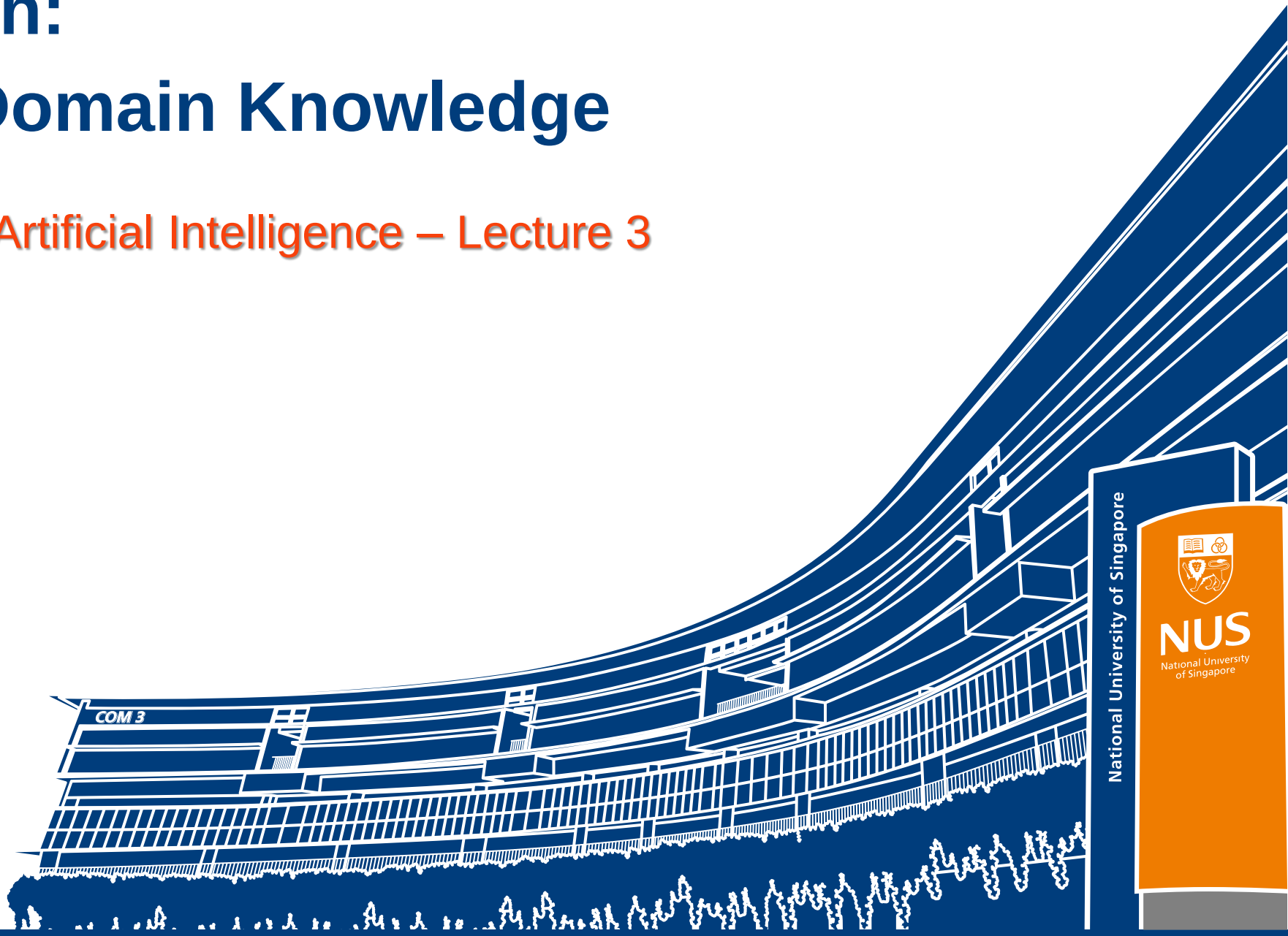
Informed Search:
Incorporating Domain Knowledge

CS3243: Introduction to Artificial Intelligence – Lecture 3



NUS | Computing
National University of Singapore

1 Administrative Matters

Project 1 (1.1 & 1.2)

- Released on Monday (28 AUG)
  - Project 1.1 is due Week 5 Sunday (15 SEP 2359 hrs)
  - Project 1.2 is due Week 6 Sunday (22 FEB 2359 hrs)
- Late submission penalties
  - Within deadline +24 hours = 80% of score
  - Within deadline +48 hours = 50% of score
  - Beyond deadline +48 hours = 0% of score

Submission on Coursemology

DO NOT COPY / SHARE CODE!

Start on Project 1 early!
Start today!

Upcoming...

- Deadlines
  - Tutorial Assignment 1 (released last week)
    - Post-tutorial submission: Due this Friday!
  - Tutorial Assignment 2 (released today)
    - Pre-tutorial Submission: Due this Sunday!
    - Post-tutorial Submission: Due next Friday!

Remember that there are attendance marks for tutorials!

Contents

- Reviewing Uninformed Search
- Greedy Best-First Search
- A* Search
- Dominant Heuristics

Reference: AIMA 4th Edition, Section 3.5

2 Reviewing Uninformed Search

Problem-Solving Agent: General Path Planning

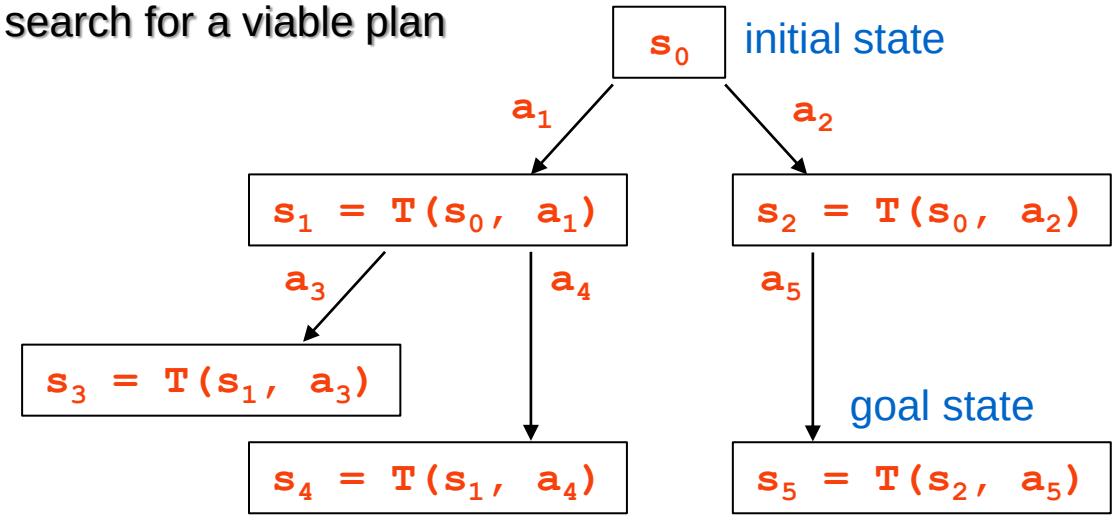
- Assumed environment
  - Fully observable
  - Deterministic
  - Discrete
  - Episodic

- General definition of a path planning problem
  - State representation ADT, s_i
    - Initial state value (s_0)
  - Actions function, $actions : s_i \rightarrow A = \{a_1, \ldots, a_k\}$
  - Transition model function, $T : (s_i, a_j) \rightarrow s_i'$
  - Goal test function, $isGoal : s_i \rightarrow \{0, 1\}$
  - Action costs function, $cost : (s_i, a_j, s_i') \rightarrow v \geq 0$

General method of defining an Agent Function for ALL path planning problems
(given a standard search problem formulation) – stronger AI (than reflex agent)

Example from Lecture 2:

- Dynamically form the search tree as we search for a viable plan



Tree 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
  - Paths to explore/search
  - Considers ALL paths (including redundant ones)

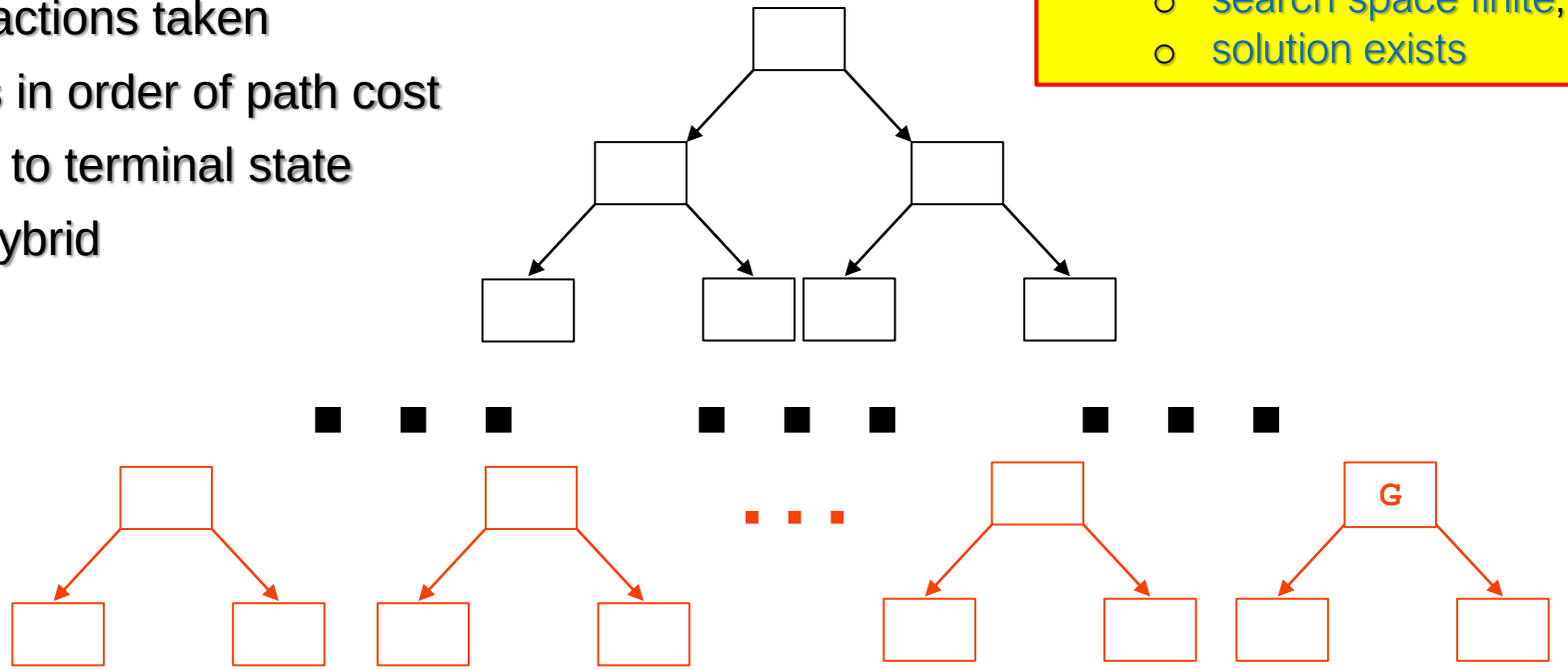
- Each element of the **`frontier`** is a **`Node`** that must include
  - Referenced end point of path, state **`s`**
  - Parent node, for Linked List reference (with Action taken, defines the path, i.e., plan)
  - Other attributes
    - Action – also facilitates backtracking in DFS – i.e., O(m) space complexity
    - Path cost – for efficient UCS (avoids traversing path Linked List)
    - Depth – for efficient DLS/IDS (avoids traversing path Linked List)

Uninformed Search Algorithms & Completeness

- Systematic traversal of search space
  - BFS: paths in order of number of actions taken
  - UCS: paths in order of path cost
  - DFS: paths to terminal state
  - DLS/IDS: hybrid

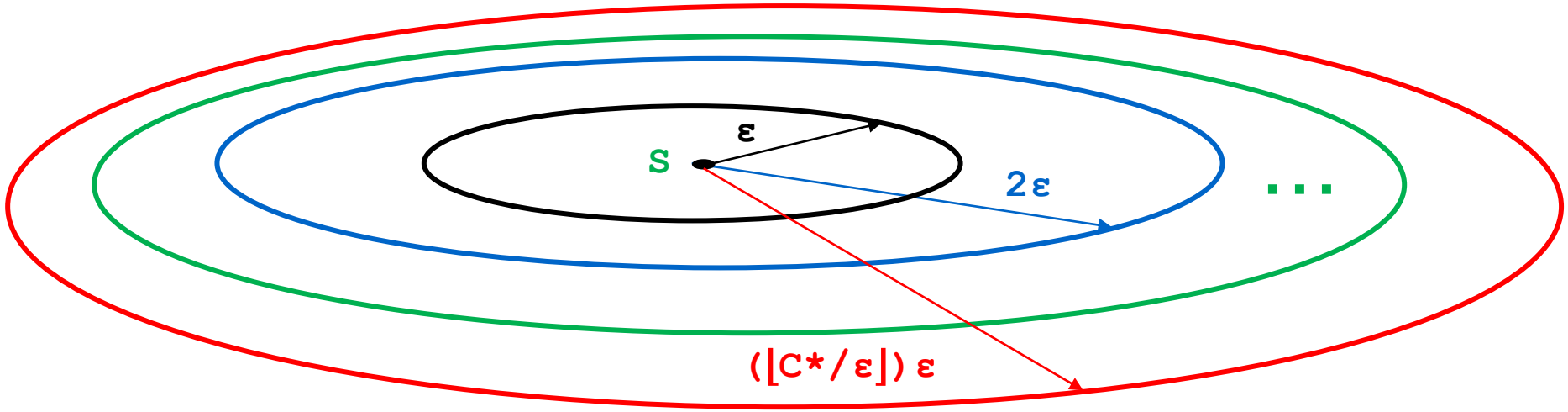
Systematic search affords completeness

- Assuming either:
  - search space finite, or
  - solution exists



UCS & Optimality

- UCS will systematically traverse paths in order of path cost



- Ordered traversal of paths based on path cost is required to ensure optimality

Tree Search 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)

Note that the given UCS complexities abstract away priority queue operation complexities

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

Since BFS / DFS / DLS / IDS all cannot guarantee optimality, using Graph Search Version 1 decreases complexity (since the resultant search tree excludes ALL redundant paths) without loss of expected performance!

