Informed Search: Incorporating Domain Knowledge

CS3243: Introduction to Artificial Intelligence – Lecture 3

24 January 2022

Contents

Administrative Matters

Project Poll

Grade distribution

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

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

23%

Version 1:
20% on competitive component

23%

199 responses

Grade distribution

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

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

77%

Version 2:
20% on hidden test cases

77%

Project 1

- Released today
  - Individual work (no groups)
  - Python 3.7
  - Graded based on test cases (+ some inspection)
    - Public test cases given in release
    - Private test cases on ***codePost***

- Deadline is 20 February 2022
  - Late penalties
    - Within deadline +24 hours = 80% of score
    - Within deadline +48 hours = 50% of score
    - Beyond deadline +48h hours = 0% of score

CNY Public Holidays

- Affected Classes
  - Monday : T02 (1300), T03 (1400), T04 (1500)
  - Tuesday: T05 (0900), T06 (1000), T07 (1100), T08 (1200)
  - Wednesday : T09 (0800), T10 (0900), T11 (1000), T12 (1100)
- Alternative Zoom Sessions
  - To be arranged by your tutors; announced next week
    - Rahul : T02, T03
    - May : T04, T07, T08
    - Bryan : T05, T06
    - Sagar : T09, T10
    - Jia Wei : T11, T12

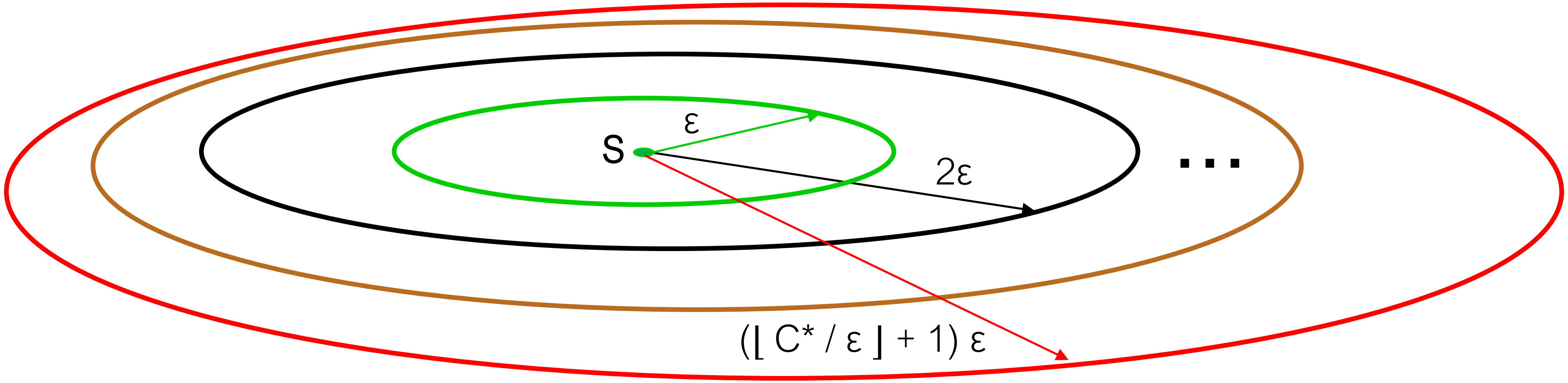
Upcoming…

- Tutorials
  - Begin this week!
    - Today: T02 (1300 hrs), T03 (1400 hrs), T04 (1500 hrs)
    - Tuesday – Friday: T05 – T17
  - Face-to-face in SR-2
    - Show negative FET → uNivUS Green Pass
- Deadlines
  - DQ3 (released today)
    - *Due this Friday (28 January), 2359 hrs*
  - TA2 (released today)
    - *Due this Sunday (30 January), 2359 hrs*
    - *Refer to the tutorial assignment instructions document on LumiNUS*

Reviewing UCS

UCS Optimality

- UCS traverses paths in order of path cost



- When UCS pops a node from the frontier
  - Minimum path cost to that sate
  - Since path costs always increase from initial state

Tree-search Versus Graph-search

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

UCS:

- Frontier = Priority Queue
- Priority of node *n*
    - Path cost of current path taken to *n*, g(*n*)

With a graph-search implementation:

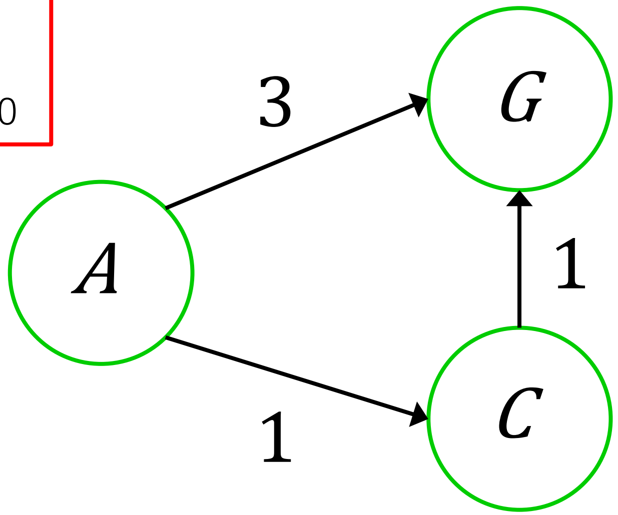
- Maintain a *reached* hash table
- Add nodes corresponding to each state reached (i.e., on push)
- Only add new node to *frontier* (and *reached*) if
    - state represented by node not previously reached
    - path to state already reached is cheaper than one stored

UCS Under Tree-Search & Graph-Search

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

Completeness assumptions:
- b finite, and state space finite or has a solution
- All action costs are > ε > 0

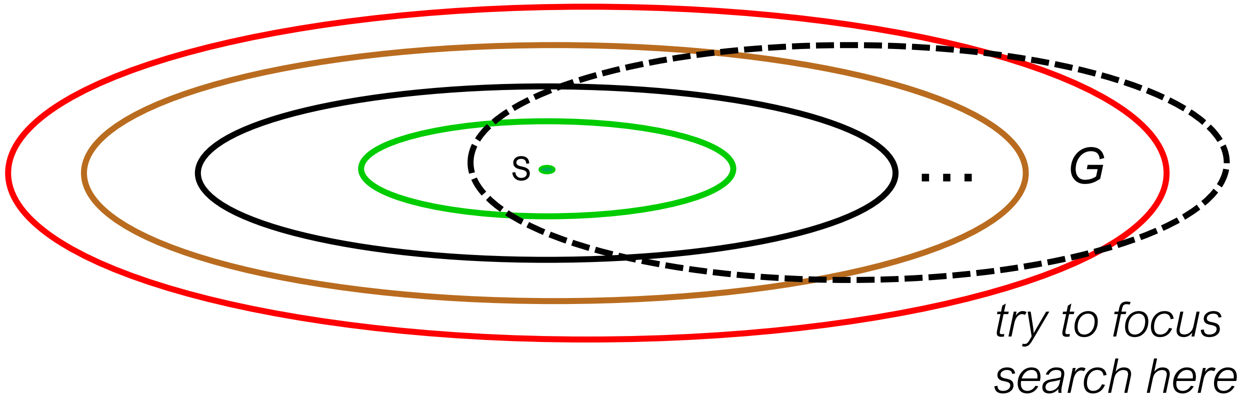
- What about graph-search?
  - Ensure optimal path not among excluded paths
  - Consider this example
    - F = {A(0)}; R = {A}
      - pop A(0), push C(1) and G(3)
    - F = {C(1), G(3)}; R = {A, C, G}
      - pop C(1), push G(2) since lower cost
    - F = {G(2), G(3)}; R = {A, C, G}
      - pop G(2),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



Notice that without the update to G while it was on the frontier, we would not have returned the optimal path

Going in the Right Direction?

- Uninformed search algorithms are systematic
  - Search outward from the initial state
  - All directions
- What can we do to try to move in the right direction?



