Uninformed Search:
Problem-Solving Agents & Path Planning

CS3243: Introduction to Artificial Intelligence – Lecture 2

NUS | Computing
National University of Singapore

1 Administrative Matters

Tutorials & Tutorial 1

- Tutorials begin next week (i.e., Week 3)
  - Tutorial Worksheet 1 released after this lecture
    - Includes Tutorial Assignment 1 (TA1)

- Tutorials not recorded
  - Unable to attend?
    - Email me and your tutor with a valid excuse (to avoid losing the 0.5% for attendance)

Assignments Relating to Tutorials

- Remember tutorial-related assessments
  - 5% course participation
    - Attendance (0.5% per tutorial – i.e., you must attend at least 10 of 11 to get the maximum mark)
  - 5% tutorial assignments
    - Graded (1% each – best 5 of the 9 tutorial assignments)
    - Pre-tutorial submission for Assignment 1 – due this Sunday, 25 AUG, 2359 hrs via Canvas
      - Only need to submit assignment question (i.e., solution for 1 question)
      - Cannot be blank; must demonstrate that a reasonable attempt was made
    - Post-tutorial submission for Assignment 1 – due next Friday, 30 AUG, 2359 hrs via Canvas
      - No need to submit this if you are happy with your pre-tutorial submission

Project 1.1 & 1.2

- Both released in Week 3
  - Forming informal discussion groups recommended (only to ideate)
  - DO NOT COPY / SHARE CODE
- Project 1.1 due Sunday of Week 5; Project 1.2 due Sunday of Week 6
  - Submission details in the Project 1.1 & 1.2 Problem Descriptions
- Grader = Coursemology
  - Requires you to register a Coursemology account; check your NUS email for the invitation
  - More information can be found in the Coursemology Guide – will be available on Canvas
- Support
  - Use the Discussion Forums on Canvas
  - Project Consultations: TBC

Contents

2 Problem-Solving Agents

Recall from Lecture 1…

- Goal in AI → determine agent function `f`
  - `f : P → a`
  - `a ∈ A`

Problem-solving Agent → Goal-based Agent
- Model problem via graph representation (i.e., define a search space)
- Find a path to a goal state (via algorithm)

- 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

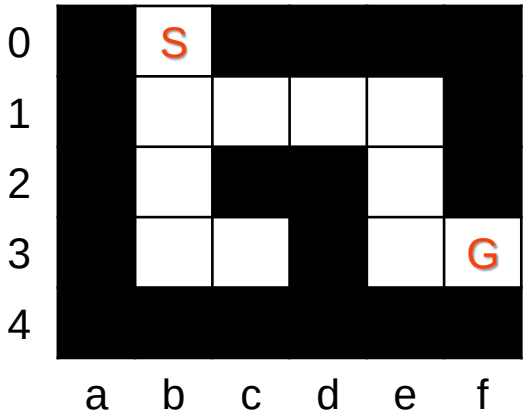
(2) Design search algorithm

Problem-solving 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 solution to ANY problem case
    - Assumes generalisable problem
      - e.g., standard state representation
  - Relatively stronger AI

Problem-solving Agent: Example Application

- Consider a Maze Puzzle problem
  - Layout known (but may have to solve different mazes)
  - Moves: ←,↑,→,↓
  - Find path from S to G
- Graph (search space) formulation
  - Vertices (states): positions in maze
  - Edges (actions): moves (e.g., b0 to b1 – i.e., ↓)

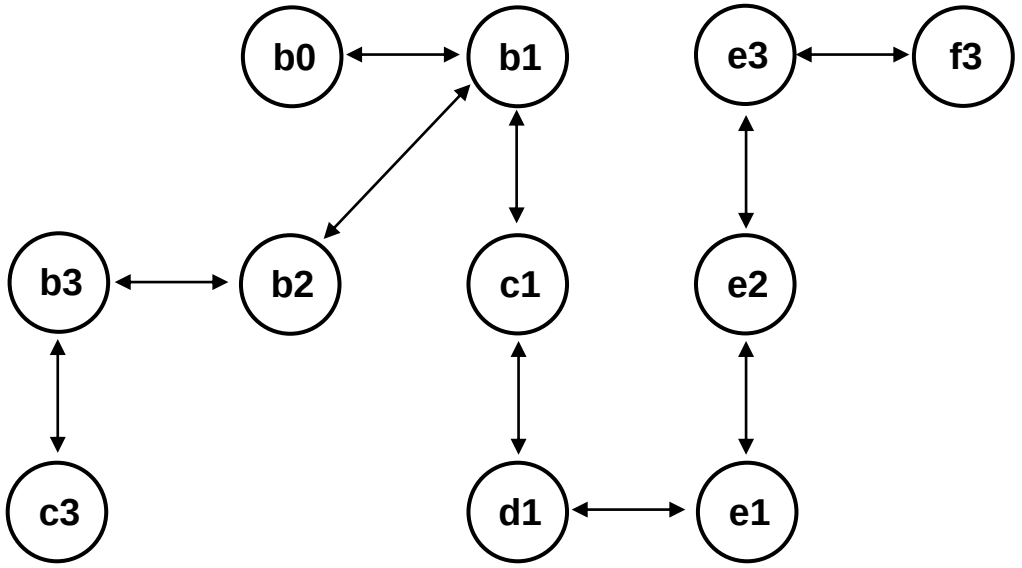


Corresponding Adjacency List

State	Possible 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 (search space)



- Solve using graph search algorithm

Corresponding Adjacency List

State	Possible 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?
  - Have complete information
  - 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
  - Agent would see new problem (e.g., new maze), formulate a plan, and execute that plan

3 Search Spaces

Search Space Definition

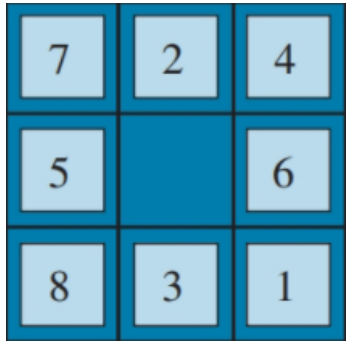
- State representation, s_i
  - Abstract data type (ADT) containing data describing an instance of the environment
  - Initial state (s_0)
- Actions, $actions : s_i \rightarrow A$
  - Function that returns the possible actions, $A = \{a_1, \ldots, a_k\}$, at a given state s_i
- Transition model, $T : (s_i, a_j) \rightarrow s_i'$
  - Function that returns the intermediate state transitioned to, s_i', when action a_j is applied at state s_i

Search Space Definition

- Goal test, `isGoal : $s_i$ → {0, 1}`
  - Function that returns `1` if given state s_i is a goal state, else returns `0`
- Action costs, `cost : ($s_i$, $a_j$, $s_i'$) → v`
  - Function that returns cost, `v`, of taking the action a_j at state s_i to reach the intermediate state s_i'
  - Generally, assume costs ≥ 0 (unless otherwise stated)

- Generally applicable to many AI problems (with slight modifications)
- A search space definition corresponds to a model of the problem
  - Model / representation → decisions about *efficient search space generalisation*

Example Problem: 8-Puzzle



Example Initial State

Goal State

- 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 `(r, c)` to the blank cell `(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 → 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 is 1 unit

This Puzzle requires the player to shift the numbered squares into the empty cell until the final pattern is obtained

Search Space Definition → Search Tree

- Use the search problem formulation
  - `actions(s)`: returns set of actions, `A = {a1, …, ak}`, available at state `s`
  - `T(s, a)`: returns the resultant state, `s'`, given action `a` is taken at state `s`
- Assume the following
  - `actions(s0) = {a1, a2}`
  - `actions(s1) = {a3, a4}`
  - `actions(s2) = {a5}`
  - `isGoal(s5) = 1`
  - `isGoal(si) = 0`, given `i = 1,2,3,4`
  - `T(si, aj) = sj`

Search Space Definition → Search Tree

- Use the search problem formulation
  - `actions(s)`: returns set of actions, `A = {a1, …, ak}`, 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

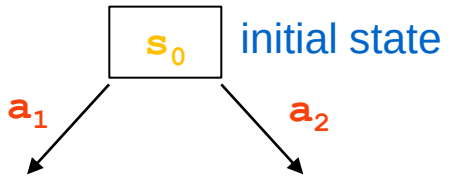
- Assume the following
  - $actions(s_0) = \{a_1, a_2\}$
  - $actions(s_1) = \{a_3, a_4\}$
  - $actions(s_2) = \{a_5\}$
  - $isGoal(s_5) = 1$
  - $isGoal(s_i) = 0$, given $i = 1,2,3,4$
  - $T(s_i, a_j) = s_j$

s_0 initial state

Search Space Definition → Search Tree

- Use the search problem formulation
  - `actions(s)`: returns set of actions, `A = {a1, …, ak}`, 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 $a_i \in actions(s_0)$

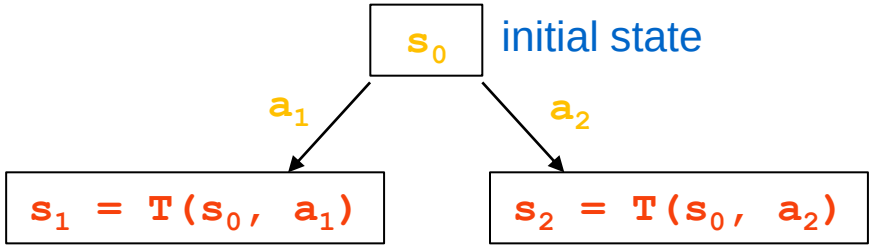
- Assume the following
  - $actions(s_0) = \{a_1, a_2\}$
  - $actions(s_1) = \{a_3, a_4\}$
  - $actions(s_2) = \{a_5\}$
  - $isGoal(s_5) = 1$
  - $isGoal(s_i) = 0$, given $i = 1,2,3,4$
  - $T(s_i, a_j) = s_j$



Search Space Definition → Search Tree

- Use the search problem formulation
  - `actions(s)`: returns set of actions, `A = {a1, …, ak}`, 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 $T(s_0, a_i)$

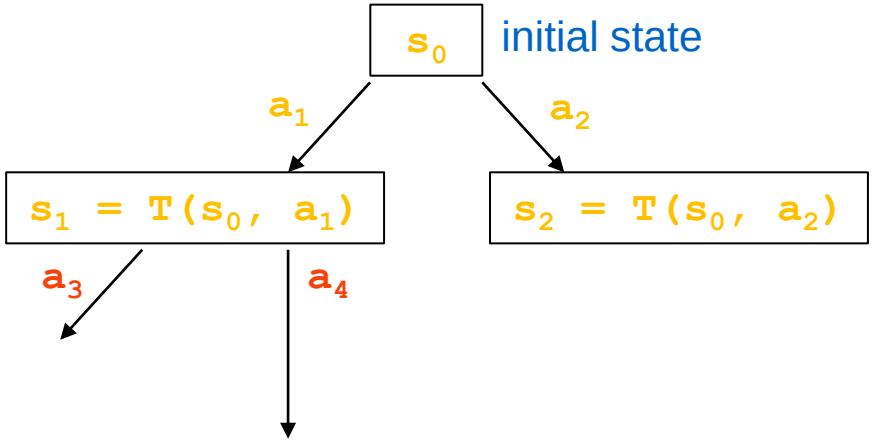
- Assume the following
  - $actions(s_0) = \{a_1, a_2\}$
  - $actions(s_1) = \{a_3, a_4\}$
  - $actions(s_2) = \{a_5\}$
  - $isGoal(s_5) = 1$
  - $isGoal(s_i) = 0$, given $i = 1,2,3,4$
  - $T(s_i, a_j) = s_j$



Search Space Definition → Search Tree

- Use the search problem formulation
  - `actions(s)`: returns set of actions, `A = {a1, …, ak}`, 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

