Uninformed Search: Problem-solving Agents & Path Planning

CS3243: Introduction to Artificial Intelligence – Lecture 2

16 January 2023

Contents

Reference: AIMA 4th Edition, Section 3.1-3.4

Administrative Matters

Tutorials & Tutorial 1

- Tutorials begin in Week 3 (i.e., next week)
  - Tutorial Worksheet 1 released today
    - Includes Tutorial Assignment 1 (TA1)
- Remember Tutorial Assessments
  - 5% participation
    - Tutors will explain expectations
  - 5% assignments
    - To be submitted DURING your Tutorial Session (e.g., TA1 in Week 1 – refer to schedule on Canvas)
    - Best 8 of 9 @ 0.625% each
- Tutorials not recorded
  - Unable to attend?
    - Email me + tutor valid excuse

Project 1

- Released in Week 3 (next week)
  - Forming informal discussion groups recommended (only to ideate)
  - DO NOT COPY/SHARE CODE
- Due in Week 6
  - Submission details in the Project 1 Problem Description
- Grader = CodePost
  - Requires you to register a CodePost account
    - Check your email
  - CodePost to be used to evaluate your code
    - Pass all test cases = full marks
- Support
  - Live Coding + Consultation: TBC

Diagnostic Quizzes

- Supports your learning
  - One DQ for each lecture
    - Recommended sequence: Lecture > DQ > Tutorials
  - Hypothesis: more valid questions attempted/reviewed ⇒ more robust understanding
    - Help foster Relational versus Instrumental understanding
    - Ask more questions about the relationships between various concepts
- Usage recommendations
  - Routines can help
    - Schedule a time each week to attempt them
    - Consistency has significant impact on learning + grades
  - Value in Peer Learning
    - Form a group to work on DQs + Tutorials
    - More people ⇒ more questions

Problem-Solving Agents

Recall from Lecture 1…

- Goal in AI → determine agent function *f*
  - $f : P \rightarrow a$
  - $a \in A$
- Key idea → AI as graph search
  - Each percept corresponds to a state in the problem (state → vertex)
  - Define the desired states → goals
  - After each action, we arrive at a new state (action → edge)
  - Construct a search space (graph)
  - Design and apply a graph search algorithm

(1) Define performance measure and search space

(2) Design search algorithm

Problem-solving Agent = Goal-based Agent

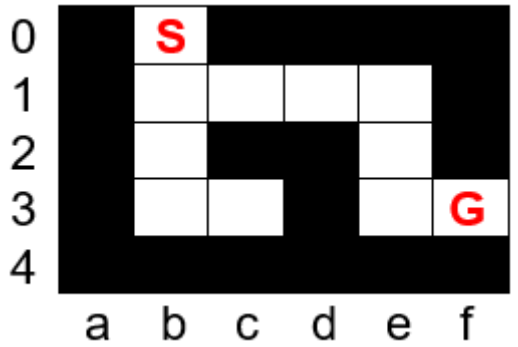
- Model Problem via Graph Representation (Search Space)
- Find a Path to a Goal State (via Algorithm)

Problem-solving Versus Reflex Agent?

- Reflex agent
  - Rules for limited cases
  - More cases → more rules (not scalable)
  - Narrow AI
- Problem-solving (goal-based) agent
  - Find path from initial to goal state (path planning)
  - Uses graph search algorithm
  - Able to find solutions to ANY problem case
    - Assumes standardised state representation
  - Relatively stronger AI

Problem-Solving Agent: Example Application

- Consider a Maze Puzzle problem
  - Layout known
  - Moves ←,↑,→,↓
  - Find path from **S** to **G**



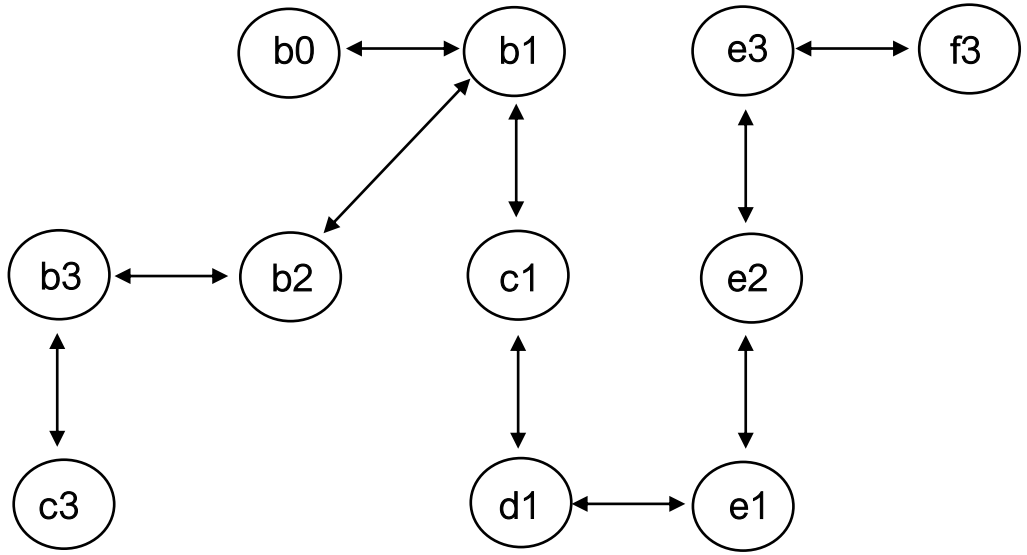
- Graph formulation
  - Vertices (states): positions in maze
  - Edges (actions): moves (e.g., b0 to b1 – i.e., ↓)