Graph Search Algorithm (Version 2)

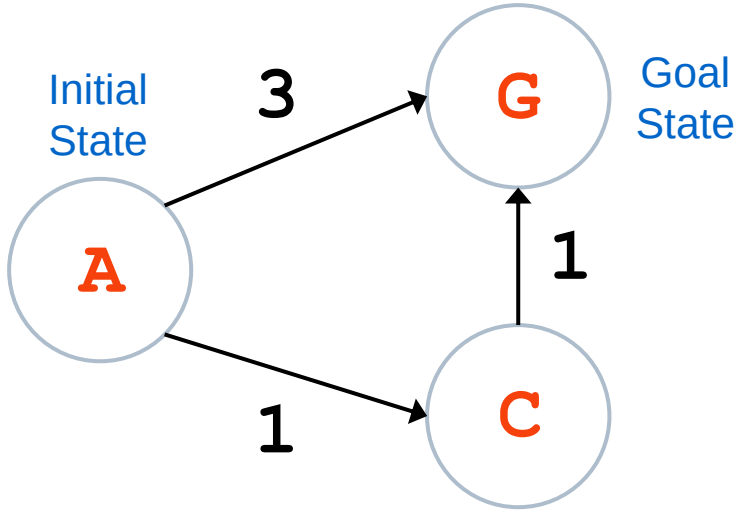
```
Function GraphSearchV2(initial_state, actions, T, isGoal, cost):
   initial_node = Node(initial_state, NULL)
   frontier = {initial_node}
   reached = {initial_state: initial_node}
   while frontier not empty:
         current = frontier.pop()
         if isGoal(current.state): return current.getPath()
         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

Tree Search Versus Graph Search

- Tree search will try ALL paths
  - No paths excluded
  - No issues with optimality
    (i.e., optimal path will not be excluded from the search)
- What about graph search (v2)?
  - Ensures optimal path not among excluded paths
  - Consider the example on the right

Completeness assumptions:
- b finite, and m finite OR has a solution
- All action costs $> \varepsilon > 0$

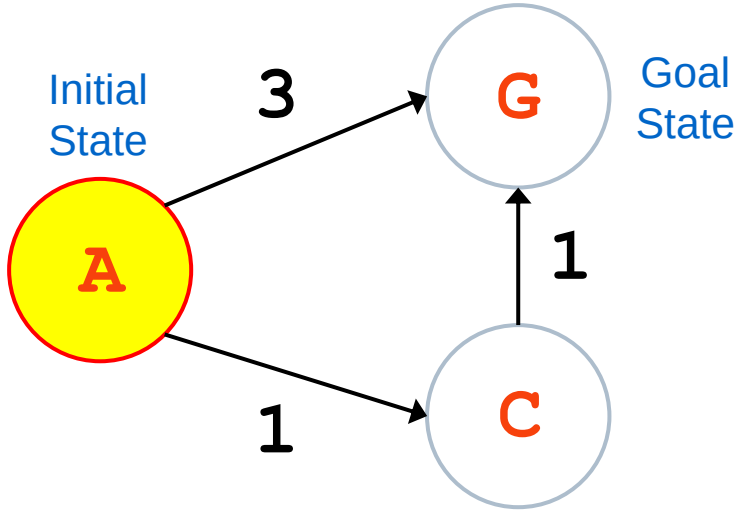


Tree Search Versus Graph Search

- Tree search will try ALL paths
  - No paths excluded
  - No issues with optimality
    (i.e., optimal path will not be excluded from the search)
- What about graph search (v2)?
  - Ensures optimal path not among excluded paths
  - Consider the example on the right
    - Iteration 1: push `(A(-),0)`
      `F = {(A(-),0)}; R = {A}`

`F`: frontier; `R`: reached

Completeness assumptions:
- `b` finite, and `m` finite OR has a solution
- All action costs > `ε` > 0

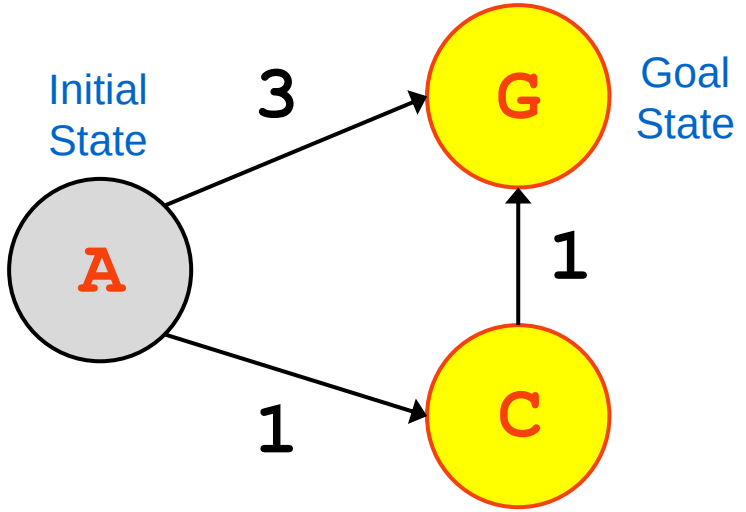


Tree Search Versus Graph Search

- Tree search will try ALL paths
  - No paths excluded
  - No issues with optimality
    (i.e., optimal path will not be excluded from the search)
- What about graph search (v2)?
  - Ensures optimal path not among excluded paths
  - Consider the example on the right
    - Iteration 1: push `(A(-),0)`
      `F = {(A(-),0)}; R = {A}`
    - Iteration 2: pop `(A(-),0)`, push `(C(A),1)` and `(G(A),3)`
      `F = {(C(A),1), (G(A),3)}; R = {A,C,G}`

`F`: frontier; `R`: reached

Completeness assumptions:
- `b` finite, and `m` finite OR has a solution
- All action costs > `ε` > 0

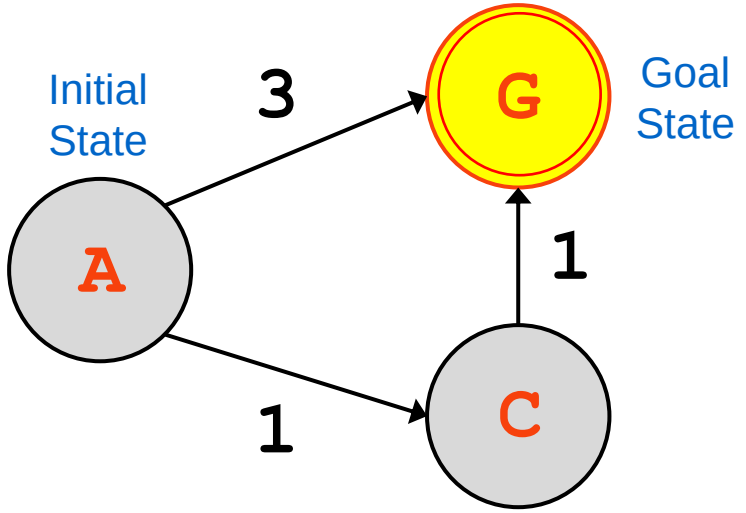


Tree Search Versus Graph Search

- Tree search will try ALL paths
  - No paths excluded
  - No issues with optimality
    (i.e., optimal path will not be excluded from the search)
- What about graph search (v2)?
  - Ensures optimal path not among excluded paths
  - Consider the example on the right
    - Iteration 1: push `(A(-),0)`
      `F = {(A(-),0)}; R = {A}`
    - Iteration 2: pop `(A(-),0)`, push `(C(A),1)` and `(G(A),3)`
      `F = {(C(A),1), (G(A),3)}; R = {A,C,G}`
    - Iteration 3: pop `(C(A),1)`, push `(G(A,C),2)` since lower cost
      `F = {(G(A,C),2), (G(A),3)}; R = {A,C,G}`

`F`: frontier; `R`: reached

Completeness assumptions:
- `b` finite, and `m` finite OR has a solution
- All action costs > `ε` > 0

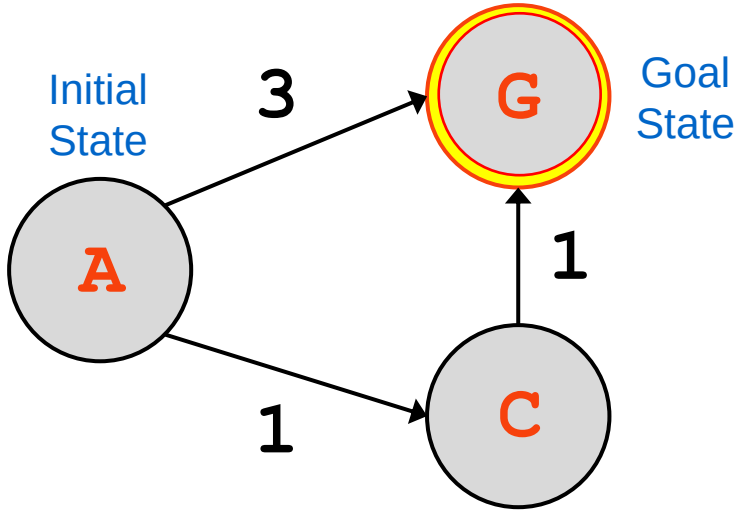


Tree Search Versus Graph Search

- Tree search will try ALL paths
  - No paths excluded
  - No issues with optimality
    (i.e., optimal path will not be excluded from the search)
- What about graph search (v2)?
  - Ensures optimal path not among excluded paths
  - Consider the example on the right
    - Iteration 1: push `(A(-),0)`
      `F = {(A(-),0)}; R = {A}`
    - Iteration 2: pop `(A(-),0)`, push `(C(A),1)` and `(G(A),3)`
      `F = {(C(A),1), (G(A),3)}; R = {A,C,G}`
    - Iteration 3: pop `(C(A),1)`, push `(G(A,C),2)` since lower cost
      `F = {(G(A,C),2), (G(A),3)}; R = {A,C,G}`
    - Iteration 4: pop `(G(A,C),2)`, return optimal path `A,C,G`

`F`: frontier; `R`: reached

Completeness assumptions:
- `b` finite, and `m` finite OR has a solution
- All action costs > `ε` > 0

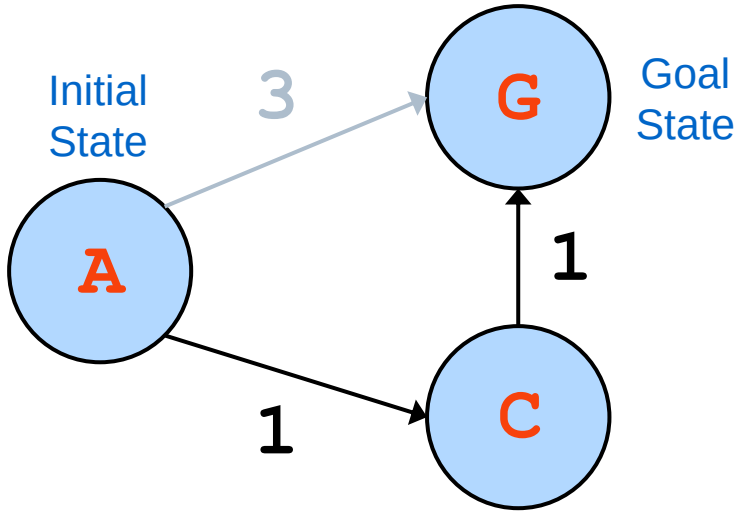


Tree Search Versus Graph Search

- Tree search will try ALL paths
  - No paths excluded
  - No issues with optimality
    (i.e., optimal path will not be excluded from the search)
- What about graph search (v2)?
  - Ensures optimal path not among excluded paths
  - Consider the example on the right
    - Iteration 1: push `(A(-),0)`
      `F = {(A(-),0)}; R = {A}`
    - Iteration 2: pop `(A(-),0)`, push `(C(A),1)` and `(G(A),3)`
      `F = {(C(A),1), (G(A),3)}; R = {A,C,G}`
    - Iteration 3: pop `(C(A),1)`, push `(G(A,C),2)` since lower cost
      `F = {(G(A,C),2), (G(A),3)}; R = {A,C,G}`
    - Iteration 4: pop `(G(A,C),2)`, return optimal path `A,C,G`

`F`: frontier; `R`: reached

Completeness assumptions:
- `b` finite, and `m` finite OR has a solution
- All action costs > `ε` > 0



Notice that without the (v2) update to `G` (when already on `R`), the optimal path would not have been returned (similar to applying an Early Goal Test)

