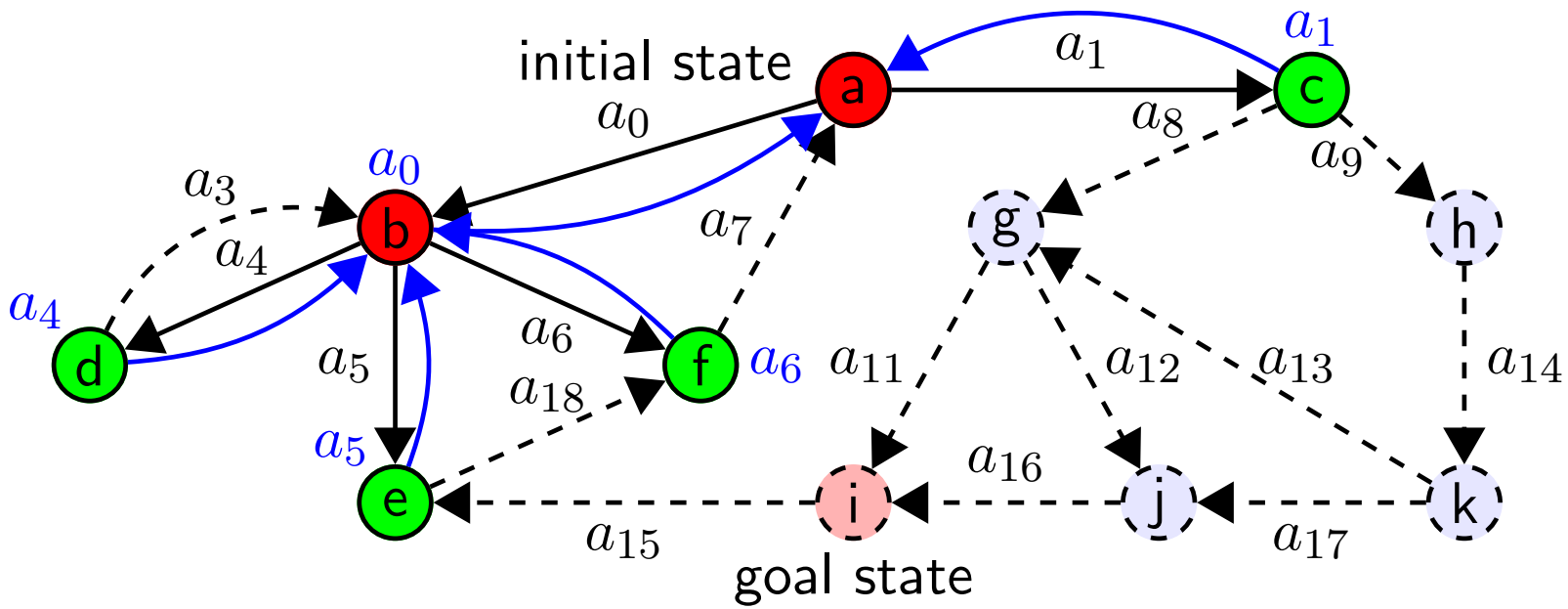
Visual trace (conceptual view) III



Remove search node of a from fringe. Put a into closed list. Expand a to get b,c. b, c are not goal states, not in fringe, not in closed list – put them into fringe.

fringe $= b, c$; closed $= a$

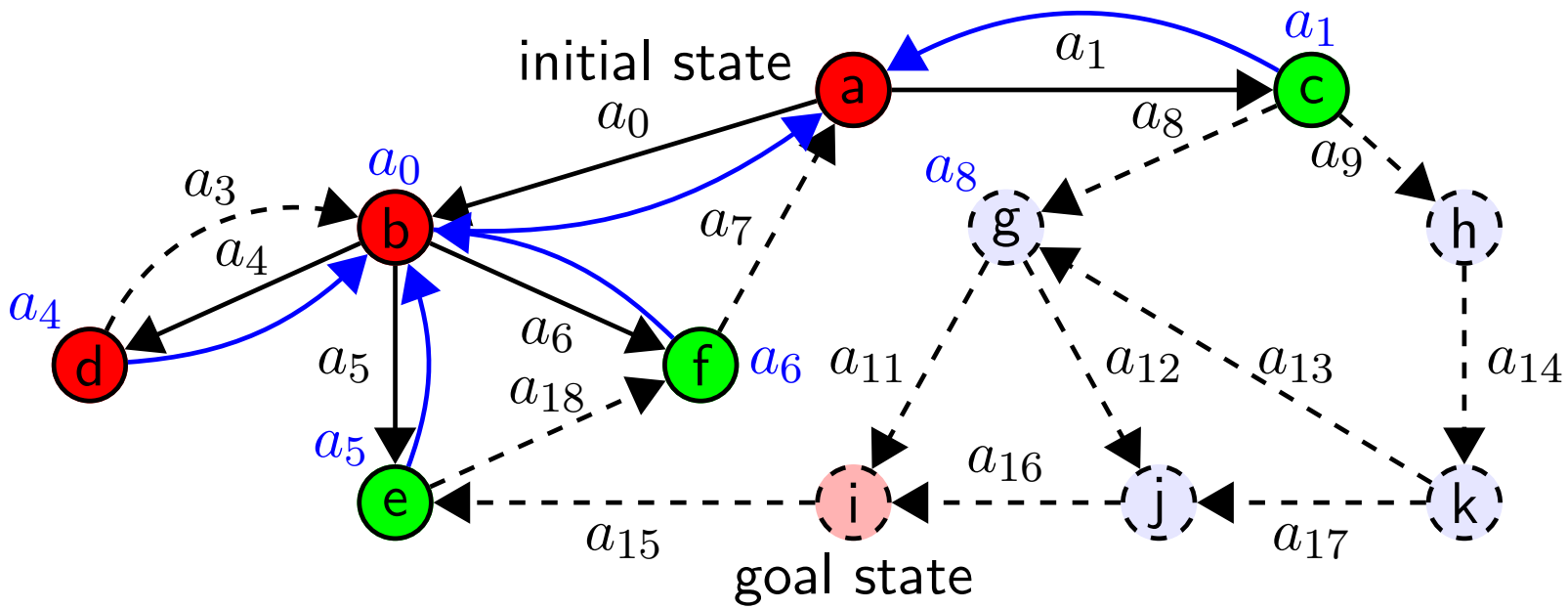
Visual trace (conceptual view) IV



Remove an option from fringe – say b. Put b in closed list. Expand b to get d,e,f – d, e, f are not goal state, not in closed list, not in fringe. Put d,e,f into fringe.

fringe $= d, e, f, c$; closed $= a, b$

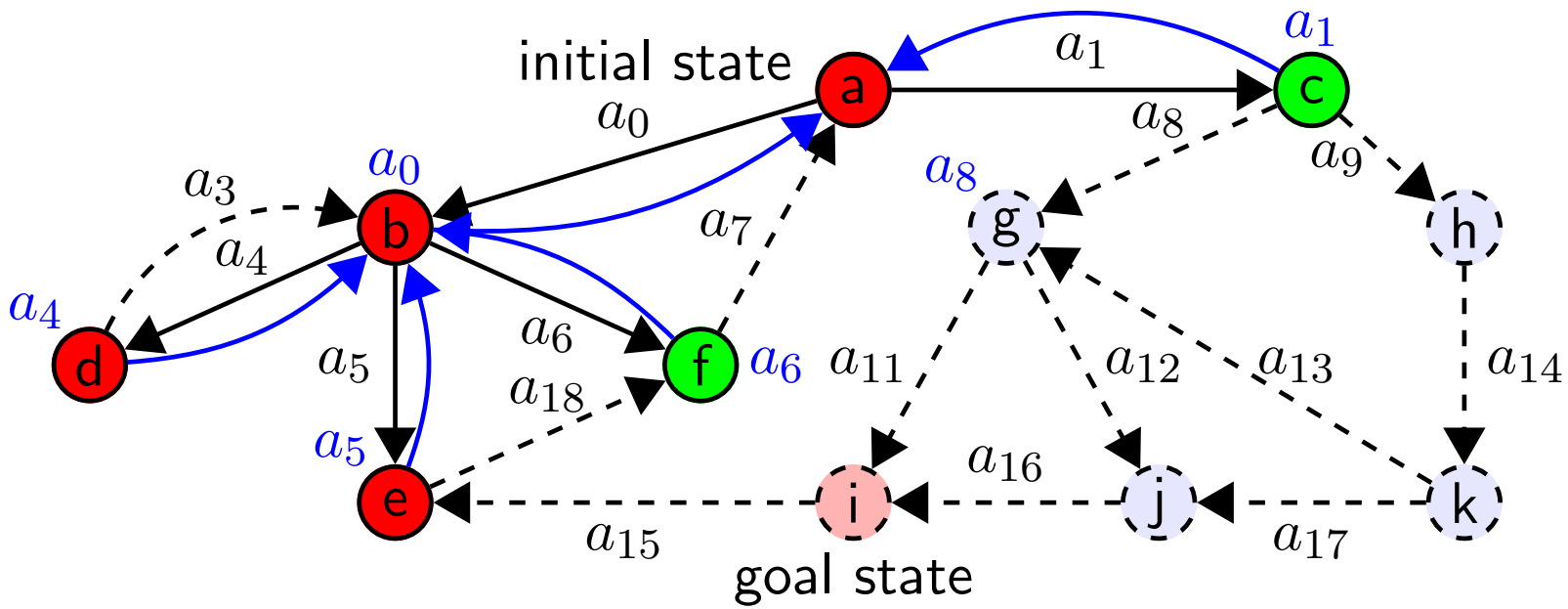
Visual trace (conceptual view) V



Remove option from fringe – say d. Put d into closed list. Expand d to get b. b is in the closed list so ignore it.

fringe $= e, f, c$; closed $= a, b, d$

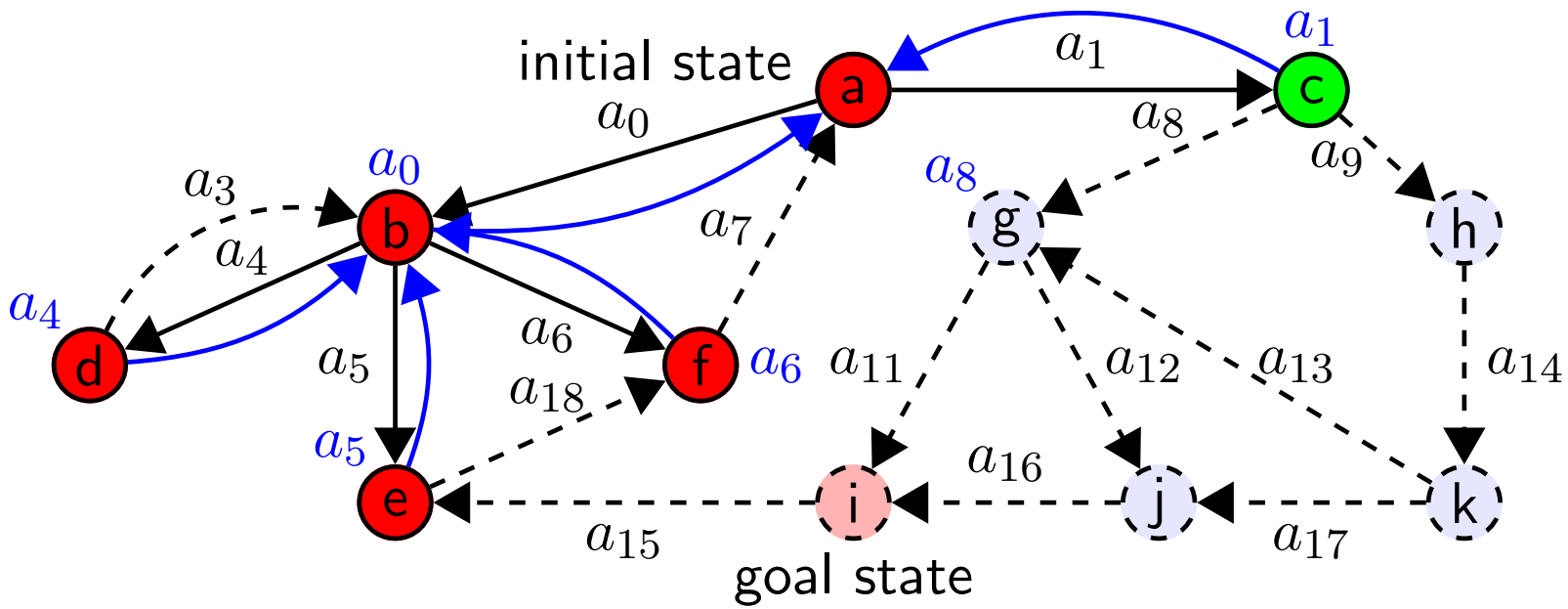
Visual trace (conceptual view) VI



Remove an option from fringe – say e. Put e into closed list. e expands to f. f is already in fringe – ignore.

fringe $= f, c$; closed $= a, b, d, e$

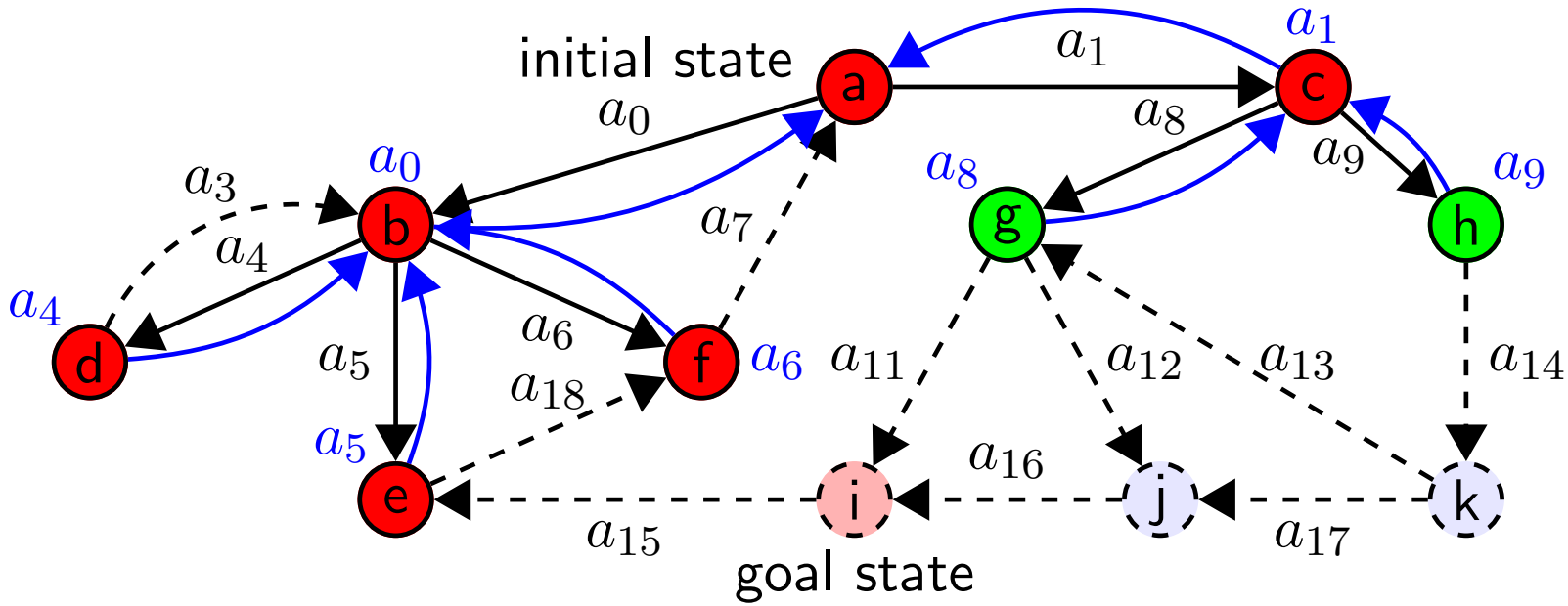
Visual trace (conceptual view) VII



Remove an option from fringe – say f. Put f in the closed list. Expand f to get a. a is in the closed list so ignore it.