Summary of Adjacency Matrix

State	Actions
b0 (S)	b1
b1	b0, b2, c1
b2	b1, b3
b3	b2, c3
c1	b1, d1
c3	b3
d1	c1, e1
e1	d1, e2
e2	e1, e3
e3	e2, f3
f3 (G)	e3

Problem-Solving Agent: Example Application

- Resultant graph



- Solve using graph search algorithm

Summary of Adjacency Matrix

State	Actions
b0 (S)	b1
b1	b0, b2, c1
b2	b1, b3
b3	b2, c3
c1	b1, d1
c3	b3
d1	c1, e1
e1	d1, e2
e2	e1, e3
e3	e2, f3
f3 (G)	e3

Path Planning Problem Properties

- Assumed environment
  - Fully observable
  - Deterministic
  - Discrete
  - Episodic

Episodic interpretation?

- Complete information
- Fully deterministic
- Able to **PLAN** → look ahead at what to do
- Execute plan once defined

- Plan is formed sequentially
  - Each action in the plan impacts the next action in the plan
  - Development of the one plan has no bearing on the next → episodic problem

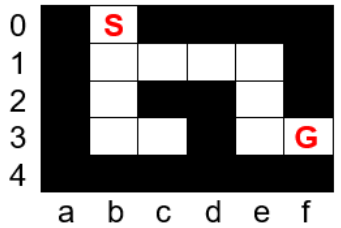
Search Spaces

Search Space Definition

- State representation, s_i
  - ADT containing data describing an instance of the environment
  - Initial state (s_0)
- Goal test, *isGoal*: $s_i \rightarrow \{0, 1\}$
  - Function that returns 1 if given state s_i is a goal state, else returns 0
- Actions, *actions*: $s_i \rightarrow A$
  - Function that returns the possible actions, $A = \{a_1, \ldots, a_k\}$, at a given state, s_i
- Action costs, *cost*: $(s_i, a_j, s_i') \rightarrow v$
  - Function that returns cost, v, of taking the action a_j at state s_i to reach state s_i'
  - Generally, assume costs ≥ 0

Search Space Definition

- Transition model, T: $(s_i, a_j) \rightarrow s_i$'
  - Function that returns the state transitioned to, s_i', when action a_j is applied at state s_i
- ***Generally applicable*** to many AI problems (with slight modifications)
- Actions / transition model / action costs functions
  - Modelling / representation → decisions for ***efficient search space generalisation***
  - Maze Problem example:



- Transition model:
  - ← = (r, c-1)
  - ↑ = (r-1, c)
  - → = (r, c+1)
  - ↓ = (r+1, c)
- State considerations:
  - Adjacency matrix, A, versus Obstacle hash table, O
  - O requires map ranges
  - Actions function references O or A for non-blocked moves
- Consider 10^3 by 10^3 grid with no obstacles
  - $|O| = 0$
  - $|A| = ?$

Assuming square grid of length n: $4\times2 + 4(n-2)\times3 + (n-2)^2\times4$

Using the Search Space Definition

- Use the search problem formulation
  - *actions*(*s*): returns set of actions, $A = \{a_1, \ldots, a_k\}$, available at state *s*
  - $T(s, a)$: returns the resultant state, *s*', given action *a* is taken at state *s*