Graph Search 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	$O(\|V\|+\|E\|)$				
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

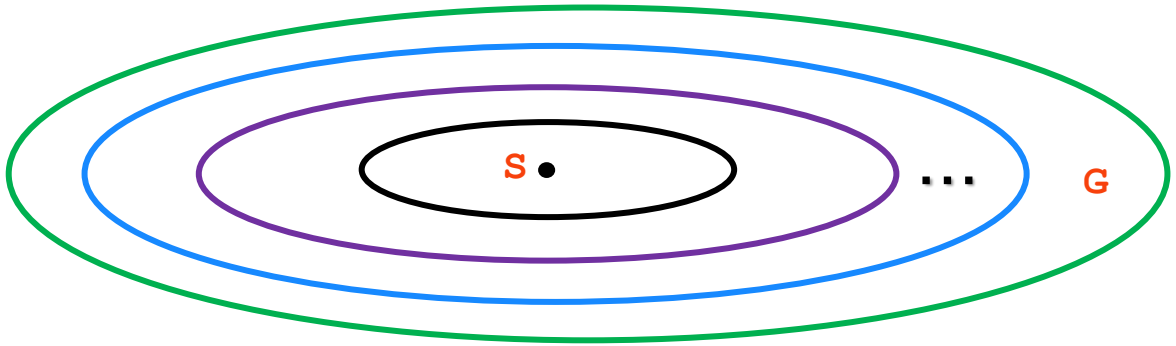
For graph search, assume V2 for UCS and V1 for other uninformed search algorithms

Searching Less?

- Uninformed search algorithms are systematic
  - Search outward from the initial state
  - Search in ALL directions
- Search spaces are LARGE

Searching Less?

- Uninformed search algorithms are systematic
  - Search outward from the initial state
  - Search in ALL directions
- Search spaces are LARGE
- Can we search more efficiently?



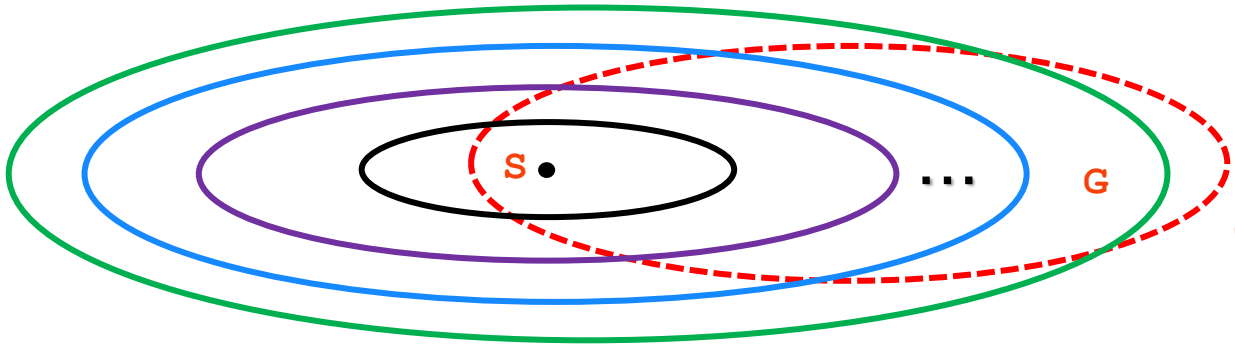
Searching Less?

- Uninformed search algorithms are systematic
  - Search outward from the initial state
  - Search in ALL directions
- Search spaces are LARGE
- Can we search more efficiently?

General Idea

Use domain knowledge about the problem environment to determine the cost required to go from a particular state to its nearest goal

→ informed search

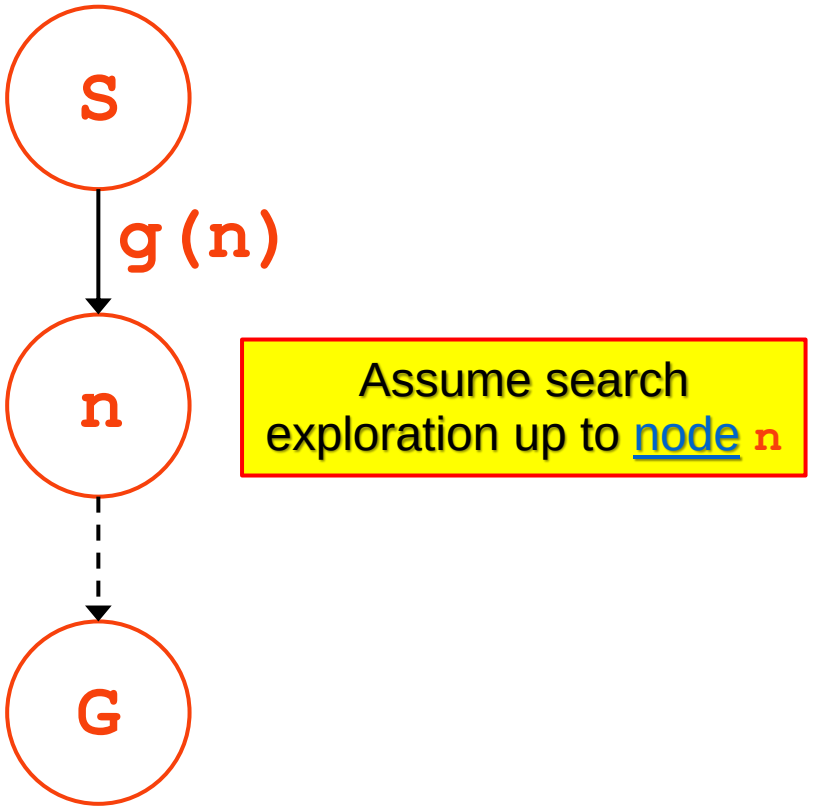


try to focus on searching in the direction of the goal

3 Informed Search

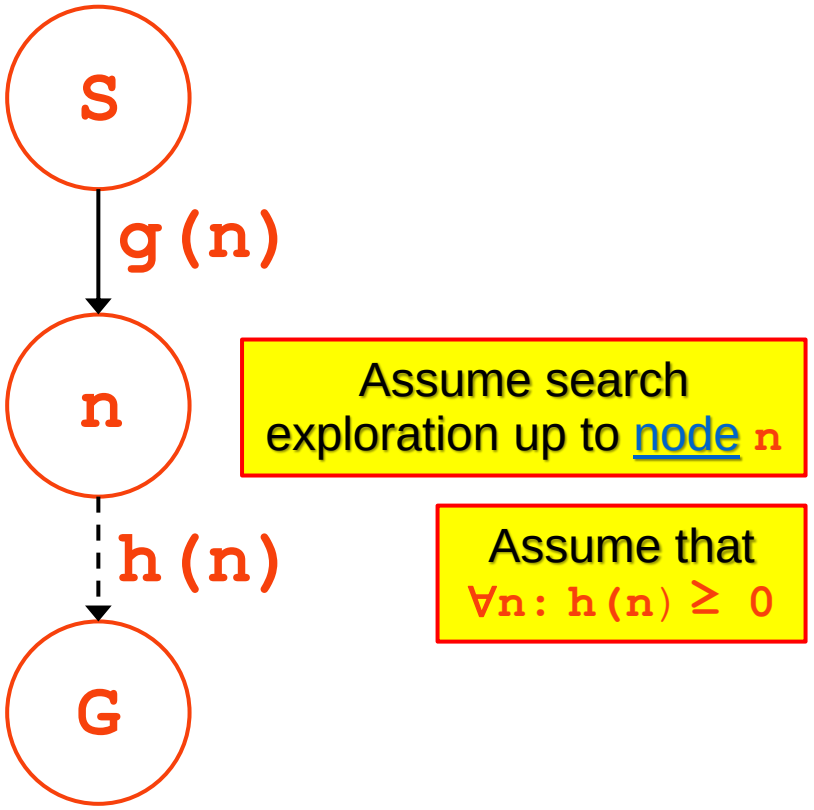
Path Costs & Heuristics

- UCS
  - Frontier → Priority Queue
  - Priority for node `n` → `g(n)`
  - `g(n)` quantifies the path cost from the initial state `S` to `n.state` as defined by `n.path`
- General idea
  - use domain knowledge to estimate the cost from `n.state` to `G`



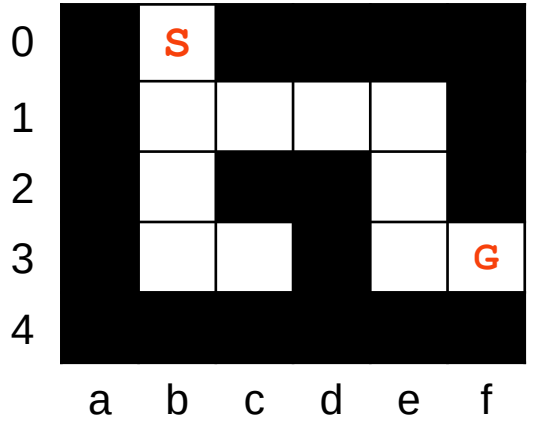
Path Costs & Heuristics

- UCS
  - Frontier → Priority Queue
  - Priority for node `n` → `g(n)`
  - `g(n)` quantifies the path cost from the initial state `S` to `n.state` as defined by `n.path`
- General idea
  - use domain knowledge to estimate the cost from `n.state` to `G`
- Define a heuristic function `h`
  - `h` approximates the path cost from `n.state` to its nearest goal (in terms of cost), `G`



Ideas on Deriving Heuristic Functions

- Consider a Maze Puzzle problem
  - Layout fully observable and deterministic
  - Moves ←,↑,→,↓
  - Find path from `S` to `G`
- Example `h`: Euclidean distance
  - `h(G)`: Euclidean distance from `n.state` to `G`
- General requirements
  - Efficient – e.g., Euclidean distance has `O(k)` complexity, where `k` = number of dimensions
  - More properties discussed later



Always to assume that `h(G) = 0`

General Idea

Use domain knowledge about the costs (e.g., distances) between a given node and its closest goal – i.e., think about how to define the function `h`

More on how to define `h` in the next lecture

Implementation with Evaluation Functions

- Keep using a priority queue for frontier
  - Use different priorities
- Define an evaluation function: `f`
  - Defines the priority for priority queue
  - Priority for node `n` → `f(n)`
  - UCS: priority → `f(n) = g(n)`
- Now consider different evaluation functions
  - Greedy Best-First Search priority: `f(n) = h(n)`
  - A* Search priority: `f(n) = g(n) + h(n)`

Best-First Search Algorithm

- General graph-search implementation

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Late Goal Test

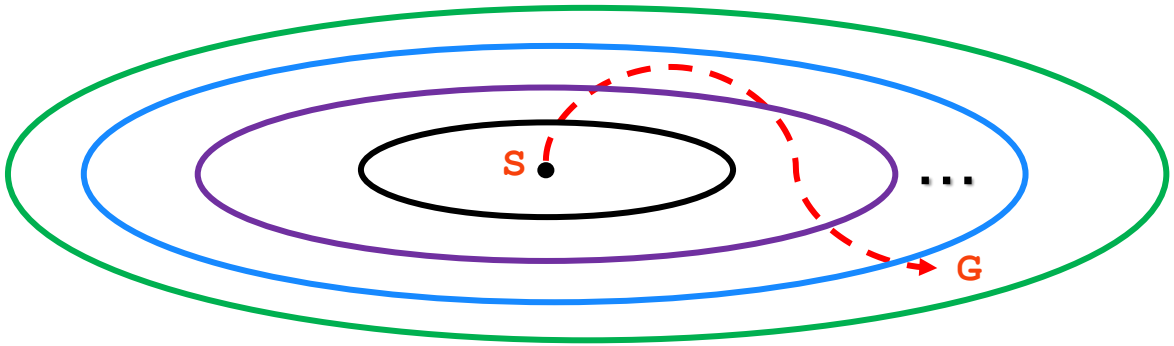
Graph Search (v2)

Utilises search problem definitions

4 Greedy Best-First Search

The Greedy Best-First Search Algorithm

- Implemented like UCS except
  - `f(n) = h(n)`
  - Search in ALL directions



General Idea

keep picking states estimated to be closest to the goal
(based on candidate paths on the frontier)

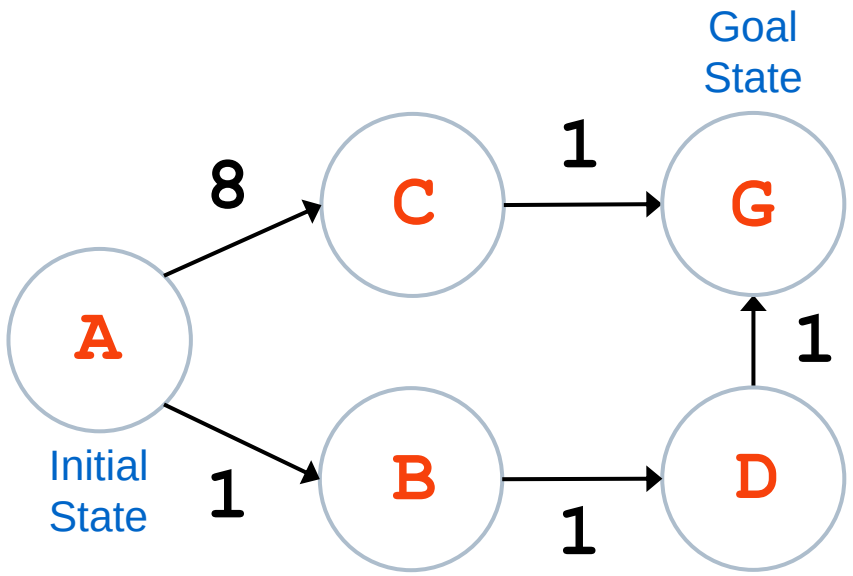
The Greedy Best-First Search Algorithm

- Example (tree-search)

Assume this `h`:

n	h(n)	h*(n)
A	3	3
B	2	2
C	1	1
D	1	1
G	0	0

`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`



Node representation: `S(P),C`
- Path `P` denotes the path taken to state `S`
- With priority `C` - i.e., path cost from `A` to `n`

Trace (note: priority `C` = `h(n)`)

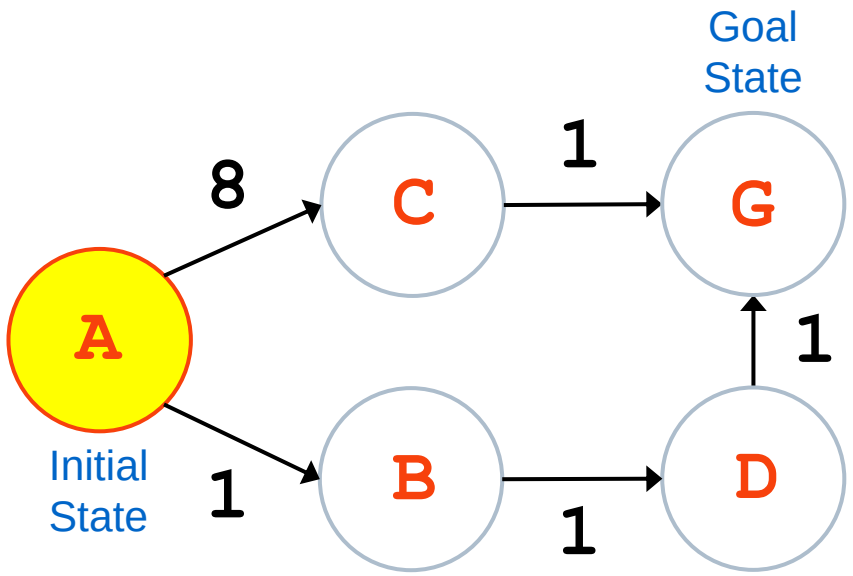
The Greedy Best-First Search Algorithm

- Example (tree-search)

Assume this `h`:

n	h(n)	h*(n)
A	3	3
B	2	2
C	1	1
D	1	1
G	0	0

`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`



Node representation: `S(P),C`
- Path `P` denotes the path taken to state `S`
- With priority `C` - i.e., path cost from `A` to `n`

Trace (note: priority `C` = `h(n)`)

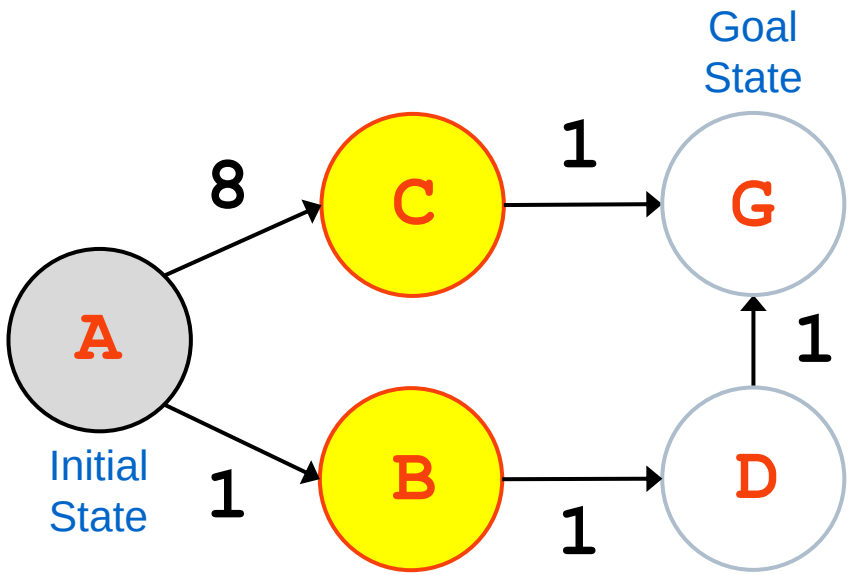
```
ITR1: {A((-),3)}
```

The Greedy Best-First Search Algorithm

- Example (tree-search)

Assume this `h`:

n	h(n)	h*(n)
A	3	3
B	2	2
C	1	1
D	1	1
G	0	0



Node representation: `S(P),C`
- Path `P` denotes the path taken to state `S`
- With priority `C` - i.e., path cost from `A` to `n`

Trace (note: priority `C` = `h(n)`)

