Informed Search: Incorporating Domain Knowledge

CS3243: Introduction to Artificial Intelligence – Lecture 3

25 January 2023 (Video Lecture)

Contents

1. Administrative Matters

2. Reviewing Uninformed Search

3. Greedy Best-First Search

4. A* Search

5. Dominant Heuristics

Administrative Matters

Project 1

- **Released this week (Week 3)**
  - Due Week 6 Sunday, 19 February, 2359 hrs (about a month to work on it)

- **Consultation session**
  - Week 4 Friday, 3 February, Time TBC
  - Online (Zoom) – will be recorded
    - Meeting link = Canvas announcement
  - Reviews FAQ + any new questions raised

- **Late submissions**
  - Penalties
    - Within deadline +24 hours = 80% of score
    - Within deadline +48 hours = 50% of score
    - Beyond deadline +48h hours = 0% of score

Live Coding Session: TBC
(Probably also in Week 4)

Start on Project 1 early!
Start today!

DO NOT COPY/SHARE CODE!

Upcoming…

- **Tutorials begin this week**
  - Only sessions on Monday/Tuesday rescheduled
    - T03 (MON 1400 hrs) → WED 1400 hrs - T03 Zoom Link (Week 3)
    - T04 (MON 1500 hrs) → WED 0800 hrs - T04 Zoom Link (Week 3)
    - T05 (TUE 0900 hrs) → FRI 1400 hrs - T05 Zoom Link (Week 3)

- **Deadlines**
  - TA1 (released last week)
    - *Due in your Week 3 tutorial session*
    - *Submit the a physical copy (more instructions on the Tutorial Worksheet)*
    - *Those with tutorials on Zoom this week → submit via email to tutors*
  - Remember to prepare for the tutorial
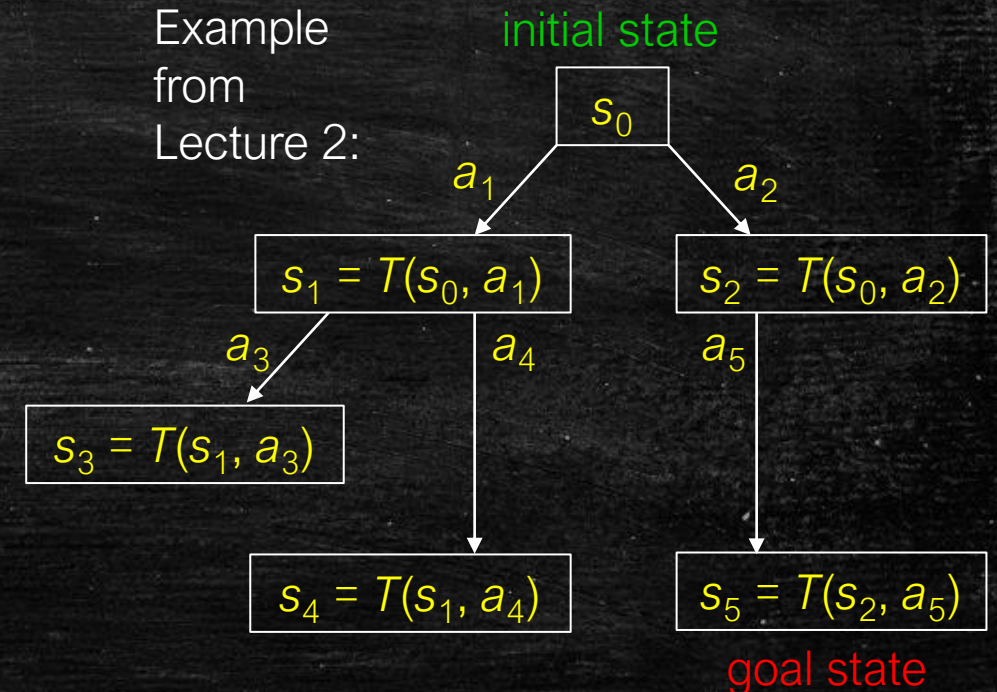   - Participation marks = 5%

Reviewing Uninformed Search

Problem-solving Agent: General Path Planning

- **Assumptions on problem environment**
  - Fully observable
  - Deterministic
  - Discrete
  - Episodic

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

General solution for ALL path planning problems – stronger AI (than reflex agent)

Example from Lecture 2:

initial state

s_0

a_1 a_2

$s_1 = T(s_0, a_1)$ $s_2 = T(s_0, a_2)$

a_3 a_4 a_5

$s_3 = T(s_1, a_3)$

$s_4 = T(s_1, a_4)$ $s_5 = T(s_2, a_5)$

goal state

Tree Search Algorithm

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

- Frontier
  - Paths to search
  - Considers ALL paths (including redundant ones)