fringe $= c$; closed $= a, b, c, e, f$

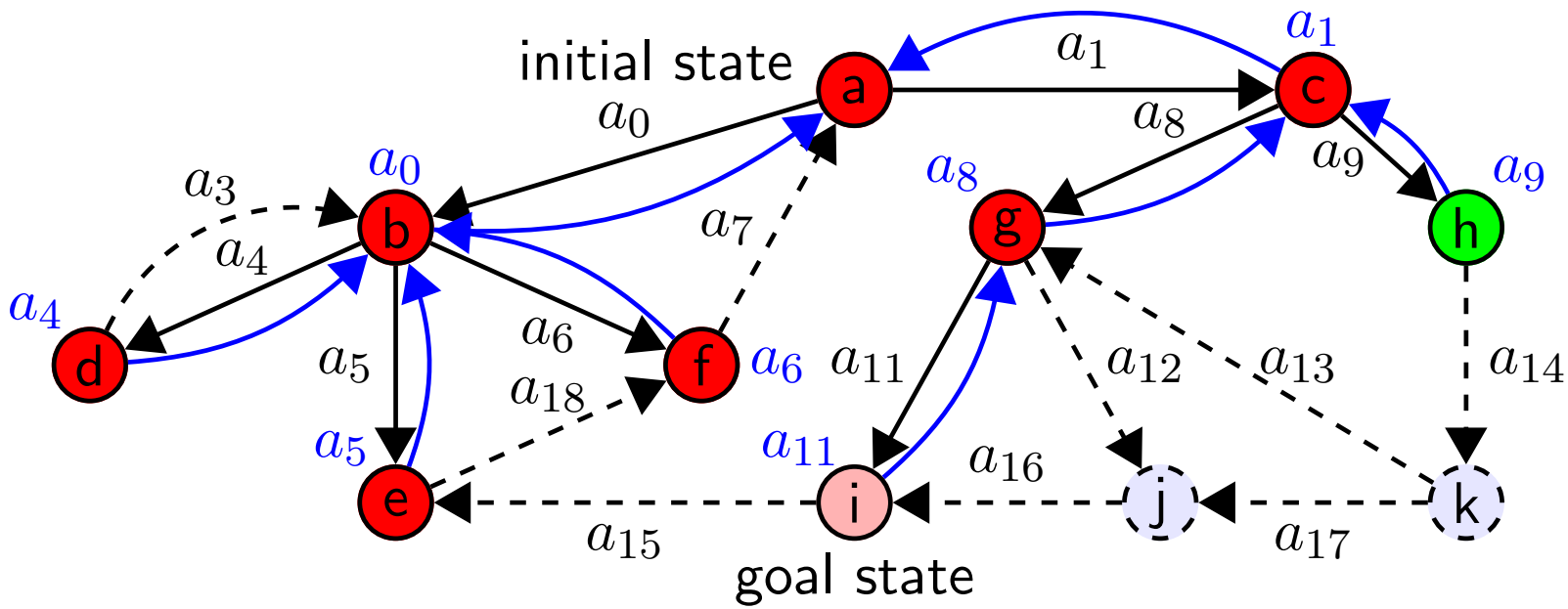
Visual trace (conceptual view) VIII



Remove an option from fringe – there's only one, c. Put c into closed list. Expand c to get g, h – both not in closed list and not in fringe. Put g, h into fringe.

fringe = g,h; closed = a,b,c,e,f,c

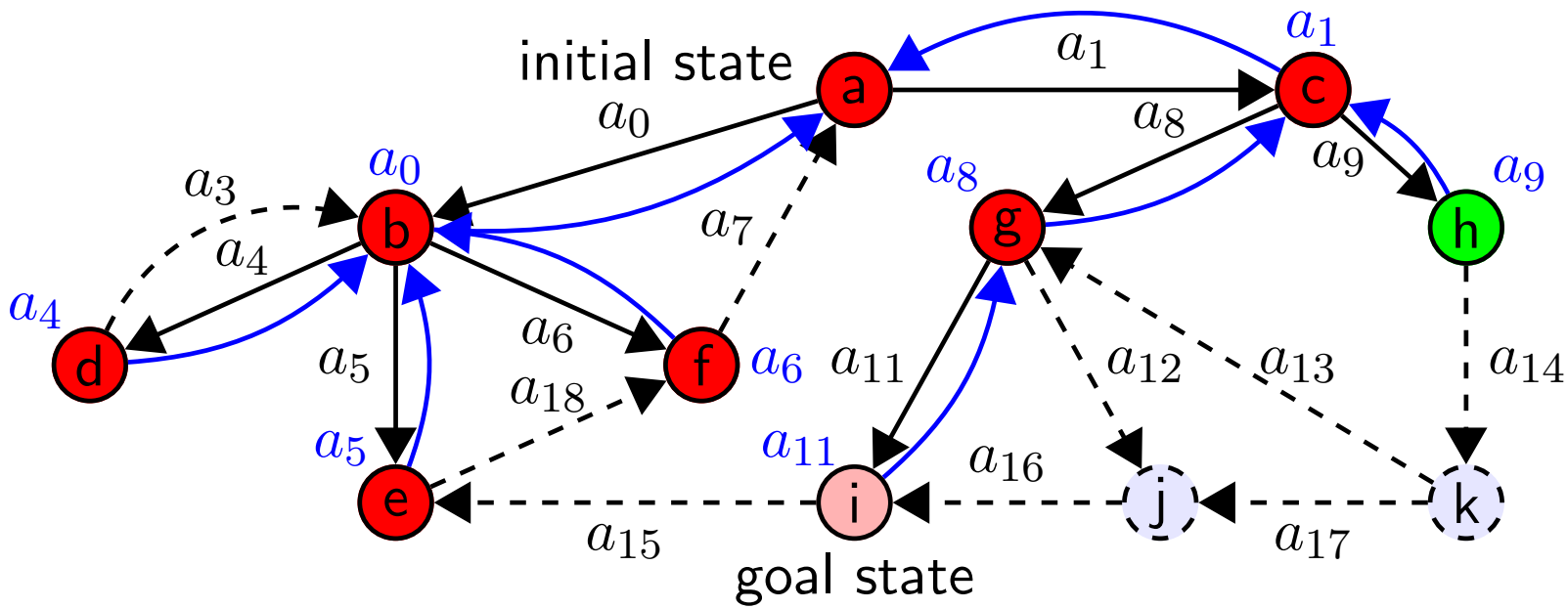
Visual trace (conceptual view) IX



Remove an option from fringe – say g. Put g into closed list. g expands to i (say i is expanded before j). i is a goal state!

fringe $= h$; closed $= a, b, c, e, f, c$

Visual trace (conceptual view) X



At i, get a_{11}, follow parent pointer to g, get a_8. Follow parent pointer to c, get a_1. Follow parent pointer to a, initial state. Get a_{11}, a_8, a_1 – reverse to get solution a_1, a_8, a_{11}.

fringe $= h$; closed $= a, b, c, e, f, c$

Visual trace (conceptual view) XI

- NOTE: The progression of the search depends on
  - Ordering of values taken out of fringe – i.e., the container organization of fringe (stack? queue? etc.)
  - Ordering of the expanded states (or actions) for a state.
    - Example: in the above, the expansion of g gives us i and j. We assumeed that i appears first. If j appears first, then j would go into fringe.

  (In class work – assignments, etc. – make sure you read and follow the instructions carefully.)
- In the pseudocode, I put the state s into the closed list at the beginning. What if I do that after processing all it's expanded states?

State space I

- For the algorithm/pseudocode/software view, we need to be precise and fix the following function/method names ...
- Recall that from a problem, we create the following
  - a graph of states and actions
  - there is an initial state
  - applying an action to a state gives you a new state
  - you can test if a state is a goal state
- `goal_test`: For a state `s`, `goal_test(s)` return true if `s` is a success state.
- `result`: For each state `s` and an action `a` available at `s`, we will write `result(s, a)` for the resulting state when we apply action `a` to `s`. WARNING: Parameter `s` is not changed.

State space II

- `step_cost` or `cost` : For each state `s` and an action `a` available at `s`, we will write `step_cost(s, a)` for the cost of the action `a` at `s`. By fault this is 1. For simplicity, I will sometimes write `cost(s, a)` or `c(s, a)`.
- `actions`: For each state `s`, `actions(s)` returns a list of available actions for `s`.
  - WARNING: The ordering of the actions will have an impact on the progress of the search.
- `successors`: For each state `s`, `successors(s)` returns a list of (a', s') where applying action a' to s gives the resulting state s'.

State space III

- There are other functions/methods which are (obviously) needed. For instance, you can compare states, print states, etc.
- For convenience, we will pack all these into a problem object. Therefore if `p` is an object from the `Problem` class, then we have
  - `p.get_initial_state()` will return the initial state for this problem
  - `p.goal_test(s)`
  - `p.result(s, a)`
  - `p.actions(s)`
  - `p.step_cost(s, a)` or `p.cost(s, a)` (or even `p.c(s, a)`)
  - `p.successors(s)`

State space IV

where `s` is a state and `a` is an action

- It's also a good idea to have a `State` class.
- NOTE: For assignments, the order of the actions in `p.actions(s)` is very important. Make sure you read the assignment in detail.)

State space V

- Instead of putting all the above into the `Problem` class, you can also put some of the methods in the `State` class. For instance if `s` is a state,
  - `s.result(a)`: resulting state after apply `a` to `s`
  - `s.actions()`: all actions available for `s`
  - `s.step_cost(a)` or `s.cost(a)`: step cost.
  - `s.successors()`: all (action, successor) pairs of `s`.

  and the rest in `Problem` class:
  - `p.get_initial_state()`
  - `p.goal_test(s)`

State space VI

- **Transition model** = Given a state and an action, what is the resulting state.
  - Our state graph is sometimes called the **state and transition model**. I prefer state graph or state space.
- Exercise: Write a class `SlidingBlockProblem` such that `SlidingBlockProblem(m)` gives you an $n^2 - 1$ problem where `m` is a 2D array of values 1, 2, 3, ..., $n^2 - 1$ and a space.

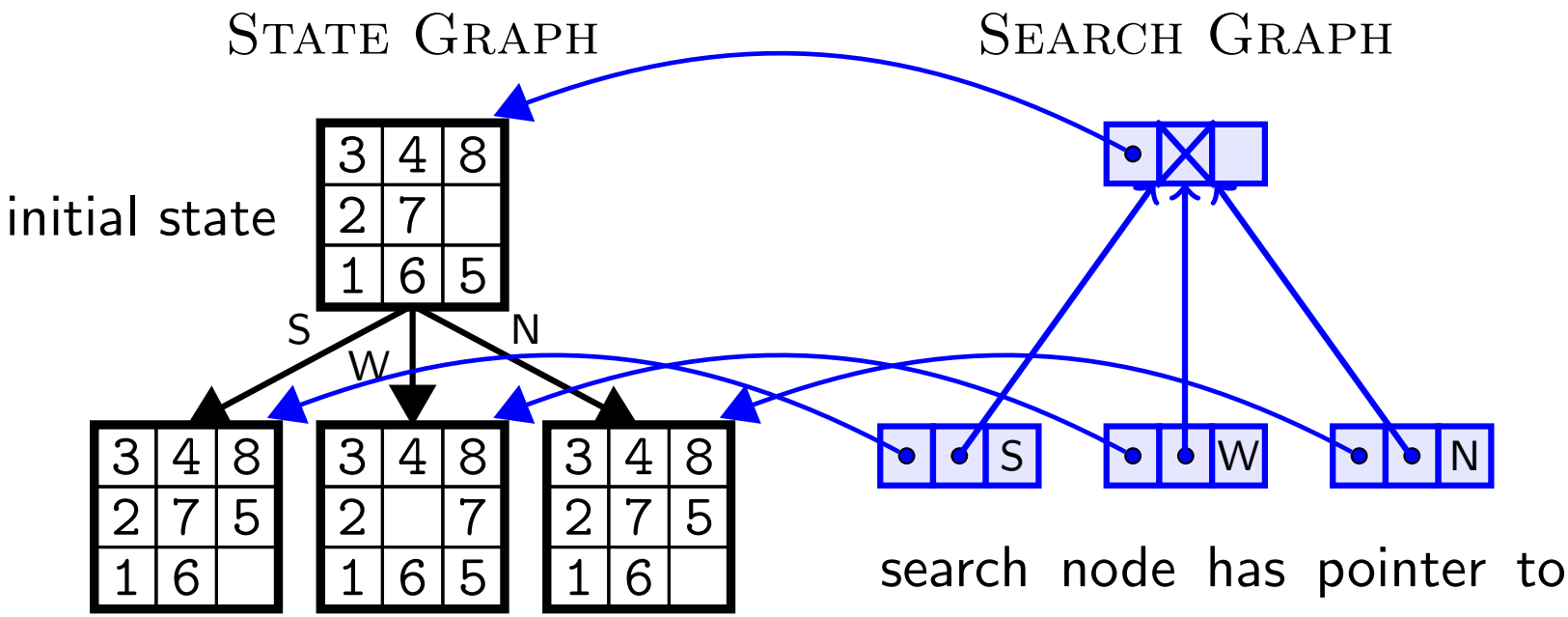
Search node space I

- Recall: Given a state space, we need the following information for each generated state as we try to find a solution starting from the initial state:
  - parent
  - parent action
  - path cost from initial state to the current state
  - etc. (For instance in some searches, a search node needs depth from initial search node, ...)
- These information are created during the search for a solution. Instead of putting these information in a state object, we will create a different type of objects – **search node** objects.