```
ITR1: {A((-),3)}
ITR2:{C((A),1),B((A),2)}
```

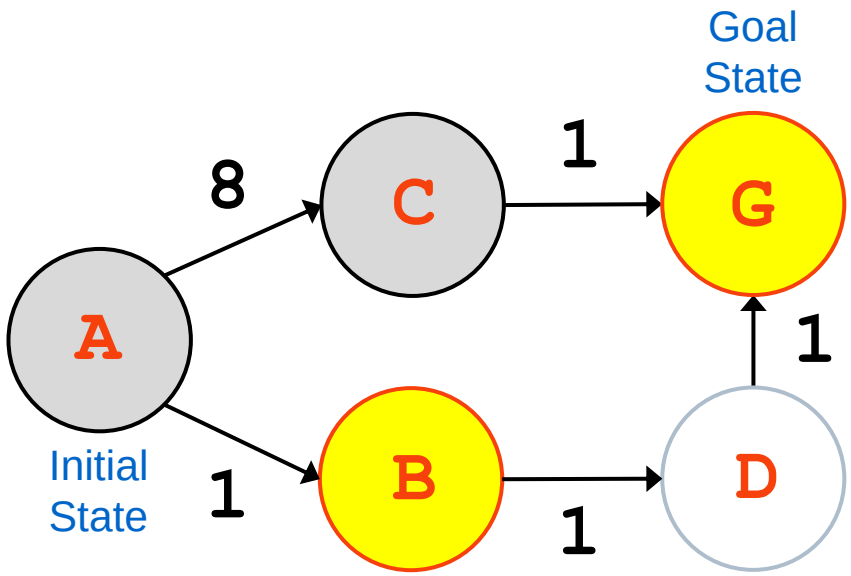
`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`

The Greedy Best-First Search Algorithm

- Example (tree-search)

Assume this `h`:

n	h(n)	h*(n)
A	3	3
B	2	2
C	1	1
D	1	1
G	0	0



Node representation: `S(P),C`
- Path `P` denotes the path taken to state `S`
- With priority `C` - i.e., path cost from `A` to `n`

Trace (note: priority `C` = `h(n)`)

```
ITR1: {A((-),3)}
ITR2:{C((A),1),B((A),2)}
ITR3: {G((A,C),0),B((A),2)}
```

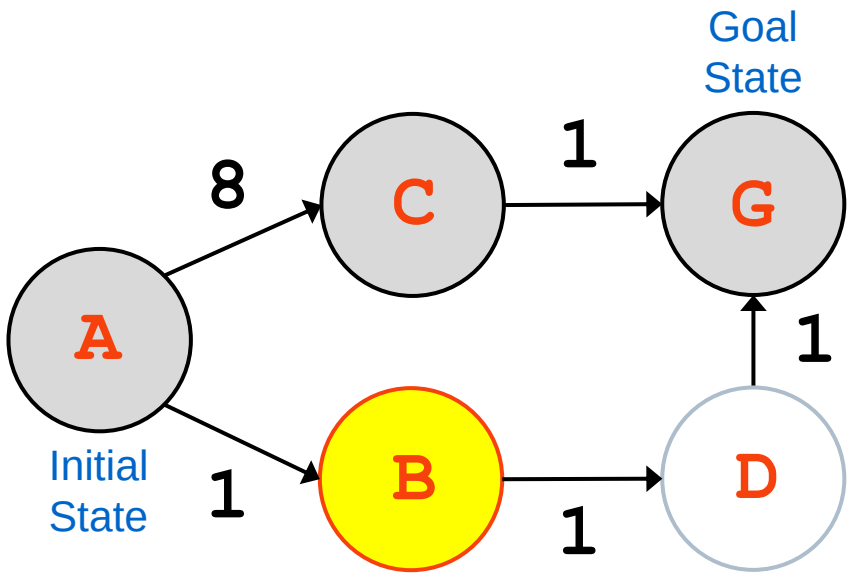
`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`

The Greedy Best-First Search Algorithm

- Example (tree-search)

Assume this `h`:

n	h(n)	h*(n)
A	3	3
B	2	2
C	1	1
D	1	1
G	0	0



Node representation: `S(P),C`
- Path `P` denotes the path taken to state `S`
- With priority `C` - i.e., path cost from `A` to `n`

Trace (note: priority `C` = `h(n)`)

```
ITR1: {A((-),3)}
ITR2:{C((A),1),B((A),2)}
ITR3: {G((A,C),0),B((A),2)}
ITR4:  DONE (A,C,G)
```

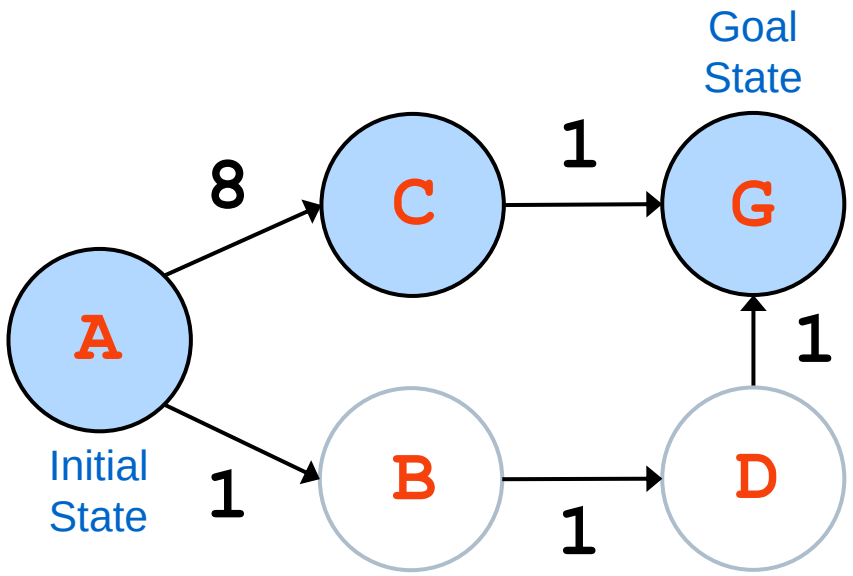
`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`

The Greedy Best-First Search Algorithm

- Example (tree-search)

Assume this `h`:

n	h(n)	h*(n)
A	3	3
B	2	2
C	1	1
D	1	1
G	0	0



Node representation: `S(P),C`
- Path `P` denotes the path taken to state `S`
- With priority `C` - i.e., path cost from `A` to `n`

Trace (note: priority `C` = `h(n)`)