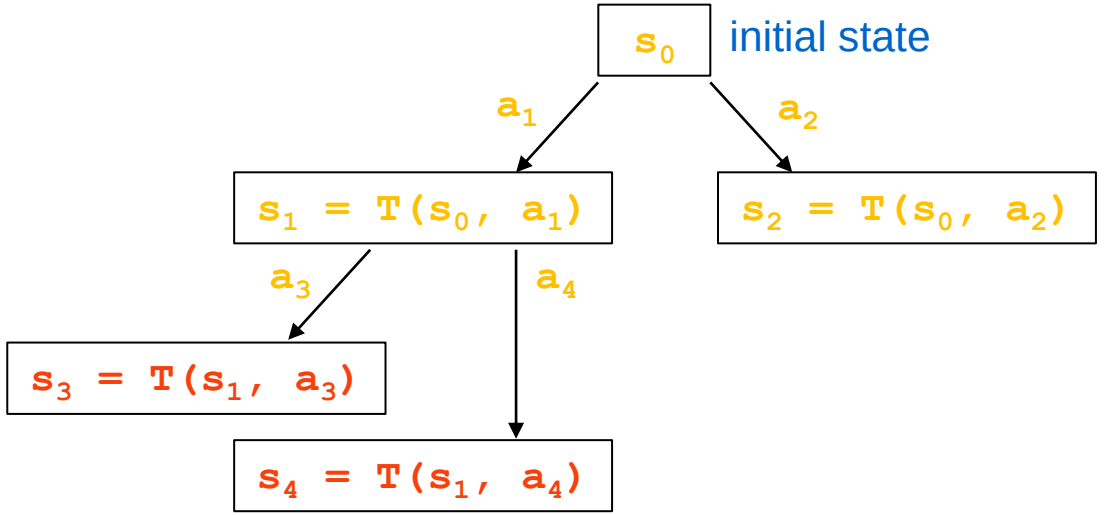
- Assume the following
  - $actions(s_0) = \{a_1, a_2\}$
  - $actions(s_1) = \{a_3, a_4\}$
  - $actions(s_2) = \{a_5\}$
  - $isGoal(s_5) = 1$
  - $isGoal(s_i) = 0$, given $i = 1,2,3,4$
  - $T(s_i, a_j) = s_j$



Search Space Definition → Search Tree

- Use the search problem formulation
  - `actions(s)`: returns set of actions, `A = {a1, …, ak}`, 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

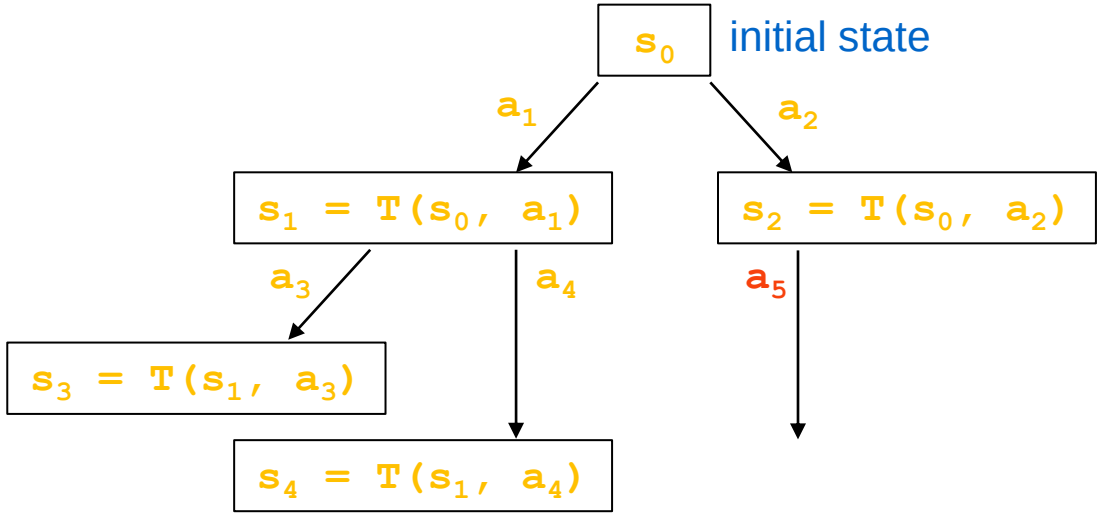
- Assume the following
  - $actions(s_0) = \{a_1, a_2\}$
  - $actions(s_1) = \{a_3, a_4\}$
  - $actions(s_2) = \{a_5\}$
  - $isGoal(s_5) = 1$
  - $isGoal(s_i) = 0$, given $i = 1,2,3,4$
  - $T(s_i, a_j) = s_j$



Search Space Definition → Search Tree

- Use the search problem formulation
  - `actions(s)`: returns set of actions, `A = {a1, …, ak}`, 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

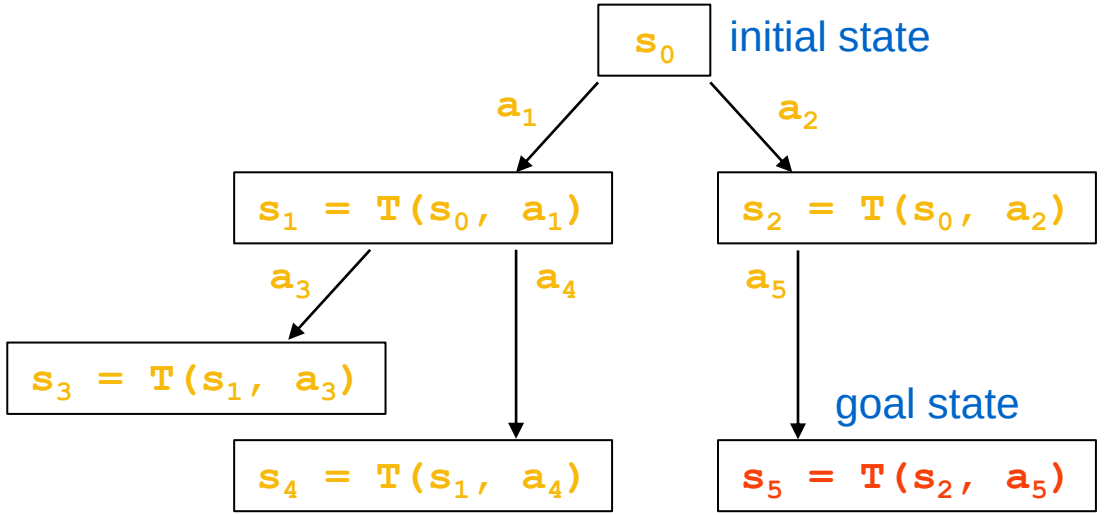
- Assume the following
  - $actions(s_0) = \{a_1, a_2\}$
  - $actions(s_1) = \{a_3, a_4\}$
  - $actions(s_2) = \{a_5\}$
  - $isGoal(s_5) = 1$
  - $isGoal(s_i) = 0$, given $i = 1,2,3,4$
  - $T(s_i, a_j) = s_j$



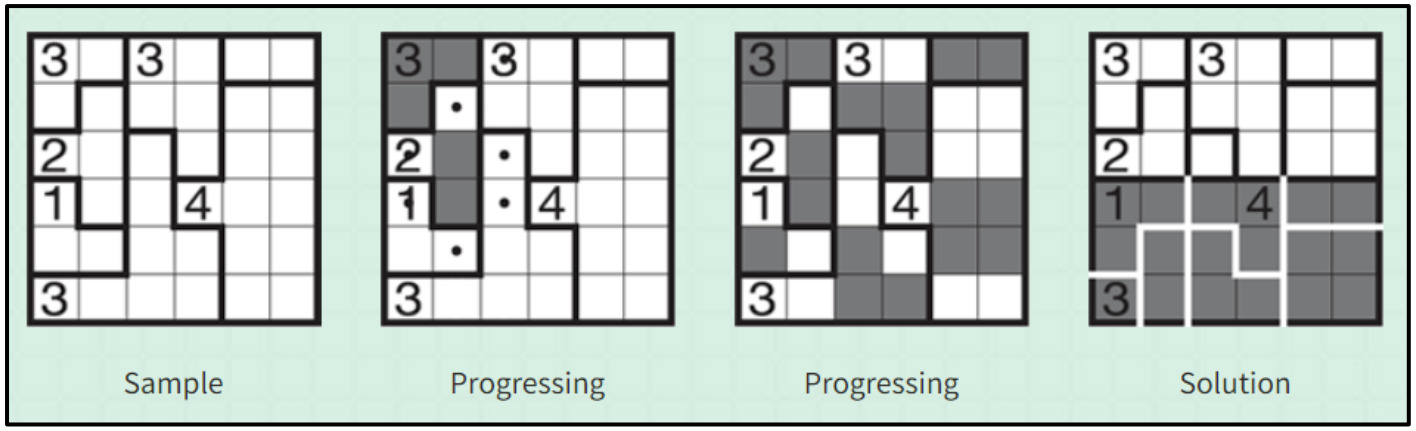
Search Space Definition → Search Tree

- Use the search problem formulation
  - `actions(s)`: returns set of actions, `A = {a1, …, ak}`, 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 the following
  - $actions(s_0) = \{a_1, a_2\}$
  - $actions(s_1) = \{a_3, a_4\}$
  - $actions(s_2) = \{a_5\}$
  - $isGoal(s_5) = 1$
  - $isGoal(s_i) = 0$, given $i = 1,2,3,4$
  - $T(s_i, a_j) = s_j$



Another Example Problem: Sto-stone Puzzle



AY22/23 S3
Midterm Q1

To be reviewed
in Tutorial 1

- Goal is to colour certain cells in the puzzle black
- Enclosed areas within bold lines are called “Rooms.” Each room contains a number indicating the number of vertically or horizontally contiguous black cells required in that room. Rooms with no number may have any number of vertically or horizontally contiguous black cells
- Vertically or horizontally contiguous black cells cannot connect across bold lines
- When all stones are “dropped” straight down (empty cells under black cells deleted) the stones must fill the bottom half of the grid without empty cells

4 Search Solutions

General Search Algorithm

```
Function TreeSearch(initial_state, actions, T, isGoal, cost):
   frontier = {Node(initial_state, NULL)}          # ADT: Node(current_state, parent_node)
   while frontier not empty:                       # add path cost, depth, action as required
         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          # i.e., no path from the initial state to any goal state
```

- Frontier
  - Part of the search space we are exploring
  - Current edge of the search tree
- 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 (up to some state)
  - Includes the following information
    - State
    - Parent node — to track current path from initial state
    - Action
    - Path cost — we will see why we need these later
    - Depth

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

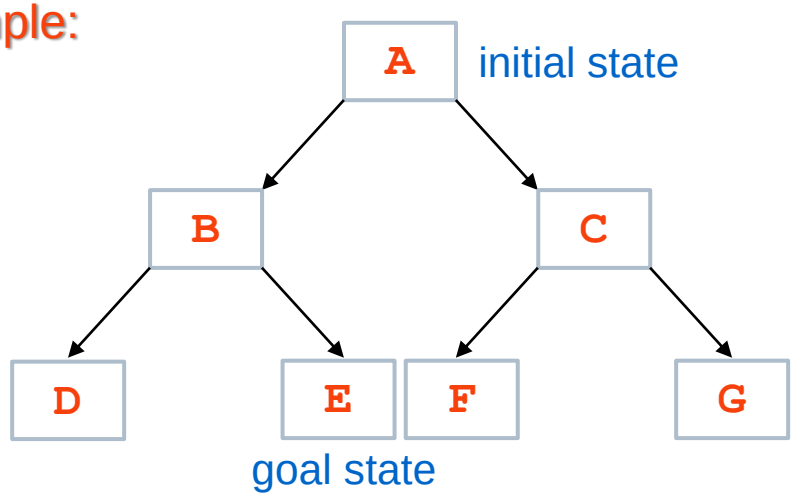
- Efficiency
  - Time complexity
  - Space complexity

- Correctness
  - An algorithm is complete if it will find a solution when one exists and correctly report failure (no solution) 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)

5 Breadth-First Search

Breadth-First Search (BFS) Algorithm

Example:



BFS Frontier: Queue

Frontier Trace:

Tie-breaking: alphabetic order on push to frontier

Undiscovered: ...

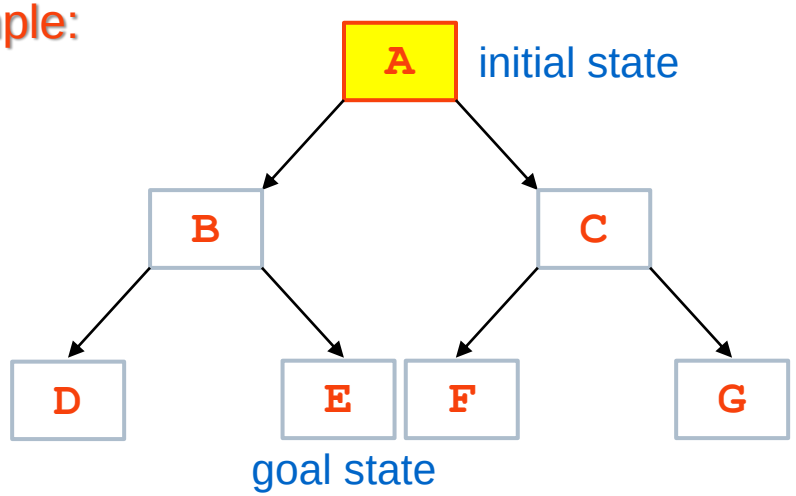
Discovered (Pushed): ...