Search node space II

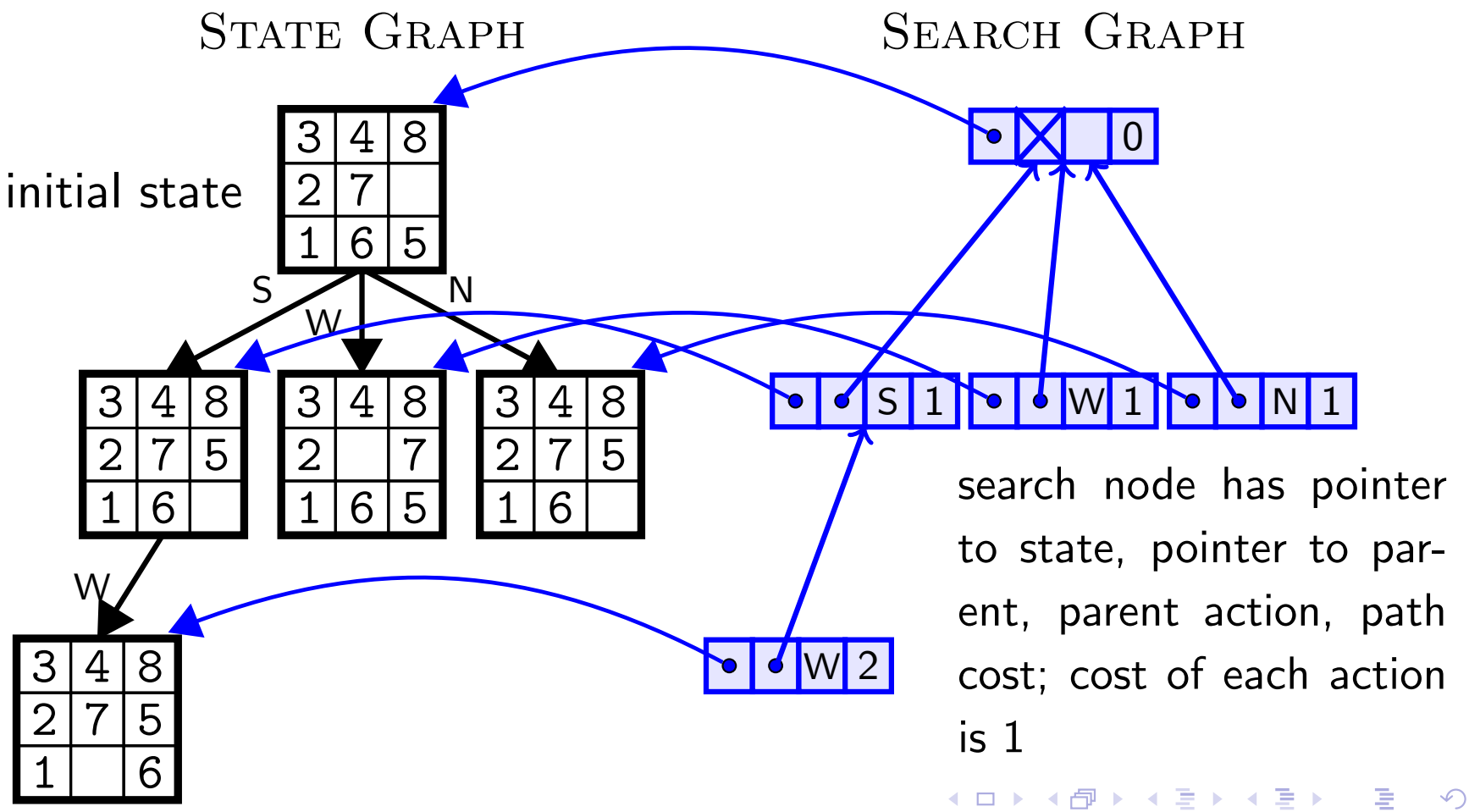
- Each object of the `SearchNode` class will contain:
  - pointer to a state
  - pointer to the parent search node
  - action from the parent search node to the current node
  - path cost from initial search node to the current node
  - Other data relevant to the search process is stored in the search node
- The **initial search node** is the search node that points to the initial state.

Search node space III



search node has pointer to state, pointer to parent, parent action

Search node space IV



Graph search algorithm I

- Now to describe the graph search algorithm ...

```
GRAPH-SEARCH (EARLY GOAL TEST VERSION)
INPUT: problem, fringe

let s = initial_state of problem
if s is a goal state: return empty solution
let n = search node with state=s and path cost=0
let closed list be empty
let fringe contain only n

while fringe is not empty:
    take a node, n, out of fringe and let s = state of n
    if s is a goal state, return n.solution()
    put s into closed list

    for a running over all actions for s:
        let s0 = resulting state when a is applied to s
        let n0 = search node with state=s0, parent=n, parent action=a,
                 path cost = n.path cost + cost(s, a)
        if s0 is a goal state, return n0.solution()
        if n0 not in fringe and s0 not in closed list:
            put n0 into the fringe

return FAILURE -- NO SOLUTION
```

```
GRAPH-SEARCH (LATE  GOAL TEST VERSION)
INPUT: problem, fringe

let s = initial_state of problem
if s is a goal state: return empty solution
let n = search node with state=s and path cost=0
let closed list be empty
let fringe contain only n

while fringe is not empty:
    take a node, n, out of fringe and let s = state of n
    if s is a goal state, return n.solution()
    put s into closed list

    for a running over all actions for s:
        let s0 = resulting state when a is applied to s
        let n0 = search node with state=s0, parent=s, parent action=a,
                 path cost = s.path cost + cost(s, a)
        if s0 is a goal state, return n0.solution()
        if n0 not in fringe and s0 not in closed list:
            put n0 into the fringe

return FAILURE -- NO SOLUTION
```

EARLY and LATE I

- Difference between early and late versions:
  - Early version: test if a state is a goal state before it enters the fringe (as a node).
  - Late version: test if a state is a goal state after it leaves the fringe (as a node).
- In more details, the following table lists the differences in the stages of the life of a state for the EARLY GOAL TEST and LATE GOAL TEST graph search.

EARLY and LATE II

Early goal test	Late goal test
generated	generated
goal tested	put into fringe
put into fringe	taken out of fringe
taken out of fringe	put into closed list
put into closed list	**goal tested**
expanded	expanded

Fringe and closed list I

- Recall we use two containers during the search: fringe and closed list.
- Note the differences between fringe and closed list.
  - Fringe: contains search nodes
  - Closed list: contains states.

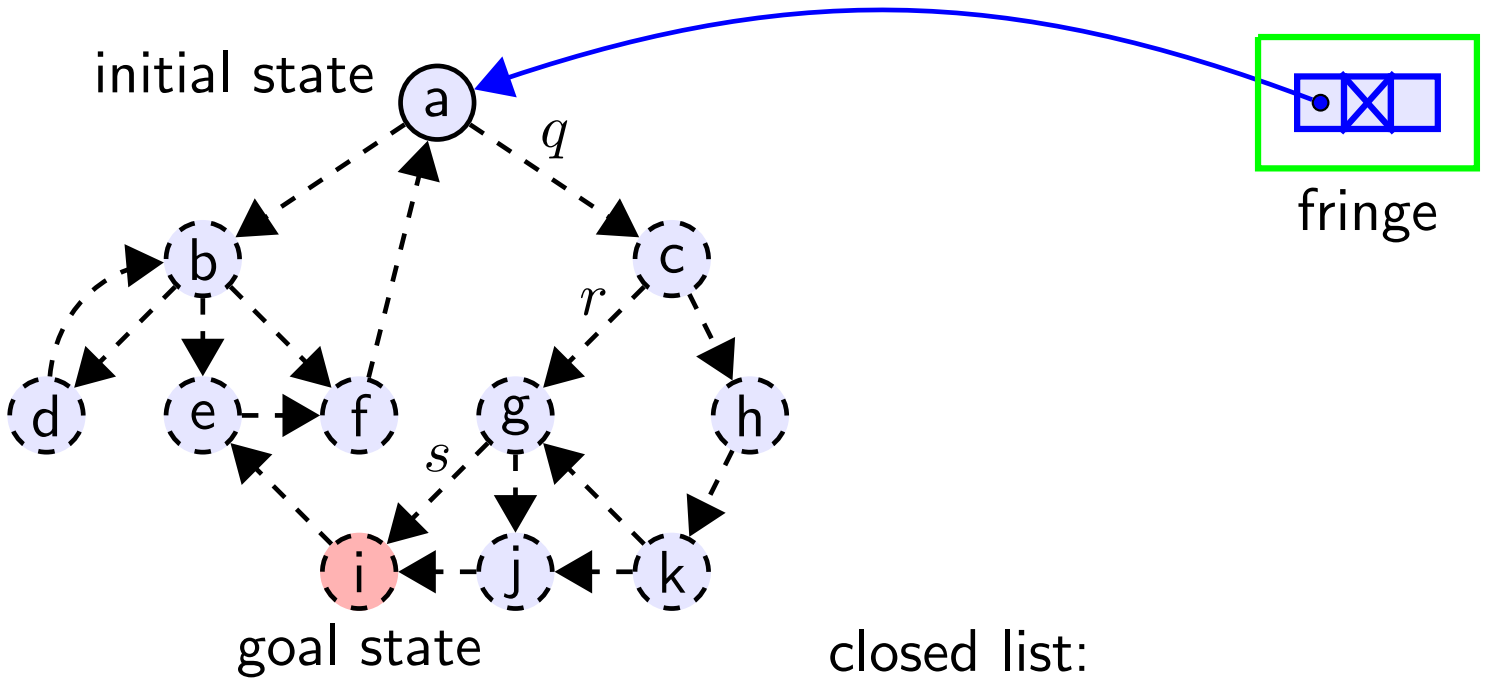
Fringe and closed list II

- Fringe:
  - Operations: insert, delete, find
    - Find means returning a reference/pointer to the value in the fringe. For some searches, the value might requires modification.
  - Can be stack, queue, etc. depending on search strategy
  - If the fringe is a stack or queue (using double ended queue), then find is $O(n)$ where $n =$ size of fringe – slow. To speed this update, include a hash table.

Fringe and closed list III

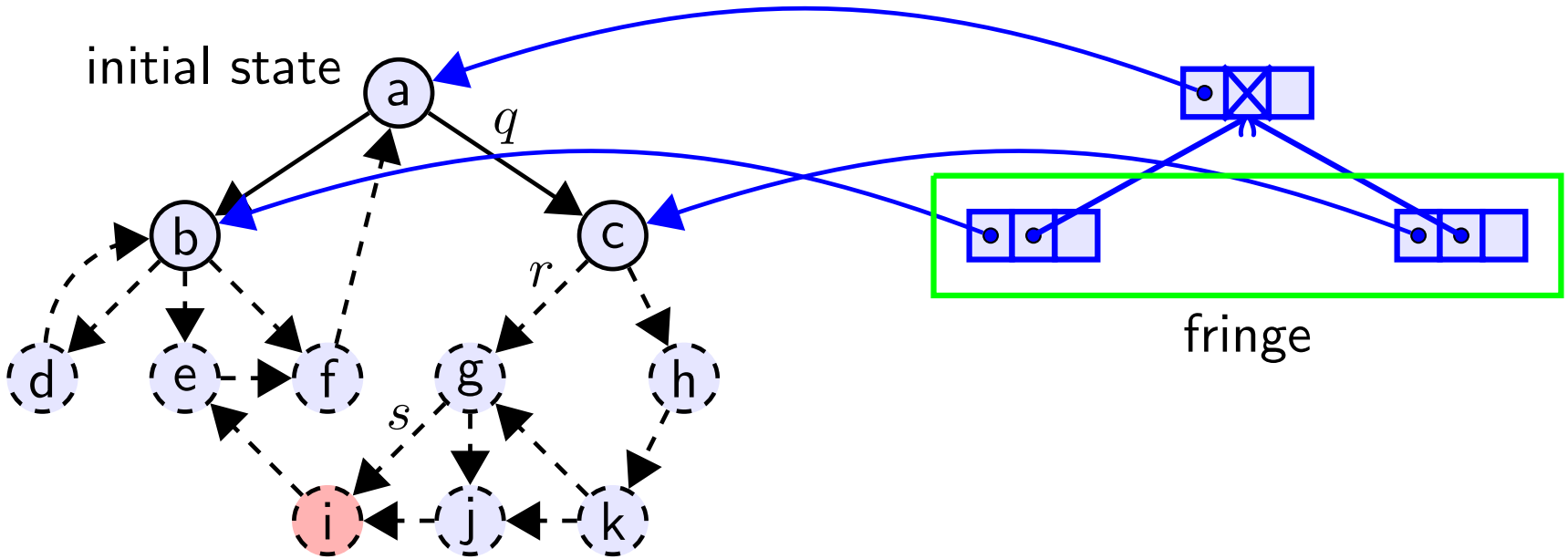
- Closed list:
  - Operations: insert, membership
    - Membership means checking if a value is in the container.
    - Values are not removed nor modified.
  - Implementation: Tree or hash table.
  - Note that the closed list only grows in size
  - Use some compression to save on memory. Will speed up find operation too.
  - Example: For 2D array of characters, might want to convert to string (array of characters) or bit sequence.
  - (There are other techniques for managing closed list such as removing useless states – will not cover in this course).
  - Exercise: For $(3^2 - 1)$–puzzle, what are some ways to compress each state?