- Concept of a Node for defining paths
  - Referenced *end point of path*, state *s*
  - Parent node, for *Linked List reference*
  - Other attributes
    - Action – facilitate *backtracking* in *DFS* – i.e., O(m) space complexity
    - Path cost – for efficient *UCS* (avoids traversing path Linked List)
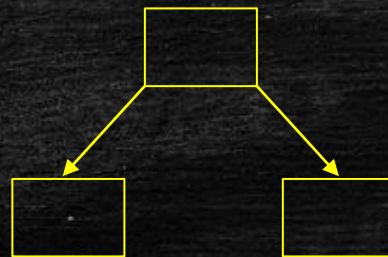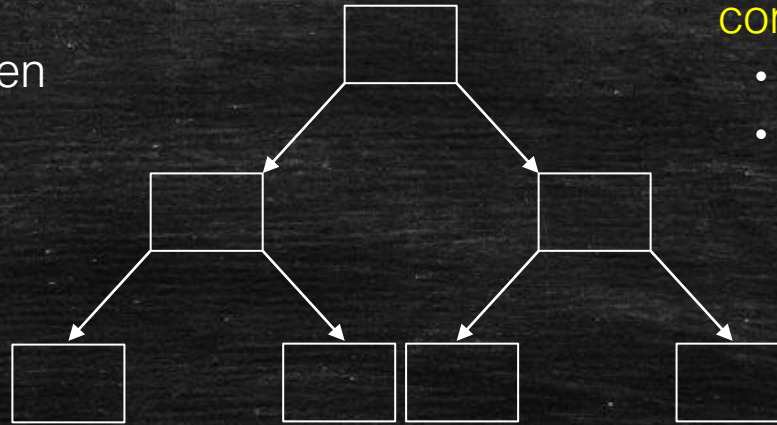   - Depth – for efficient *DLS*/*IDS* (avoids traversing path Linked List)

Uninformed Search Algorithms
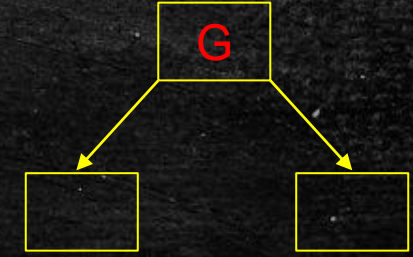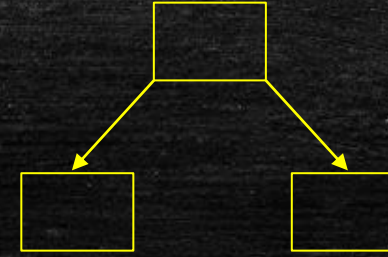
- Systematic traversal of search space
  - BFS: paths in order of number of actions taken
  - DFS: paths to terminal state
  - DLS/IDS: hybrid

- Systematic search allows completeness
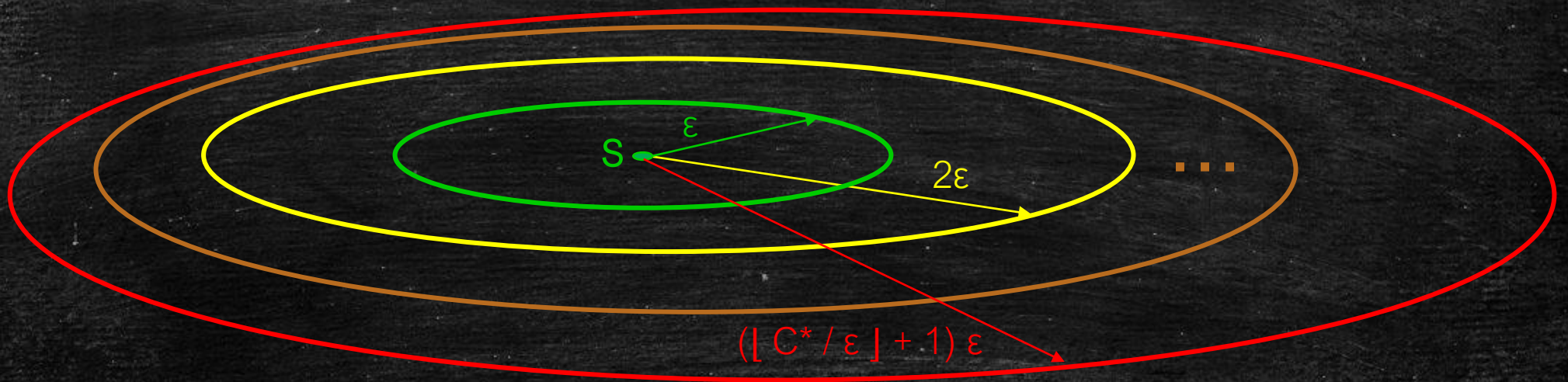 - Assuming search space finite
  - Assuming solution exists

G

UCS & Optimality

- UCS traverses paths in order of path cost

- Ordered traversal of path costs required to ensure optimality

Summary under tree search

- ## Performance of search algorithms

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

1. Complete if b finite and either has a solution or m finite
2. Complete if all actions costs are $> \varepsilon > 0$
3. DFS is incomplete unless the search space 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 not complete (even under 1) as even if a solution exists, it may infinitely traverse a path without a solution (note the "or")
- DFS space complexity may be improved to $O(m)$ with backtracking (similar for DLS and IDS)

