Uninformed Search: Problem-solving Agents & Path Planning

CS3243: Introduction to Artificial Intelligence – Lecture 2

17 January 2022

Contents

Administrative Matters

Upcoming…

- Tutorials
  - Increased to 16 classes
  - Begin next week (i.e., Week 3)
  - Face-to-face in SR-2

- Deadlines
  - DQ0, DQ1 (released last Monday)
    - *Due this Friday (21 January), 2359 hrs*
  - DQ2 (released today)
    - *Due this Friday (21 January), 2359 hrs*
  - TA1 (released today)
    - *Due this Sunday (23 January), 2359 hrs*
    - *Refer to the tutorial assignment instructions document on LumiNUS*

Project Poll

Grade distribution

- 20% competitive - i.e., driven by performance of the solutions submitted by your peers
  - Specifically, percentage_obtained = 20 * [(total_students - (your_rank - 1)) / total_students]
  - Ranks based on 1224 ranking system
- 30% tough cases - i.e., some optimisations would be required to clear
- 50% applications on standard difficulty - i.e., basic implementations

24%

Note that only the 20% component is different between the two options.

Version 1:
20% on competitive component

24%

156 responses

Grade distribution

- 20% hidden test cases - i.e., hidden cases requiring several optimisations
- 30% tough cases - i.e., some optimisations would be required to clear
- 50% applications on standard difficulty - i.e., basic implementations

76%

Note that only the 20% component is different between the two options.

Version 2:
20% on hidden test cases

76%

Administrative Questions Before We Begin?

- Any pressing questions about the administrative matters?

- Channels
  - Verbally on Zoom
  - On Archipelago
  - Via Zoom Chat

OR https://archipelago.rocks/app/resend-invite/54568232609

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

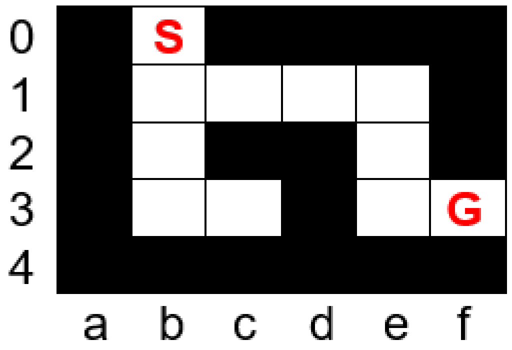
(1) Define performance measure and search space

(2) Design search algorithm

Goal-based Agent

Problem-Solving Agent: Example Application

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



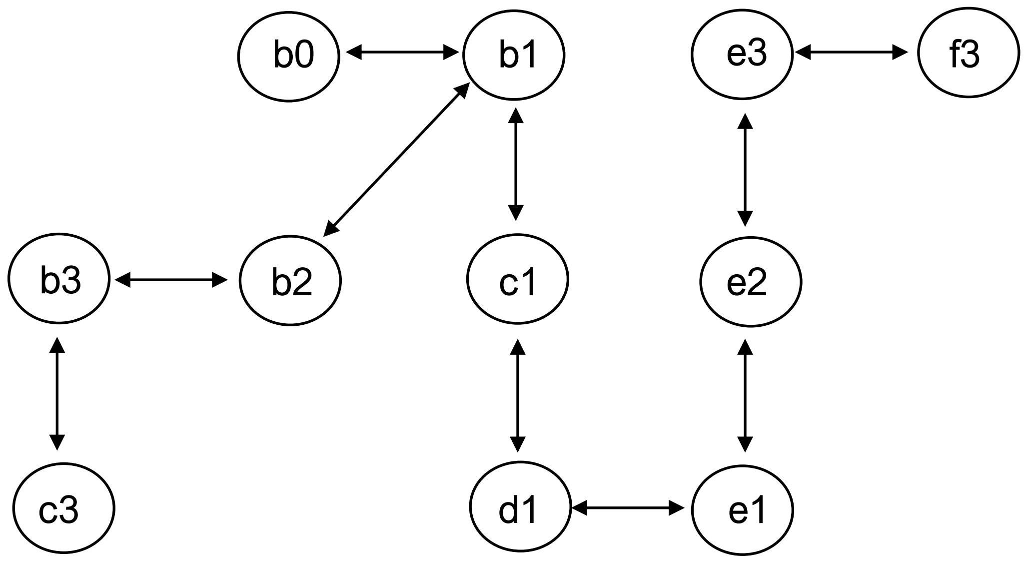
- Graph formulation
  - Vertices (states): positions in maze
  - Edges (actions): moves (e.g., b0 to b1 – i.e., ↓)

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



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

- Solve using graph search algorithm

Path Planning Problem Properties

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

Episodic interpretation?