Explored (Popped): ...

Node representation: `S (P)`,
where path `P` denotes the path taken to state `S`

Breadth-First Search (BFS) Algorithm

Example:



BFS Frontier: Queue

Frontier Trace:

```
Iteration 1: { A(-) }
```

Tie-breaking: alphabetic order on push to frontier

Undiscovered: ...

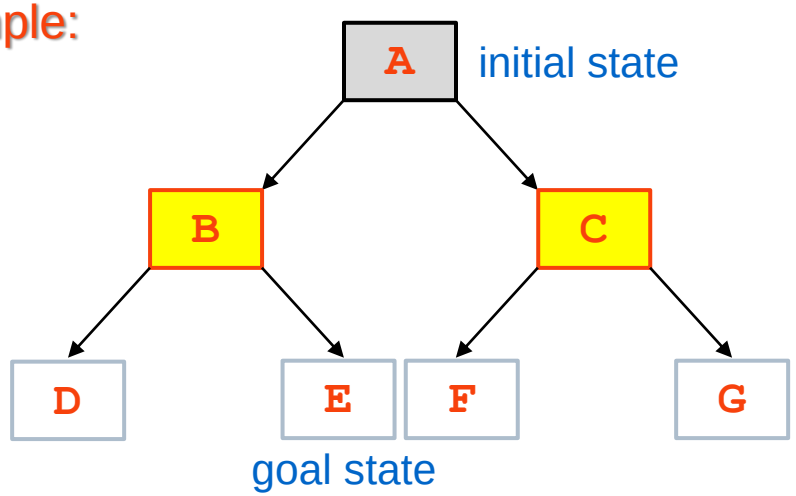
Discovered (Pushed): ...

Explored (Popped): ...

Node representation: `S(P)`,
where path `P` denotes the path taken to state `S`

Breadth-First Search (BFS) Algorithm

Example:



BFS Frontier: Queue

Frontier Trace:

```
Iteration 1: { A(-) }
Iteration 2: { B(A), C(A) }
```

Tie-breaking: alphabetic order on push to frontier

Undiscovered: ...

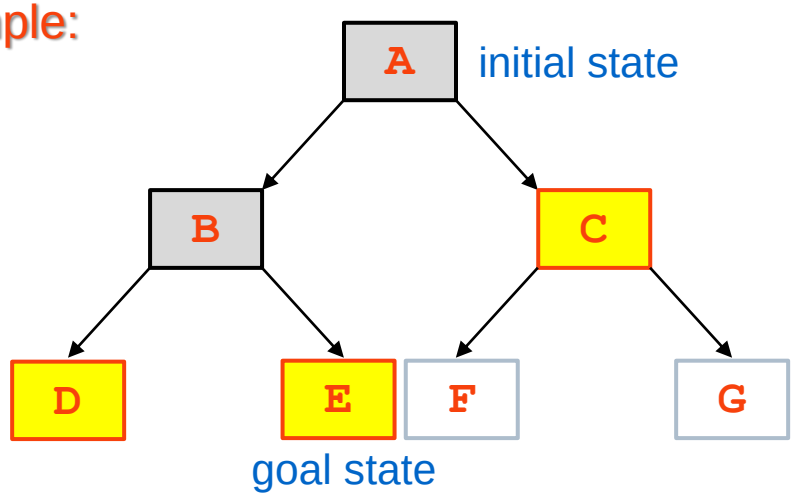
Discovered (Pushed): ...

Explored (Popped): ...

Node representation: `S(P)`,
where path `P` denotes the path taken to state `S`

Breadth-First Search (BFS) Algorithm

Example:



BFS Frontier: Queue

Frontier Trace:

```
Iteration 1: { A(-) }
Iteration 2: { B(A), C(A) }
Iteration 3: { C(A), D(A,B), E(A,B) }
```

Tie-breaking: alphabetic order on push to frontier

Undiscovered: ...

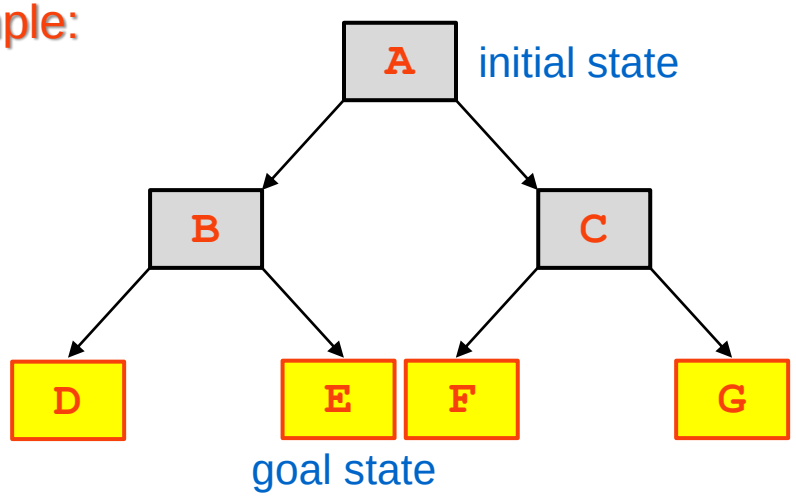
Discovered (Pushed): ...

Explored (Popped): ...

Node representation: `S(P)`,
where path `P` denotes the path taken to state `S`

Breadth-First Search (BFS) Algorithm

Example:



BFS Frontier: Queue

Frontier Trace:

```
Iteration 1: { A(-) }
Iteration 2: { B(A), C(A) }
Iteration 3: { C(A), D(A,B), E(A,B) }
Iteration 4: { D(A,B), E(A,B), F(A,C), G(A,C) }
```

Tie-breaking: alphabetic order on push to frontier

Undiscovered: ...

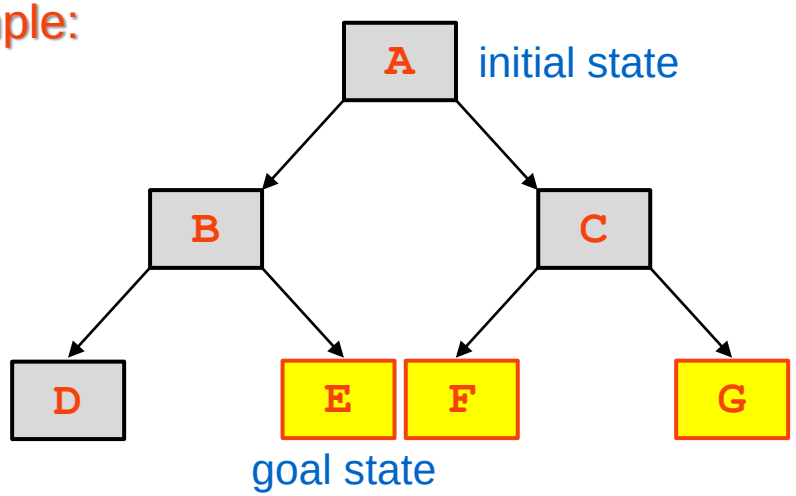
Discovered (Pushed): ...

Explored (Popped): ...

Node representation: `S(P)`,
where path `P` denotes the path taken to state `S`

Breadth-First Search (BFS) Algorithm

Example:



BFS Frontier: Queue

Frontier Trace:

```
Iteration 1: { A(-) }
Iteration 2: { B(A), C(A) }
Iteration 3: { C(A), D(A,B), E(A,B) }
Iteration 4: { D(A,B), E(A,B), F(A,C), G(A,C) }
Iteration 5: { E(A,B), F(A,C), G(A,C) }
```

Tie-breaking: alphabetic order on push to frontier

Undiscovered: ...

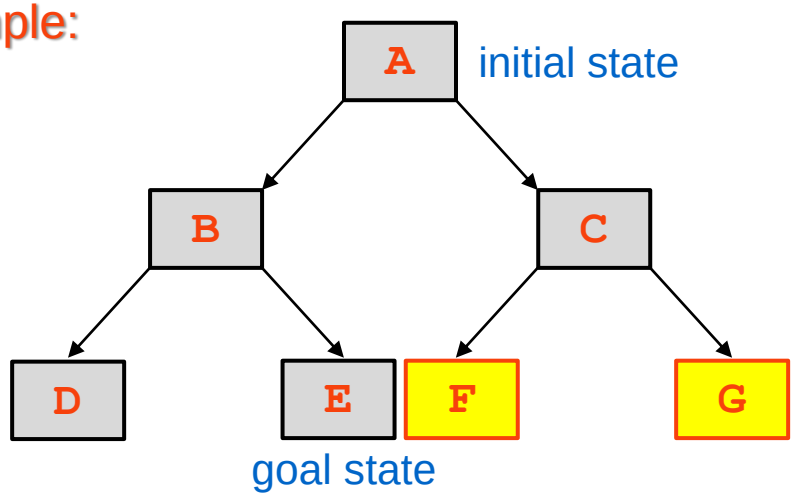
Discovered (Pushed): ...

Explored (Popped): ...

Node representation: `S(P)`,
where path `P` denotes the path taken to state `S`

Breadth-First Search (BFS) Algorithm

Example:



BFS Frontier: Queue

Frontier Trace:

```
Iteration 1: { A(-) }
Iteration 2: { B(A), C(A) }
Iteration 3: { C(A), D(A,B), E(A,B) }
Iteration 4: { D(A,B), E(A,B), F(A,C), G(A,C) }
Iteration 5: { E(A,B), F(A,C), G(A,C) }
Iteration 6:   DONE (A,B,E)
```

Tie-breaking: alphabetic order on push to frontier

Undiscovered: ...

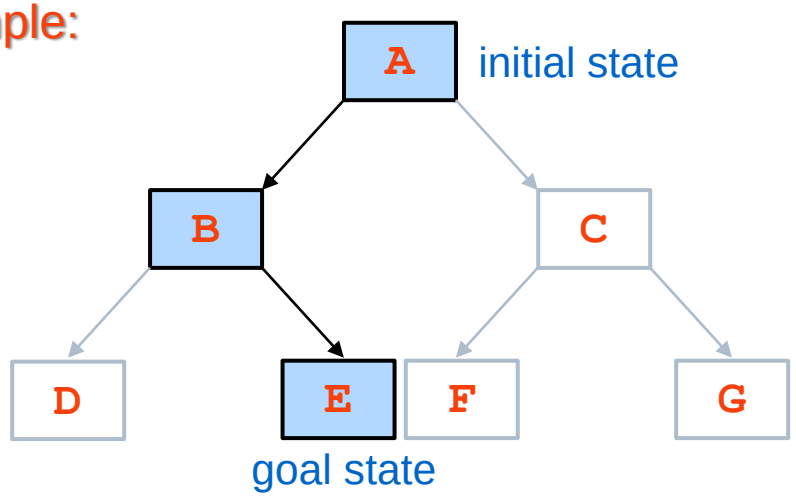
Discovered (Pushed): ...

Explored (Popped): ...

Node representation: `S(P)`,
where path `P` denotes the path taken to state `S`

Breadth-First Search (BFS) Algorithm

Example:



BFS Frontier: Queue

Frontier Trace:

```
Iteration 1: { A(-) }
Iteration 2: { B(A), C(A) }
Iteration 3: { C(A), D(A,B), E(A,B) }
Iteration 4: { D(A,B), E(A,B), F(A,C), G(A,C) }
Iteration 5: { E(A,B), F(A,C), G(A,C) }
Iteration 6:   DONE (A,B,E)
```

Tie-breaking: alphabetic order on push to frontier

Undiscovered: ...

Discovered (Pushed): ...

Explored (Popped): ...