```
ITR1: {A((-),3)}
ITR2:{C((A),1),B((A),2)}
ITR3: {G((A,C),0),B((A),2)}
ITR4:  DONE  (A,C,G)
```

`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`

Notice that even with the perfect heuristic, we may not get the optimal solution. Why?

Algorithm never exploits information on path already travelled; as such, it cannot order the paths in terms of path cost!

Completeness & Optimality of the Greedy Best-First Search Algorithm

- Tree-search implementation is incomplete!
  - General idea
    - Can get stuck in a loop between nodes where `h` values are lowest
    - Prove with completeness counterexample – T02 Q1a
- Graph-search implementation is complete if search space is finite
  - General idea
    - With no revisits, in finite state space, will visit entire space eventually
    - Prove – T02 Q1b
- Not optimal under either tree search or graph search
  - As shown in example on previous slide
  - Find another example – T02 Q1c

General Idea

Ordered traversal of paths in increasing order of path cost required to ensure optimality

Questions about the Lecture?

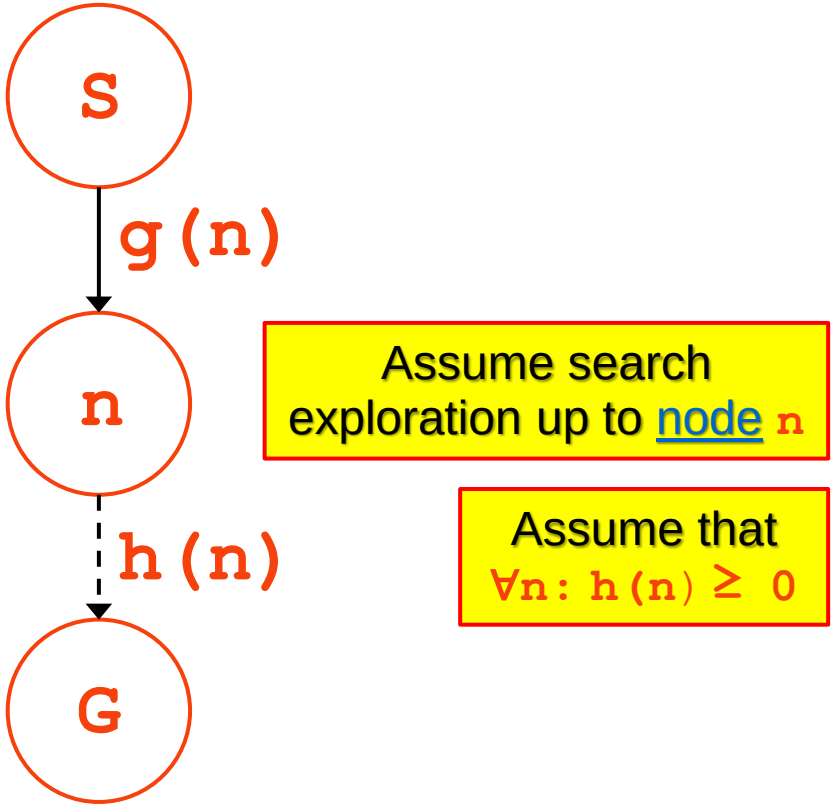
- Was anything unclear?
- Do you need to clarify anything?
- Ask on Archipelago
  - Specify a question
  - Upvote someone else's question

Invitation Link (Use NUS Email --- starts with E)
https://archipelago.rocks/app/resend-invite/86826503199

5 A* Search

The A* Search Algorithm

- Greedy Best-First Search
  - With greedy, `f(n) = h(n)`
  - Does not consider `g(n)`
  - Means that you cannot ensure a strictly monotonically increasing ordering of paths → cannot ensure optimality
- Accounting for costs already incurred: A* Search
  - With A*, `f(n) = g(n) + h(n)`
- A* priorities
  - Based on total path cost estimates from `S` to `G`
    - Actual cost of path `S` to `n` (i.e., node `n`) is known
    - Path `n` to `G` is unknown; here, we approximate the cost to nearest goal



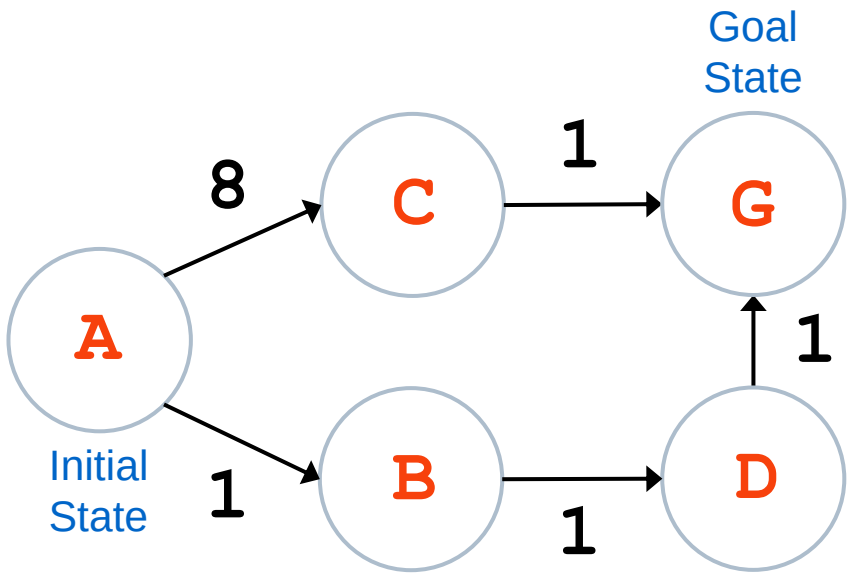
The A* Search Algorithm

- Example (tree-search)

Assume this `h`:

n	h(n)	h*(n)
A	3	3
B	2	2
C	1	1
D	1	1
G	0	0

`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`



Node representation: `S(P),C`
- Path `P` denotes the path taken to state `S`
- With priority `C` (i.e., path cost from `A` to `n` + approx. path cost from `n` to `G`)

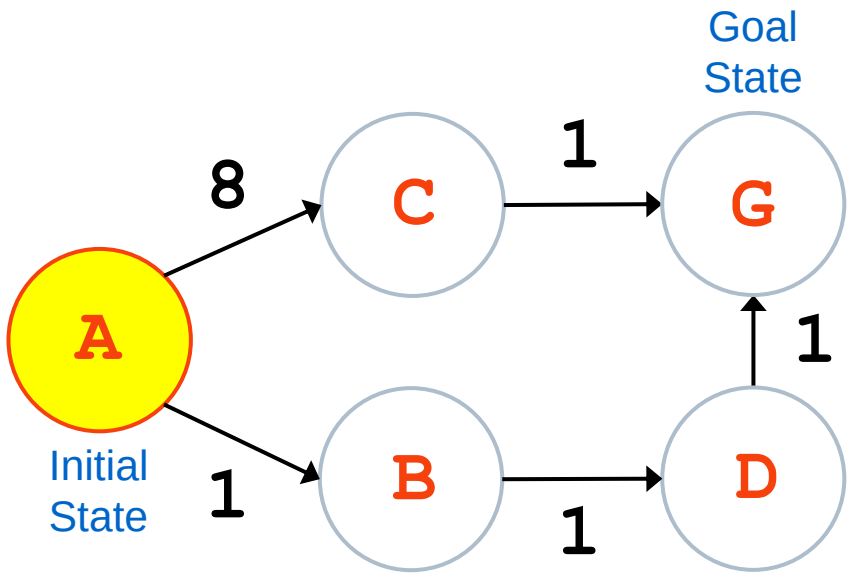
Trace (note: priority `C` = `g(n)+h(n)`)

The A* Search Algorithm

- Example (tree-search)

Assume this `h`:

n	h(n)	h*(n)
A	3	3
B	2	2
C	1	1
D	1	1
G	0	0



`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`

Node representation: `S(P),C`
- Path `P` denotes the path taken to state `S`
- With priority `C` (i.e., path cost from `A` to `n` + approx. path cost from `n` to `G`)

Trace (note: priority `C` = `g(n)+h(n)`)

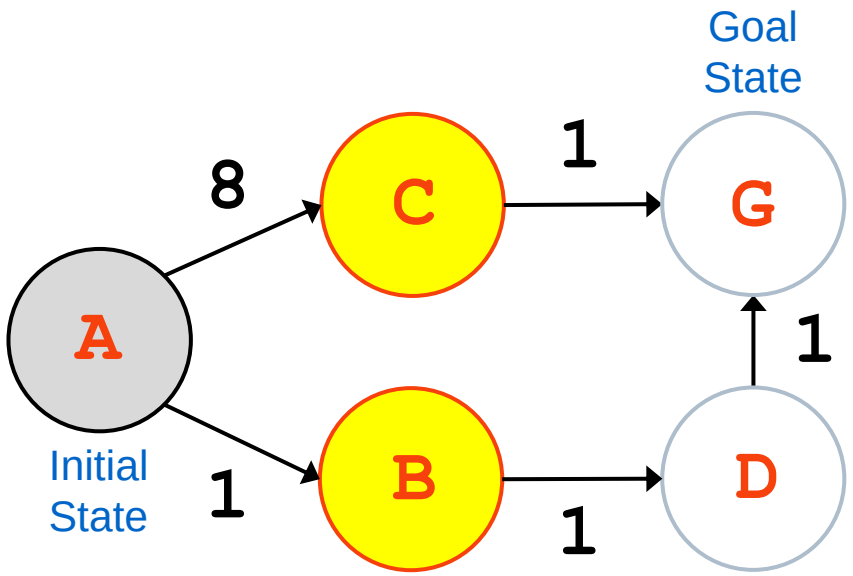
`ITR1: {A((-),0+3)}`

The A* Search Algorithm

- Example (tree-search)

Assume this `h`:

n	h(n)	h*(n)
A	3	3
B	2	2
C	1	1
D	1	1
G	0	0



`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`

Node representation: `S(P),C`
- Path `P` denotes the path taken to state `S`
- With priority `C` (i.e., path cost from `A` to `n` + approx. path cost from `n` to `G`)

Trace (note: priority `C = g(n)+h(n)`)

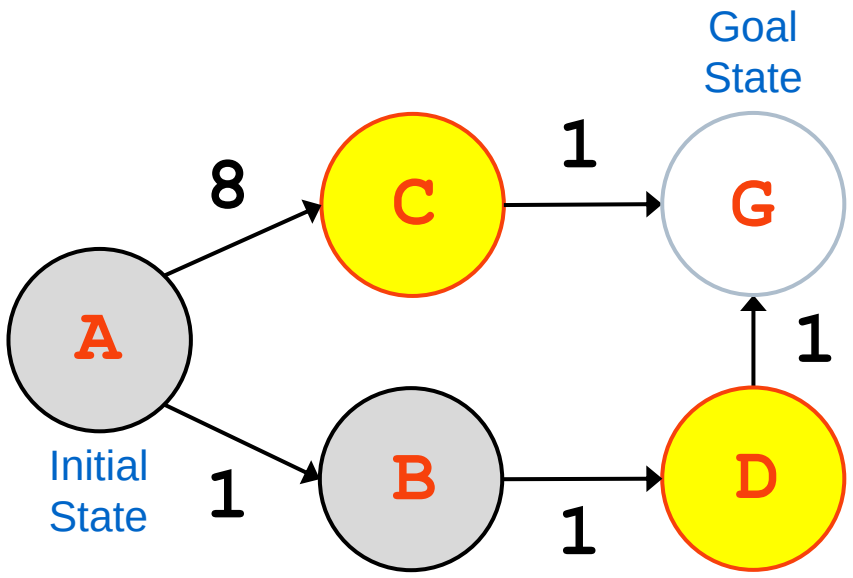
```
ITR1: {A((-),0+3)}
ITR2: {B((A),1+2),C((A),8+1)}
```

The A* Search Algorithm

- Example (tree-search)

Assume this `h`:

n	h(n)	h*(n)
A	3	3
B	2	2
C	1	1
D	1	1
G	0	0



`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`

Node representation: `S(P),C`
- Path `P` denotes the path taken to state `S`
- With priority `C` (i.e., path cost from `A` to `n` + approx. path cost from `n` to `G`)

Trace (note: priority `C = g(n)+h(n)`)

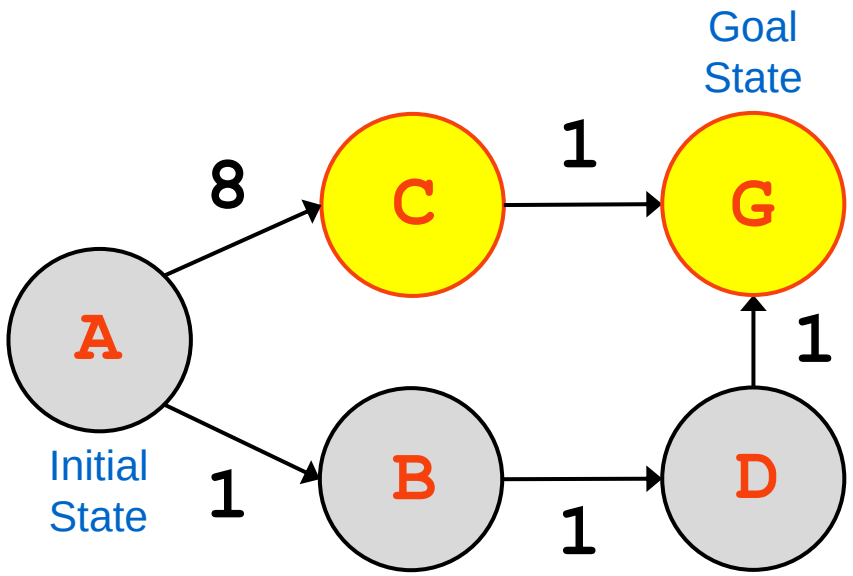
```
ITR1: {A((-),0+3)}
ITR2: {B((A),1+2),C((A),8+1)}
ITR3: {D((A,B),2+1),C((A),8+1)}
```

The A* Search Algorithm

- Example (tree-search)

Assume this `h`:

n	h(n)	h*(n)
A	3	3
B	2	2
C	1	1
D	1	1
G	0	0



`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`

Node representation: `S(P),C`
- Path `P` denotes the path taken to state `S`
- With priority `C` (i.e., path cost from `A` to `n` + approx. path cost from `n` to `G`)

Trace (note: priority `C = g(n)+h(n)`)

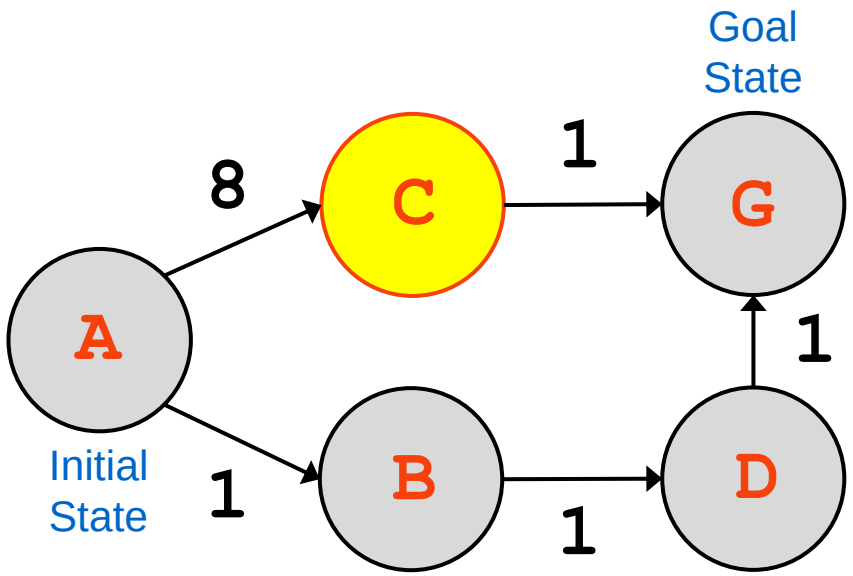
```
ITR1: {A((-),0+3)}
ITR2: {B((A),1+2),C((A),8+1)}
ITR3: {D((A,B),2+1),C((A),8+1)}
ITR4: {G((A,B,D),3+0),C((A),8+1)}
```

The A* Search Algorithm

- Example (tree-search)

Assume this `h`:

n	h(n)	h*(n)
A	3	3
B	2	2
C	1	1
D	1	1
G	0	0



`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`

Node representation: `S(P),C`
- Path `P` denotes the path taken to state `S`
- With priority `C` (i.e., path cost from `A` to `n` + approx. path cost from `n` to `G`)

Trace (note: priority `C = g(n)+h(n)`)

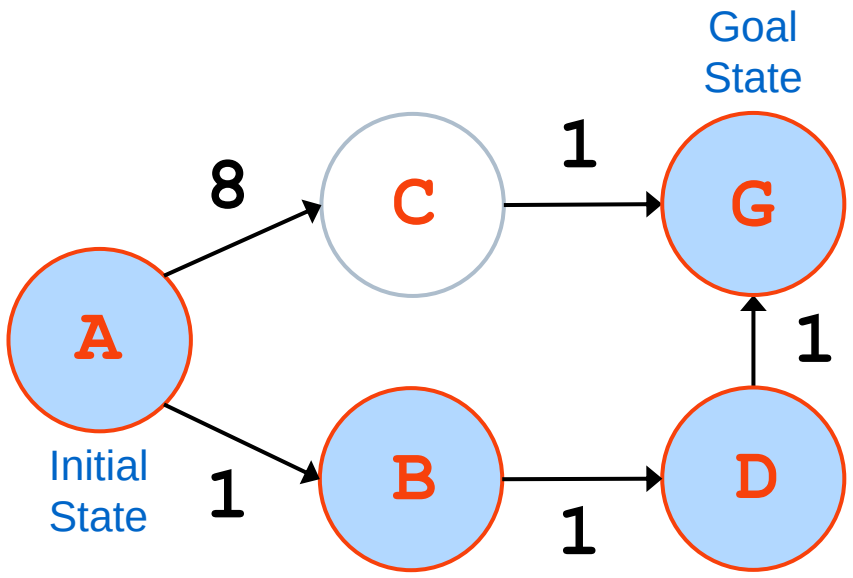
```
ITR1: {A((-),0+3)}
ITR2: {B((A),1+2),C((A),8+1)}
ITR3: {D((A,B),2+1),C((A),8+1)}
ITR4: {G((A,B,D),3+0),C((A),8+1)}
ITR5: DONE (A,B,D,G)
```

The A* Search Algorithm

- Example (tree-search)

Assume this `h`:

n	h(n)	h*(n)
A	3	3
B	2	2
C	1	1
D	1	1
G	0	0



`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`

Node representation: `S(P),C`
- Path `P` denotes the path taken to state `S`
- With priority `C` (i.e., path cost from `A` to `n` + approx. path cost from `n` to `G`)

Trace (note: priority `C = g(n)+h(n)`)