General idea

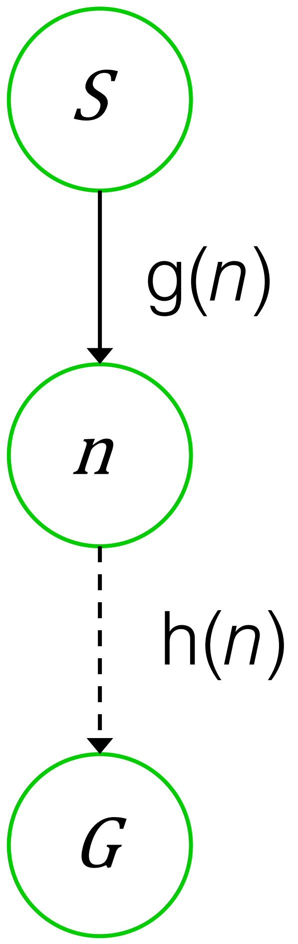
Use domain knowledge about the problem environment to determine the cost required to go from a particular state to its nearest goal

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*

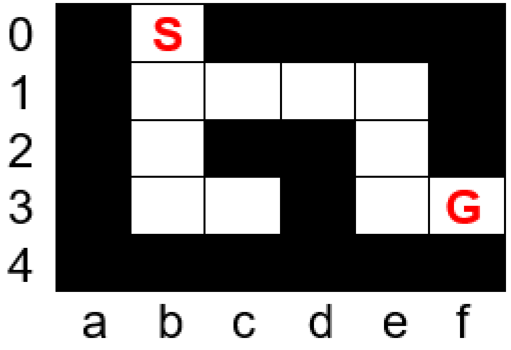
- General idea: use domain knowledge to estimate cost from *n.state* to *G*

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



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***

h(*G*) = 0 requirement

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

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 this 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
```

Late Goal Test

Graph-search