Visual trace (implementation view) I



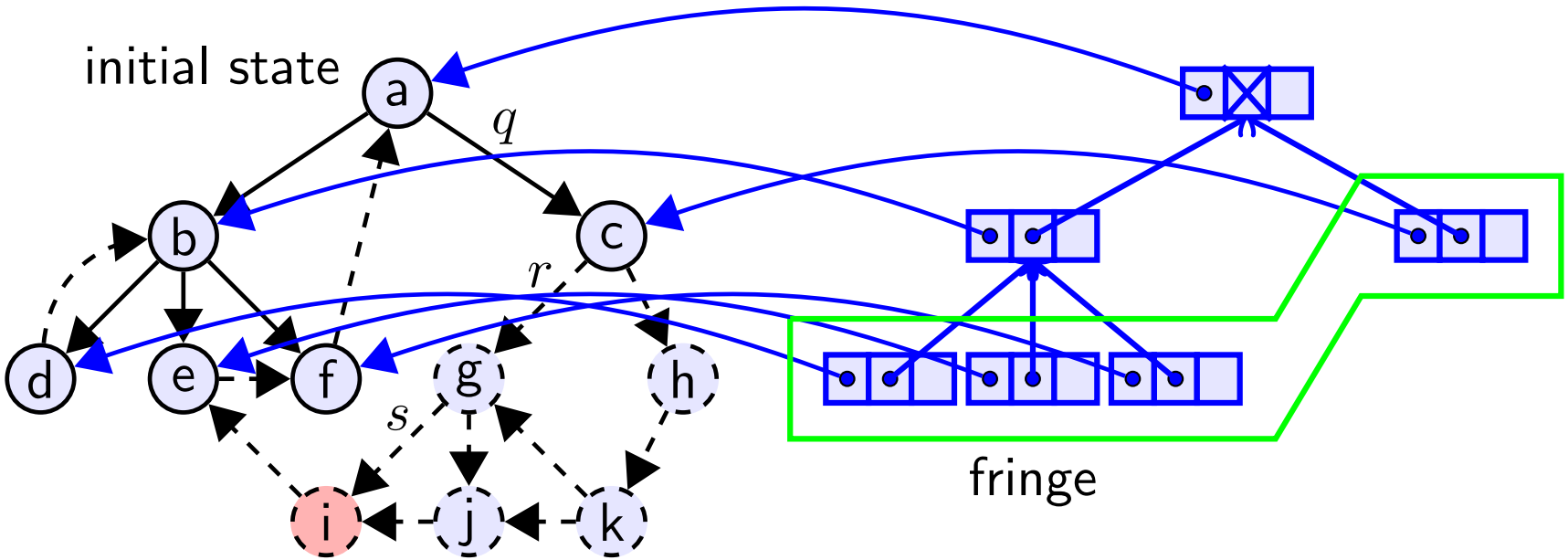
Initialization: creating search node for a and put it into fringe. At this point, memory usage is one state and one search node.

Visual trace (implementation view) II



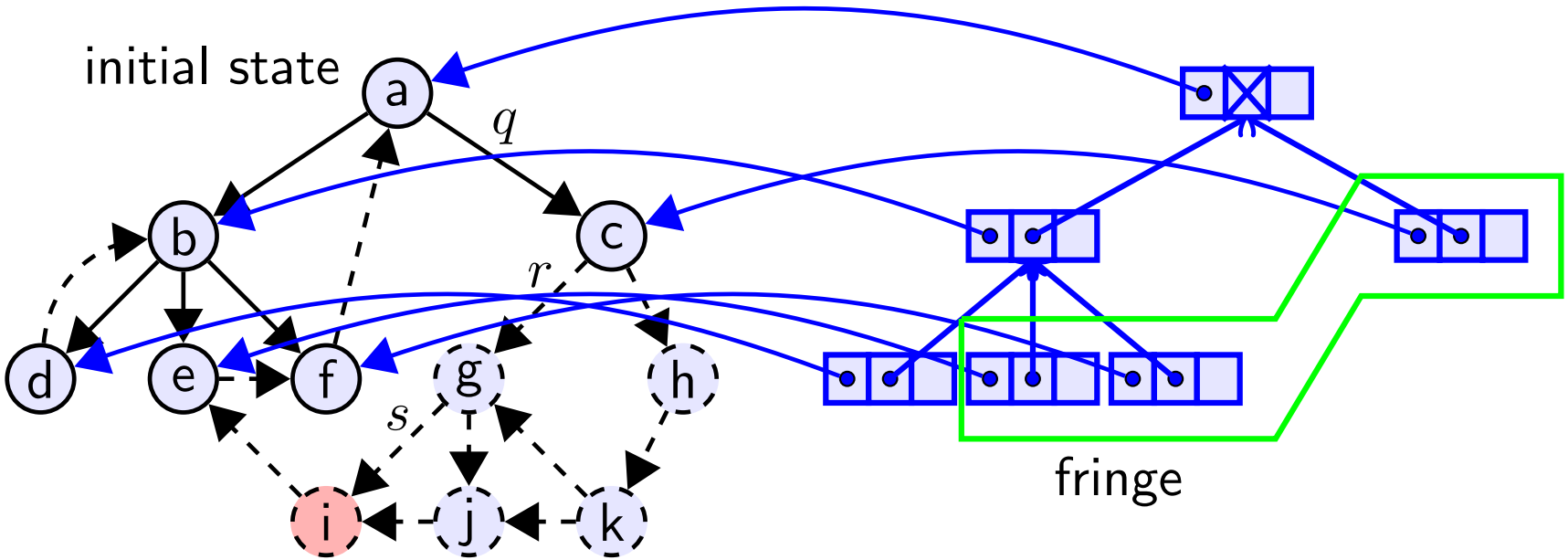
Remove search node of a from fringe. Put state a into closed list. Expand a to get b and c. b and c are not goal states: put them into fringe as search nodes.

Visual trace (implementation view) III



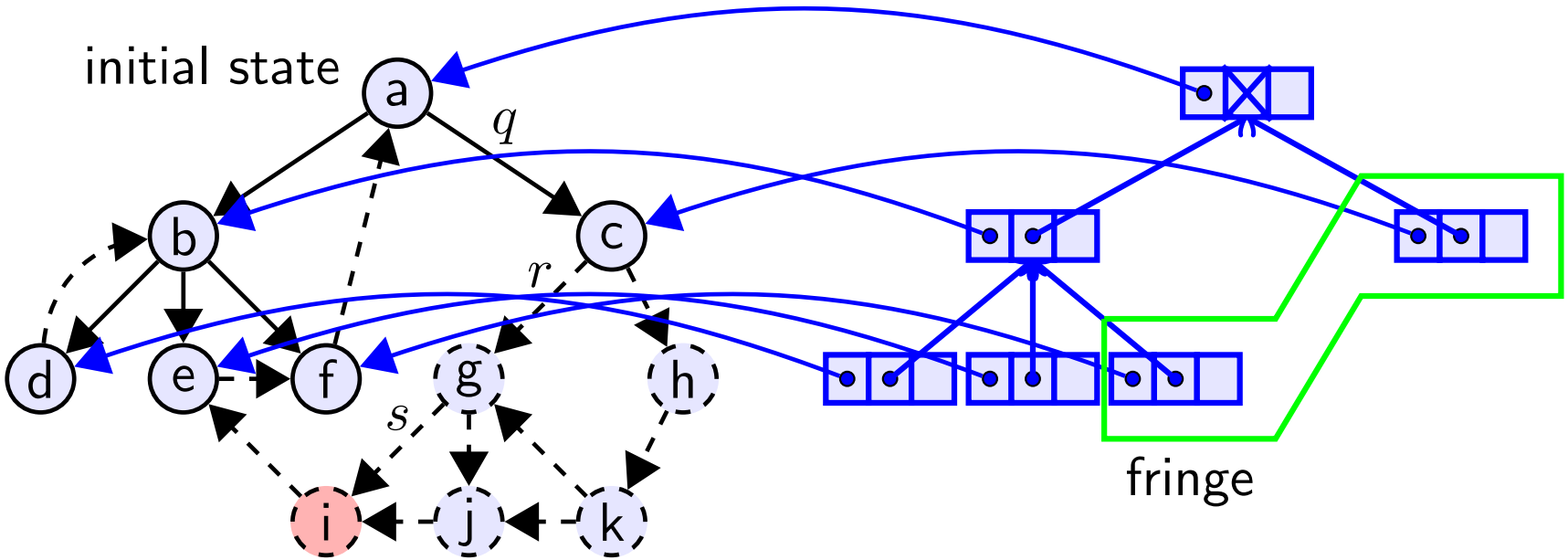
Remove a search node from fringe – say search node for b. Put b in closed list. Expand b to get d,e,f – none are goal state and none are in closed list. Put search node of d,e,f into fringe.

Visual trace (implementation view) IV



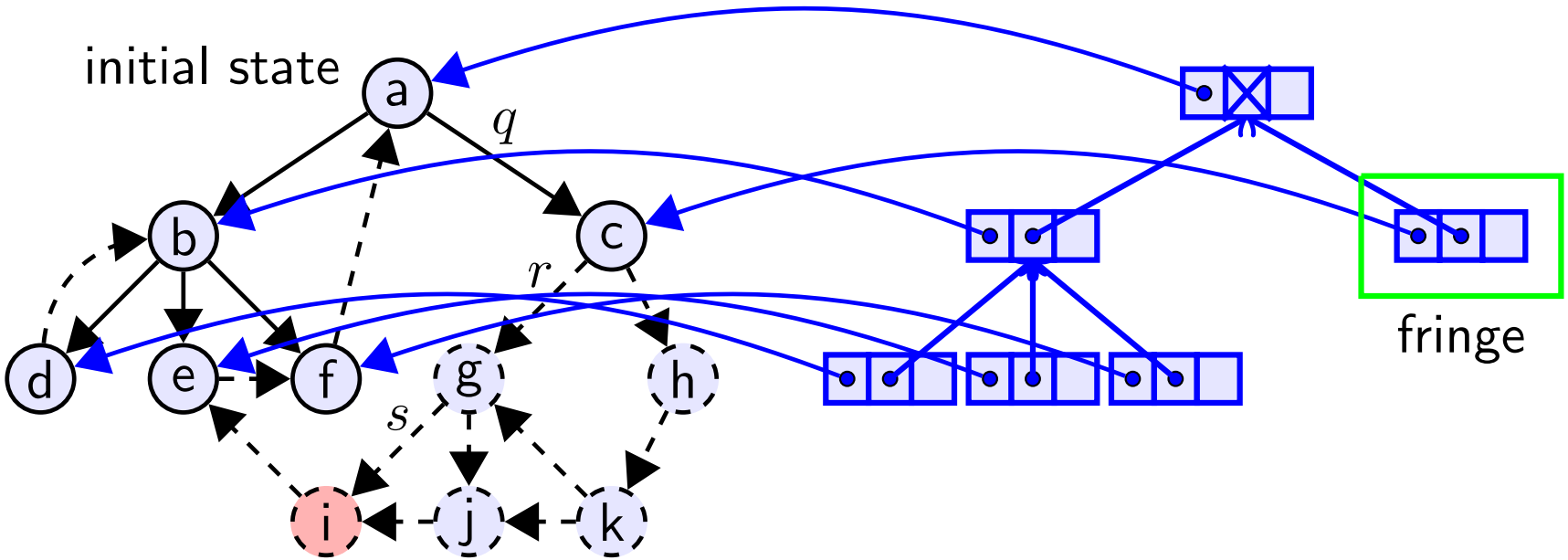
Remove a search node from fringe – say search node for d. Put d into closed list. Expand d to get b. b is in the closed list so ignore it.

Visual trace (implementation view) V



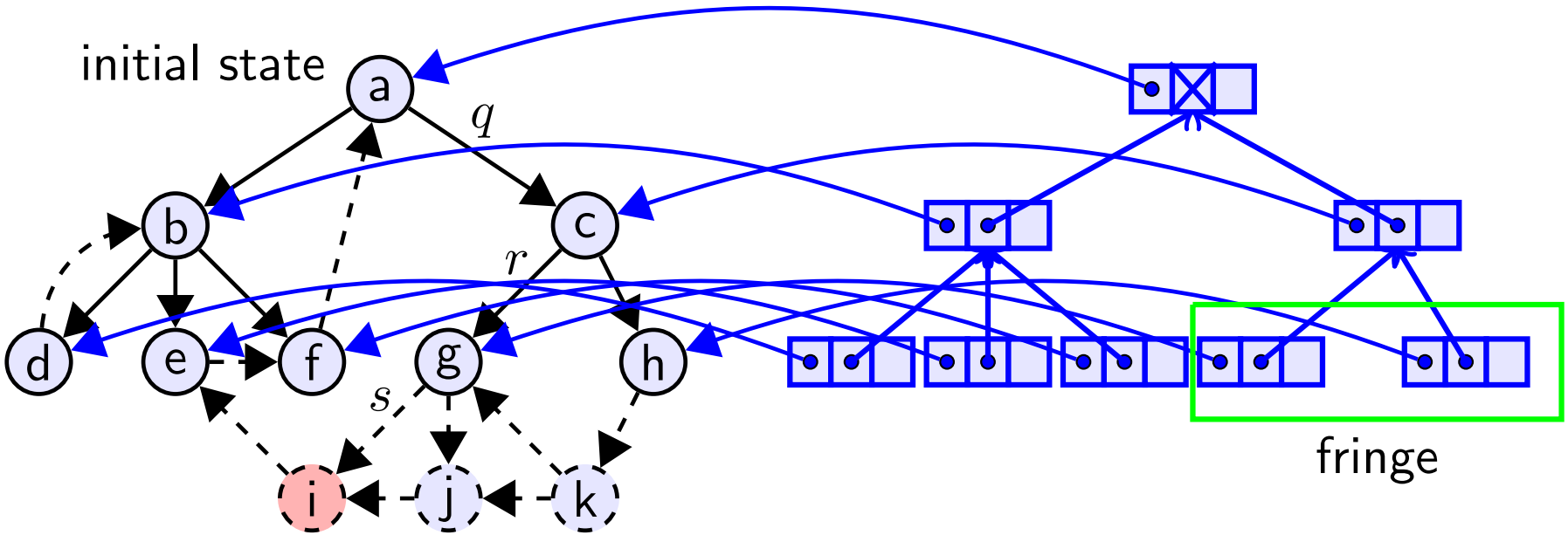
Remove a search node from fringe – say search node for e. Put e into closed list. e expands to f; f is already in fringe – ignore.