```
ITR1: {A((-),0+3)}
ITR2: {B((A),1+2),C((A),8+1)}
ITR3: {D((A,B),2+1),C((A),8+1)}
ITR4: {G((A,B,D),3+0),C((A),8+1)}
ITR5: DONE (A,B,D,G)
```

A* outputs the optimal solution, unlike the Greedy Best-First Search

Will it always be optimal? What about graph search?

A* Search Algorithm: Completeness & Optimality

- Completeness
  - Same criteria as UCS
    - b finite, and m finite OR has a solution
    - All action costs > ε > 0
- Optimality
  - Depends on the properties of h

Recall: An ascending traversal of paths based on path cost is required to ensure optimality

Admissible Heuristics

- $h(n)$ is admissible if $\forall n, h(n) \leq h^*(n)$
  - $h(n)$ never overestimates the cost
    - Implications
      - Paths not ending at a goal are never over-estimated
        - Evaluation function for non-goal is never over-estimated
        - At non-goal n, $f(n) = \boxed{g(n) + h(n)} \leq \boxed{g(n) + h^*(n)}$
          approximation via h ; actual cost (to goal nearest n)
      - Paths ending at a goal are exact
        - Evaluation function of value of a goal is exact
        - At goal m, $f(m) = g(m) + h(m)$, where $h(m) = 0$
- Theorem: If $h(n)$ is admissible, then A* using tree search is optimal

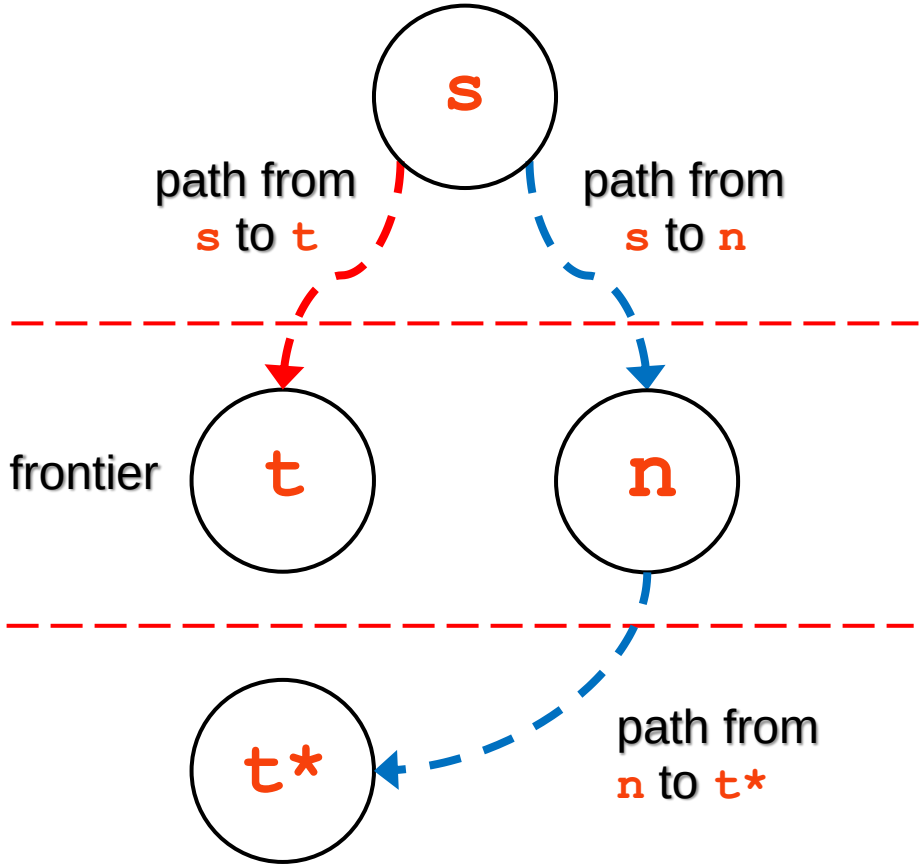
Main Idea

By the time we explore a path to a goal, P, all paths with actual costs less than P must be searched. Although, not necessarily strictly in ascending order of path cost.

Example h: Euclidean distance in the maze environment (as it always underestimates the actual distance)

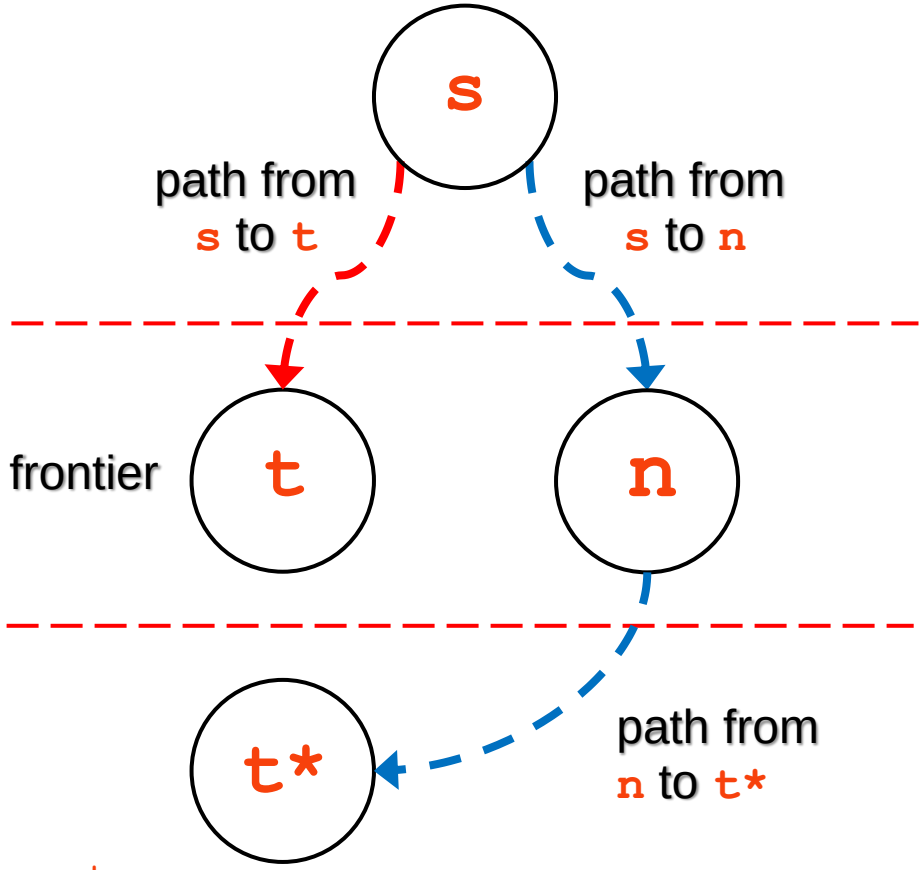
Optimality of A* Using Tree Search

- Proof sketch (proposition for proof by contradiction)
  - A* is optimal → A* returns optimal path
    - Let `s` to `n` to `t*` be the optimal path
  - If A* is not optimal:
    - Must explore path to `t` first
    - Where `t` is another (non-optimal) goal
  - Not optimal → explore `t` before `n`
    (while both `t` and `n` are on the frontier)
  - Proposition:
    Assert that A* will expand non-optimal `t` before `n`
    (where `n` is along the optimal path)
    ⇒ `f(n) > f(t)`



Optimality of A* Using Tree Search

- Proof sketch (working with proposition: `f(n) > f(t)`)
  - Assuming tree-search (search all paths)
    - i.e., no paths are left out
  - All sub-paths** along a single path to a goal must be searched before the path including that goal
    - For non-goal `m`, $f(m) \leq g(m) + h^*(m)$ (since admissible)
    - If goal `m*` on path from `m`, $f(m) \leq f(m^*)$ (given ε)
  - Since `t*` is a goal on the optimal path
    ⇒ $f(n) < f(t^*) < f(t)$ (asserting `t` is non-optimal goal)
  - The above contradicts our proposition
    - Prove more formally – T02 Q2a



** Consider a path, `P`, from an initial state, `s`, to a goal state, `t`, to be $s \rightarrow n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_k \rightarrow t$.
Let a sub-path of `P`, `P'` be any path $s \rightarrow n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_i$, w here $1 \leq i < k$.

A* Using Graph Search

- Difference between tree search and graph search
  - Under admissibility and tree search
    - All nodes leading to a goal (i.e., sub-paths) are expanded before the node representing the goal
    - Optimal path will be found
  - Under graph search we may skip some paths (due to no revisiting) so may skip the optimal path
- Under graph search (version 2), non-redundant paths never skipped
  - Graph search checks and allows revisits if path is non-redundant
    - If a path is cheaper, Graph Search Version 2 will allow it onto the frontier even if must revisit
  - Still optimal since equivalent to tree search
- Under graph search (version 1) may skip optimal path!
  - Not optimal!

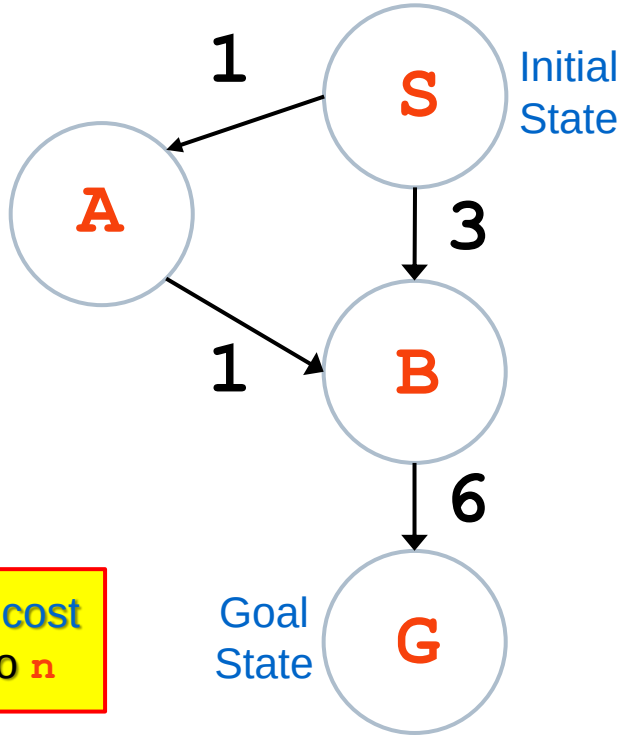
Is A* Optimal Under Graph Search (Version 1)?

- Example (tree-search)

Assume this admissible `h`:

n	h(n)	h*(n)
S	8	8
A	7	7
B	0	6
G	0	0

`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`



Trace (note: priority `c = g(n)+h(n)`)

Frontier: Reached:

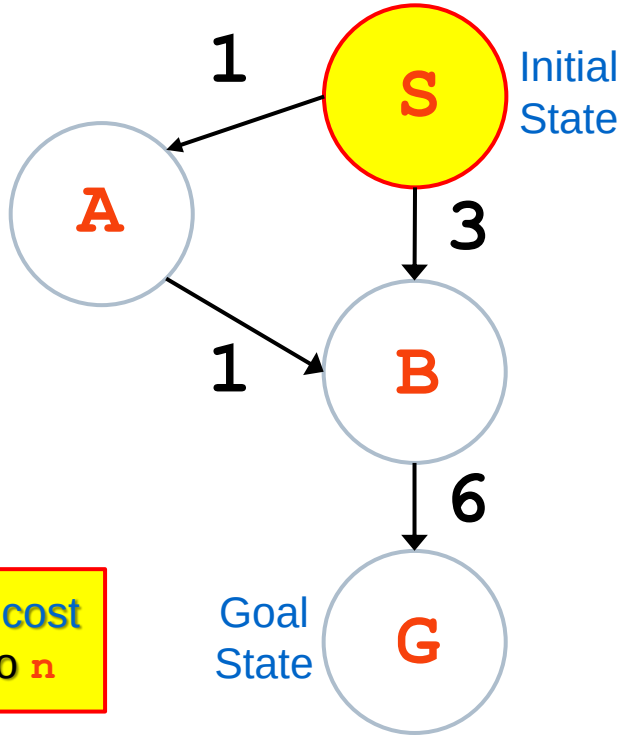
Is A* Optimal Under Graph Search (Version 1)?

- Example (tree-search)

Assume this admissible `h`:

n	h(n)	h*(n)
S	8	8
A	7	7
B	0	6
G	0	0

`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`



Trace (note: priority `c = g(n)+h(n)`)

	Frontier:	Reached:
ITR1:	{ S((-),0) }	{ S }

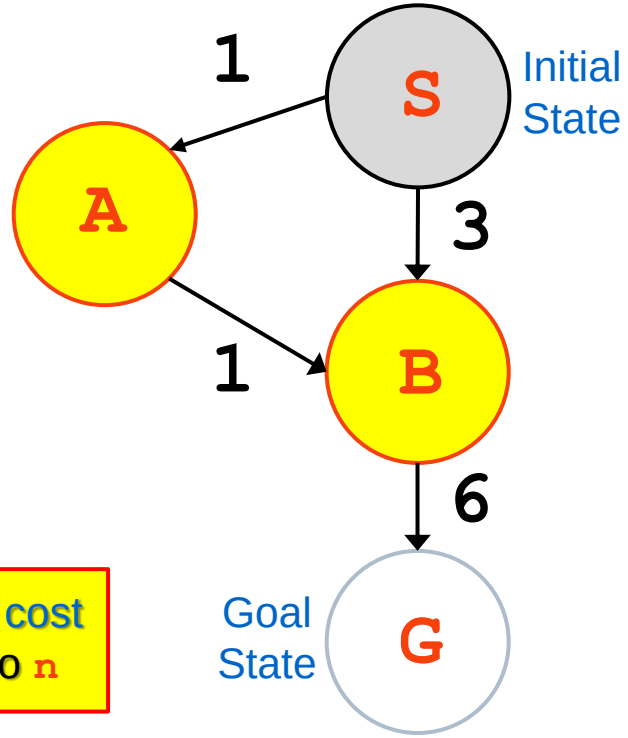
Is A* Optimal Under Graph Search (Version 1)?

- Example (tree-search)

Assume this admissible `h`:

n	h(n)	h*(n)
S	8	8
A	7	7
B	0	6
G	0	0