- Static + complete information
- Taking actions only changes your position
- 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
  - Potential *search space generalisation*
  - Example
    - Transition model:
      - $\leftarrow$ = (r, c-1)
      - $\uparrow$ = (r-1, c)
      - $\rightarrow$ = (r, c+1)
      - $\downarrow$ = (r+1, c)
    - Obstacle hash table, O (assuming less obstacles than valid adjacency cells)
    - Map dimensions
    - Actions function determines non-blocked moves
    - Consider 10^3 by 10^3 grid with no obstacles
      - Adj. matrix size 10^6
      - $|O| = 0$

Search Solutions

General Search Algorithm

```
frontier = {initial state} // frontier is a data structure
while frontier not empty:
      current = frontier.pop()
      if isGoal(current) return path found
      for a in actions(current):
            frontier.push(T(current, a))
return failure
```

- Frontier
  - Part of the search space we are exploring
  - Current edge of the search tree
- Note that each element of the frontier must include
  - Referenced state *s*
  - Path that was taken to get to *s*

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)

Questions on the Lecture so far?

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

- Channels
  - Verbally on Zoom
  - On Archipelago
  - Via Zoom Chat

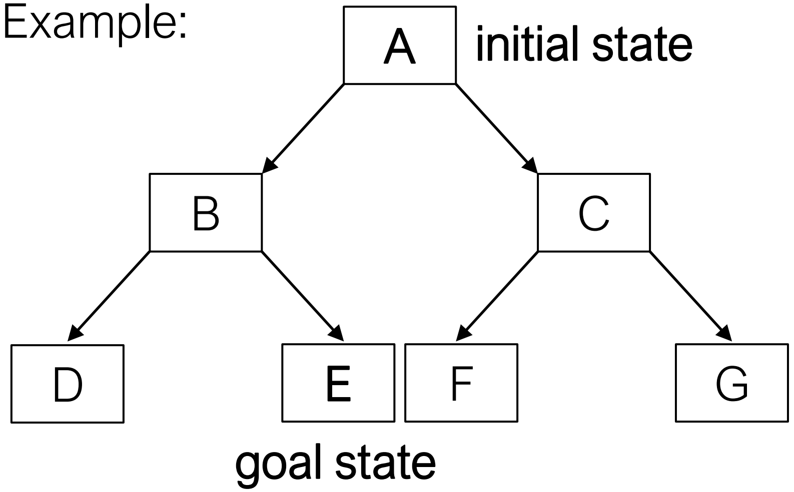
OR https://archipelago.rocks/app/resend-invite/54568232609

Breadth-First Search

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:



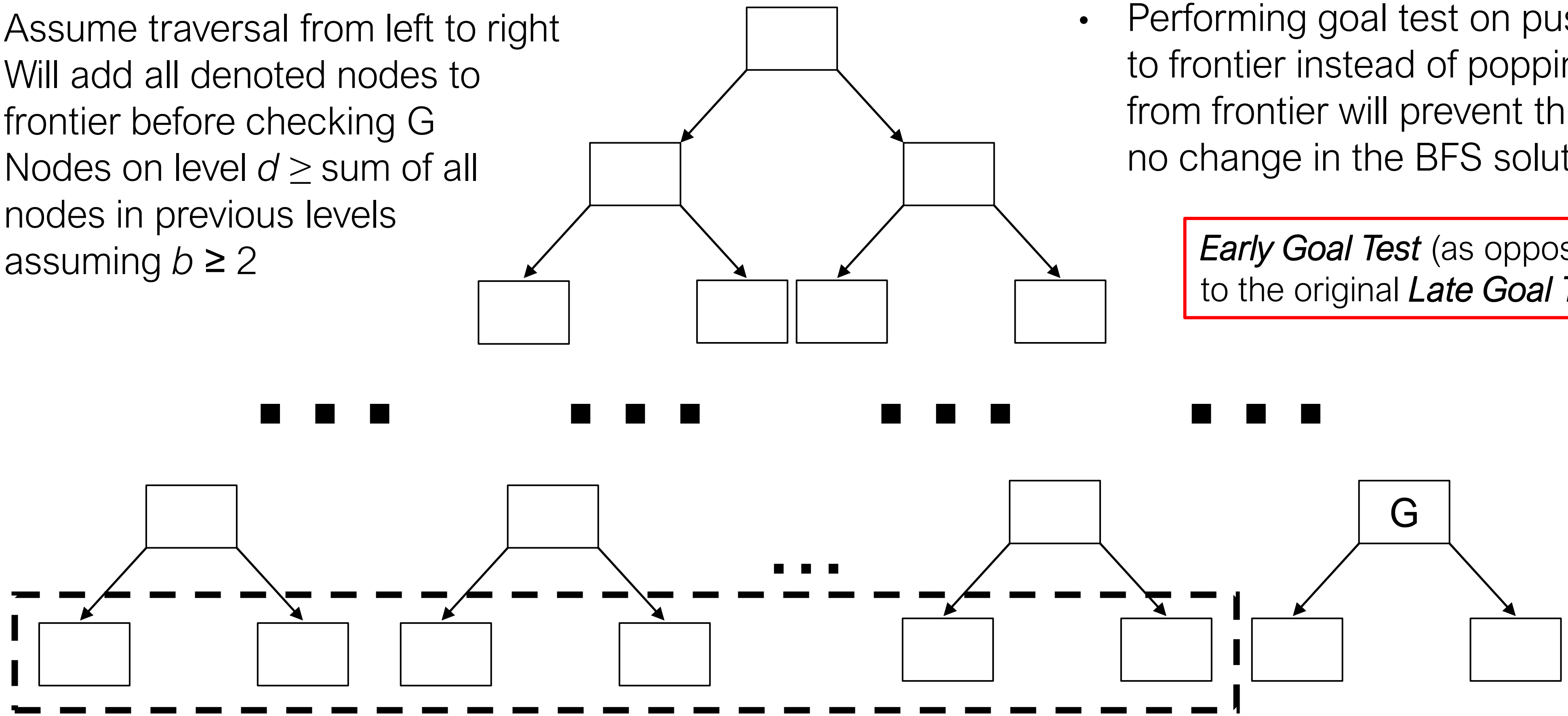
Tie-breaking: alphabetic order on push to frontier