Node representation: `S(P)`,
where path `P` denotes the path taken to state `S`

Breadth-First Search (BFS) Algorithm

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

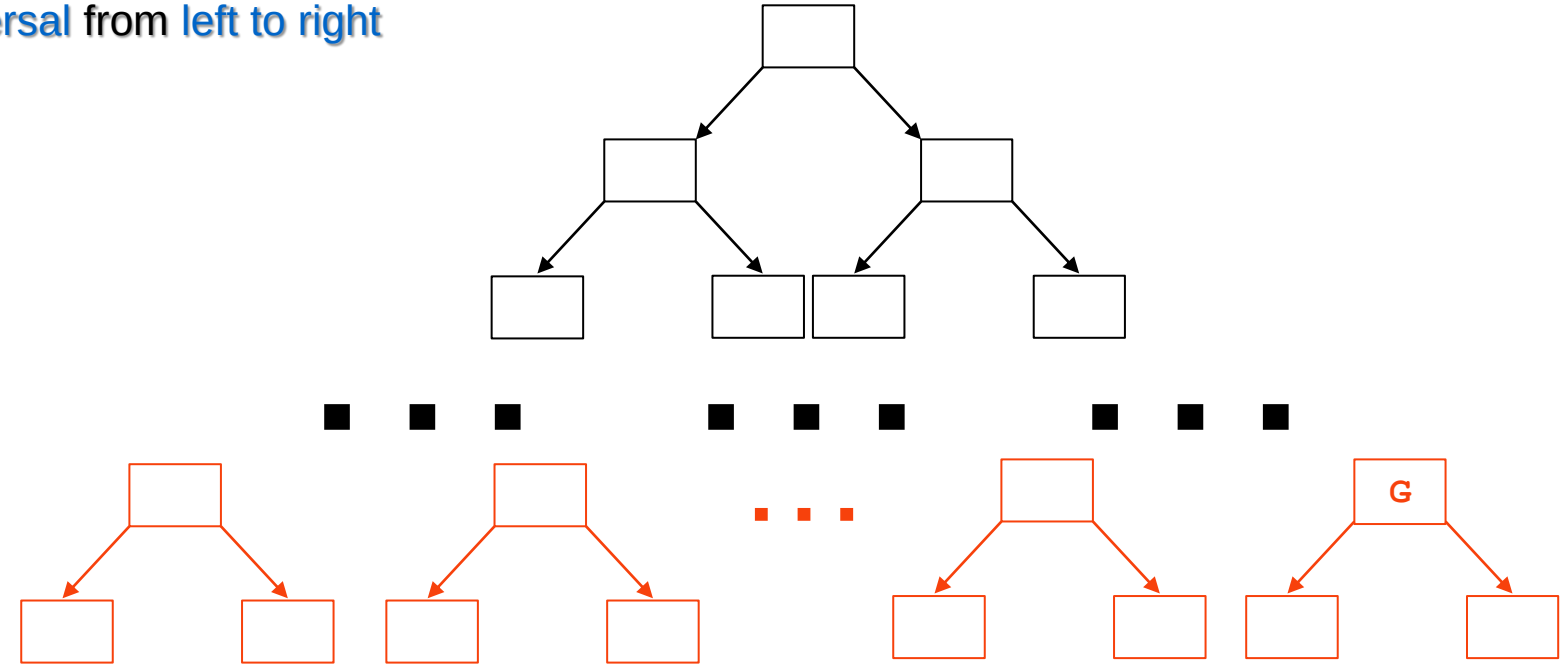
`b`: branching factor
`d`: depth of shallowest goal
`m`: maximum depth
1: if finite `b` AND (finite `m` OR contains solution)
2: optimal if action costs uniform (and some other cases)

Default assumptions on search spaces:
- `b` finite
- `d` finite (contains a solution)
- `m` infinite
- All actions costs ≥ 0

Exercise questions:
- Show that when `b` is infinite, BFS is incomplete.
- Show that when `b` is finite, but `m` is infinite AND there is no solution, BFS is incomplete.
- Under what (other) cases is BFS optimal?
- Why is the complexity of BFS $O(b^d)$? Is this accurate?

BFS: Complexity $O(b^{d+1})$ or $O(b^d)$?

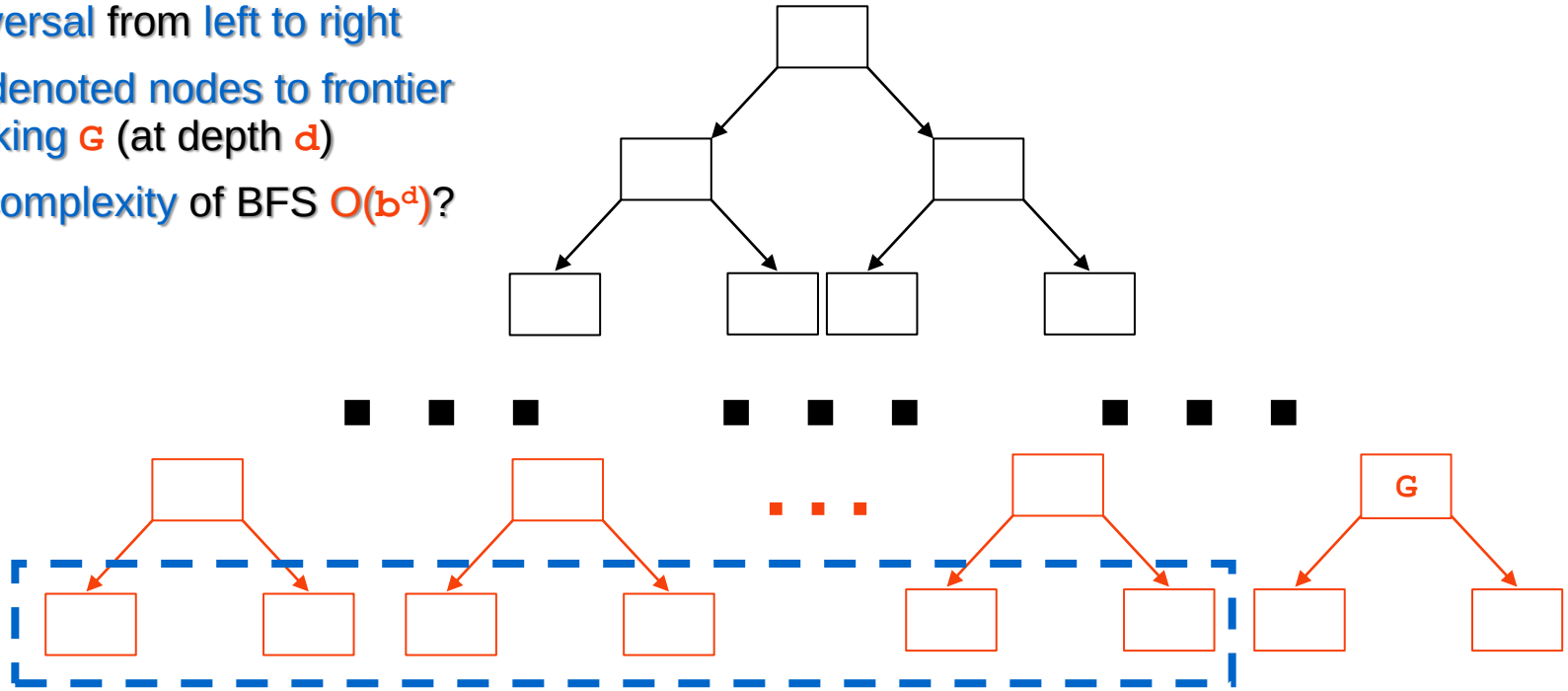
- Assume traversal from left to right



- Note: sum of nodes up to depth d:
  Sum of geometric series (i.e., $O(b^d)$ nodes)

BFS: Complexity $O(b^{d+1})$ or $O(b^d)$?

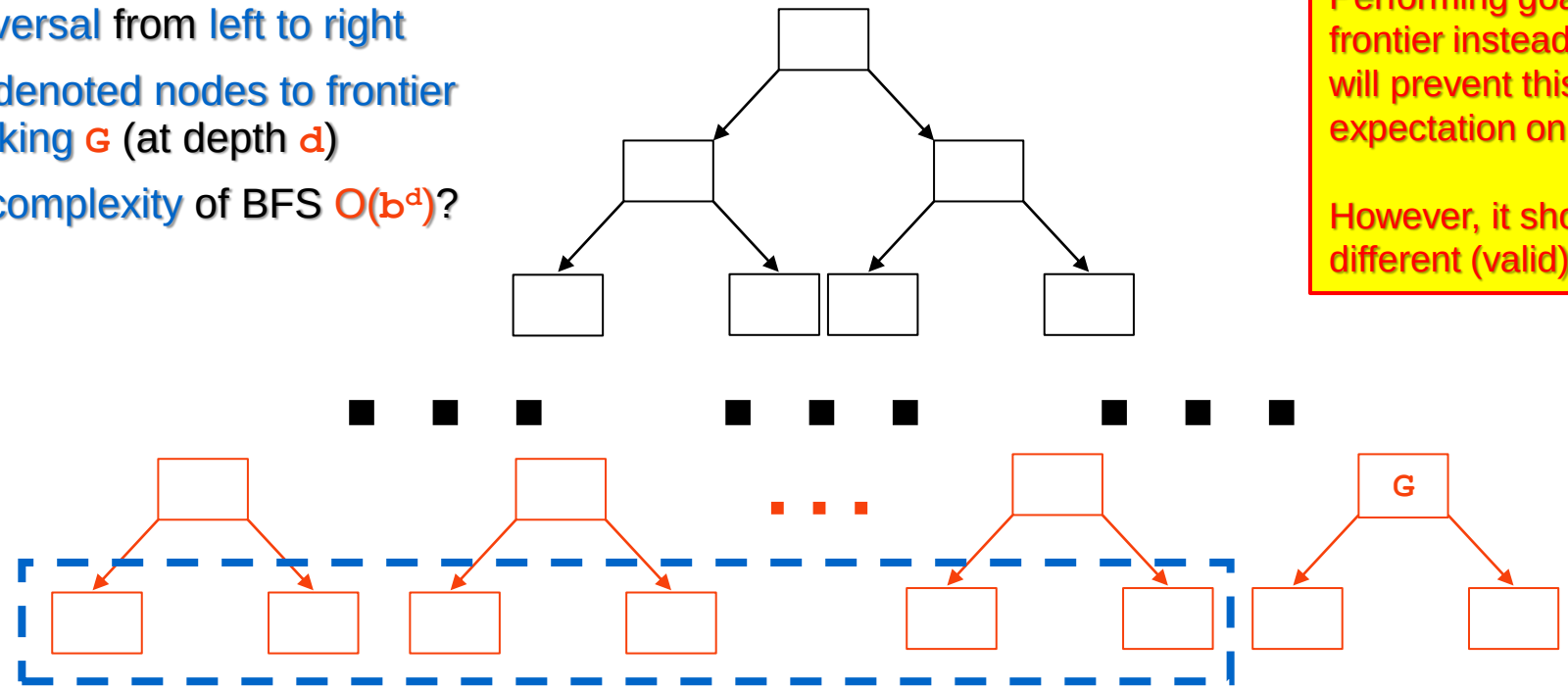
- Assume traversal from left to right
- Will add all denoted nodes to frontier before checking `G` (at depth `d`)
- Why is the complexity of BFS $O(b^d)$?



- Note: sum of nodes up to depth `d`:
  Sum of geometric series (i.e., $O(b^d)$ nodes)

BFS: Complexity $O(b^{d+1})$ or $O(b^d)$?

- Assume traversal from left to right
- Will add all denoted nodes to frontier before checking `G` (at depth `d`)
- Why is the complexity of BFS $O(b^d)$?



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

However, it should be noted that a different (valid) path may be output

- Note: sum of nodes up to depth `d`: Sum of geometric series (i.e., $O(b^d)$ nodes)

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?