Visual trace (implementation view) VI



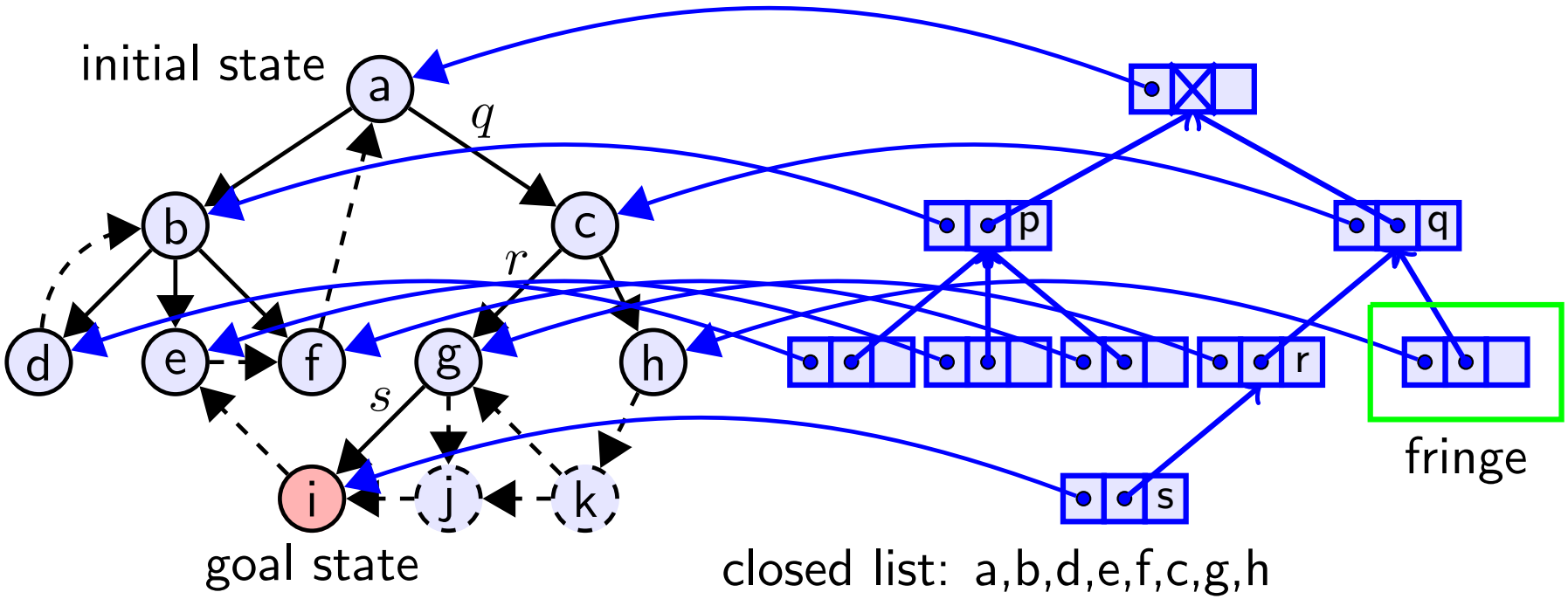
Remove a search node from fringe – say search node for f. Put f in the closed list. Expand f to get a. a is in the closed list so ignore it.

Visual trace (implementation view) VII



Remove a search node from fringe – there's only one, the search node for c. Put c into closed list. Expand c to get g, h – both not in closed list. Put the search nodes of g, h into fringe.

Visual trace (implementation view) VIII



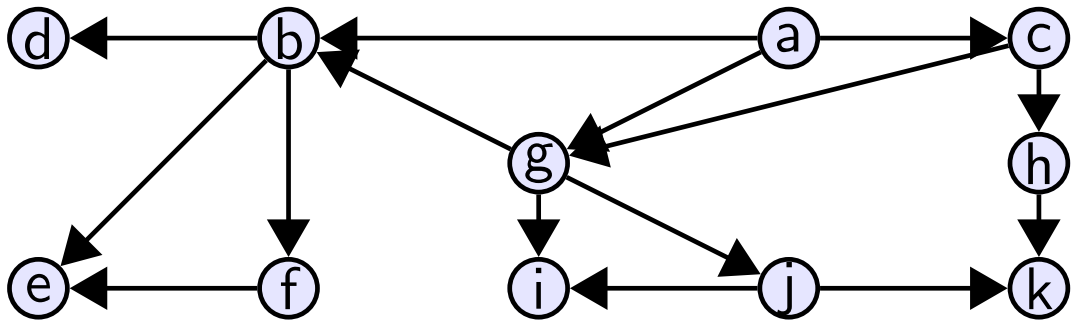
Remove a node from fringe – say search node for g. g expands to i (say i is expanded before j). i is a goal state! Create search node for i. (Last two steps not really necessary.)

Visual trace (implementation view) IX

- Note that a state of the state graph is exactly one of the following:
  - in the closed list
  - in the fringe (in the sense that a search node points to it)
  - not yet generated (i.e., not created in memory)
- You can think of the fringe as a wave front sweeping toward the states which are not yet created.

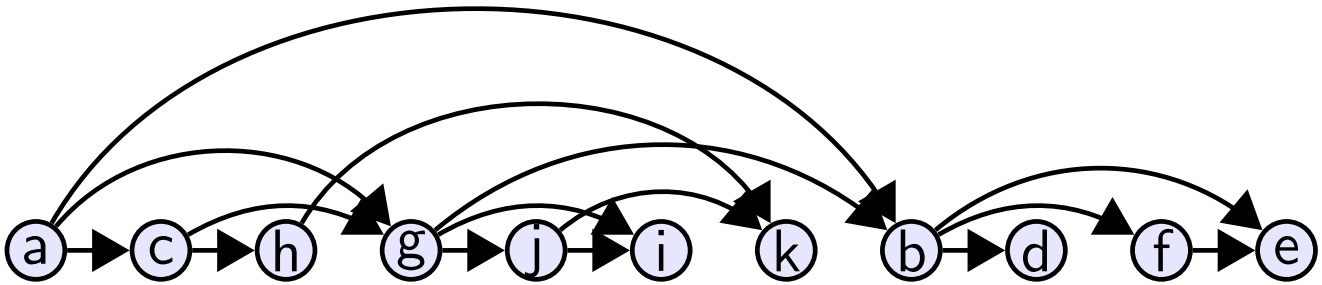
Tree search algorithm I

- What if the state space graph has some special properties?
- **Directed acyclic graph (DAG)** = directed graph with no cycles (i.e., no directed path that starts and ends at the same vertex)
- Example: The following is a DAG.



Tree search algorithm II

- The nodes of a DAG can be sorted so that directed edges only point to the right. This is called **topological sorting**.



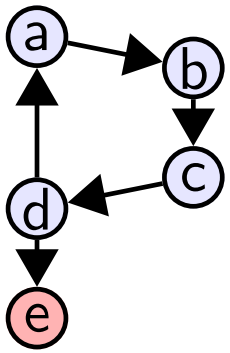
This organization is very important for theoretical and practical reasons.

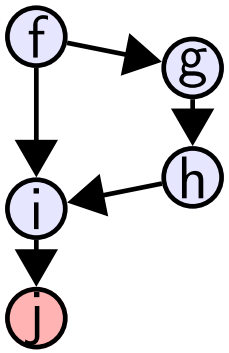
Tree search algorithm III

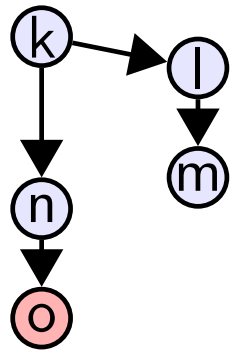
- Exercise. Can a DAG be topologically sorted in two different ways?
- Exercise. Given a DAG, print the vertex names in a topological sorted order.
- Exercise. There are several examples in this set of notes. Do you see any state space graph that is a DAG?

Tree search algorithm IV

- Look at the following three state graphs where a, f, k are initial states and e, j, o are goal states. .







- Left graph: Not DAG.
  - If you do not use closed list, search might go into infinite loop during search (if a is taken out of fringe before e).
  - If you use GRAPH-SEARCH (with a closed list), a can be placed in closed list. d will then expand to a, e but only put e into fringe – no infinite loop during search.

Tree search algorithm V

- Middle graph: DAG, some state has more than one parent.
  - There are multiple (in this case two) paths from f to i: f, i and f, g, h, i. i visited twice. This is a waste if the goal is to find any path to goal state.
  - But if the goal is to find a *least cost path*, then you *do* have to compute both paths, i.e., allow search nodes with the same state to enter fringe multiple times. Fringe will choose the one with the least path cost.
  - Use GRAPH-SEARCH if you want to find any solution.
  - Use TREE-SEARCH if you want to find least cost solution, allow some search nodes to enter fringe more than once.
- Right graph: DAG, every state has at most one parent. Closed list is not necessary, i.e., use TREE-SEARCH.

Tree search algorithm VI

- Tree search: Basically graph search except that the state space as a graph is a tree, i.e., no cycles. Therefore closed list is not needed.

```
ALGORITHM: TREE-SEARCH (EARLY GOAL TEST VERSION)
INPUT: problem, fringe

let s = the initial state in problem
if s is a goal state: return empty solution
let n = search node with state=s and path cost=0
let closed list be empty
let fringe contain only n

while fringe is not empty:
    take a node, n, out of fringe and let s = state of n
    if s is a goal state, return n.solution()
    put s into closed list

    for a running over all actions for s:
        let s0 = resulting state when a is applied to s
        let n0 = search node with state=s0, parent=s, parent action=a,
                 path cost = s.path cost + cost(s, a)
        if s0 is a goal state, return n0.solution()
        if n0 not in fringe and s0 not in closed list:
            put n0 into fringe

return FAILURE -- NO SOLUTION
```

```
ALGORITHM: TREE-SEARCH (LATE  GOAL TEST VERSION)
INPUT: problem, fringe

let s = the initial state in problem
if s is a goal state: return empty solution
let n = search node with state=s and path cost=0
let closed list be empty
let fringe contain only n

while fringe is not empty:
    take a node, n, out of fringe and let s = state of n
    if s is a goal state, return n.solution()
    put s into closed list

    for a running over all actions for s:
        let s0 = resulting state when a is applied to s
        let n0 = search node with state=s0, parent=s, parent action=a,
                 path cost = s.path cost + cost(s, a)
        if s0 is a goal state, return n.solution()
        if n0 not in fringe and s0 not in closed list:
            put n0 into fringe

return FAILURE -- NO SOLUTION
```

Cleanup – combining the algorithms I

- Combining graph and tree search: Have another closed list class, `DummyClosedList`, that does not do anything and every time you check if a state is in an object of this closed list class, it always return false. Add a closed list parameter to the graph search
- In the graph search algorithm

```
if n0 not in fringe and s0 not in closed list:
    put n0 into fringe
```

let fringe decide whether to add `n0` to the fringe or not:

```
if s0 not in closed list:
    put n0 into fringe (note: fringe might reject n0)
```

Cleanup – combining the algorithms II

- In the graph and tree search algorithm, we use the concept of state and node. To clean up, you can create methods in the search node class that mirrors the methods in the state class. For instance add an `actions()` method so that you can compute the actions available for a search node. Etc. Then the graph and tree search algorithm will only reference search nodes. Add method in closed list to accept nodes.

```
GRAPH-SEARCH (EARLY GOAL TEST VERSION)
INPUT: problem, fringe, closed list (use dummy closed list for
                                        tree search)
let n = initial_node of problem with path cost 0
if n is a goal node: return empty solution
let closed list be empty
let fringe contain only n

while fringe is not empty:
    take a node, n, out of fringe
    if n is a goal node, return n.solution()
    put n into closed list

    for a running over all actions for n:
        let n0 = resulting node of n on action a, with parent=n,
                 parent action=a,
                 path cost = n.path cost + cost(n, a)
        if n0 is a goal node, return n0.solution()
        if n0 not in closed list:
            put n0 into the fringe

return FAILURE -- NO SOLUTION
```

```
GRAPH-SEARCH (LATE  GOAL TEST VERSION)
INPUT: problem, fringe, closed list (use dummy closed list for
                                          tree search)
let n = initial_node of problem with path cost 0
if n is a goal node: return empty solution
let closed list be empty
let fringe contain only n

while fringe is not empty:
    take a node, n, out of fringe
    if n is a goal node, return n.solution()
    put n into closed list

    for a running over all actions for n:
        let n0 = resulting node of n on action a, with parent=n,
                 parent action=a,
                 path cost = n.path cost + cost(n, a)
        if n0 is a goal node, return n0.solution()
        if n0 not in closed list:
            put n0 into the fringe

return FAILURE -- NO SOLUTION
```

Measuring Performance I

- RECALL. Measure search algorithm with 4 metrics:
- **Completeness**: Can algorithm find solution if there is one?
- **Optimality**: Can algorithm find optimal solution (i.e. lowest path cost)?
- **Time complexity**
- **Space complexity**
  - Do not count initial state (since that's an input). But if you do include, it won't change the big-O.
  - There are memory usage for fringe and closed list. Closed list is 0 if you use TREE SEARCH.

Measuring Performance II

- Relevant notation:
  - b = branching factor = the maximum number of resulting states for any state. For the $n^2 - 1$ problem, $b = 4$.
  - d = depth of shallowest goal from initial state
  - m = max length of path in state space

Uninformed search I

- **Uninformed/blind/brute force search**: Info used about states are
  - Given a state, what are the valid actions and new states?
  - Is the state a goal state?
- Later ... **informed/heuristic search**: Info used about states are
  - Above ... and ...
  - Heuristic (i.e., imperfect) info about how "close" a state is to a goal state. Use this to choose action that will probably lead to goal state.

Uninformed search II

- Why is it called uninformed/blind/brute force?
  - You don't know until you actually arrive at goal state, i.e., at each state, you have to try every action and take a step to the next state.
  - Heuristic search uses heuristics to guide which states to explore earlier, i.e., before you take the action to the next state you have a approximate idea of whether the next state is a goal state.