```
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)
```

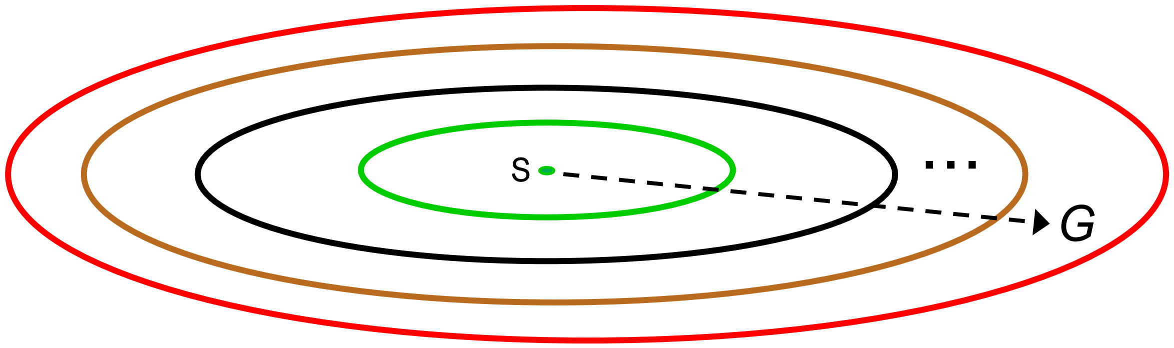
Utilises search problem definitions

Greedy Best-First Search

The Greedy Best-First Search Algorithm

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

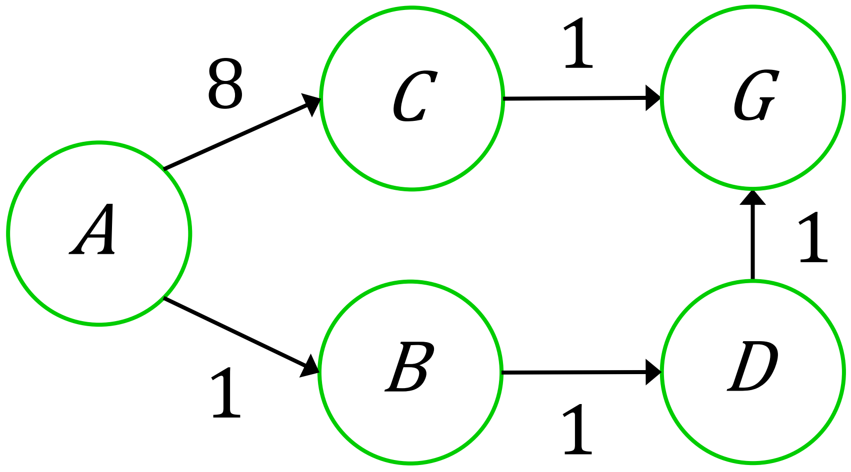
- General idea
  - Given all nodes along the frontier
  - Explore next reachable state that you estimate is closest to a goal



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 & Optimality

- Tree-search version is incomplete
  - General idea
    - Can get stuck in a loop between nodes where h values are lowest
  - Prove with counter example - T02 Q1a
- Graph-search is complete as long as 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 last slide
  - Find another example – T02 Q1c

Questions on the Lecture so far?

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

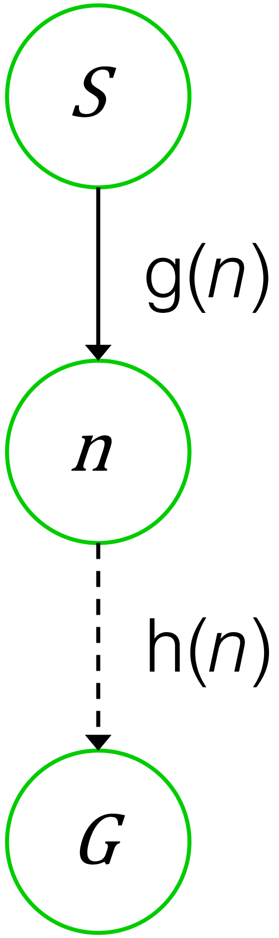
- Channels
  - Verbally on Zoom
  - On Archipelago →
  - Via Zoom Chat

OR https://archipelago.rocks/app/resend-invite/71722702648