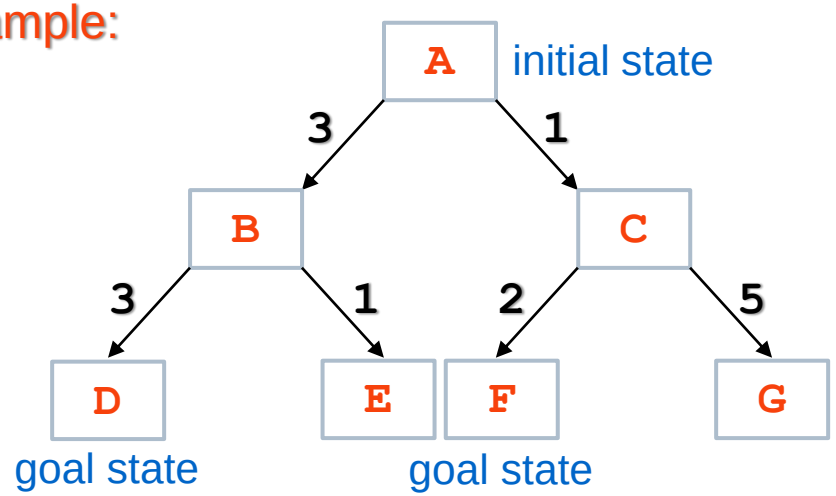
- Was anything unclear?
- Do you need to clarify anything?
- Ask on Archipelago
  - Specify a question
  - Upvote someone else's question

Invitation Link (Use NUS Email --- starts with E)
https://archipelago.rocks/app/resend-invite/78380593193

6 Uniform-Cost Search (Dijkstra's Algorithm)

Uniform-Cost Search (UCS) Algorithm

Example:



UCS Frontier: Priority Queue[1]

Frontier Trace:

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

Undiscovered: ...

Discovered (Pushed): ...

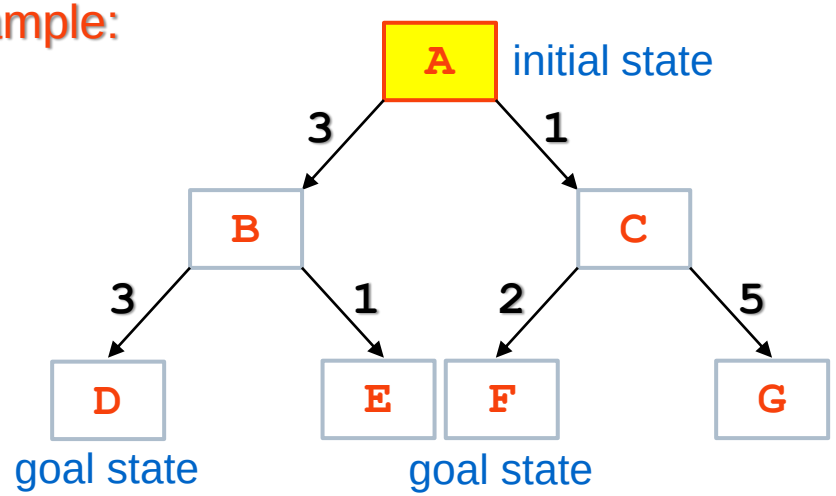
Explored (Popped): ...

Node representation: `S((P),C)`, where path `P` denotes the path taken to state `S`, with path cost `C`

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

Uniform-Cost Search (UCS) Algorithm

Example:



UCS Frontier: Priority Queue[1]

Frontier Trace:

```
Iteration 1: { A((-),0) }
```

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

Undiscovered: ...

Discovered (Pushed): ...

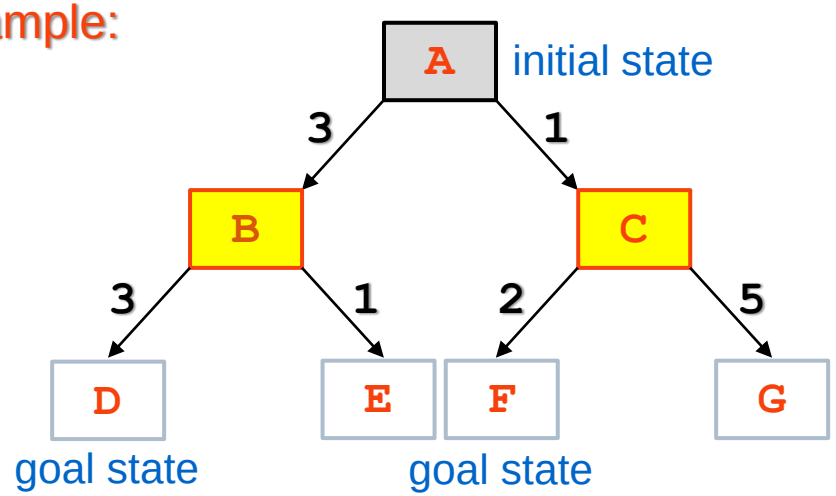
Explored (Popped): ...

Node representation: `S((P),C)`, where path `P` denotes the path taken to state `S`, with path cost `C`

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

Uniform-Cost Search (UCS) Algorithm

Example:



UCS Frontier: Priority Queue[1]

Frontier Trace:

```
Iteration 1: { A((-),0) }
Iteration 2: { C((A),1), B((A),3) }
```

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

Undiscovered: ...

Discovered (Pushed): ...

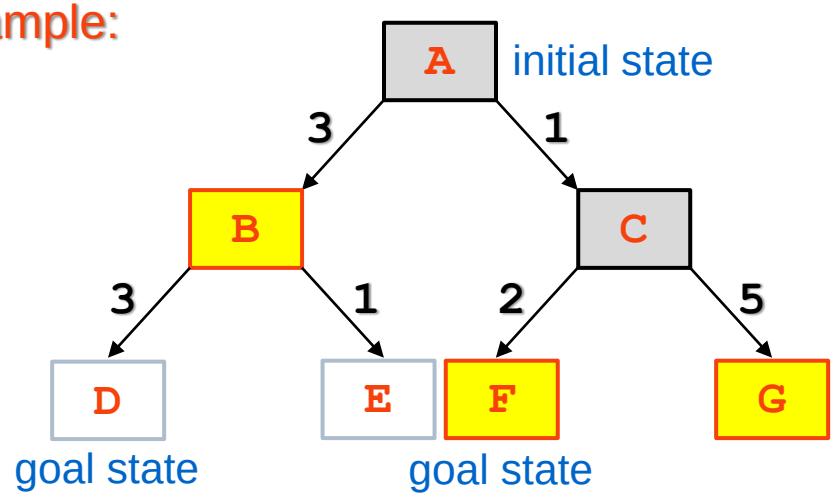
Explored (Popped): ...

Node representation: `S((P),C)`, where path `P` denotes the path taken to state `S`, with path cost `C`

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

Uniform-Cost Search (UCS) Algorithm

Example:



UCS Frontier: Priority Queue[1]

Frontier Trace:

```
Iteration 1: { A((-),0) }
Iteration 2: { C((A),1), B((A),3) }
Iteration 3: { B((A),3), F((A,C),3), G((A,C),6) }
```

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

Undiscovered: ...

Discovered (Pushed): ...

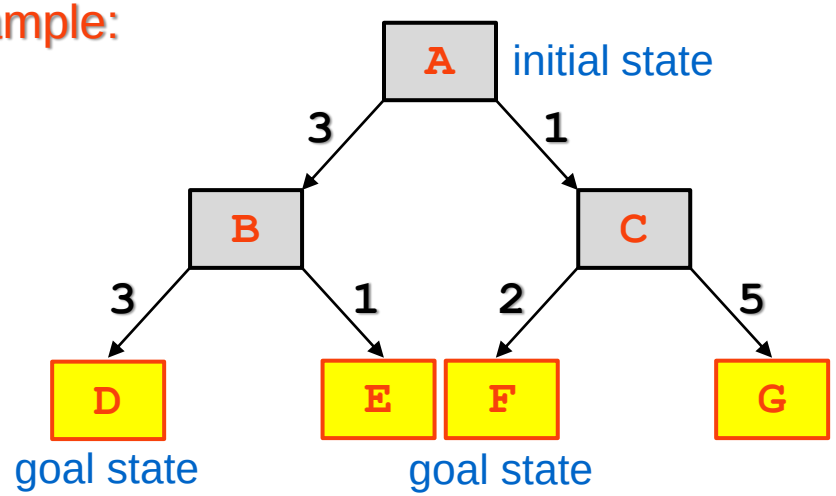
Explored (Popped): ...

Node representation: `S((P),C)`, where path `P` denotes the path taken to state `S`, with path cost `C`

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

Uniform-Cost Search (UCS) Algorithm

Example:



UCS Frontier: Priority Queue[1]

Frontier Trace:

```
Iteration 1: { A((-),0) }
Iteration 2: { C((A),1), B((A),3) }
Iteration 3: { B((A),3), F((A,C),3), G((A,C),6) }
Iteration 4: { F((A,C),3), E((A,B),4),
               D((A,B),6), G((A,C),6) }
```

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

Undiscovered: ...

Discovered (Pushed): ...

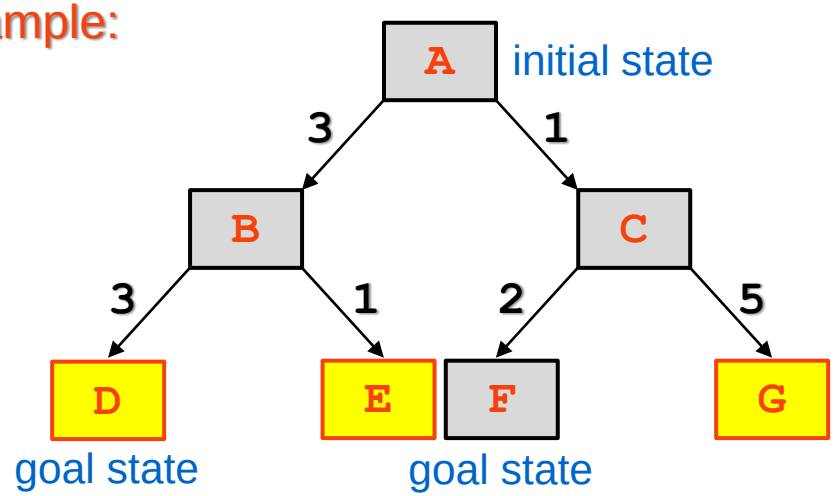
Explored (Popped): ...

Node representation: `S((P),C)`, where path `P` denotes the path taken to state `S`, with path cost `C`

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

Uniform-Cost Search (UCS) Algorithm

Example:



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

Undiscovered: ...

Discovered (Pushed): ...

Explored (Popped): ...

UCS Frontier: Priority Queue[1]

Frontier Trace:

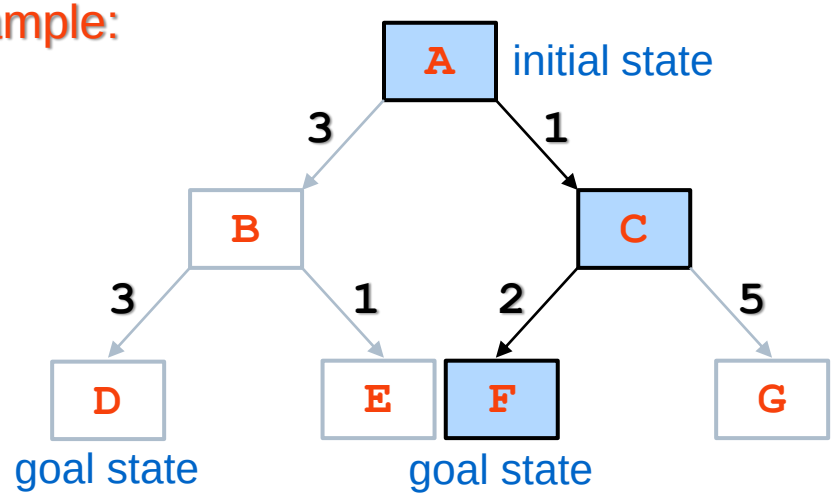
```
Iteration 1: { A((-),0) }
Iteration 2: { C((A),1), B((A),3) }
Iteration 3: { B((A),3), F((A,C),3), G((A,C),6) }
Iteration 4: { F((A,C),3), E((A,B),4),
               D((A,B),6), G((A,C),6) }
Iteration 5:   DONE (A,C,F),3
```

Node representation: `S((P),C)`, where path `P` denotes the path taken to state `S`, with path cost `C`

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

Uniform-Cost Search (UCS) Algorithm

Example:



UCS Frontier: Priority Queue[1]

Frontier Trace:

```
Iteration 1: { A((-),0) }
Iteration 2: { C((A),1), B((A),3) }
Iteration 3: { B((A),3), F((A,C),3), G((A,C),6) }
Iteration 4: { F((A,C),3), E((A,B),4),
               D((A,B),6), G((A,C),6) }
Iteration 5:   DONE (A,C,F),3
```

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

Undiscovered: ...