Uninformed search III

- Some uninformed search strategies:
  - Breadth-first search (BFS)
  - Uniform-cost search
  - Depth-first search (DFS)
  - Backtracking search
  - Depth-limited search
  - Iterative deepening DFS (ID DFS)
  - Iterative lengthening search
  - Bidirectional search

BFS I

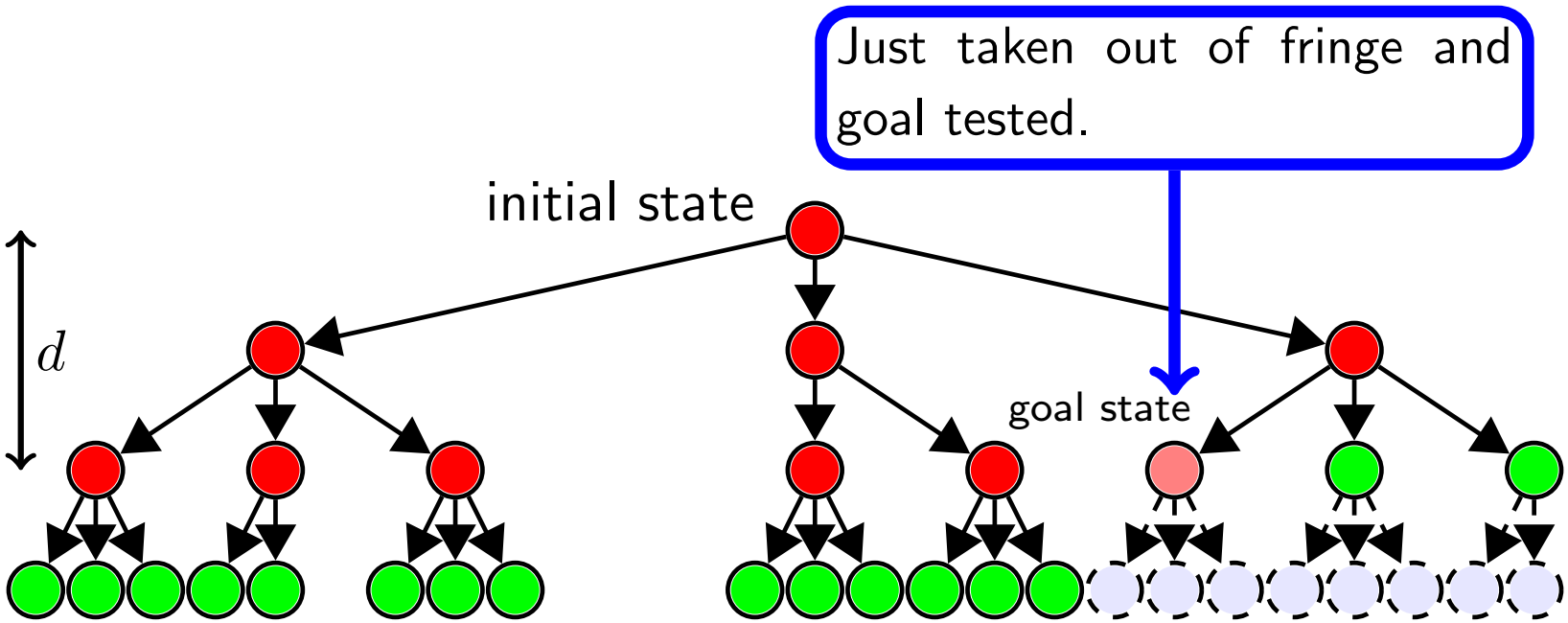
- **Breadth-first search (BFS)**: state at depth k is expanded before state at depth $k+1$.
- If state space is a graph: Use GRAPH-SEARCH EARLY GOAL TEST VERSION where fringe = queue.
- If state space is a tree: Use TREE-SEARCH EARLY GOAL TEST VERSION where fringe = queue

BFS: Performance I

- Complete? Yes if b finite
- Optimal?
  - Not necessarily (Exercise: Find an example.)
  - Yes if path cost is nondecreasing function of depth; (Example: path cost $=$ depth).
- Now to compute time and space complexity. I will do this for both late and early goal test version of BFS. Suppose d is the depth of a shallowest goal.

BFS: Performance II

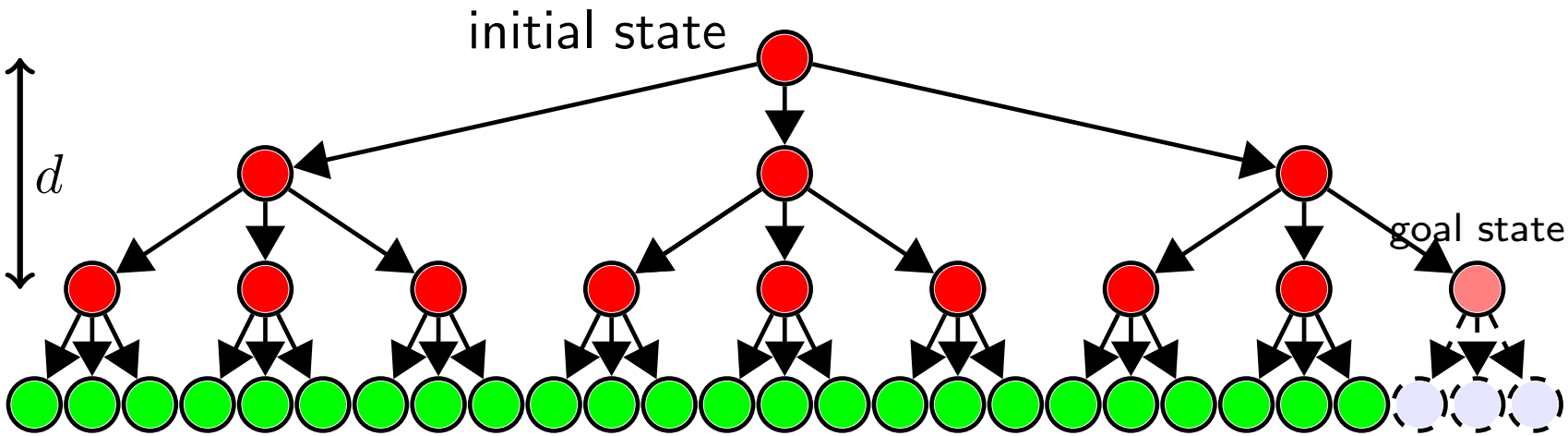
- Assume we use *late goal test* version of BFS. State is tested from the search node of that state is taken out of the fringe.



(Only generated part of state space drawn.)

BFS: Performance III

For the worse case there are as many states generated as possible, i.e., when every vertex (except the leaves) has b children and the goal state is on the furthest right:



BFS: Performance IV

Total number of generated states:

$$b + b^2 + \cdots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

(Initial state is not counted since it's an input). This is the runtime. Why? Because graph search performs a constant amount of work on each generated state (look at the stages of a state).

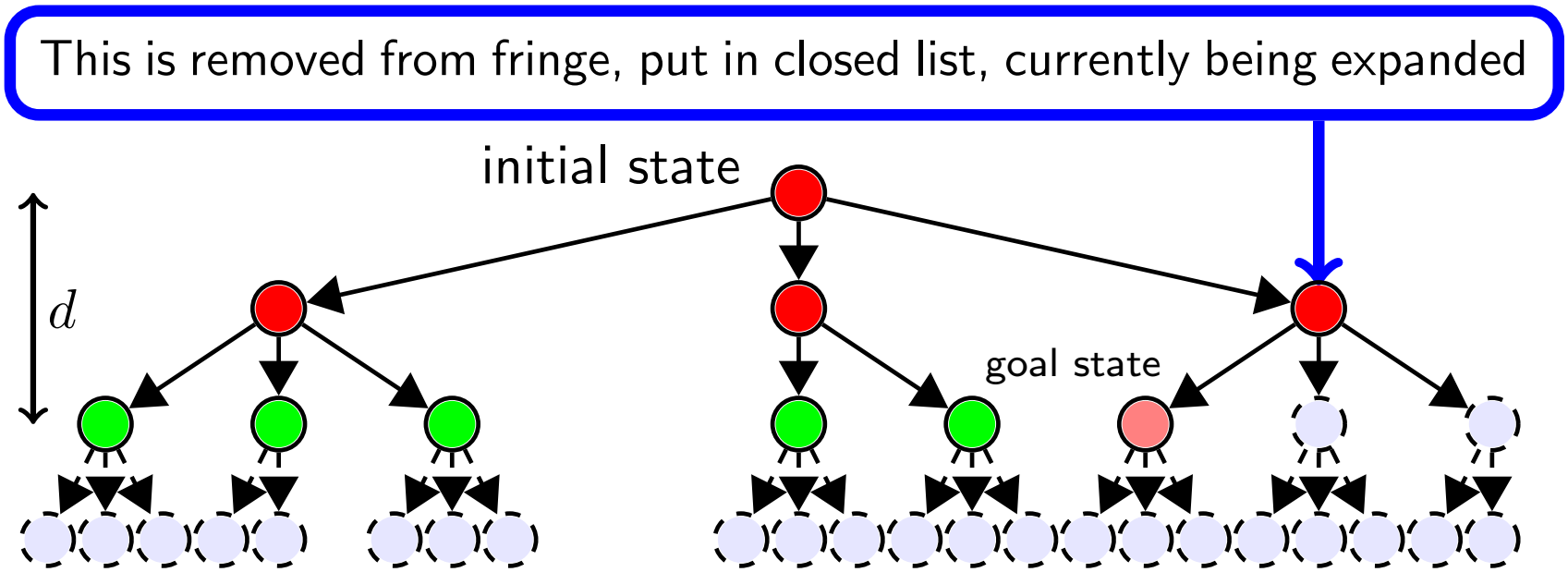
- Space complexity:
  - Space for fringe $= O(b^{d+1} - b) = O(b^{d+1})$
  - Space for closed list $= 1 + b + \cdots + b^d = O(b^d)$
- POINT:

BFS: Performance V

- Number of states at a depth $d+1$ is factor of b of number of states *in depth* $= 0, 1, 2, ..., d$!!!
- Unfortunately, for late goal test BFS, the next level after goal state is expanded – VERY COSTLY!!!

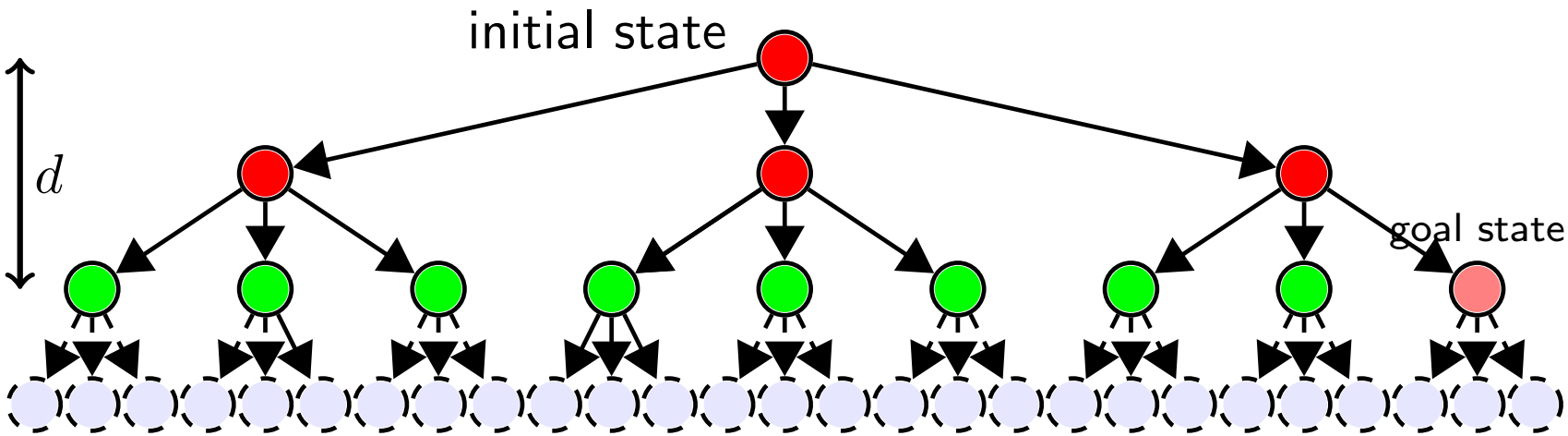
BFS: Performance VI

- Now assume we use *early goal test* version of BFS. State is tested when it is expanded from a parent state after the search node for that parent state is taken out of fringe.



BFS: Performance VII

- Time: For the worse case there are as many states generated as possible, i.e., when every vertex (except the leaves) has b children and the goal state is on the furthest right:



BFS: Performance VIII

The number of states generated is

$$b + b^2 + \cdots + b^d = O(b^d)$$

This is the runtime.

- Space:
  - Space for fringe $= O(b^d)$
  - Space for closed list $= 1 + b + \cdots + b^{d-1} = O(b^{d-1})$

Quick review of big-O I

- RECALL: Let $f(x)$ and $g(x)$ be functions. We write $f = O(g)$ if there is some C and N such that

$$|f(x)| \leq C \cdot |g(x)| \qquad \text{for } x \geq N$$

- For simplicity, assume functions take positive values In that case the above condition becomes

$$f(x) \leq C \cdot g(x) \qquad \text{for } x \geq N$$

- Example: $5n^3 + 7n + 1000 = O(10n^3 + 2n + 1)$
- Example: $5n^3 + 7n + 1000 = O(n^3 + 2000n^2)$

Quick review of big-O II

- For instance if $p(n)$ is a polynomial of degree d, then

$$p = O(n^d)$$

- Example: $5n^3 + 7n + 1000 = O(n^3)$
- Example: $\log n = O(n)$, but $n \neq O(\log n)$
- The big-O notation measures growth of a function
- Informal way to look at big-O: $f = O(g)$ means that if you zoom out the graph of f and g far enough, and you look at the graphs of f and g for "large" values of n, you see that g is "roughly" the same as or is above f.

Quick review of big-O III

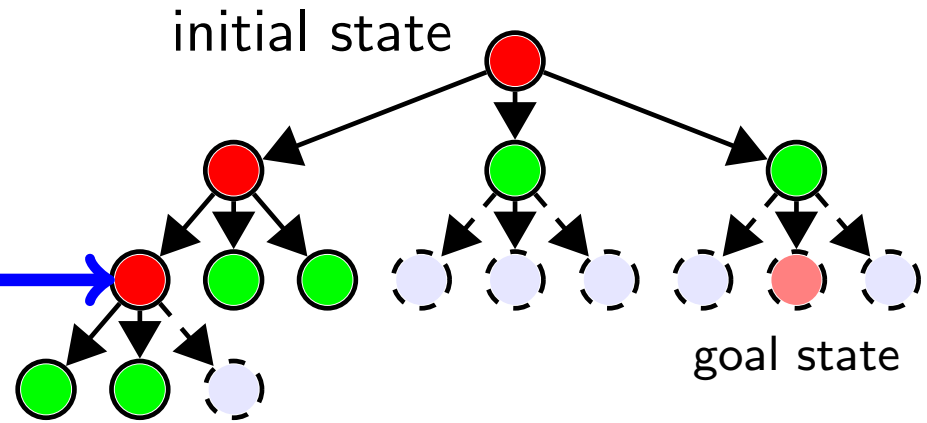
- That's only an informal way to look at it and is not rigorous enough. Details: CISS350/358.
- Exercise: Use your graphing calculator and graph $5n^3 + 7n + 1000$ and n^3. Zoom out and make sure you see that the two graphs are almost the same.

DFS I

- **Depth-first search (DFS)**: state that is expanded is a state of a search node in the fringe with greatest depth.
- If state graph is a graph: use GRAPH-SEARCH (EARLY GOAL TEST VERSION) where fringe = stack.
- If state graph is a tree, use TREE-SEARCH (EARLY GOAL TEST VERSION) where fringe = stack.

DFS: Performance I

This just taken out of fringe, put in closed list, finished expanding two successors.