Generalisation of the graph

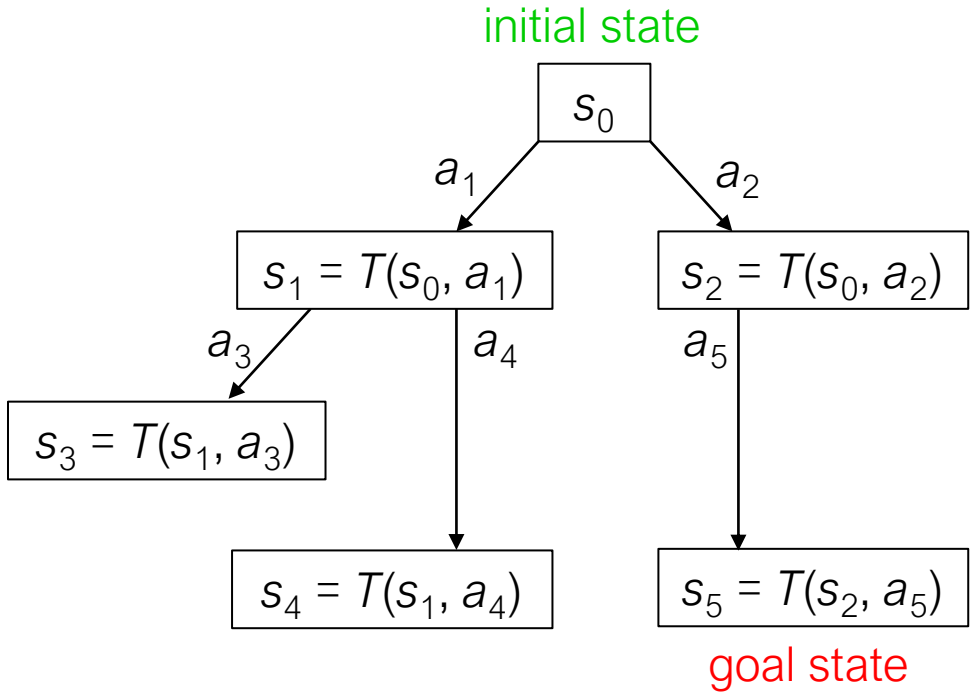
- Start at initial state s_0
- Consider each $a_i \in actions(s_0)$
- Repeat for each intermediate state $T(s_0, a_i)$
- Continue until the goal state is found

Assume:

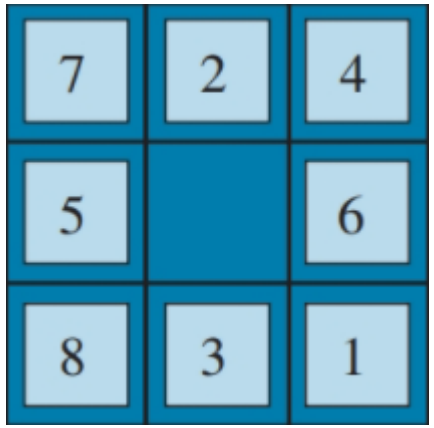
$actions(s_0) = \{a_1, a_2\}$ $\quad isGoal(s_5) = 1$

$actions(s_1) = \{a_3, a_4\}$ $\quad isGoal(s_i) = 0$, given $i = 1,2,3,4$

$actions(s_2) = \{a_5\}$ $\quad T(s_i, a_j) = s_j$



Another Example Problem: 8-Puzzle



Example Initial State

Goal State

- Puzzle requires the player to shift the numbered squares into the empty cell until the final pattern is obtained
- Search Problem Specification
  - State Representation (Initial State):
    - Matrix representing the grid, with each (r, c) ∈ {0-8}
    - 0 is the blank cell
  - Actions:
    - Move a chosen cell adjacent to the blank, (r, c) into the blank (r', c')
    - Same as moving blank cell (r', c') to a chosen cell adjacent (r, c)
      - i.e., legal (r'-1, c'), (r'+1, c'), (r', c-1), (r', c+1) cells correspond to possible actions
  - Goal Test:
    - Current state matrix = goal state matrix
  - Transition Model:
    - Swap the contents of (r, c) and (r', c')
  - Cost Function:
    - Each action cost 1 unit

Search Solutions

General Search Algorithm

```
frontier = {Node(initial state), NULL}
while frontier not empty:
        current = frontier.pop()
        if isGoal(current.state): return current.getPath()
        for a in actions(current.state):
                successor = Node(T(current.state, a), current)
                frontier.push(successor)
return failure
```

- Frontier
  - Part of the search space we are exploring
  - Current edge of the search tree
- Notice that each element of the frontier is a Node that must include
  - Referenced state *s*
  - Path that was taken to get to *s*
  - Frontier contains paths to check

States Versus Nodes

- State
  - A representation of the environment at some timestamp
- Node
  - Element in the frontier representing current path traversed
  - Includes the following information
    - State
    - Parent node – to track current path from initial state
    - Action
    - Path cost
    - Depth