Discovered (Pushed): ...

Explored (Popped): ...

Node representation: `S((P),C)`, where path `P` denotes the path taken to state `S`, with path cost `C`

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

Uniform-Cost Search (UCS) Algorithm

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

Updated default assumptions on search spaces:

- All action costs > ε > 0

Exercise questions:

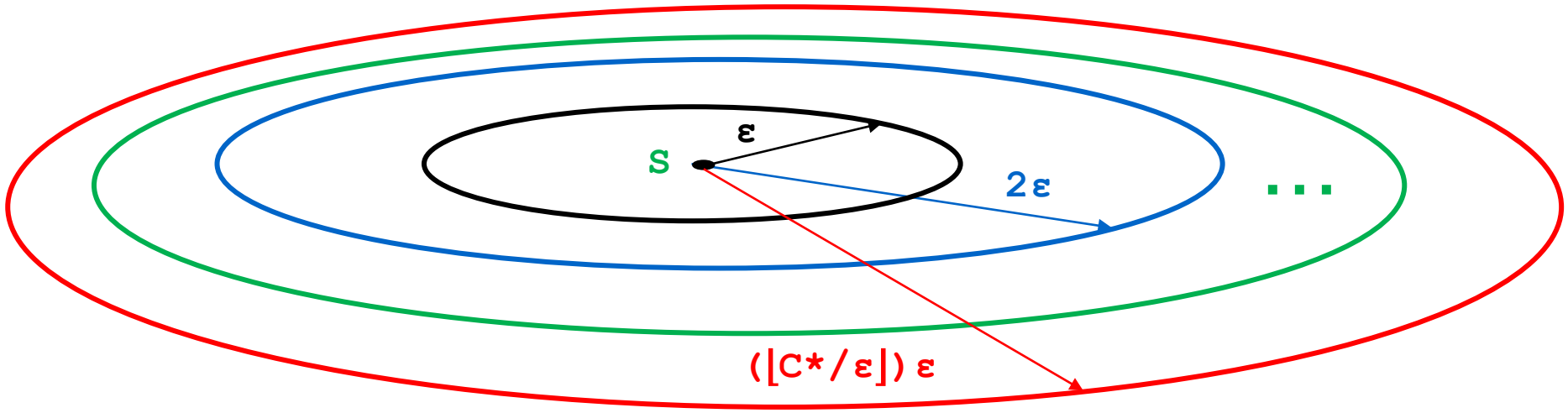
- Can you show that UCS under BFS completeness assumptions is incomplete when action costs are not > ε > 0 (e.g., when $\varepsilon = 0$)?
- Why is the complexity of UCS $O(b^e)$?
- Why is UCS optimal?
- Would UCS still be optimal with an early-goal test?

b: branching factor
e: $1 + \lfloor C^*/\varepsilon \rfloor$, where C^* is the optimal path cost and ε is some small positive constant
1: prioritising lower path cost, $g(n)$, where $g(n)$ = path cost of the path taken to reach n
2: requires BFS completeness criteria and that action costs > ε > 0
Note: determining path cost for each node is O(1) since we store the current path cost in each node

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

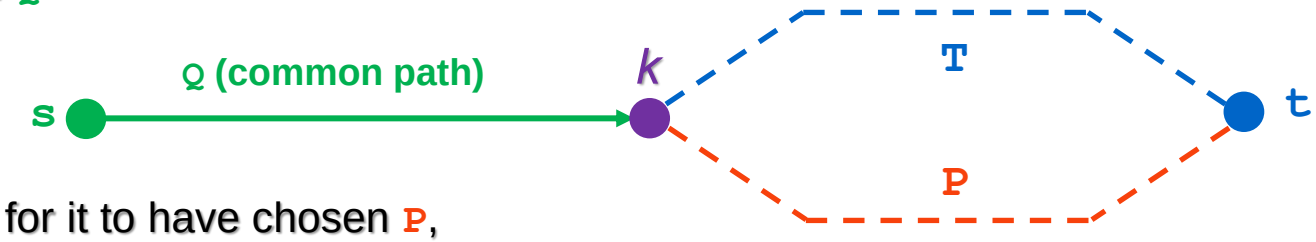
- UCS explores all paths radiating from the initial state
- UCS explores paths in increments of ε (smallest action cost)
  - With each step it extends paths by at least ε (from the initial state)
  - Considers paths with cost 0 (initial state only), then cost ε, then 2ε, etc.
  - Expected to reach goal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

note that $e = 1 + \lfloor C^*/\varepsilon \rfloor$ due to the late goal test



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 path cost of P
- UCS finds the optimal path to each node
  - Suppose UCS outputs path $Q + P$ as the solution for s to t
  - Suppose the optimal path from s to t is instead $Q + T$

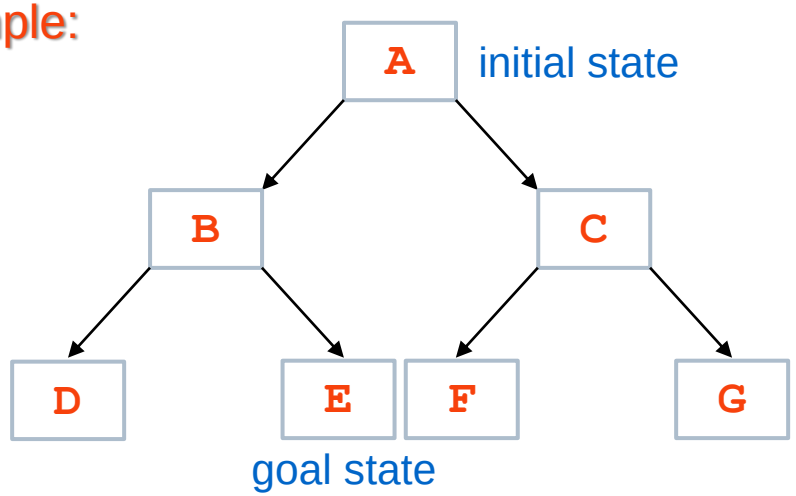


  - UCS must skip shorter paths between k and t for it to have chosen P, which is a contradiction since it always chooses shorter paths to explore first

7 Depth-First Search

Depth-First Search (DFS) Algorithm

Example:



BFS Frontier: Stack

Frontier Trace:

Tie-breaking: ***reverse*** alphabetic order on push to frontier

Undiscovered: ...

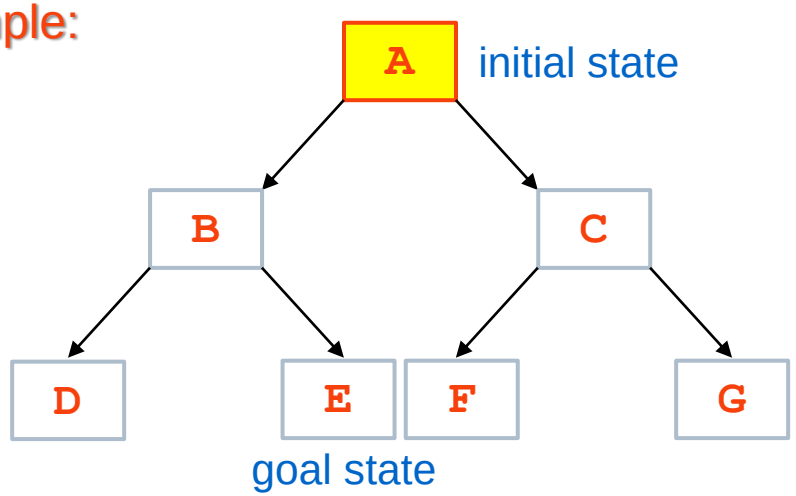
Discovered (Pushed): ...

Explored (Popped): ...

Node representation: `S (P)`,
where path `P` denotes the path taken to state `S`

Depth-First Search (DFS) Algorithm

Example:



BFS Frontier: Stack

Frontier Trace:

```
Iteration 1: { A(-) }
```

Tie-breaking: ***reverse*** alphabetic order on push to frontier

Undiscovered: ...

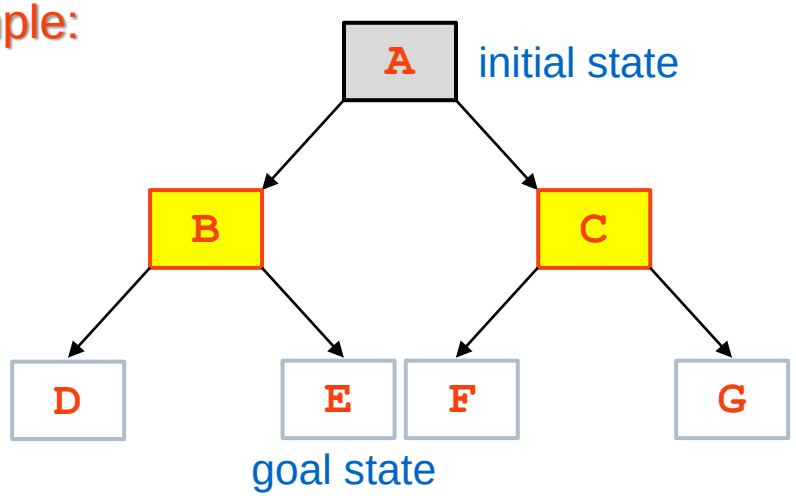
Discovered (Pushed): ...

Explored (Popped): ...

Node representation: `S(P)`,
where path `P` denotes the path taken to state `S`

Depth-First Search (DFS) Algorithm

Example:



BFS Frontier: Stack

Frontier Trace:

```
Iteration 1: { A(-) }
Iteration 2: { B(A), C(A) }
```

Tie-breaking: ***reverse*** alphabetic order on push to frontier

Undiscovered: ...

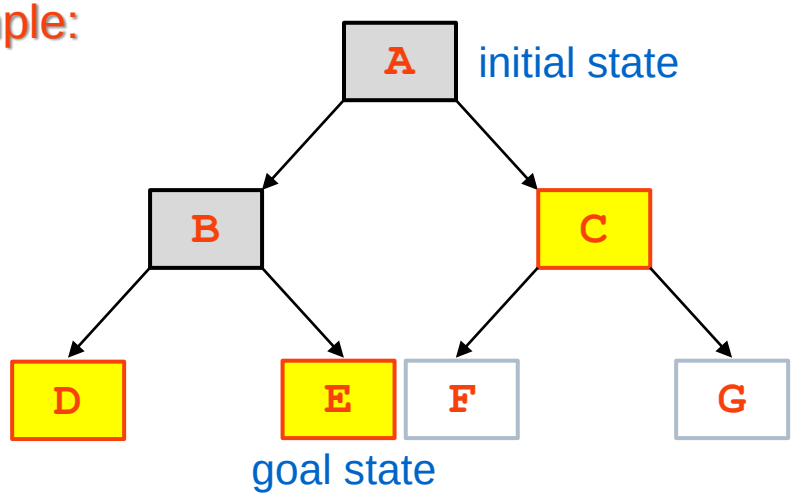
Discovered (Pushed): ...

Explored (Popped): ...

Node representation: `S(P)`,
where path `P` denotes the path taken to state `S`

Depth-First Search (DFS) Algorithm

Example:



BFS Frontier: Stack

Frontier Trace:

```
Iteration 1: { A(-) }
Iteration 2: { B(A), C(A) }
Iteration 3: { D(A,B), E(A,B), C(A) }
```

Tie-breaking: ***reverse*** alphabetic order on push to frontier

Undiscovered: ...

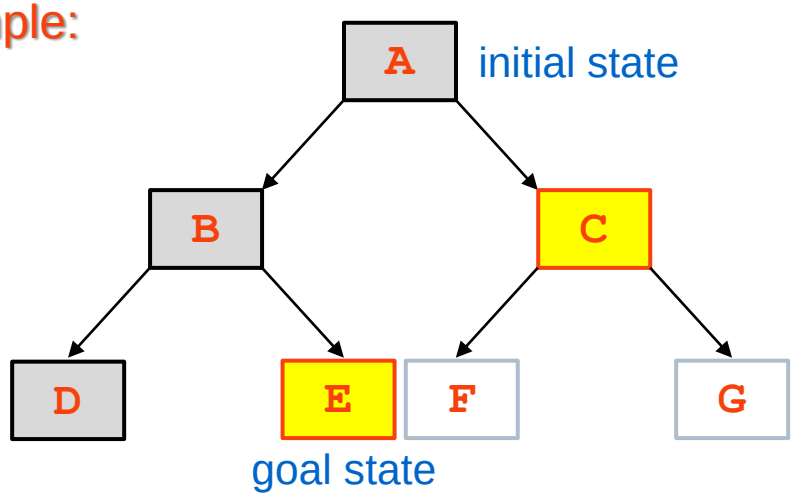
Discovered (Pushed): ...