- Complete:
  - Problem when there is no maximal depth, i.e., when $m = \infty$.
  - Assume now that m is finite.
  - Yes if use GRAPH SEARCH
  - Yes if use TREE SEARCH and state graph is a tree

DFS: Performance II

- No if use TREE SEARCH and state graph is not a tree (possibility of infinite loop during search)

- Optimal: No. (Why?)
- Time: $O(bm)$
- Space (for fringe): $1 + bm$ nodes where $m =$ max depth
- Can also use recursion instead of GRAPH-/TREE-SEARCH algorithm.
- Closed list optimization: Recall that the closed list prevents (1) infinite loops and (2) paths that overap. To save memory, can remove descendents of a state if all descendents are explored

Backtrack search (BTS) I

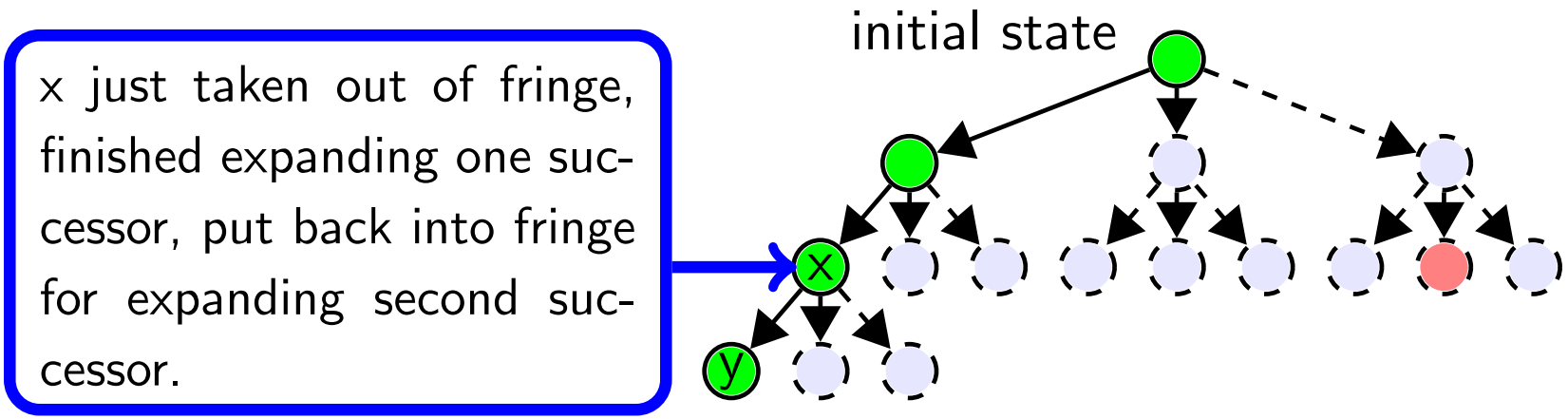
- **Backtrack search** is a form of depth first search (i.e., search depth-wise rather than breadth-wise) except that
  - In DFS, during an expansion phase of state s, *all* successors are computed. After that s is not used for successor generation.
  - In backtrack search, during one expansion phase for state s, *one* successor of s is generated. If necessary, s will generate other successors in later expansion phases.
- Two ways to implement backtrack search:
  - Use recursion (see CISS350 notes).
  - Use GRAPH-/TREE-SEARCH

Backtrack search (BTS) II

- Why is it called *backtrack*?
  - In the recursive implementatation of backtrack search, each function call holds one state s, and make one function call with s', *one* successor of s. If s' leads to a dead end, then function return will go back – *backtrack* – to the function holding s which will make another function call with *another* successor s'' of s.
  - Similar for GRAPH-/TREE-SEARCH implementation of backtrack – still need to backtrack to a previous state to compute a subsequent successor.

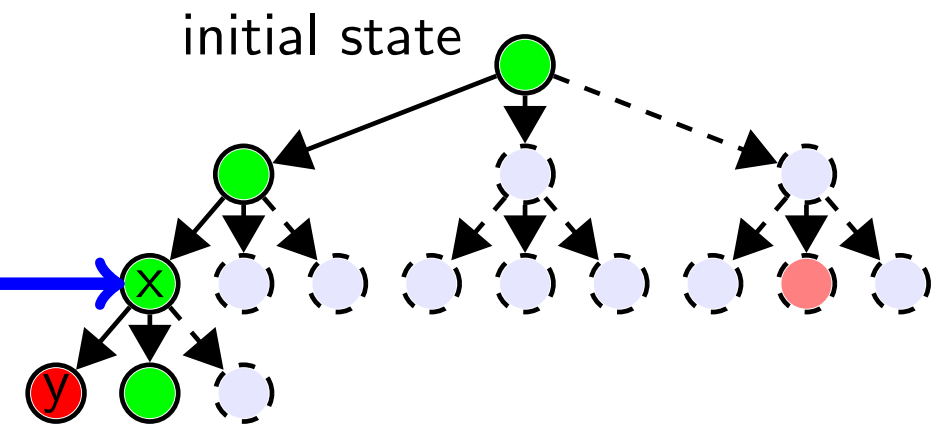
Backtrack search (BTS) III

- In the following diagrams, you can view the nodes of the graph as function calls (recursive implementation of backtrack) or as search node (GRAPH/TREE search implementation of backtrack search)



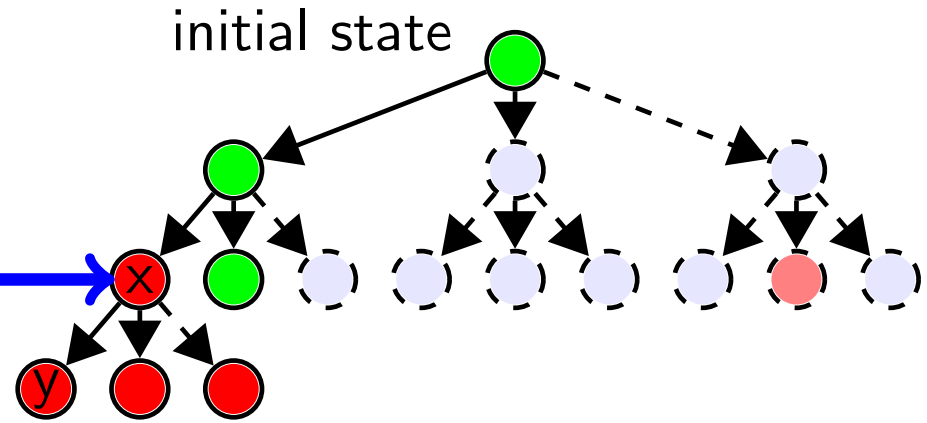
Backtrack search (BTS) IV

y has expanded all successors – goal not found. y go into closed list. Take x out of fringe to expand second successor. x and second successor go into fringe.



Backtrack search (BTS) V

This is the picture after x has expanded all successors (and none are goal states) and the parent of x has just expanded one of x's sibling.



- You see that the space complexity of the fringe is

$$m - 1 = O(m)$$

states where m is the maximum depth. Compare this with DFS which is $O(bm)$.

Backtrack search (BTS) VI

- Backtrack search is a very important AI search technique because of small memory usage. Particularly important when branching factor b is large.
- Example: Robotics where for instance a robot have b degrees of freedom in turning on a plane and b is large (example: 360).

Backtrack: using recursion I

- Recursive backtracking search:
  - Uses recursion. See CISS350 notes.
  - One problem: recursion limit.
- Idea:
  - Each function frame holds one state s.
  - Each function makes recursive function calls for each successor of s, the state that it holds. i.e., a recursive function call is a single state expansion of s.
  - If a function call reaches a goal state, return true.
  - If a function receive true (i.e., further down the path of function calls, a goal state was reached), it adds the relevant action to the solution and returns true. Very similar to parent pointer of search nodes.

Backtrack: using recursion II

- If a function call does not receive true and has finished making all its recursive function calls (i.e., every one of its successor states is a non–goal state), it return false.

- For our GRAPH/TREE SEARCH algorithms:
  - We use search nodes to keep track of parent search node. In the case of backtrack, the function frame for each function call is retained so that parent pointer is not necessary.
  - Within each function call, we iterate through actions for one state and keep that action in the solution if the recursive function call tells us that a future descendent state is a goal state. So keeping track of the parent action is also not necessary.

Backtrack: using recursion III

Therefore for recursive backtrack, search nodes are *not* necessary.

- The recursive BT search is for graph. What if the search space is a tree? Write a tree version.

```
RECURSIVE-BACKTRACK-SEARCH(problem, closed) VERSION 1:
let s = copy of problem.initial_state
set closed to be empty and let solution be an empty solution
if BT_HELPER(problem, s, closed, solution) is true:
    return solution
else:
    return FAILURE
```

```
BT_HELPER(problem, s, closed, solution):
INPUTS: closed and solution are passed by reference
OUTPUT: return true if s goal state is reached

if s passes goal test:
    return true
else:
    for a running over all possible actions of s:
        s0 = state after applying a to s
        if s0 is not in closed:
            add s0 to closed
            if BT_HELPER(s0, closed, solution) is true:
                add a to solution and return true
    return false
```

Backtrack search (BTS): Optimization I

- For the space complexity, the measure is the maximum amount of memory at any point in time during the execution of the algorithm.

Backtrack search (BTS): Optimization II

- Look at these:

```
// Code A
j = 42
for i = 0, 1, 2, ..., n - 1:
    let k = j + i
    print k
```

```
// Code B
j = 42
for i = 0, 1, 2, ..., n - 1:
    j = j + i
    print j
    j = j - i
```

  - Code A: `n` different `k`'s were created (and removed when each goes out of scope); there's only one `k` at any point in time in the loop.
  - Code B: modify `j` so that it takes the role of `k`; after that, reset `j` back to what it was.

- The space complexity for Code A and Code B are the same.

Backtrack search (BTS): Optimization III

- So what's the difference?
  - In Code A, `k` is created-destroyed many times which can be costly if `k` is a complicated object.
  - In Code B, you have to reset `j`.
  - Usually the reset of a state object is usually small when compared with creating-destroying a complicated state object.
- Example: In the $n^2 - 1$ puzzle for $n = 10$, a state is made up of 100 integers. So creating and copying 100 integer from an array to another is defintely costlier than applying an action to a state which requires swapping two integers.
- The above idea can be used to optimize RECURSIVE-BACKTRACK-SEARCH VERSION 1.

Backtrack search (BTS): Optimization IV

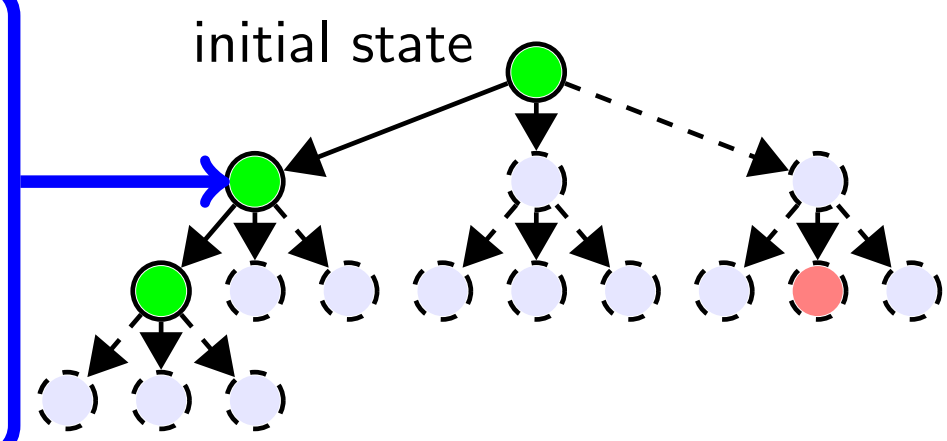
- In VERSION 1 of RECURSIVE-BACKTRACK-SEARCH, the function call does this:

```
s0 = result(s, a)
```

which make a copy of `s`, modifies it (through action `a`), and returns it to `s0`. If `s` has 3 actions, then 3 copies of `s` are made (i.e., there are really 3 different `s0` during the complete execution of the loop although only one exists at any point in time).

Backtrack search (BTS): Optimization V

VERSION 1: Function call for this state uses one `s` (suitably modified) for each of the three function calls that will be made.



More efficient to keep one `s` during the execution of the for-loop, modifying it so that `s` can play the role of `s0`. See VERSION 2 below. `result` is replaced by `modify` which operates on a state that is passed in by reference. The "opposite" of `modify` is called `undo`.

Backtrack search (BTS): Optimization VI

- Memory usage is 1 state and $O(m)$ recursive function frames. Each function is essentially used to store one action. Therefore the memory usage is 1 state and $O(m)$ actions.
- If modify the initial state (i.e., we do not need to save it), then memory usage is $O(m)$ actions. However, the initial state is usually not changed.
- Exercise. Is the recursion tail or non-tail? If it's not, can you convert it to a tail? (Unfortunately the standard python interpreter cannot perform tail recursion optimization.)

```
RECURSIVE-BACKTRACK-SEARCH(problem, closed, solution) VERSION 2:
let s = copy of problem.initial_state
set closed to be empty and let solution be an empty solution
if BT_HELPER(problem, s, closed, solution) is true:
    return solution
else:
    return FAILURE
```

```
BT_HELPER(problem, s, closed, solution):
INPUTS:
  closed and solution are passed by reference

if s passes goal test:
    return true

for a running over the actions of s:
    modify(s, a)
    if s is not in closed:
        add s to closed
        if BT_HELPER(s, closed, solution) is true:
            add a to solution and return true
    undo(s, a)
return false
```

Backtrack: using GRAPH SEARCH algorithm I

- Problem with RECURSIVE BACKTRACK-SEARCH: maximum recursion depth in function calls.
- GRAPH-SEARCH implementation: the search node in the fringe needs to remember a where a is the action to be used for producing its successor.
  Example: $n^2 - 1$:
  - Goal test initial state `s0`.
  - Put `(s0, N)` into fringe.
  - Remove `(s0, N)` from fringe, put `(s0, S)` into fringe. Compute `s1 = result(s0, N)`. If `s1` not in closed list and not in fringe: goal test `s1` (say it fails) and put `(s1,N)` into fringe.

Backtrack: using GRAPH SEARCH algorithm II

- Take `(s1, N)` out of fringe and put `(s2, S)` into fringe and say this gives `s2` (say it's not goal). Suppose `s2` has only one action `N`: Put `(s2, N)` into fringe.
- Take `(s2, N)` out of fringe and goal test (say it fails). Say all successors of `s2` are in closed list or fringe.

- In general, take `(s, a)` out of fringe and generate `s' = result(s, a)`. Goal test `s'`. If it's a goal: FOUND. Otherwise, put `(s, a')` where `a'` is next action available for `s` after `a` in fringe, put `(s', a'')` into fringe where `a''` is first action available for `s'`. Of course if any of the above "next action" is not available, you don't put that into the fringe.