} we will see why we need these later

Uninformed Search Algorithms

- Uninformed → no domain knowledge beyond search problem formulation
- Algorithm differences largely based on frontier implementation
  - Breadth-First Search (BFS): frontier = queue
  - Uniform-Cost Search (UCS): frontier = priority queue
  - Depth-First Search (DFS): frontier = stack
  - Depth-Limited Search (DLS): variation of DFS with max depth
  - Iterative Deepening Search (IDS): iterative version of DLS

Algorithm Criteria

- Efficiency
  - Time complexity
  - Space complexity

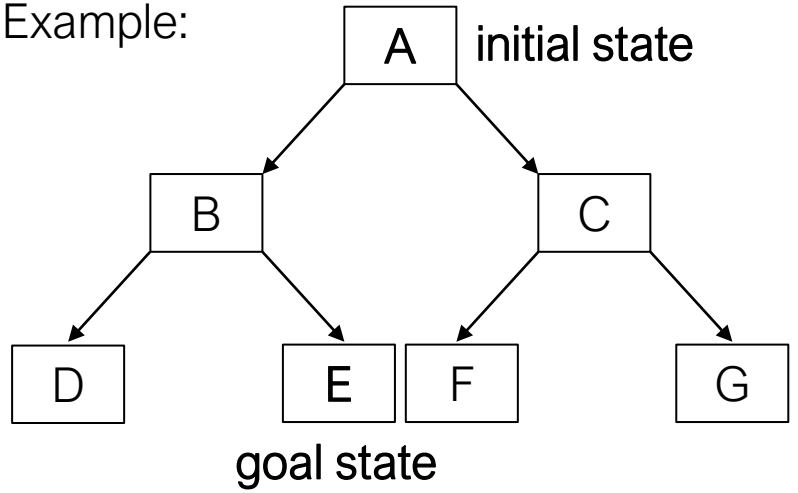
- Correctness
  - An algorithm is ***complete*** if it will find a solution when one exists and correctly report failure when it does not
  - An algorithm is ***optimal*** if it finds a solution with the lowest path cost among all solutions (i.e., path cost optimal)

Breadth-First Search (Recap)

Breadth-First Search (BFS) Algorithm: A Recap

- Frontier: Queue
- Time Complexity: $O(b^d)$
- Space Complexity: $O(b^d)$
- Complete: Yes[1]
- Optimal: No[2]

Example:



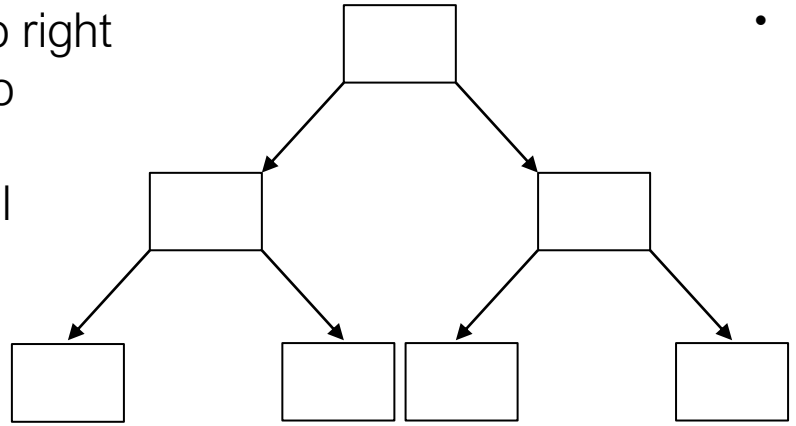
Tie-breaking: alphabetic order on push to frontier

Frontier Trace:
ITR1 = [A(-)]
ITR2 = [B(A), C(A)]
ITR3 = [C(A), D(A,B), E(A,B)]
ITR4 = [D(A,B), E(A,B), F(A,C), G(A,C)]
ITR5 = [E(A,B), F(A,C), G(A,C)]
ITR6 = DONE (A,B,E)

b: branching factor
d: depth of shallowest goal
1: if (i) finite *b* AND (ii) finite state space OR contains solution
2: optimal if costs uniform (and some other cases)