`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`



Trace (note: priority `C = g(n)+h(n)`)

	Frontier:	Reached:
ITR1:	{ S((-),0) }	{ S }
ITR2:	{ B((S),3+0), A((S),1+7) }	{ S,A,B }

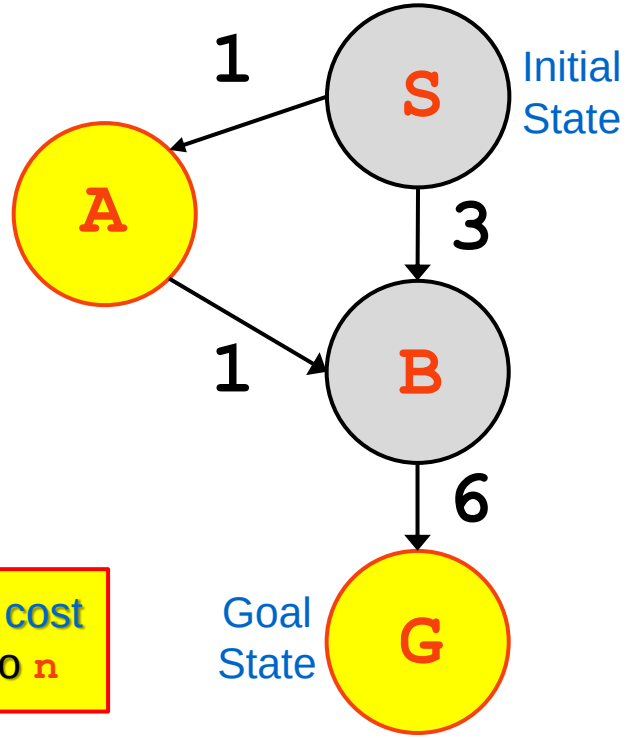
Is A* Optimal Under Graph Search (Version 1)?

- Example (tree-search)

Assume this admissible `h`:

n	h(n)	h*(n)
S	8	8
A	7	7
B	0	6
G	0	0

`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`



Trace (note: priority `C = g(n)+h(n)`)

	Frontier:	Reached:
ITR1:	{ S((-),0) }	{ S }
ITR2:	{ B((S),3+0), A((S),1+7) }	{ S,A,B }
ITR3:	{ A((S),1+7), G((S,B),9+0) }	{ S,A,B,G }

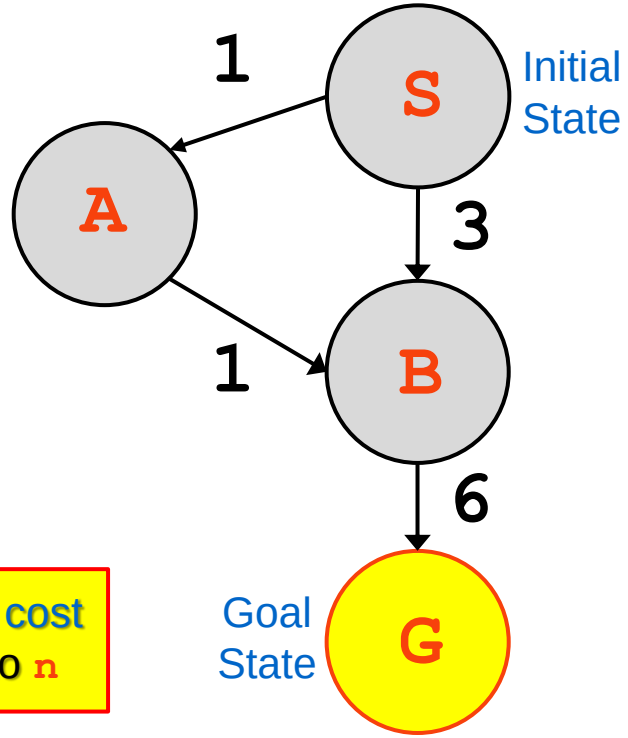
Is A* Optimal Under Graph Search (Version 1)?

- Example (tree-search)

Assume this admissible `h`:

n	h(n)	h*(n)
S	8	8
A	7	7
B	0	6
G	0	0

`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`



Trace (note: priority `C = g(n)+h(n)`)

	Frontier:	Reached:
ITR1:	{ S((-),0) }	{ S }
ITR2:	{ B((S),3+0), A((S),1+7) }	{ S,A,B }
ITR3:	{ A((S),1+7), G((S,B),9+0) }	{ S,A,B,G }
ITR4:	{ G((S,B),9+0) }	{ S,A,B,G }

cannot push `B` since on reached

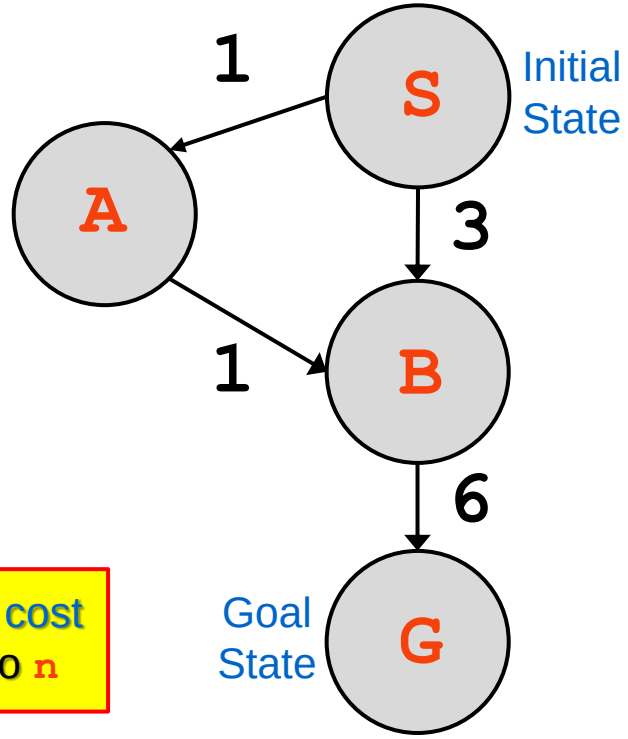
Is A* Optimal Under Graph Search (Version 1)?

- Example (tree-search)

Assume this admissible `h`:

n	h(n)	h*(n)
S	8	8
A	7	7
B	0	6
G	0	0

`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`



Trace (note: priority `C = g(n)+h(n)`)

	Frontier:	Reached:
ITR1:	{ S((-),0) }	{ S }
ITR2:	{ B((S),3+0), A((S),1+7) }	{ S,A,B }
ITR3:	{ A((S),1+7), G((S,B),9+0) }	{ S,A,B,G }
ITR4:	{ G((S,B),9+0) }	{ S,A,B,G }
ITR5:	DONE((S,B,G),9)	

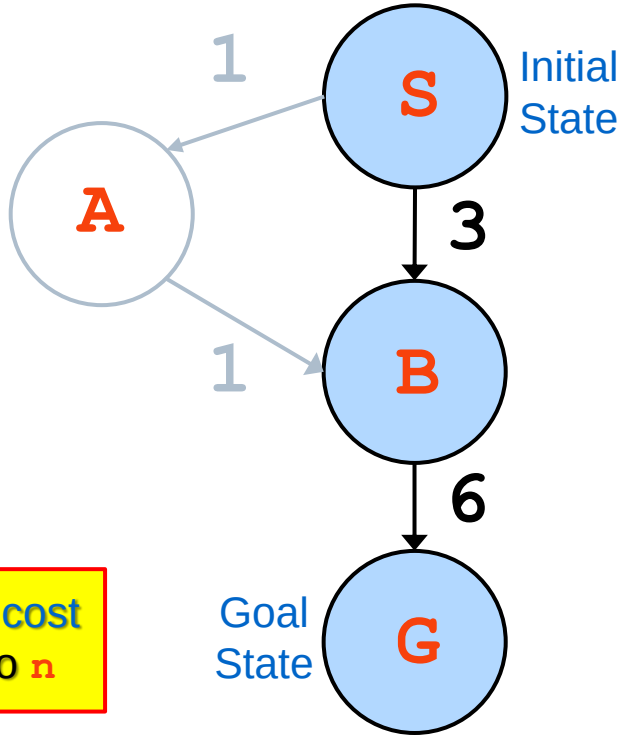
Is A* Optimal Under Graph Search (Version 1)?

- Example (tree-search)

Assume this admissible `h`:

n	h(n)	h*(n)
S	8	8
A	7	7
B	0	6
G	0	0

`h*(n)` denotes the true path cost from `n` to the nearest goal to `n`



Trace (note: priority `C = g(n)+h(n)`)

	Frontier:	Reached:
ITR1:	{ S((-),0) }	{ S }
ITR2:	{ B((S),3+0), A((S),1+7) }	{ S,A,B }
ITR3:	{ A((S),1+7), G((S,B),9+0) }	{ S,A,B,G }
ITR4:	{ G((S,B),9+0) }	{ S,A,B,G }
ITR5:	DONE((S,B,G),9)	// Non-optimal path!

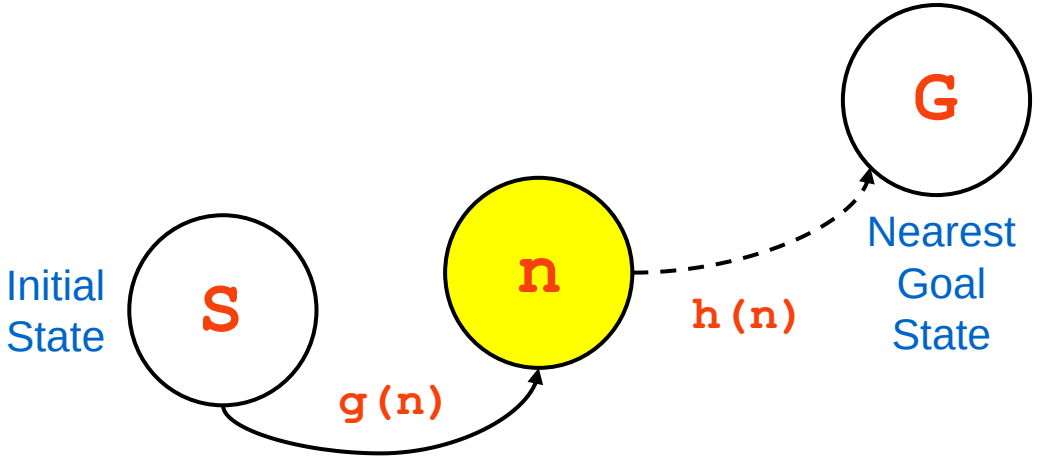
We need a tighter constraint on `h`; like UCS → contours of search progression - i.e., stricter ordering of paths

Consistent Heuristics

- Forming contours
  - Under tree search `g` costs are monotonically increasing
    - `f` costs are not since we can underestimate `h` by differing amounts
    - But ALL paths with path costs less than the optimal path cost will be explored first
  - For `f` costs to be monotonically increasing along a path
    - Assume `n` is a predecessor of `n'` along a path
    - We need: `g(n) + h(n) ≤ g(n) + cost(n,a,n') + h(n')`

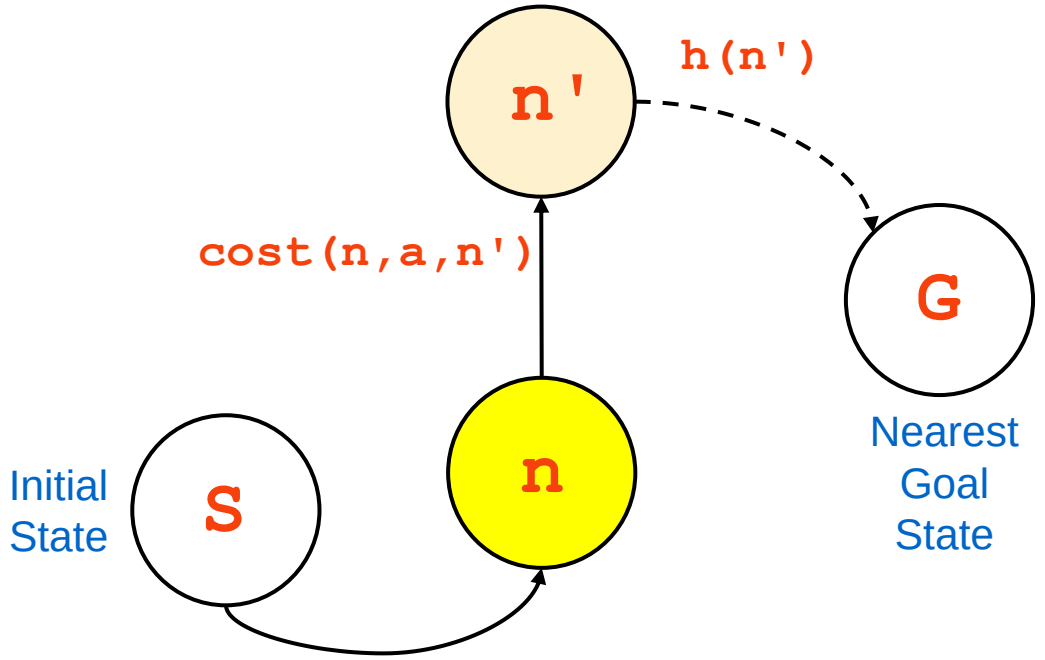
Consistent Heuristics

- Forming contours
  - Under tree search `g` costs are monotonically increasing
    - `f` costs are not since we can underestimate `h` by differing amounts
    - But ALL paths with path costs less than the optimal path cost will be explored first
  - For `f` costs to be monotonically increasing along a path
    - Assume `n` is a predecessor of `n'` along a path
    - We need: `g(n) + h(n)` ≤ `g(n) + cost(n,a,n') + h(n')`



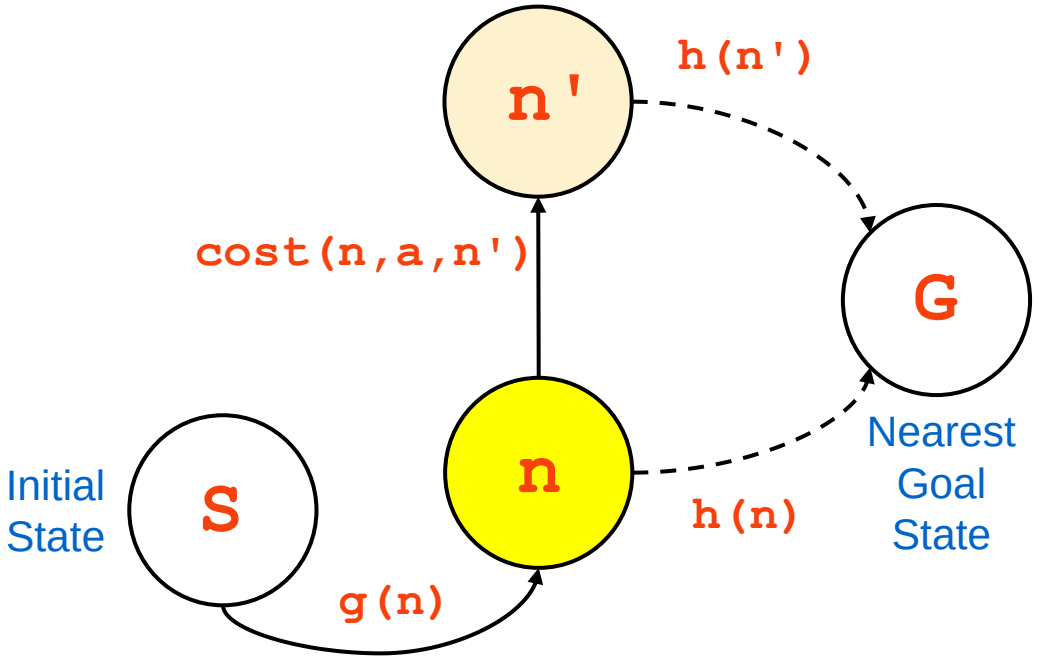
Consistent Heuristics

- Forming contours
  - Under tree search `g` costs are monotonically increasing
    - `f` costs are not since we can underestimate `h` by differing amounts
    - But ALL paths with path costs less than the optimal path cost will be explored first
  - For `f` costs to be monotonically increasing along a path
    - Assume `n` is a predecessor of `n'` along a path
    - We need: `g(n) + h(n)` ≤ `g(n) + cost(n,a,n') + h(n')`



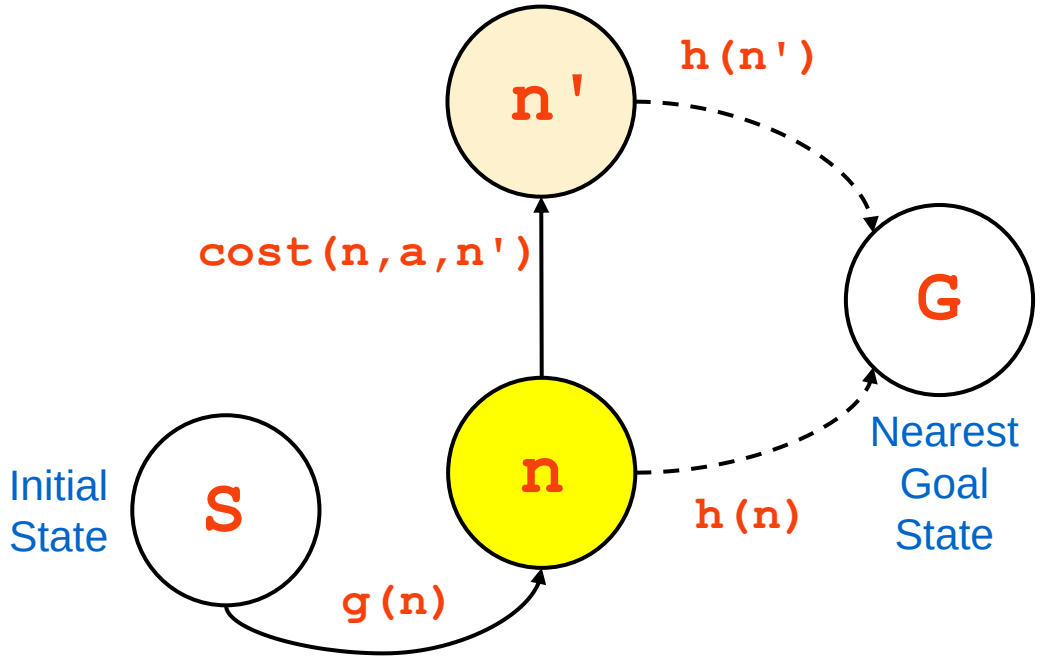
Consistent Heuristics

- Forming contours
  - Under tree search `g` costs are monotonically increasing
    - `f` costs are not since we can underestimate `h` by differing amounts
    - But ALL paths with path costs less than the optimal path cost will be explored first
  - For `f` costs to be monotonically increasing along a path
    - Assume `n` is a predecessor of `n'` along a path
    - We need: `g(n) + h(n) ≤ g(n) + cost(n,a,n') + h(n')`
    - And thus need: `h(n) ≤ cost(n,a,n') + h(n')`



Consistent Heuristics

- Forming contours
  - Under tree search `g` costs are monotonically increasing
    - `f` costs are not since we can underestimate `h` by differing amounts
    - But ALL paths with path costs less than the optimal path cost will be explored first
  - For `f` costs to be monotonically increasing along a path