Graph Search Algorithm (Version 1)

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

- Reached (sometimes also labelled: Visited)
  - Hash table that tracks all nodes already reached or visited via prior searching

- This version ensures that nodes are never revisited
  - Omits all redundant paths
  - May omit optimal path
  - No impact on completeness

Graph Search Algorithm (Version 2)

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

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

Tree Search Versus Graph Search

- **Tree search will try ALL paths**
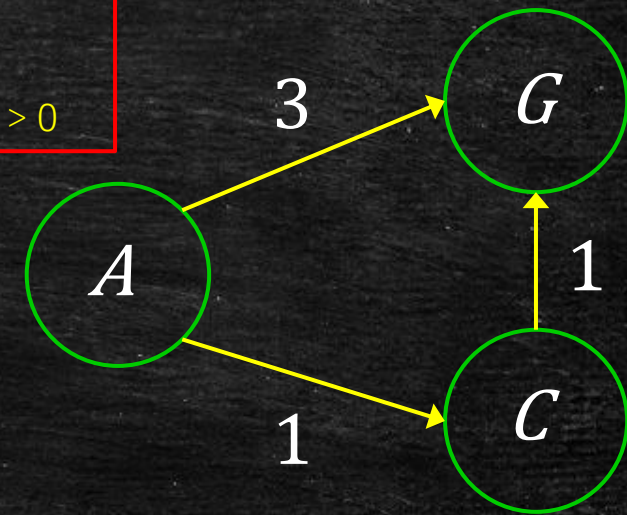 - No paths excluded
  - No issues with optimality

- **What about graph search?**
  - Ensure optimal path not among excluded paths
  - Consider this example
    - F = {A(0)}; R = {A}
      - pop A(0), push C(1) and G(3)
    - F = {C(1), G(3)}; R = {A, C, G}
      - pop C(1), push G(2) since lower cost
    - F = {G(2), G(3)}; R = {A, C, G}
      - pop G(2), path is A → C → G

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

Notice that without the update to G while it was on the frontier, we would not have returned the optimal path (similar to applying Early Goal Test)

A G C

3

1

1

Graph Search Properties

- **Performance under graph search implementations**

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

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

– 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, assume graph search Version 1 is used unless otherwise mentioned

DONE? SOLVED?

Informed Search

Searching Less?

- **Uninformed search algorithms are systematic**
  - Search outward from the initial state
  - All directions

- **Search spaces are LARGE**

- **Can we search more efficiently?**

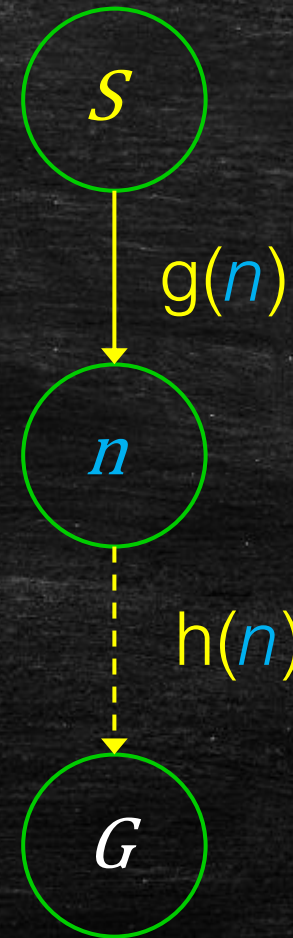General idea

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

try to focus on searching in the direction of the goal
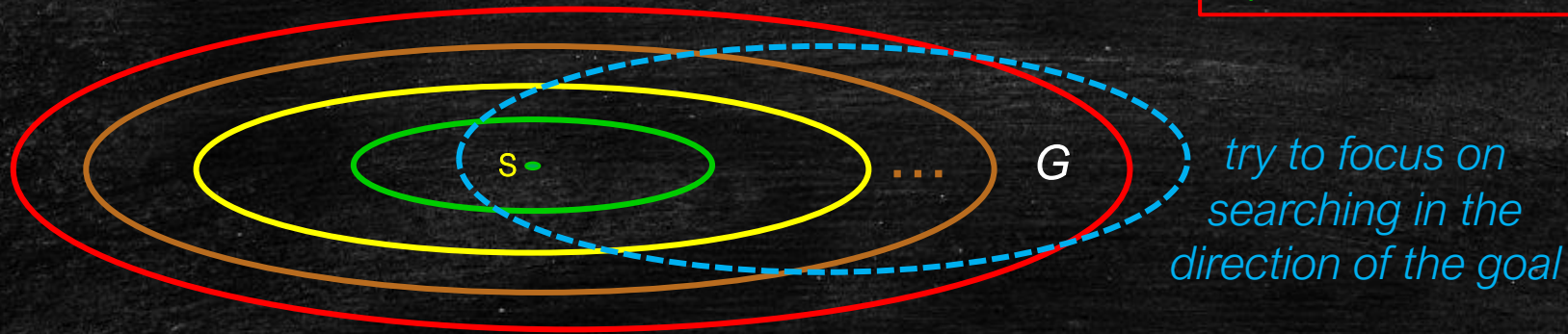
Path Costs & Heuristics