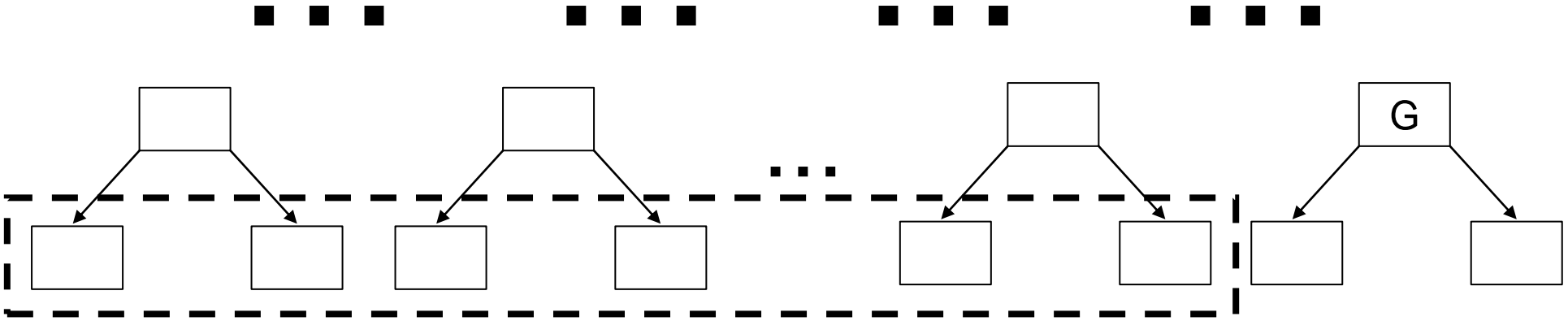
BFS: A Simple Improvement

- Assume traversal from left to right
- Will add all denoted nodes to frontier before checking G
- Nodes on level $d \geq$ sum of all nodes in previous levels assuming $b \geq 2$



- Performing goal test on pushing to frontier instead of popping from frontier will prevent this with no change in the BFS solution

Early Goal Test (as opposed to the original ***Late Goal Test***) – from this point assume BFS to use early goal test (unless otherwise stated)



Questions about the Lecture?

- Was anything unclear?
- Do you need to clarify anything?

- Ask on Archipelago
  - Specify a question
  - Upvote someone else's question

Invitation Link (Use NUS Email --- starts with E)
https://archipelago.rocks/app/resend-invite/22846952979

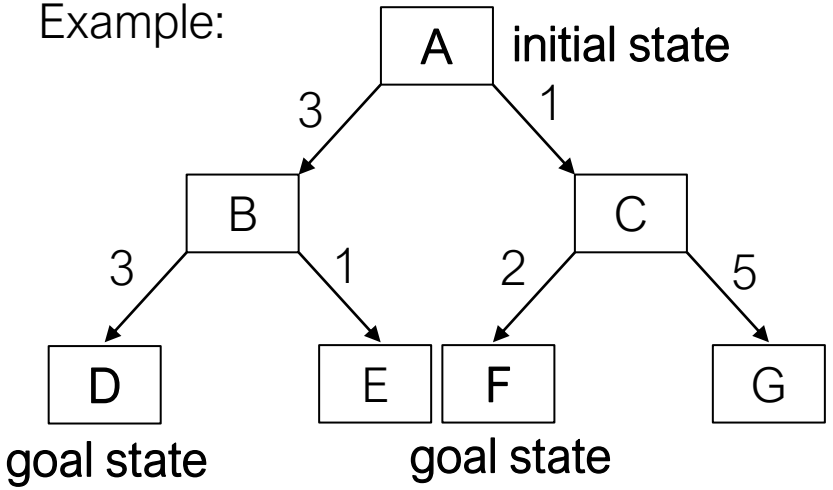
Uniform-Cost Search

Uniform-Cost Search (UCS) Algorithm: A Recap

UCS is basically Dijkstra's Algorithm

- Frontier: Priority Queue[1]
- Time Complexity: $O(b^e)$
- Space Complexity: $O(b^e)$
- Complete: Yes[2]
- Optimal: Yes

Example:



Frontier Trace:

ITR1 = [A((-),0)]

ITR2 = [C((A),1), B((A),3)]

ITR3 = [B((A),3), F((A,C),3), G((A,C),6)]

ITR4 = [F((A,C),3), E((A,B),4), D((A,B),6), G((A,C),6)]

ITR5 = DONE (A,C,F)

Tie-breaking: nodes ordered alphabetically when priority is the same

Note:
Updating path cost from each node is O(1) since we store the current path cost

1: prioritising lower path cost, g(n), where g(n) = path cost of the path taken to reach n