    - Assume `n` is a predecessor of `n'` along a path
    - We need: `g(n) + h(n) ≤ g(n) + cost(n,a,n') + h(n')`
    - And thus need: `h(n) ≤ cost(n,a,n') + h(n')`
  - We will use the above requirement



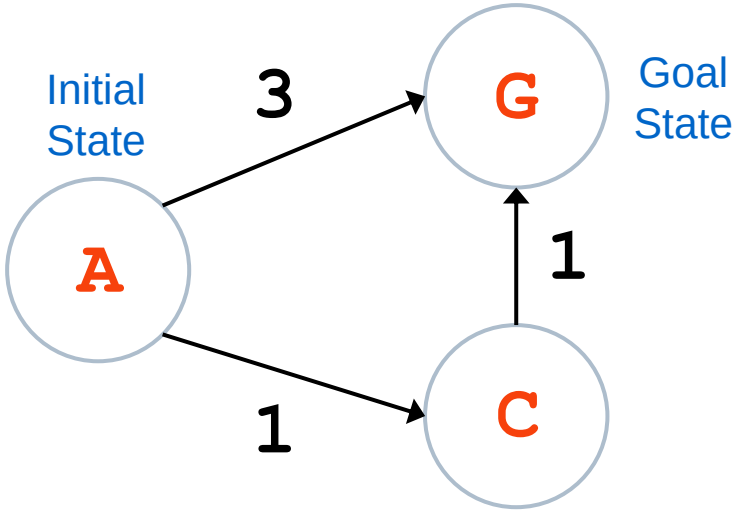
- `h(n)` is consistent, if $\forall n, n' : h(n) \leq cost(n,a,n') + h(n')$
- Theorem: If `h(n)` is consistent, then A* using graph search is optimal

Note: consistency ⇒ admissibility
- Proof – T02 Q3a

Still an Issue!

- Refer to the UCS example earlier in the lecture
  - Assume graph search version 1 with consistent heuristic `h = h*`
  - With this example, we have the trace is as follows

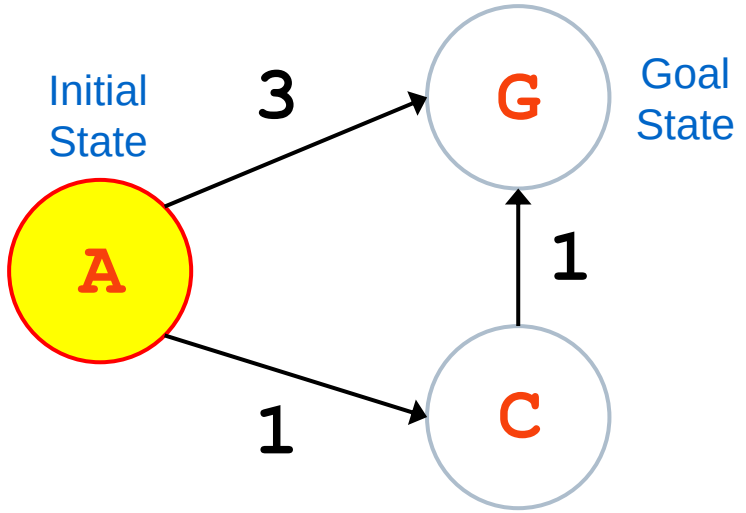
`F`: frontier; `R`: reached



Still an Issue!

- Refer to the UCS example earlier in the lecture
  - Assume graph search version 1 with consistent heuristic `h = h*`
  - With this example, we have the trace is as follows
    - Iteration 1: push `(A(-),0+2)`
      `F = {(A(-),0+2)}; R = {A}`

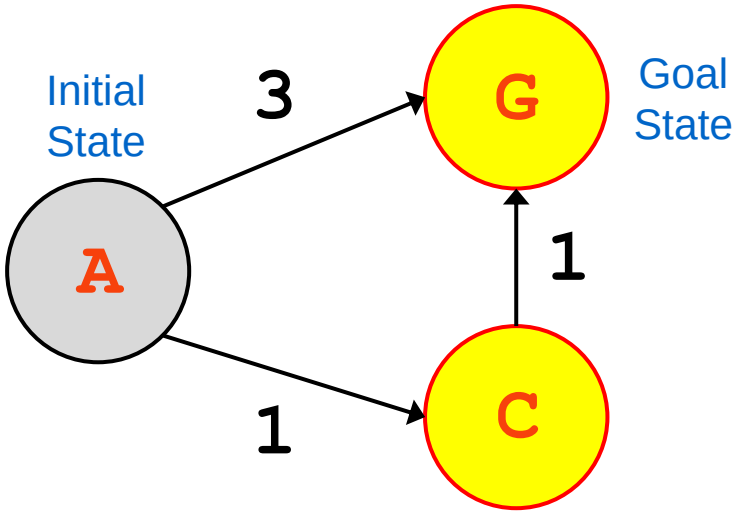
`F`: frontier; `R`: reached



Still an Issue!

- Refer to the UCS example earlier in the lecture
  - Assume graph search version 1 with consistent heuristic `h = h*`
  - With this example, we have the trace is as follows
    - Iteration 1: push `(A(-),0+2)`
      `F = {(A(-),0+2)}; R = {A}`
    - Iteration 2: pop `(A(-),0+2)`, push `(C(A),1+1)` and `(G(A),3+0)`
      `F = {(C(A),1+1), (G(A),3+0)}; R = {A,C,G}`

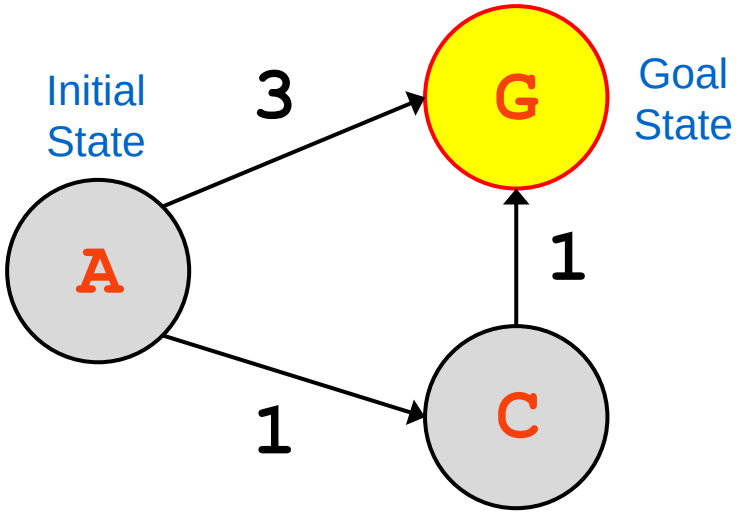
`F`: frontier; `R`: reached



Still an Issue!

- Refer to the UCS example earlier in the lecture
  - Assume graph search version 1 with consistent heuristic `h = h*`
  - With this example, we have the trace is as follows
    - Iteration 1: push `(A(-),0+2)`
      `F = {(A(-),0+2)}; R = {A}`
    - Iteration 2: pop `(A(-),0+2)`, push `(C(A),1+1)` and `(G(A),3+0)`
      `F = {(C(A),1+1), (G(A),3+0)}; R = {A,C,G}`
    - Iteration 3: pop `(C(A),1+1)`, cannot push `(G(A,C),2+0)`
      `F = {(G(A),3+0)}; R = {A,C,G}`

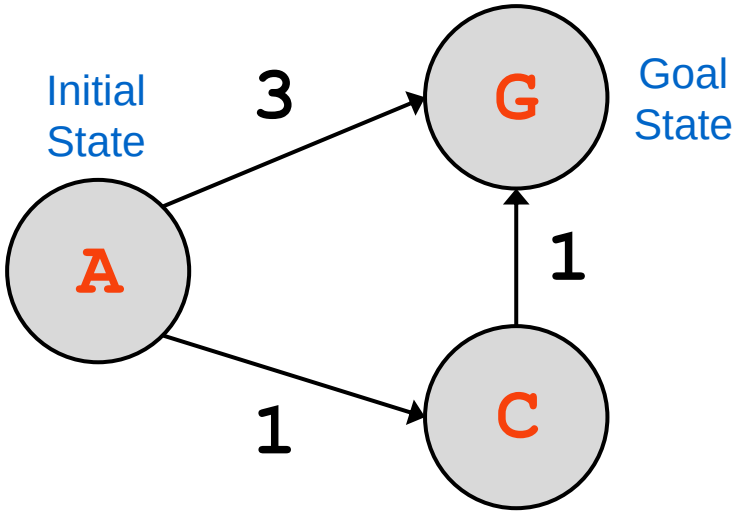
`F`: frontier; `R`: reached



Still an Issue!

- Refer to the UCS example earlier in the lecture
  - Assume graph search version 1 with consistent heuristic `h = h*`
  - With this example, we have the trace is as follows
    - Iteration 1: push `(A(-),0+2)`
      `F = {(A(-),0+2)}; R = {A}`
    - Iteration 2: pop `(A(-),0+2)`, push `(C(A),1+1)` and `(G(A),3+0)`
      `F = {(C(A),1+1), (G(A),3+0)}; R = {A,C,G}`
    - Iteration 3: pop `(C(A),1+1)`, cannot push `(G(A,C),2+0)`
      `F = {(G(A),3+0)}; R = {A,C,G}`
    - Iteration 4: pop `(G(A),3+0)`, return non-optimal path `A,G`

`F`: frontier; `R`: reached



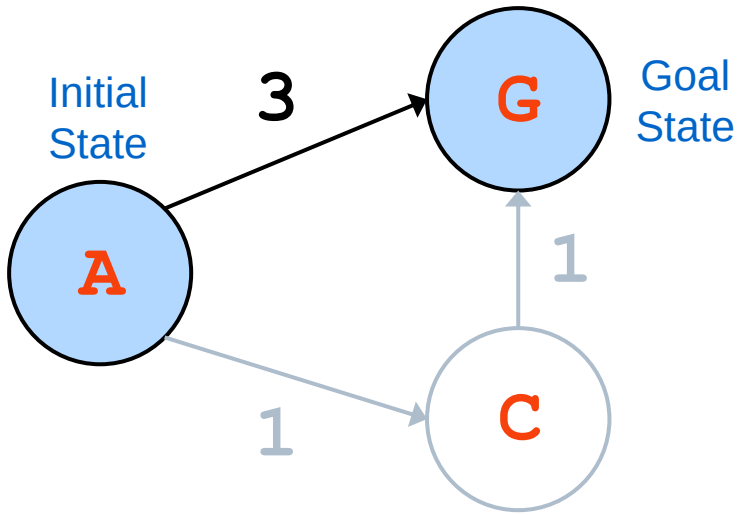
Still an Issue!

- Refer to the UCS example earlier in the lecture
  - Assume graph search version 1 with consistent heuristic `h = h*`
  - With this example, we have the trace is as follows
    - Iteration 1: push `(A(-),0+2)`
      `F = {(A(-),0+2)}; R = {A}`
    - Iteration 2: pop `(A(-),0+2)`, push `(C(A),1+1)` and `(G(A),3+0)`
      `F = {(C(A),1+1), (G(A),3+0)}; R = {A,C,G}`
    - Iteration 3: pop `(C(A),1+1)`, cannot push `(G(A,C),2+0)`
      `F = {(G(A),3+0)}; R = {A,C,G}`
    - Iteration 4: pop `(G(A),3+0)`, return non-optimal path `A,G`

`F`: frontier; `R`: reached

As with the UCS example at the start of the lecture, Graph Search Version 2 elevates this issue.

However, are there any alternatives?



Graph Search Algorithm (Version 3)

```
Function GraphSearchV3(initial_state, actions, T, isGoal, cost):
   frontier = {Node(initial_state, NULL)}
   reached = {}
   while frontier not empty:
         current = frontier.pop()
         reached.insert(current.state: current)
         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)
   return failure
```

- Only adds a node to reached when it is popped
- Proof requires this version (given counterexample for Version 1 on the previous slide)
- Prove this in a similar manner to the UCS proof (contours) – T02 Q2b

6 Dominant Heuristics

Efficiency & Dominance

- Efficiency of A* depends on the accuracy of its heuristics
  - Higher heuristic accuracy means we need to try fewer paths
  - Specifics not covered in CS3243

- Which heuristics are better?

- If , $h_1(n) \geq h_2(n)$, then h_1 dominates h_2
  - If h_1 is also admissible
    - h_1 must be closer to h^* than h_2
    - h_1 is usually more efficient than h_2 when used with A*

Note: with typical interpretations, dominance requires admissibility.

We apply this in CS3243.

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/86826503199