- **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 cost from $n.state$ to G**

- **Define a *heuristic* function h**
  - h approximates the path cost from $n.state$ to its nearest goal G

S

g(n)

n

h(n)

G

Ideas on Deriving Heuristic Functions

- **Consider a Maze Puzzle problem**
  - Layout known
  - Moves ←,↑,→,↓
  - Find path from *S* to *G*
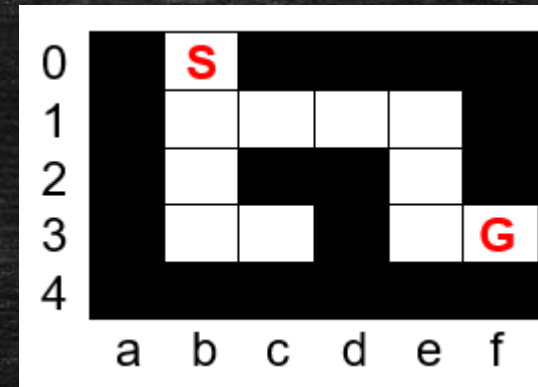
- **Example: Euclidean distance**
  - h(*n*) = Euclidean distance from *n* to *G*

- **General requirements**
  - Efficient – e.g., Euclidean distance is O(*m*), where *m* = no. dimensions
  - More properties discussed later