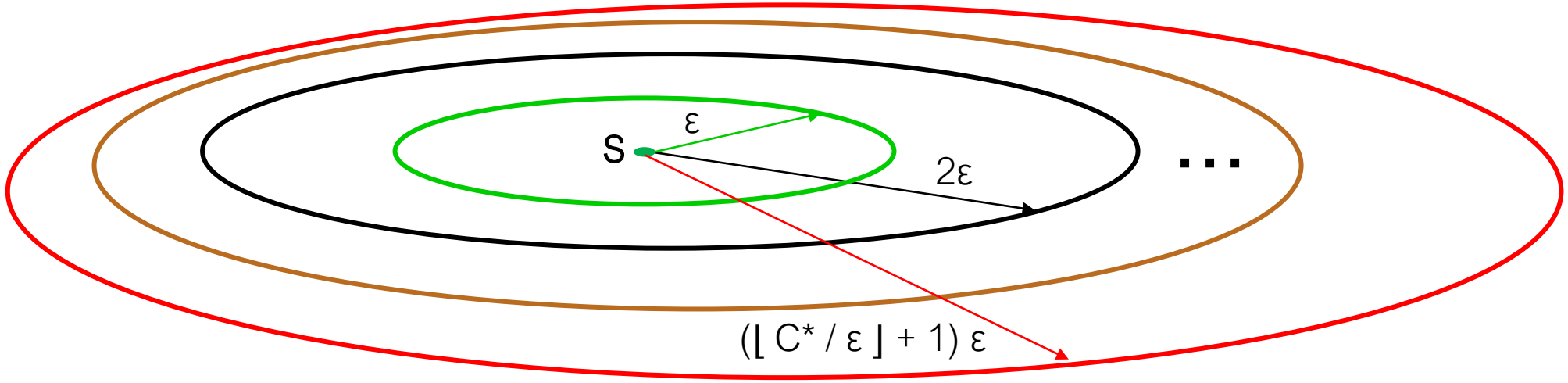
b: branching factor

e: $1 + \lfloor C^* / \varepsilon \rfloor$, where C^* is the optimal path cost and ε is some small positive constant

2: requires same completeness criteria as BFS and that actions costs are $> \varepsilon > 0$

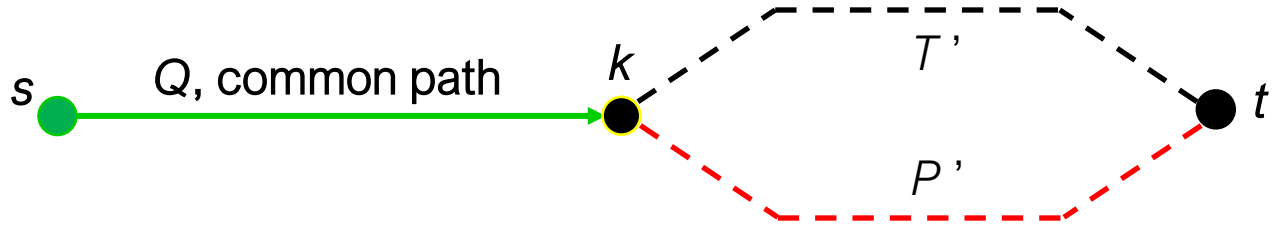
Why $O(b^e)$ Complexity for UCS?

- UCS explores all paths radiating from the initial state
- UCS explores paths in increments of ε (smallest action cost)
  - With each *step* it extends paths by at least ε (from the initial state)
  - Considers paths with cost 0 (initial state only), then cost ε, then 2ε, etc.
  - Expected to reach goal in ⌊ C* / ε ⌋ + 1 steps, where C* is the optimal path cost



Why is UCS Optimal? A General Idea

- UCS traverses paths in order of path cost
  - This is because path costs from the initial state are always increasing (given ε)
    - i.e., whenever a node, *n*, is added to a path, *P*, the new path, *P*' must have a path cost that is at least ε greater than the past cost of *P*
- UCS finds the optimal path to each node
  - Suppose UCS outputs path *P* = *Q* + *P* ' as the solution for *s* to *t*
  - Suppose the optimal path from *s* to *t* is instead *T* = *Q* + *T* '



  - UCS must skip shorter paths between *k* and *t* for it to have chosen *P*, which is a contradiction since it always chooses shorter paths to explore first

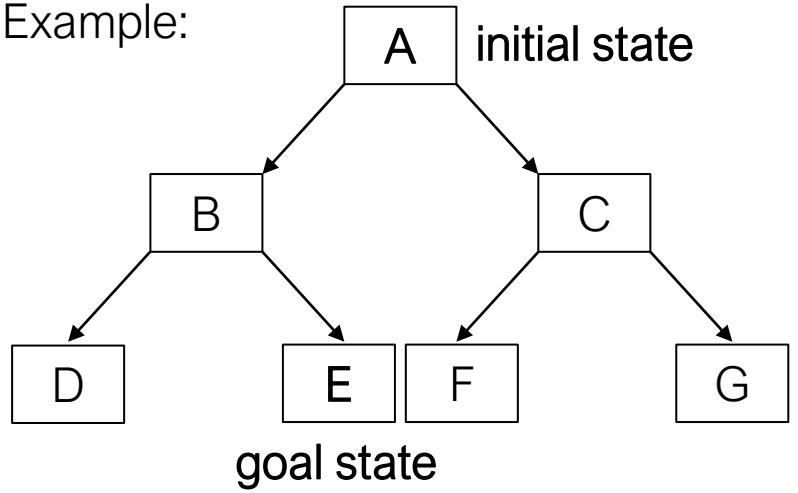
Depth-First Search (Recap)