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) b finite AND (ii) state space finite 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*)

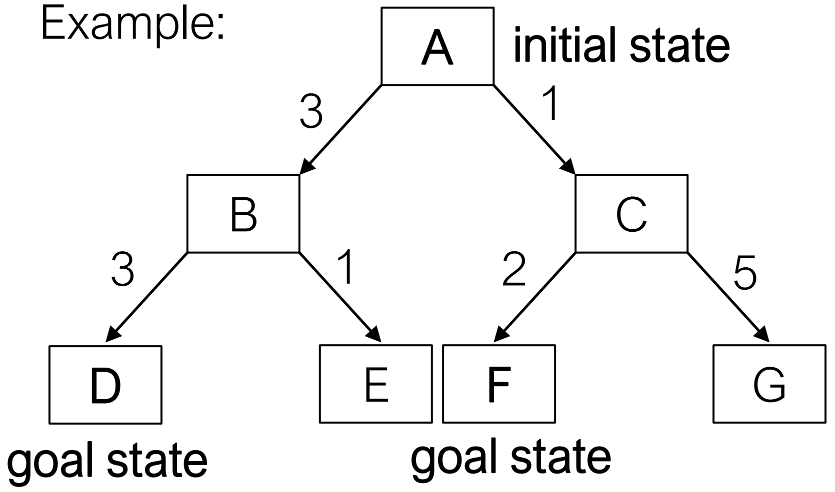
Uniform-Cost Search

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)

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

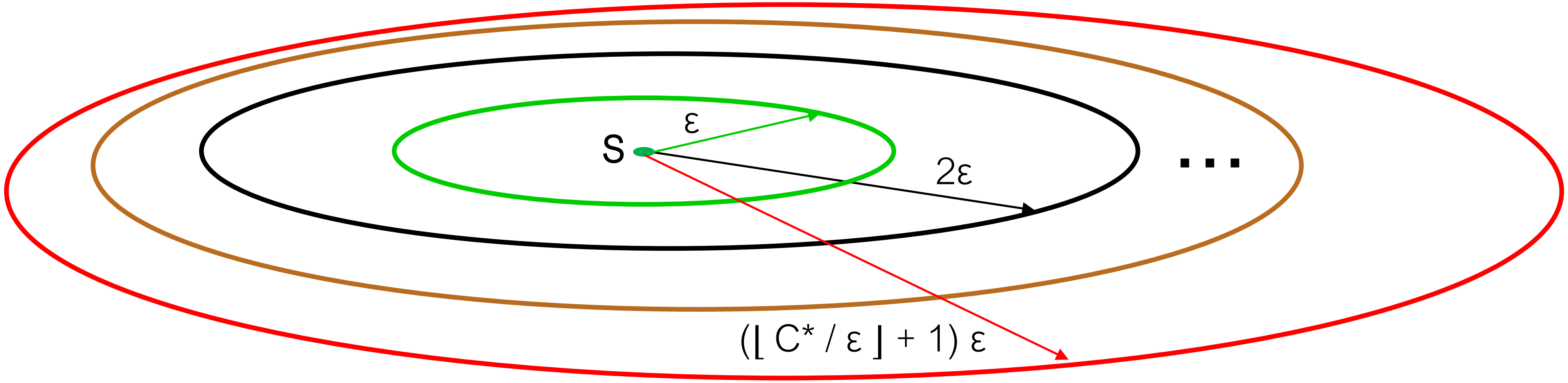
b: branching factor
e: 1 + ⌊ C* / ε ⌋, where C* is the optimal path cost and ε is some small positive constant
2: requires same completeness criteria as BFS and that actions costs are > ε > 0