h(*G*) = 0 requirement

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 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
  - 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)
```
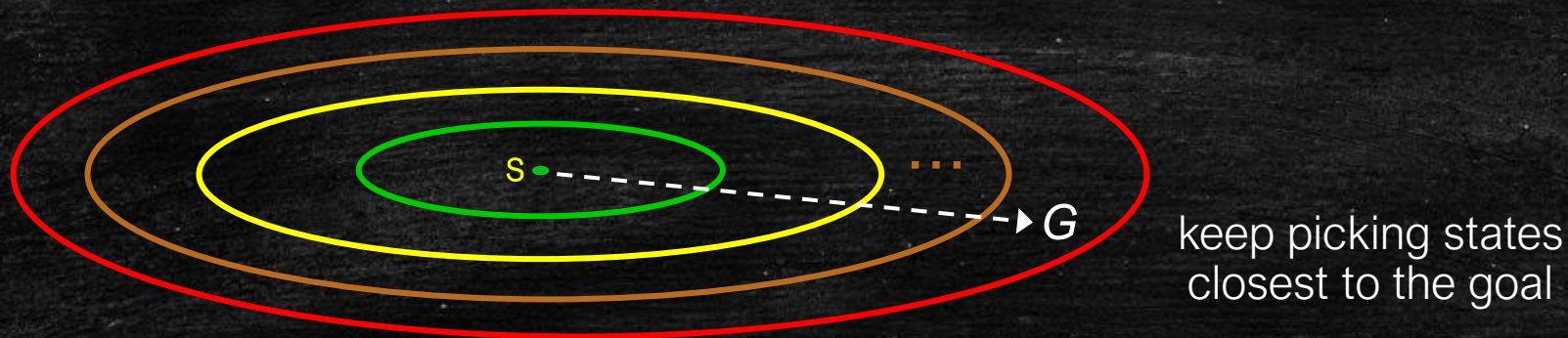
Late Goal Test

Graph search (v2)

Utilises search problem definitions
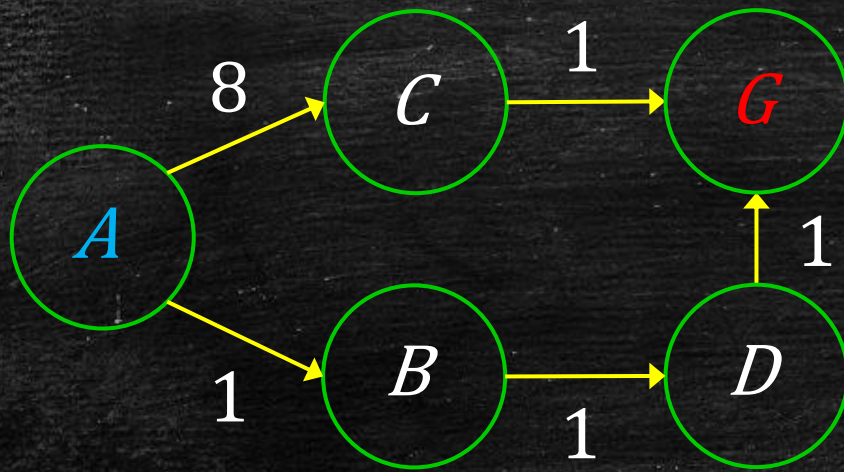
Greedy Best-First Search

The Greedy Best-First Search Algorithm

- Implemented like UCS except
  - $f(n) = h(n)$

- General idea
  - Given all nodes along the frontier
  - Explore next reachable state that you estimate is closest to a goal

keep picking states
closest to the goal

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

Trace:
ITR1 = [A((-),3]
ITR2 = [C((A),1), B((A),2)]
ITR3 = [G((A,C),0), B((A),2)]
ITR4 = DONE (A,C,G)

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

h*(n) = true path cost from n to nearest goal

Algorithm never exploits information on path already travelled.

Completeness and Optimality

- Tree-search implementation is incomplete!
  - General idea
    - Can get stuck in a loop between nodes where h values are lowest
  - Prove with completeness counter example - 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
  - Prove – T02 Q1b

- Not optimal under either tree search or graph search
  - As shown in example on previous slide
  - Find another example – T02 Q1c

Ordered traversal of path in order of path costs required to ensure optimality

Questions about the Lecture?

- Do you need to clarify anything?

- Ask on Canvas > CS3243 > Discussion > Week 3 Content
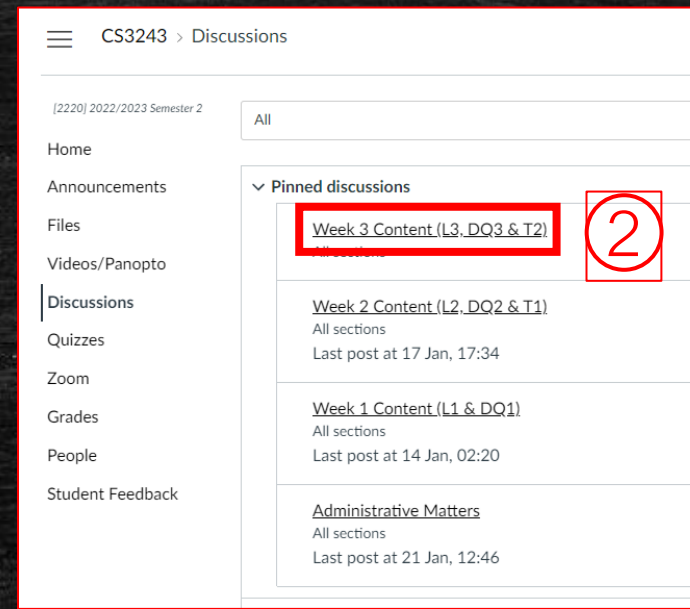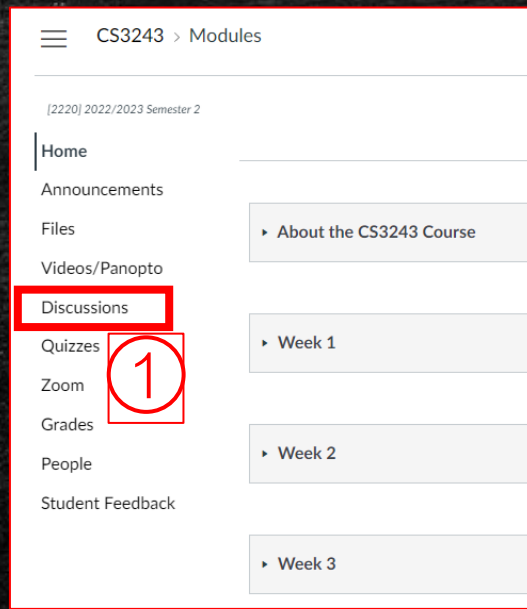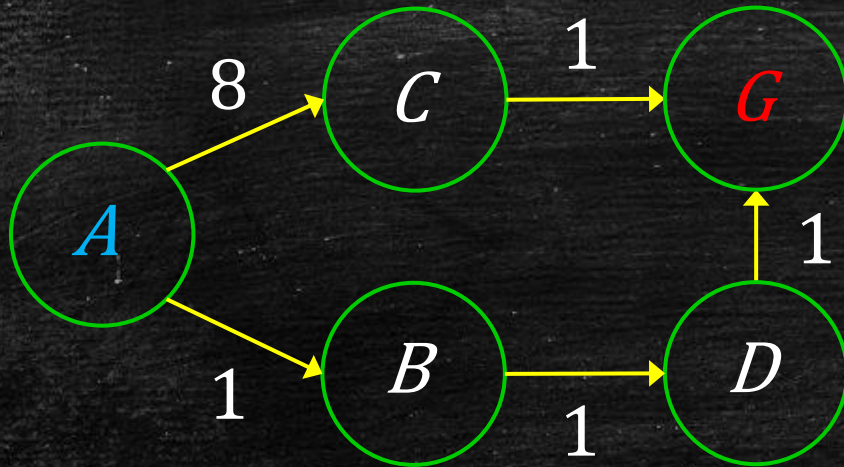
A* Search

The A* Search Algorithm

- **Greedy Best-First Search**
  - With greedy, f(n) = h(n)
  - Does not consider cost of path already taken

- **Accounting for costs already incurred: A***
  - With A*, f(n) = g(n) + h(n)