Depth-First Search (DFS) Algorithm: A Recap

- Frontier: Stack
- Time Complexity: $O(b^m)$
- Space Complexity: $O(bm)$
- Complete: No
- Optimal: No

b: branching factor
m: maximum depth

Example:



Tie-breaking: reverse alphabetic order on push to frontier

Frontier Trace:
ITR1 = [A(-)]
ITR2 = [B(A), C(A)]
ITR3 = [D(A,B), E(A,B), C(A)]
ITR4 = [E(A,B), C(A)]
ITR5 = DONE (A,B,E)

Why is DFS incomplete even under the same assumptions of completeness for BFS?

DFS may be incomplete even if a solution exists

Note:
Space efficiency may be improved to O(*m*) by simply backtracking – i.e., tracing back to parent and last action taken (assuming fixed order of actions – recall that we store parent node and action taken at parent)

Depth-Limited & Iterative Deepening Search

Depth-Limited Search (DLS)

- DFS with a depth limit, ℓ
  - Search only up to depth ℓ
  - Assume no actions may be taken from nodes at depth ℓ

- Same guarantees as DFS with ℓ in place of m
  - Time complexity: $O(b^{\ell})$
  - Space complexity: $O(b\ell)$
  - Complete: No
  - Optimal: No

Iterative Deepening Search

- Idea: use DLS iteratively, each time increasing ℓ by 1
  - Will be completely search based on depth
  - Completeness of BFS with space complexity of DFS
- Overheads: will rerun top levels many times
  - Assuming branching factor *b* and depth ℓ,nodes generated by DLS:
    - $O(b^0) + O(b^1) + O(b^2) + \cdots + O(b^{\ell-2}) + O(b^{\ell-1}) + O(b^{\ell})$
  - Nodes generated by IDS to depth *d* with branching factor *b*:
    - $(d+1)O(b^0) + dO(b^1) + (d-1)O(b^2) + \cdots 3O(b^{d-2}) + 2O(b^{d-1}) + O(b^d)$

Iterative Deepening Search

- Example, b = 10 and d = 5
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$
  - Overhead $\approx 11\%$
- IDS properties
  - Time: $O(b^d)$
  - Space: $O(bd)$
  - Complete: Yes (b finite and d finite or contains solution) – same as BFS
  - Optimal: No (optimal if costs uniform (and some other cases)) – same as BFS

Summary

- Performance of search algorithms

Criterion	BFS	UCS	DFS	DLS	IDS
Complete?	Yes[1]	Yes[1,2]	No[3]	No	Yes[1]
Optimal Cost?	Yes[4]	Yes	No	No	Yes[4]
Time	$O(b^d)$	$O(b^{1 + \lfloor C^* / \varepsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1 + \lfloor C^* / \varepsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$

1. Complete if *b* finite and either has a solution or *m* finite
2. Complete if all actions costs are $> \varepsilon > 0$
3. DFS is incomplete unless the search space if finite – i.e., when *b* finite and *m* finite
4. Cost optimal if action costs are all identical (and several other cases)

- Recall that an Early Goal Test is assumed for BFS
- UCS must perform a Late Goal Test to be optimal (this also accounts for the +1 in the index of its complexity)
- DFS is not complete (even under 1) as even if a solution exists, it may infinitely traverse a path without a solution (note the "or")
- DFS space complexity may be improved to $O(m)$ with backtracking (similar for DLS and IDS)

Tree Search Versus Graph Search

Cycles & Redundant Paths

- Cycle → cyclic graph
  - Infinite loops (incomplete)
  - May greatly increase necessary computation
- Redundant path to s_i → more expensive paths from s_0 to s_i
  - Should not consider these if optimality is required
- Typical practice → graph-search implementation
  - Maintain a *reached* (or *visited*) hash table
  - Add reached states
  - Only add new node to *frontier* and *reached* if
    - state represented by node not previously reached
    - path to state already reached is cheaper than one stored

Alternative → tree search implementation
– Allows revisits
– All previous analysis assumed tree search