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

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

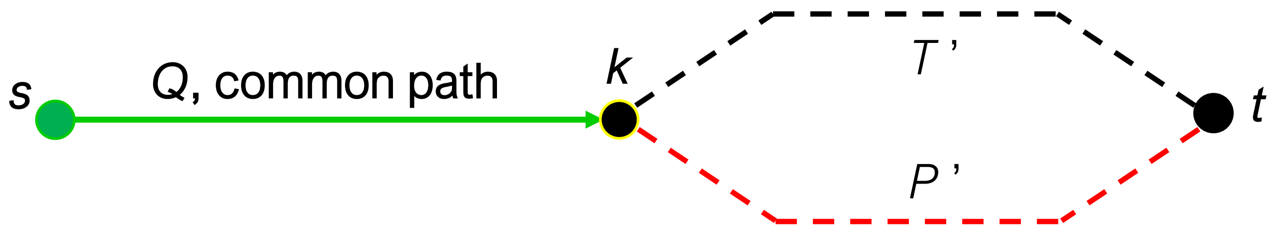
- UCS explores all paths radiating from the initial state
- UCS explores paths in increments of ε (smallest action cost)
  - With each *step* it extends paths by at least ε (from the initial state)
  - Considers paths with cost 0 (initial state only), then cost ε, then 2ε, etc.
  - Expected to reach goal in $\lfloor C^* / \varepsilon \rfloor + 1$ 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

Depth-First Search (DFS) Algorithm: A Recap

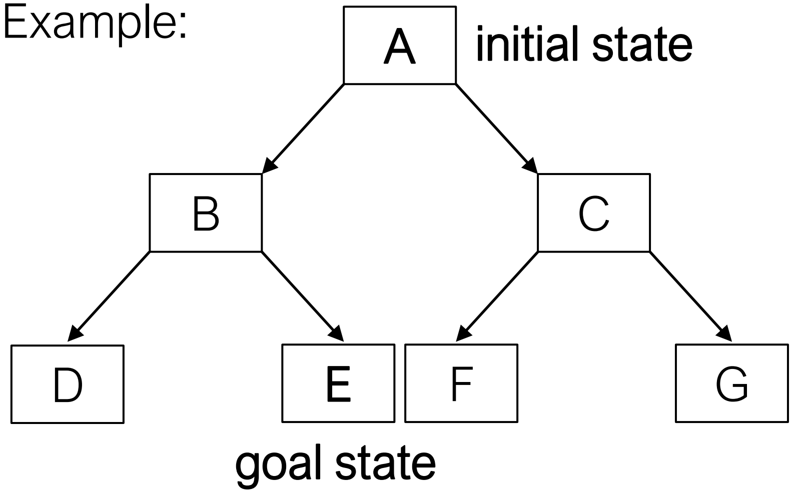
- Frontier: Stack
- Time Complexity: $O(b^m)$
- Space Complexity: $O(bm)$
- Complete: No
- Optimal: No

b: branching factor
m: maximum depth

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

DFS might get caught in a cycle

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)

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 under tree-search

Criterion	BFS	UCS	DFS	DLS	IDS
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]
Optimal Cost?	Yes[3]	Yes	No	No	Yes[3]
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 state space either finite or has a solution
2. Complete if all actions costs are > ε > 0
3. Cost optimal if action costs are all identical (and several other cases)

- Recall that an Early Goal Test on BFS may improve runtime practically
- 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 it might get caught in a cycle
- 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 use these in solution if optimality is required
- Typical practice → graph-search implementation
  - Maintain a *reached* hash table
  - Add nodes corresponding to each state reached
  - 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 is tree-search implementation → allow revisits (all we reviewed earlier was done under tree-search)

Graph-Search Implementations

- Performance under graph-search

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

1. Complete if b finite and state space either finite or has a solution
2. Complete if all actions costs are > ε > 0
3. Cost optimal if action costs are all identical (and several other cases)

- DFS under graph search is complete, assuming a finite state space
- Time and space complexities are now bounded by the size of the state space - i.e., the number of vertices and edges, $|V| + |E|$
- Note that we ***do not*** need to update under BFS and DFS since costs play no part in algorithm and they cannot guarantee an optimal solution anyway

Questions on the Lecture?

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

- Channels
  - Verbally on Zoom
  - On Archipelago
  - Via Zoom Chat

OR https://archipelago.rocks/app/resend-invite/54568232609