    - g(n): actual path cost from S to n
    - h(n): estimated cheapest path cost from n to G

- **A* priorities**
  - Total path cost estimates from S to G
  - Gets more accurate as paths get explored
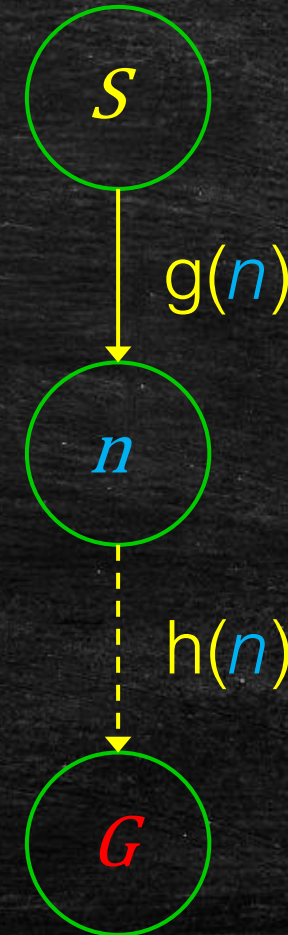
The A* Search Algorithm

- **Example (tree search)**
  same example as used on greedy (Slide 25)

Assume this h:
again, same as before (Slide 25)

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

Trace:
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?

h*(n) = true path cost from n to nearest goal

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

  Ordered traversal of path in order of path costs 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) = g(n) + h(n) $\leq$ g(n) + h*(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

- Examples:
  - Euclidean distance in the maze environment (always underestimates)

- Theorem: If h(n) is admissible, then A* using tree search is optimal

> Main idea, by the time we visit a path to a goal, *P*, all paths with actual costs less than *P* must be searched
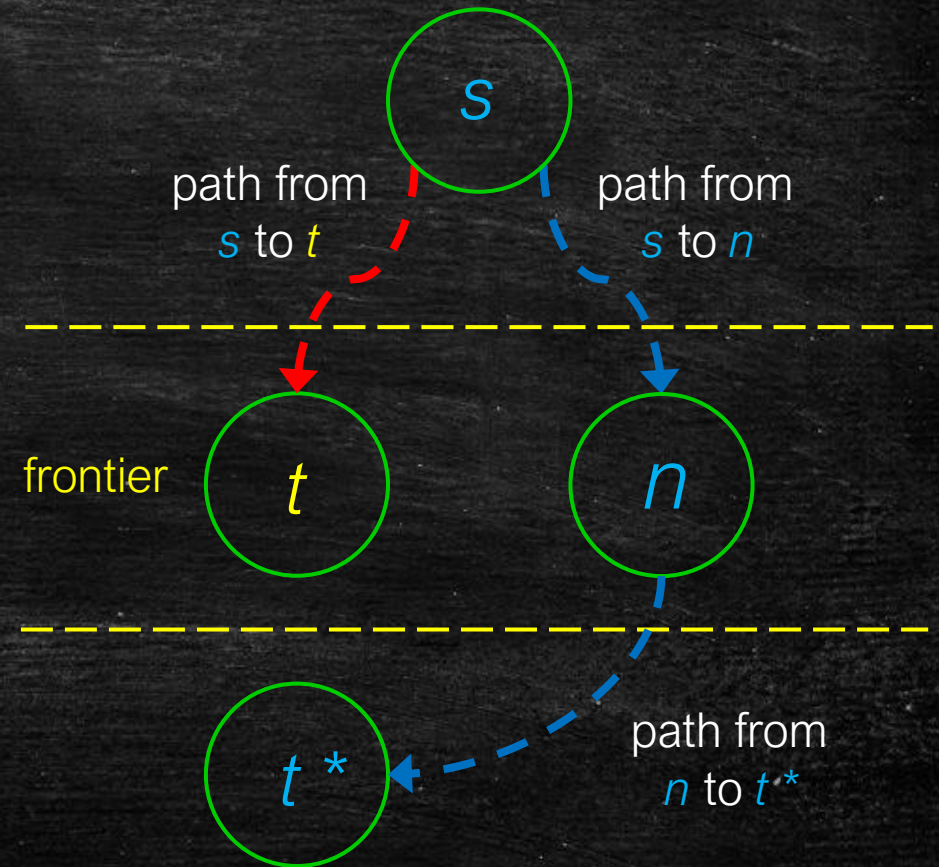
Optimality of A* Using Tree Search

- **Proving the Theorem**
  - T02 Q2a

- **Consider the following**
  - A* is optimal → returns optimal path
    - Let s to n to t^* be the optimal path
  - If not optimal:
    - Must explore path to t first
    - Where t is a goal
  - Not optimal → explore t before n

s

path from
s to t

path from
s to n

frontier

t

n

t^*

path from
n to t^*
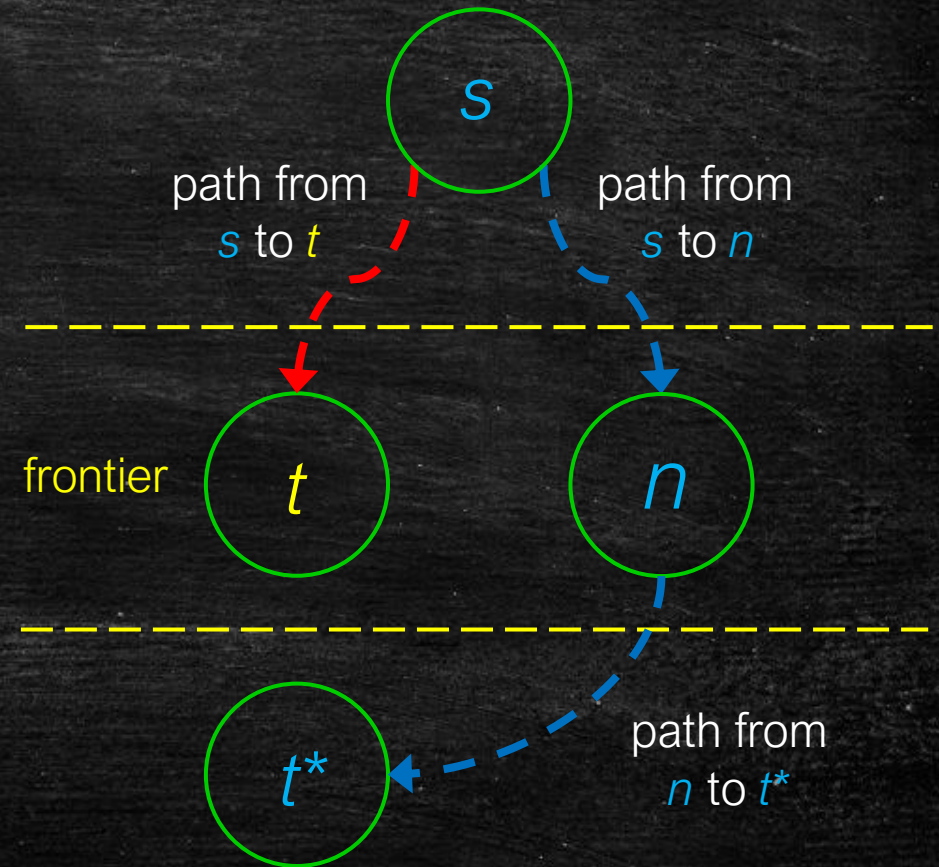
Optimality of Admissible A* Under Tree Search

- **Assume non-optimal t expanded before n (on optimal path)**
  - $f(t) < f(n)$

- **Assuming tree-search**
  - All paths searched
  - All sub-paths* along a single path to a goal must be searched before 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 goal on optimal path
    - $f(n) < f(t^*) < f(t)$
    - *CONTRADICTION*

path from s to t

path from s to n