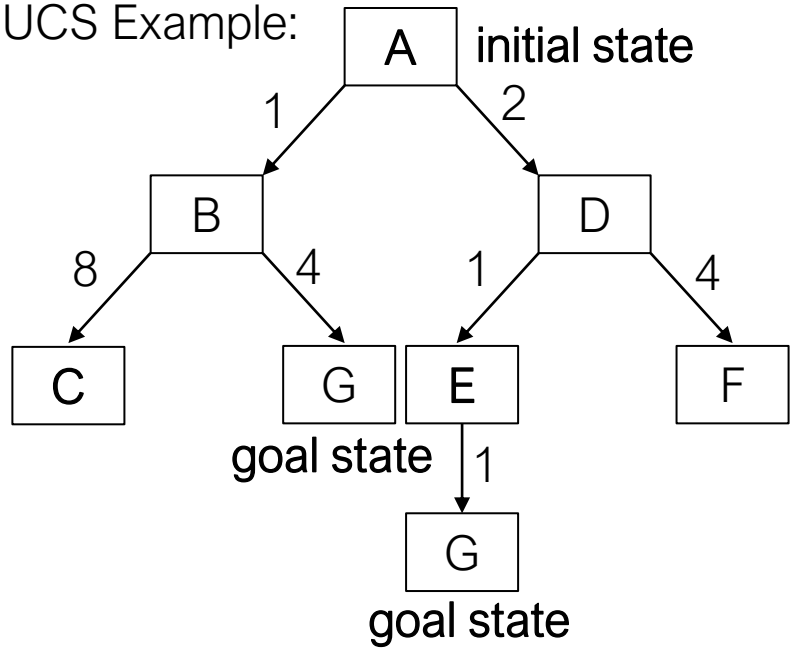
Graph Search Algorithm (Version 1)

```
frontier = {Node(initial state, NULL)}
reached = {initial state: Node(initial state)}
while frontier not empty:
        current = frontier.pop()
        if isGoal(current.state): return current.getPath()
        for a in actions(current.state):
                successor = Node(T(current.state, a), current)
                if successor.state not in reached:
                        frontier.push(successor)
                        reached.insert(successor.state: successor)
return failure
```

- Reached (sometimes also labelled: Visited)
  - Hash table that tracks all nodes already reached or visited via prior searching
- This version ensures that nodes are never revisited
  - Omits all redundant paths
  - May omit optimal path

Graph Search Algorithm (Version 1)

UCS Example:



Tie-breaking: nodes ordered alphabetically when priority is the same

	Frontier:	Reached:
ITR1 =	[A((-),0)]	[A]
ITR2 =	[B((A),1), D((A),2)]	[A,B,D]
ITR3 =	[D((A),2), G((A,B),5), C((A,B),9)]	[A,B,C,D,G]
ITR4 =	[E((A,D),3), G((A,B),5), F((A,D),6), C((A,B),9)]	[A,B,C,D,E,F,G]
ITR5 =	[G((A,B),5), F((A,D),6), C((A,B),9]	[A,B,C,D,E,F,G]
ITR6 =	DONE (A,B,G) **Non-optimal path!**	

Graph Search Algorithm (Version 2)

```
frontier = {Node(initial state, NULL)}
reached = {initial state: Node(initial state)}
while frontier not empty:
        current = frontier.pop()
        if isGoal(current.state): return current.getPath()
        for a in actions(current.state):
                successor = Node(T(current.state, a), current)
                if successor.state not in reached or
                   successor.getCost() < reached[successor.state].getCost():
                        frontier.push(successor)
                        reached.insert(successor.state: successor)
return failure
```

- More relaxed constraint on paths that are considered
  - Also considers paths with lower path cost

Graph Search Properties

- Performance under graph search implementations

Criterion	BFS	UCS	DFS	DLS	IDS
Complete?	Yes[1]	Yes[1,2]	No[3]	No	Yes[1]
Optimal Cost?	Yes[4]	Yes	No	No	Yes[3]
Time	$O(\|V\| + \|E\|)$				
Space					

1. Complete if *b* finite and either has a solution or *m* finite
2. Complete if all actions costs are $> \varepsilon > 0$
3. DFS is incomplete unless the search space is finite – i.e., when *b* finite and *m* finite
4. Cost optimal if action costs are all identical (and several other cases)

For CS3243, assume graph search Version 1 is used unless otherwise mentioned

- Time and space complexities are now bounded by the size of the search space - i.e., the number of vertices (nodes) and edges, $|V| + |E|$
- Note that we ***do not*** need to check for cheaper paths under graph search for BFS and DFS since costs play no part in those algorithms and they cannot guarantee an optimal solution anyway

Questions about the Lecture?

- Was anything unclear?
- Do you need to clarify anything?

- Ask on Archipelago
  - Specify a question
  - Upvote someone else's question

Invitation Link (Use NUS Email --- starts with E)
https://archipelago.rocks/app/resend-invite/22846952979