Backtrack: using GRAPH SEARCH algorithm III

- Exercise. Looks like the GRAPH-SEARCH algorithm has to be changed. Is it possible not to change it?
  SOLUTION. Instead of modifying the GRAPH-SEARCH algorithm, you can modify the fringe.
  - remove operation: When you perform a remove operation from the fringe, you take (n, a) out of the fringe and put (n, a') back into the fringe and return (n, a) where a' is the next action after a for the state of n; return n.
  - insert operation: When you put n into the fringe, the fringe puts (n, a) into the fringe where a is the first operation of s.

Uniform cost search (UCS) I

- Goal: Suppose each action has a cost. The goal is to find a solution (of actions) such that the **path cost of the solution** (= sum of cost of actions of the solution) is the smallest.
- For most cases, we assume step cost ≥ 0.
- **Optimal solution** = solution with least path cost.
- Example: Energy used by a vacuum cleaner robot to move forward for 1 sec, energy to turn clockwise for 1 sec, energy used to suck for 1 sec, etc. might all be different. It is natural to find a solution that minimizes energy usage.

Uniform cost search (UCS) II

- Notation: Let s be a state. Define $g(s) =$ least path cost from initial state to s. If n is a search node with state s, we write $g(n)$ for $g(s)$.
- **Uniform cost search (UCS)**: search node that is selected for expansion is a node with least path cost in the fringe
- Implementation of UCS: Use GRAPH-SEARCH (LATE GOAL TEST VERSION) where search node contains path cost and fringe is UCS fringe (see below).

Uniform cost search (UCS) III

- Search node contains path cost. The path cost for the initial node is 0. If action a when applied to state s gives s' and suppose that n is a search node for s while n' is a search node for s'. Then

$$\text{path\_cost}(n') = \text{path\_cost}(n) + \text{step\_cost}(s, a)$$

During the execution of UCS, the path cost for a state (stored in a search node) can change.

Uniform cost search (UCS) IV

- UCS Fringe: Priority queue that favors lower path cost. Furthemore if a new search node n' is added to the fringe which contains n and both n and n' are search nodes for the same state, then n is replaced by n' if

$$\text{path\_cost}(n') < \text{path\_cost}(n)$$

Otherwise n' is rejected. Note:

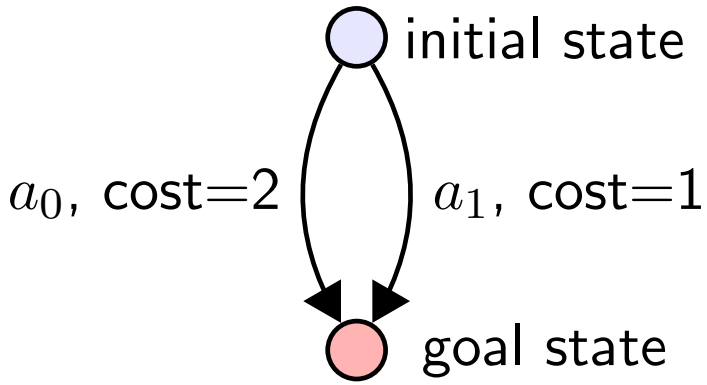
  - If there's a tie, $g(n') = g(n)$, then the new node n' is rejected.
  - For two search nodes with the same path cost (but different states), the one that was in the fringe longest will be ahead.

Uniform cost search (UCS): comparison with BFS I

- BFS finds solution which is shallowest, i.e., number of edges to initial state is smallest. So if cost of each action is 1, then BFS = UCS where least path cost = number of edges.
- Can view UCS as a generalization of BFS
- If each edge cost is constant, UCS = BFS in the sense that the solution found by both would have the same path cost.
- Exercise. If the edge cost is always 1, will UCS and BFS find the same solution?

Uniform cost search (UCS): comparison with BFS II

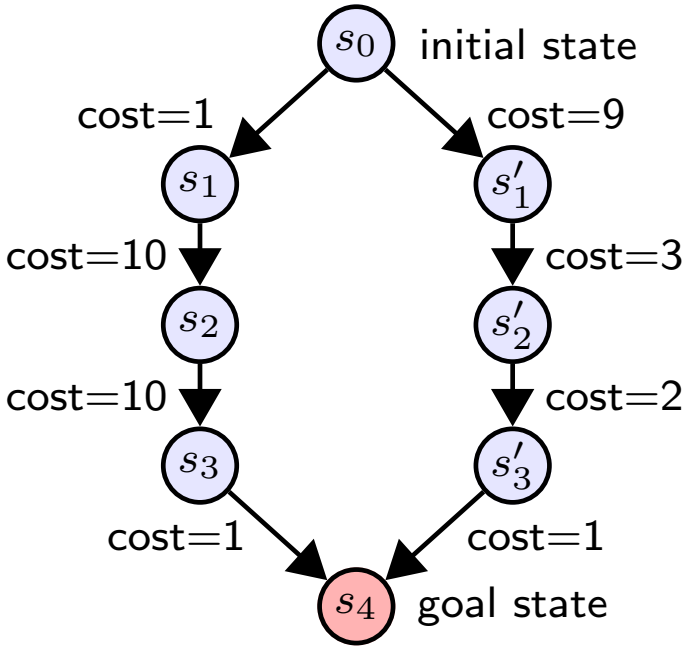
- Why use LATE GOAL TEST VERSION? If you use EARLY GOAL TEST VERSION: when node n_0 goes into fringe and has goal state, solution is returned. However if there's another node n_1 that has the same state as n_0 and has a lower path cost that will be inserted next, then the search will miss n_1. For instance look at the following state graph where action a_0 was used for expanding before a_1:



Uniform cost search (UCS): comparison with BFS III

- Exercise. To understand UCS, trace the search with the following state graph. For each iteration, write down the search node taken out of the fringe for expansion and write down the fringe after each iteration, writing down the states with decreasing priority (i.e., increasing path cost).

Uniform cost search (UCS): comparison with BFS IV



- Exercise. What happens if you want to find an optimal solution and the state space graph is a tree?

Uniform cost search (UCS): comparison with BFS V

- Exercise. Assume that all edge costs are positive. Why is it OK to put a state into the closed list once a search node of that state has been expanded?

Uniform cost search (UCS): Performance I

- UCS might go into an infinite loop if
  - $b = \infty$ (branching factor), or
  - there is an path of infinitely many distinct states with no goal state and all edge costs $= 0$.
- Complete? Yes if b (branching factor) is finite and all step cost is $> \epsilon$ for some fixed positive ϵ.
- Optimal? Yes if b (branching factor) is finite and all step cost is $> \epsilon$ for some fixed positive ϵ.
- Exercise. Write down an example where UCS is not complete even though the branching factor b is finite.
- Exercise. Is UCS complete if all edge costs are > 0 (instead of $> \epsilon$ for some fixed constant ϵ)?

Uniform cost search (UCS): Performance II

- Time $= O\left(b^{1+\lfloor C^*/\epsilon \rfloor}\right)$ where C^* is cost of optimal path. Why? Suppose the optimal path has length k and every cost is $\geq \epsilon$ where $\epsilon > 0$ is a constant. Then

$$C^* \geq k\epsilon$$
$$\therefore \quad C^*/\epsilon \geq k$$

Since k is an integer:

$$\lfloor C^*/\epsilon \rfloor \geq k$$

Uniform cost search (UCS): Performance III

i.e., there are at most $\lfloor C^*/\epsilon \rfloor$ edges in the optimal path. Therefore the number of states generated is at most

$$b + b^2 + \cdots + b^{\lfloor C^*/\epsilon \rfloor} + b^{\lfloor C^*/\epsilon \rfloor + 1} - b = O\left(b^{\lfloor C^*/\epsilon \rfloor + 1}\right)$$

(Recall that we're using the LATE VERSION – so the last power has a "$+1$").

- Space: same as time.

Uniform cost search (UCS): Performance IV

- Exercise. Suppose that an optimal goal is at depth d and the branching factor is b.
  - Does UCS expand only one state at depth d, more thae one state at depth d, all states at depth d?
  - Will any states at depth $> d$ be generated?
  - At worse, how many siblings of a goal state s is generated? (0?, 1? ...) How many children of the siblings of s are generated?

Uniform cost search (UCS): Fringe implementation I

- The UCS fringe is a priority queue prioritized by path cost; lower path cost = higher priority. Use min-heap.
- However, the min-heap insert for UCS is slightly different from usual min-heap: if node n is inserted into the fringe that already has node n' such that

$$\text{state of } n = \text{state of } n'$$

  then n overwrites n' if n has a higher priority than n'.
- Therefore:
  - Need a find operaton to locate a node (by its state).
  - Need to re-prioritize a node already in the priority queue.

Uniform cost search (UCS): Fringe implementation II

- METHOD 1. Use a min-heap.
  - Find: use linear search
  - Re-prioritize: use heapify-up

  Therefore
  - Time for insert $= O(n)$
  - Time for remove $= O(\lg n)$

  (n = size of min-heap = size of fringe)

Uniform cost search (UCS): Fringe implementation III

- METHOD 2. Use a min-heap and a hashtable. The purpose of the hashtable is to speed up find operation to locate the search node in the min-heap. In the hashtable, a (key, value) pair is:
  - key = state
  - value = the index in the min-heap array where the node (for that state) occurs.

  In this case:
  - Find: use hashtable
  - Re-prioritize: heapify-up and also update hashtable

  Therefore:
  - Time for insert = $O(\lg n)$
  - Time for remove = $O(\lg n)$ (need to remove from hashtable)

Uniform cost search (UCS): Fringe implementation IV

- Comments:
  - In METHOD 2, we use a hashtable to improve search for a search node in the min-heap (by state). Any data structure that speeds up linear search and has good runtime for insert, find, delete also works.
  - If we used a balanced tree, then the insert time would be $O(\lg n) + O(\lg n)$ (AVL tree search and heapify-up) compared against $O(1) + O(\lg n)$ (hashtable search and heapify-up), i.e., same runtime but less memory usage.

Uniform cost search (UCS) I

- Note that the path cost of n might be updated during the execution of UCS. Assuming the edge costs are ≥ 0, the path cost of n either stays the same or becomes smaller. Of course

$$g(n) \leq \text{path\_cost}(n)$$

- However assuming that all edge costs are ≥ 0, then when a node n is extracted from the fringe for expansion, the path cost of n in fact reaches $g(n)$

Uniform cost search (UCS) II

- What is the connection with Dijkstra's algorithm?
  https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=6&cad=rja&uact=8&ved=0ahUKEwjuisertYbOAhVhxoMKHamQBi0QFghFMAU&url=http%3A%2F%2Fwww.aaai.org%2Focs%2Findex.php%2FSOCS%2FSOCS11%2Fpaper%2Fdownload%2F4017%2F4357&usg=AFQjCNHdDJIJ1CuJDrpCS98R-5xmSTqgOg&bvm=bv.127521224,d.amc

Depth-Limited Search I

- **Depth–Limited Search (DLS)**: Search that is limited to states with depth $\leq$ a fixed constant. This is applied to depth-first type searches (DFS, BTS).
- Why?
  - BFS is complete and optimal (if branching factor is finite)
  - Depth-first type searchs (DFS, BTS) is complete if the max depth is finite. They are not optimal. If the maximum depth is ∞, then they are not guaranteed to halt.
  - But DFS and BTS use less memory.
  - Sometimes, finding a solution quickly is better than finding an optimal solution if it takes an astronomical amount of time (and memory) to find the optimal solution. .

Depth-Limited Search II

- If you know that a solution exists at depth $\leq D$ where D is a constant, then you can use depth-limit search (with DFS or BTS).
- In each search node keep a depth:
  - The root node has depth 0 (duh)
  - The depth of a node is the depth of the parent plus 1
- Modify your search algorithm so that it accepts a max depth argument. Once the depth of a search node has reached max depth, do not expand that search node.
- Related concept: Given a graph, the **diameter** of a graph is the length of the longest simple path in the graph.

Depth-Limited Search III

- TRICK 1: Instead of modifying the GRAPH-/TREE-SEARCH algorithm, you can also modify the fringe: The depth limit can be stored in the fringe and the fringe will reject a node (during insertion) if the depth if $>$ the depth limit. WIth this, the GRAPH-/TREE-search does not require change! You only need to subclass the relevant fringe class to get a depth-limited version.
- TRICK 2: Rewrite the GRAPH-/TREE-SEARCH algorithm so that it works with search nodes instead of states. In that case, for the pseudocode

```
for a running over all actions of state s:
    ...
```

Depth-Limited Search IV

we have

```
for a running over all actions of node n:
    ...
```

Again, you do not need to change the GRAPH-/TREE-SEARCH algorithm. You only need to subclass the relevant problem class so that the depth limit is stored in the problem object and the method for getting all actions for a state will return an empty list of actions if the depth of the node has reached the depth limit.

Depth-Limited Search V

- Exercise. John just read about depth–limited search for DFS and BTS. He had a brilliant idea: Why not use breadth–limited search for BFS? In other words, only expand up to the first B successors where B is a constant. Analyze this idea.

Iterative Deepening DFS I

- Perform depth-limited search for `max_depth` = 0,1,2,3,4,...
- Seems to be wasteful since process is repeated ... but ...
- How many nodes generated?

Depth	No. of nodes	No. of times each generated
1	b	d
2	b^2	$d-1$
3	b^3	$d-2$
$\vdots$	$\vdots$	$\vdots$
d	b^d	1

Iterative Deepening DFS II

Therefore the number of nodes generated is

$$bd + b^2(d-1) + \cdots + b^d(1) = O(b^d)$$

- Compare:
  - No. nodes generated for BFS $= O(b^{d+1})$
  - No. nodes generated for ID DFS $= O(b^d)$
- THE POINT: BFS generates notes at $d+1$ level; ID DFS does not.

Iterative Lengthening Searching I

- Same as iterative deepening DFS except you use path cost instead of depth

Bidirectional I

- Start two search, one starting with initial state and another from goal state using BFS.
- When they both reach a common node u, then you have a path from initial state to u and goal state to u.
- Patch the paths up to get a path of actions from initial to goal state
- Time and space $= O(b^{d/2})$
- Compare this to BFS $= O(b^{d+1})$

Comparison I

Criterion	BFS	Unif cost	DFS	Depth ltd	ID DFS	Bi-dir
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^{d+1})$	$O(b^{1+C^*/e})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{1+C^*/e})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

b = branching factor
d = depth of shallowest solution
m = max depth of search tree
l = depth limit
C^* = cost of optimal solution
e : each step cost $\geq e$

[a] complete if b finite
[b] complete if step costs $\geq e$ for $e \geq 0$
[c] complete if step costs identical
[d] if both directions use BFS