frontier

t

n

s

t^*

path from n to t^*

* Consider a path, P, from an initial state s to a goal state t, to be $s > n_1 > n_2 > ... > n_k > t$
Let a sub-path to P, P' be any path $s > n_1 > n_2 > ... > n_i$, where $1 \leq i \leq k$

A* Using Graph Search

- **Difference between tree search and graph search**
  - Under admissibility and tree search
    - All nodes leading to a goal 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)

- **Under graph search (version 2), non-redundant paths never skipped**
  - Graph search checks and allows revisits if path is non-redundant
    - As long as a path is cheaper, allow it onto the frontier even if must revisit
  - Still optimal since equivalent to tree search

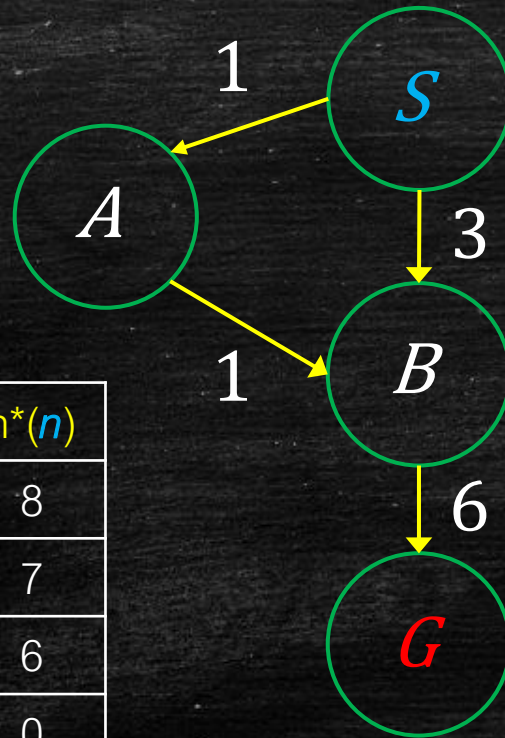- **Under graph search (version 1) may skip optimal path**

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

- Will A* be optimal under graph search Version 1?

- Example

T02 Q3b - construct an alternative example

Assume this admissible h:

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

Trace:
ITR1 = [S((-),0+8)]
ITR2 = [B((S),3+0), A((S),1+7)]
ITR3 = [A((S),1+7), G((S,B),9+0)]
ITR4 = [G((S,B),9+0)] as B already visited, do not revisit
ITR5 = DONE (S,B,G) not the optimal path!

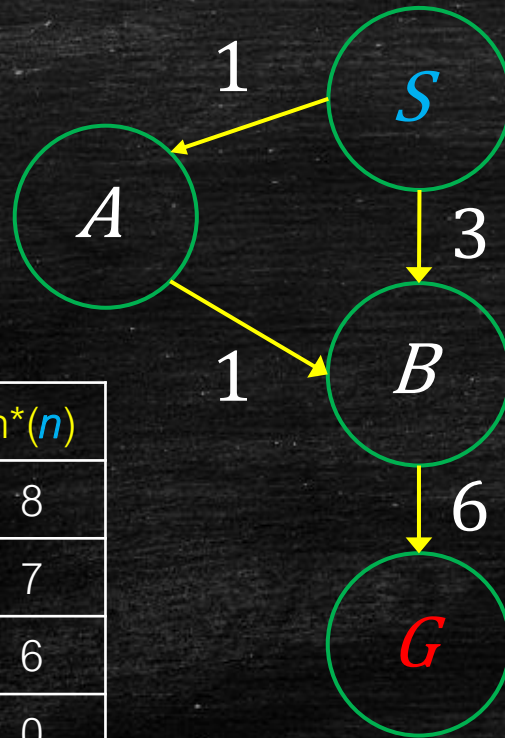We need a tighter constraint on h

Similar to what UCS offers →
contours of search progression
- i.e., stricter ordering of paths

Why Not Optimal?

- Consider the previous example

Assume this admissible h:

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

Observe the sequence of f(n) = g(n)+ h(n) values along each path:

path to n (from S)	g(n)+ h(n)	g(n) + h*(n)	
S	0+8	0+8	
S > A	1+7	1+7	
S > A > B	2+0	2+6	Dip!
S > A > B > G	8+0	8+0	

path to n (from S)	g(n)+ h(n)	g(n) + h*(n)	
S	0+8	0+9	
S > B	3+0	3+6	Dip!
S > B > G	9+0	6	

Want h that ensures paths traversed in order of true path cost!

Consistent Heuristics