Explored (Popped): ...

Node representation: `S(P)`,
where path `P` denotes the path taken to state `S`

Depth-First Search (DFS) Algorithm

Example:



BFS Frontier: Stack

Frontier Trace:

```
Iteration 1: { A(-) }
Iteration 2: { B(A), C(A) }
Iteration 3: { D(A,B), E(A,B), C(A) }
Iteration 4: { E(A,B), C(A) }
```

Tie-breaking: ***reverse*** alphabetic order on push to frontier

Undiscovered: ...

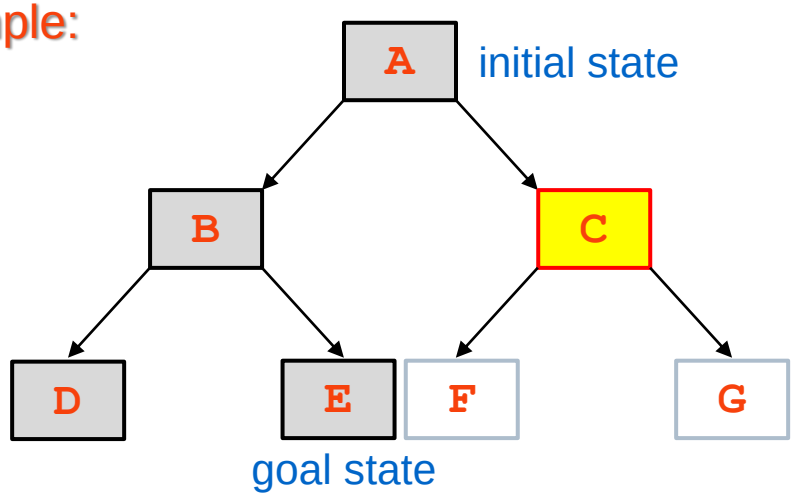
Discovered (Pushed): ...

Explored (Popped): ...

Node representation: `S(P)`,
where path `P` denotes the path taken to state `S`

Depth-First Search (DFS) Algorithm

Example:



BFS Frontier: Stack

Frontier Trace:

```
Iteration 1: { A(-) }
Iteration 2: { B(A), C(A) }
Iteration 3: { D(A,B), E(A,B), C(A) }
Iteration 4: { E(A,B), C(A) }
Iteration 5:   DONE (A,B,E)
```

Tie-breaking: ***reverse*** alphabetic order on push to frontier

Undiscovered: ...

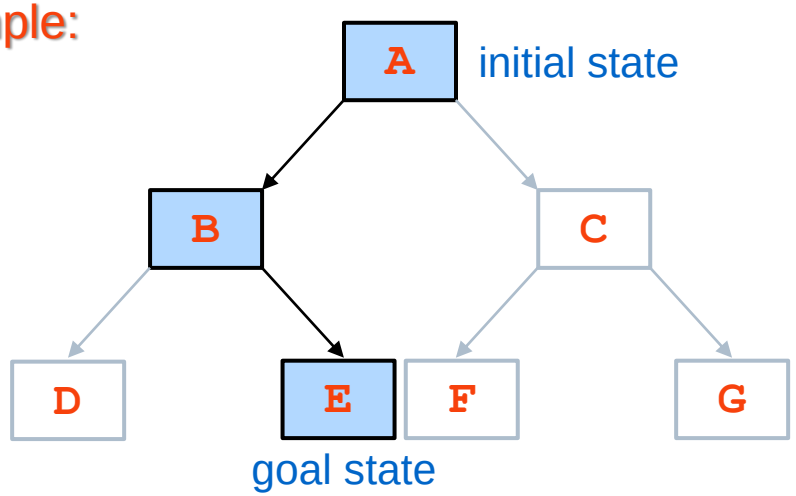
Discovered (Pushed): ...

Explored (Popped): ...

Node representation: `S(P)`,
where path `P` denotes the path taken to state `S`

Depth-First Search (DFS) Algorithm

Example:



BFS Frontier: Stack

Frontier Trace:

```
Iteration 1: { A(-) }
Iteration 2: { B(A), C(A) }
Iteration 3: { D(A,B), E(A,B), C(A) }
Iteration 4: { E(A,B), C(A) }
Iteration 5:   DONE (A,B,E)
```

Tie-breaking: ***reverse*** alphabetic order on push to frontier

Undiscovered: ...

Discovered (Pushed): ...

Explored (Popped): ...

Node representation: `S(P)`,
where path `P` denotes the path taken to state `S`

Depth-First Search (DFS) Algorithm

- Frontier: Stack
- Time Complexity: $O(b^m)$
- Space Complexity: $O(bm)$
- Complete: No[1]
- Optimal: No

b: branching factor
m: maximum depth
1: DFS may be incomplete even if a solution exists

Exercise questions:

- Show that DFS under BFS completeness assumptions is incomplete.
- Can space complexity be improved?

The space complexity of DFS 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)

8 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 (default)
  - Will completely search sub-trees up to depth limit
  - Completeness of BFS with space complexity of DFS
- Overheads: will rerun top levels many times
  - Assuming branching factor b and depth ℓ, the number of nodes generated by DLS:
    - $O(b^0) + O(b^1) + O(b^2) + \ldots + O(b^{\ell-2}) + O(b^{\ell-1}) + O(b^{\ell})$
  - Nodes generated by IDS to depth d with branching factor b:
    - $(d+1)\cdot O(b^0) + d\cdot O(b^1) + (d-1)\cdot O(b^2) + \ldots + 3\cdot O(b^{d-2}) + 2\cdot O(b^{d-1}) + O(b^d)$

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 ≈ 11%
- IDS properties
  - Time: $O(b^d)$
  - Space: $O(bd)$
  - Complete: Yes (is `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

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

Default assumptions on search spaces:

- b finite
- d finite (contains solution)
- m infinite
- All actions costs are $> \varepsilon > 0$

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)

- 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 incomplete (even under 1 – note the "or"); if a solution exists, it may infinitely traverse a path without a solution
- DFS space complexity may be improved to O(m) with backtracking (similar for DLS and IDS)

9 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 (including cycles)
  - Should not consider these if optimality is required
- Typical practice → graph search implementation
  - Maintain a `reached` (or `visited`) hash table
  - Add redundant 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)