- **Forming contours**
  - Under tree search g costs are monotonically increasing
  - For f costs to be monotonically increasing along a path
    - Assume n is a predecessor of n' along a path
    - We need: $g(n) + h(n) \le g(n) + \text{cost}(n, a, n') + h(n')$
    - And thus $h(n) \le \text{cost}(n, a, n') + h(n')$
  - We will use the above requirement

- $h(n)$ is ***consistent*** if $\forall n$, and successor of n, n',
$$h(n) \le \text{cost}(n, a, n') + h(n')$$

- Theorem: If $h(n)$ is consistent, then A* using graph search is optimal

Since true path to n must be less than true path cost to n', we want to ensure h maintains this ordering

Note that:
consistency $\Rightarrow$ admissibility
- Proof – T02 Q3a

Still an Issue!

- **Refer to the UCS example on Slide 15**

- **Assume consistent heuristic h***
  - With this example, we have
    - F = {A(g(A)=0 + h(A)=2)(-)}; R = {A}
      - pop A(0+2)(-), push C(1+1)(A) and G(3+0)(A)
    - F = {C(2)(A), G(3)(A)}; R = {A, C, G}
      - pop C(2)(A), cannot push G(2+0)(A,C) since already in R!
    - F = {G(3)(A)}; R = {A, C, G}
      - pop G(3)(A), path is A → G
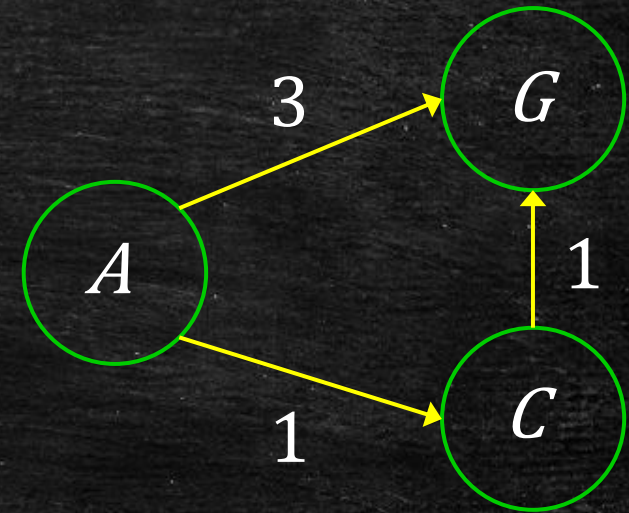
Considering G as reached too early under graph search for A* is similar to applying Early Goal Test to UCS!

Graph Search Algorithm (Version 3)

```
frontier = {Node(initial state)}
while frontier not empty:
      current = frontier.pop()
      reached.insert(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 Slide 39)

- Prove this in a similar manner to the UCS proof (contours) – T02 Q2b

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 $h_1(n) \geq h_2(n)$ for all 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 must be more efficient than h_2

Note: with typical interpretations, dominance requires admissibility. We apply this in CS3243.

Questions about the Lecture?

- Do you need to clarify anything?

- Ask on Canvas > CS3243 > Discussion > Week 3 Content