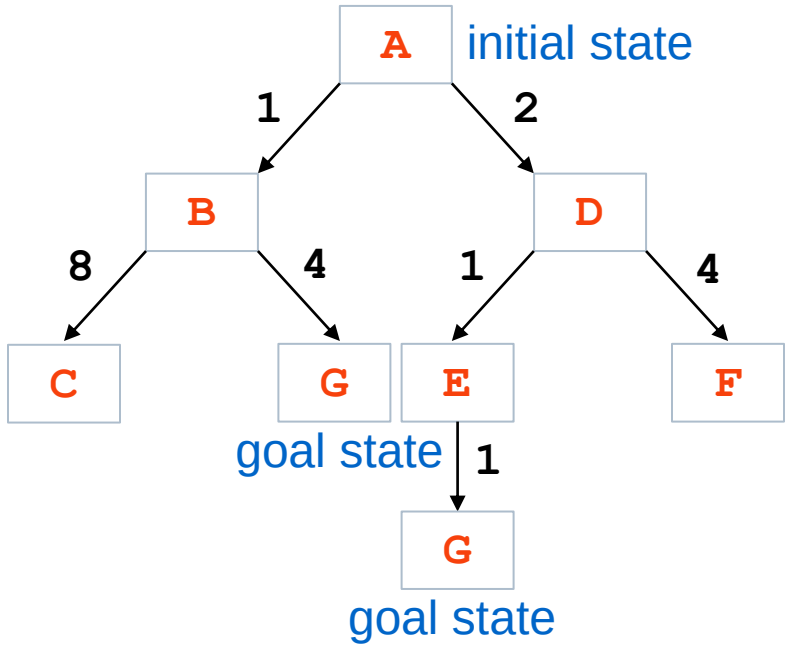
```
Function GraphSearchV1(initial_state, actions, T, isGoal, cost):
   initial_node = Node(initial_state, NULL)
   frontier = {initial_node}
   reached = {initial_state: initial_node}
   while frontier not empty:
         current = frontier.pop()
         if isGoal(current.state): return current.getPath()
         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 but may also omit optimal path

Exercise:
- Which uninformed algorithms may benefit from this implementation?

Graph Search Algorithm (Version 1)

UCS Example:



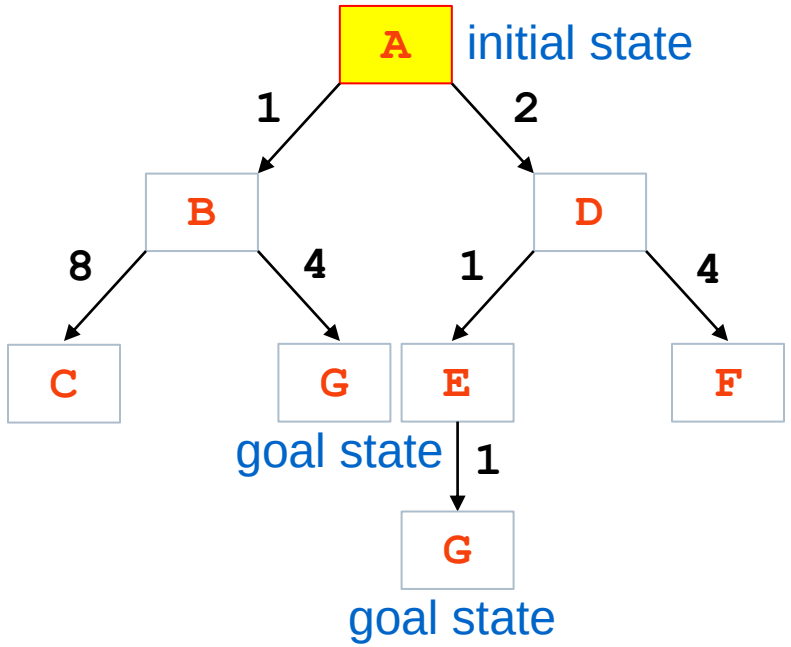
Frontier:

Reached:

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

Graph Search Algorithm (Version 1)

UCS Example:

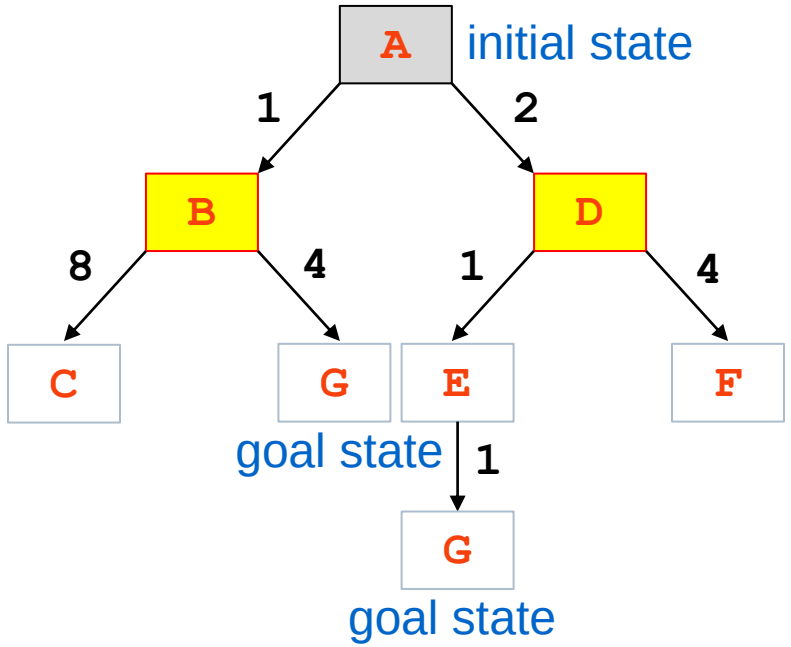


	Frontier:	Reached:
Iteration 1:	{ A((-),0) }	{ A }

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

Graph Search Algorithm (Version 1)

UCS Example:

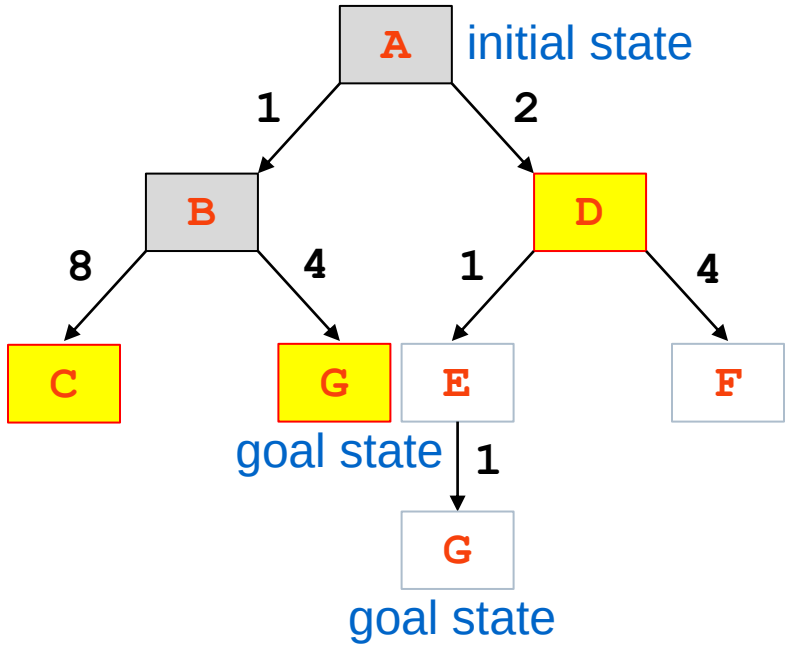


	Frontier:	Reached:
Iteration 1:	{ A((-),0) }	{ A }
Iteration 2:	{ B((A),1), D((A),2) }	{ A,B,D }

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

Graph Search Algorithm (Version 1)

UCS Example:

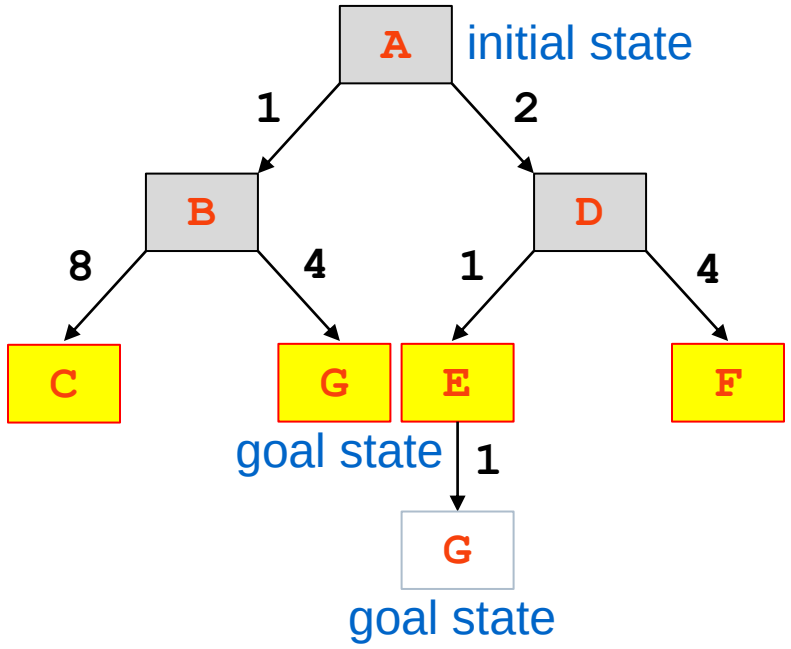


	Frontier:	Reached:
Iteration 1:	{ A((-),0) }	{ A }
Iteration 2:	{ B((A),1), D((A),2) }	{ A,B,D }
Iteration 3:	{ D((A),2), G((A,B),5), C((A,B),9) }	{ A,B,C,D,G }

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

Graph Search Algorithm (Version 1)

UCS Example:

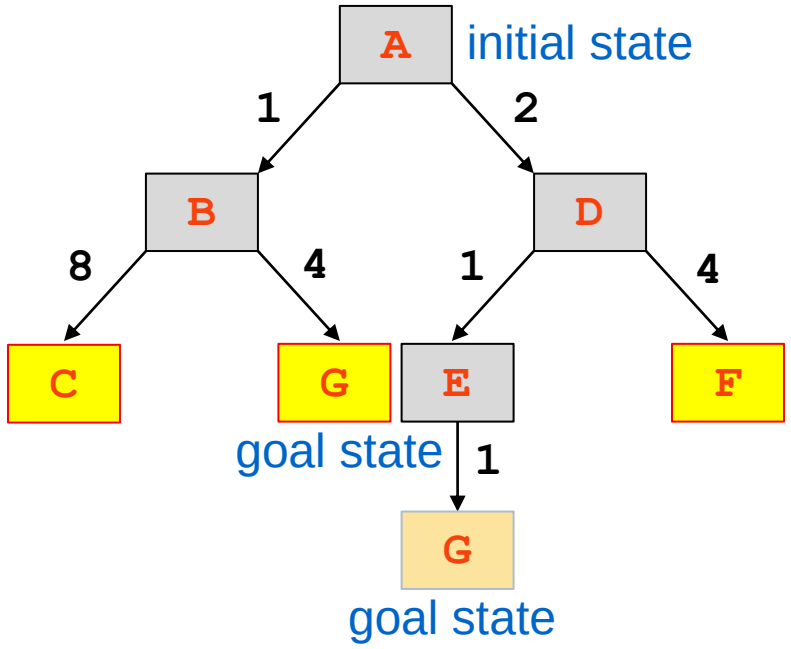


	Frontier:	Reached:
Iteration 1:	{ A((-),0) }	{ A }
Iteration 2:	{ B((A),1), D((A),2) }	{ A,B,D }
Iteration 3:	{ D((A),2), G((A,B),5), C((A,B),9) }	{ A,B,C,D,G }
Iteration 4:	{ E((A,D),3), G((A,B),5), F((A,D),6), C((A,B),9) }	{ A,B,C,D,E,F,G }

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

Graph Search Algorithm (Version 1)

UCS Example:



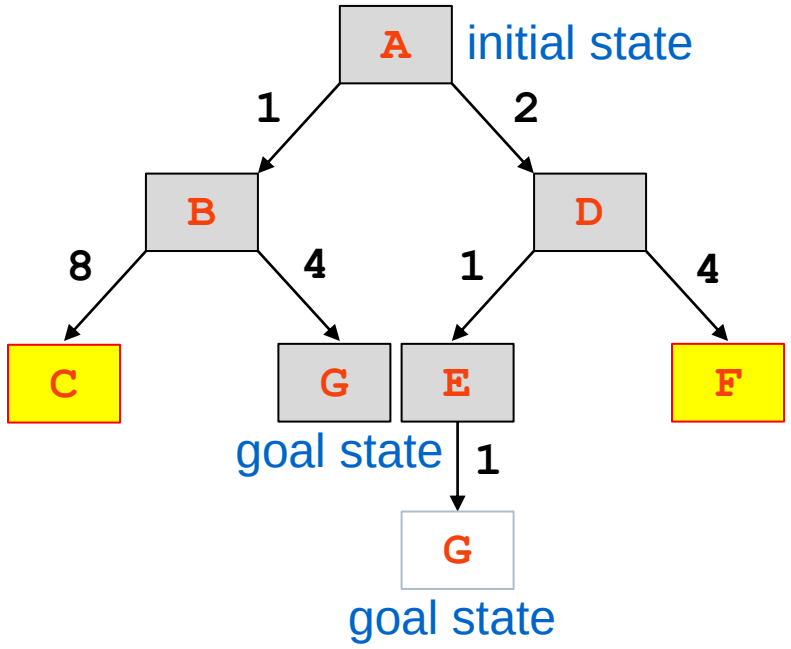
Tie-breaking: nodes ordered alphabetically when priority is the same

	Frontier:	Reached:
Iteration 1:	{ A((-),0) }	{ A }
Iteration 2:	{ B((A),1), D((A),2) }	{ A,B,D }
Iteration 3:	{ D((A),2), G((A,B),5), C((A,B),9) }	{ A,B,C,D,G }
Iteration 4:	{ E((A,D),3), G((A,B),5), F((A,D),6), C((A,B),9) }	{ A,B,C,D,E,F,G }
Iteration 5:	{ G((A,B),5), F((A,D),6), C((A,B),9}	{ A,B,C,D,E,F,G }

cannot push G to frontier again since it has been previously visited

Graph Search Algorithm (Version 1)

UCS Example:

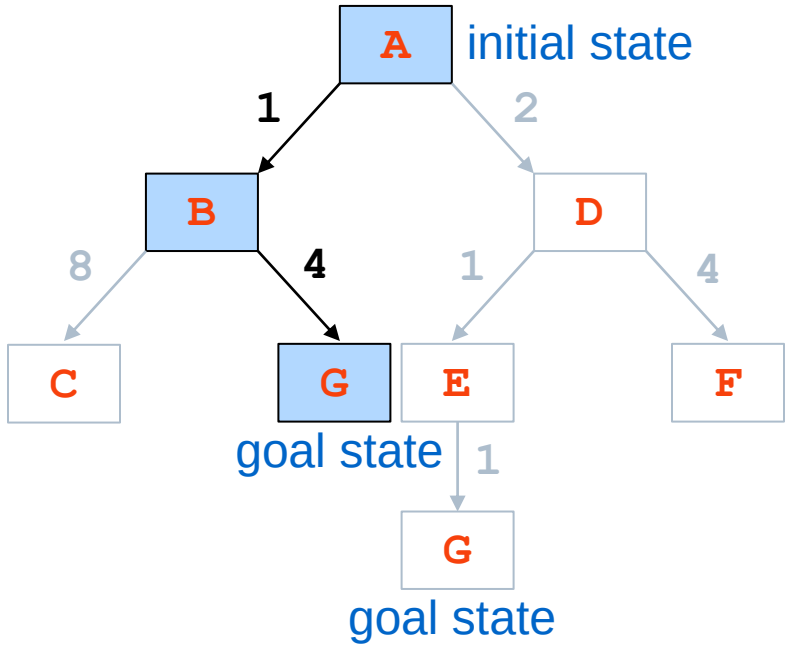


Tie-breaking: nodes ordered alphabetically when priority is the same

	Frontier:	Reached:
Iteration 1:	{ A((-),0) }	{ A }
Iteration 2:	{ B((A),1), D((A),2) }	{ A,B,D }
Iteration 3:	{ D((A),2), G((A,B),5), C((A,B),9) }	{ A,B,C,D,G }
Iteration 4:	{ E((A,D),3), G((A,B),5), F((A,D),6), C((A,B),9) }	{ A,B,C,D,E,F,G }
Iteration 5:	{ G((A,B),5), F((A,D),6), C((A,B),9}	{ A,B,C,D,E,F,G }
Iteration 6:	DONE (A,B,G)	

Graph Search Algorithm (Version 1)

UCS Example:



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

	Frontier:	Reached:
Iteration 1:	{ A((-),0) }	{ A }
Iteration 2:	{ B((A),1), D((A),2) }	{ A,B,D }
Iteration 3:	{ D((A),2), G((A,B),5), C((A,B),9) }	{ A,B,C,D,G }
Iteration 4:	{ E((A,D),3), G((A,B),5), F((A,D),6), C((A,B),9) }	{ A,B,C,D,E,F,G }
Iteration 5:	{ G((A,B),5), F((A,D),6), C((A,B),9}	{ A,B,C,D,E,F,G }
Iteration 6:	DONE (A,B,G) // Non-optimal path!	

Graph Search Algorithm (Version 2)

```
Function GraphSearchV2(initial_state, actions, T, isGoal, cost):
   initial_node = Node(initial_state, NULL)
   frontier = {initial_node}
   reached = {initial_state: initial_node}
   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

Summary (Graph Search Implementations)

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

Default assumptions on search spaces:

- b finite
- d finite (contains solution)
- m infinite
- All actions costs are $> \varepsilon > 0$

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)

- 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

For CS3243, unless otherwise mentioned, assume tree search

For graph search, assume V2 for UCS and V1 for other uninformed search algorithms

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/78380593193