A* Search

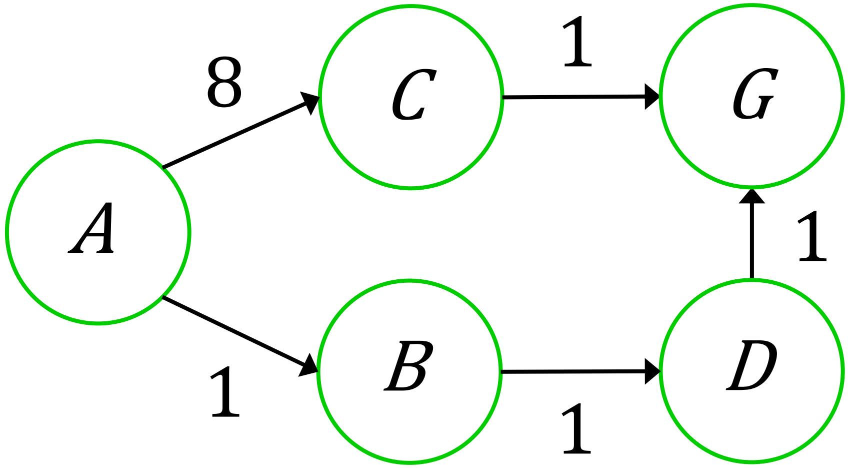
The A* Search Algorithm

- 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 19)



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

Will it always be optimal?
What about graph-search?

Assume this h:
again, same as before (slide 19)

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)

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

Completeness & Optimality

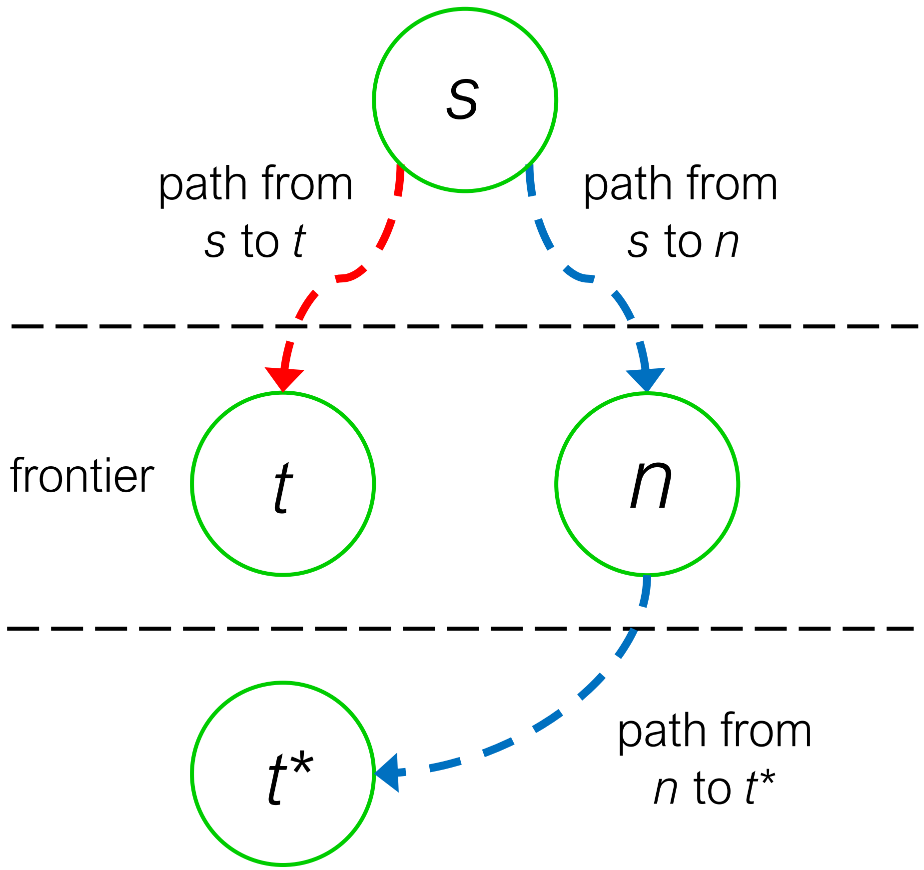
- Completeness
  - Same criteria as UCS
    - b finite, and state space finite or has a solution
    - All action costs > ε > 0
- Optimality
  - Depends on the properties of h

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 under-estimated
        - Evaluation function of value of a non-goal is under-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

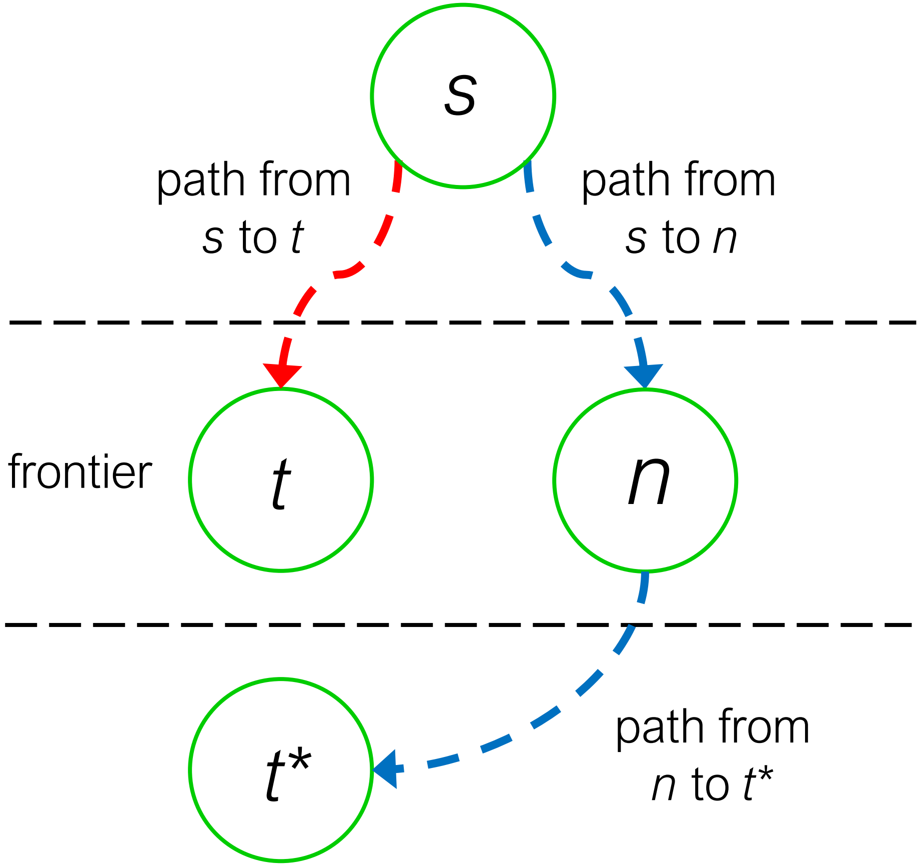
Optimality of A* Using Tree-Search

- 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



Optimality of Admissible A* Under Tree-Search

- Assume t expanded before n
  - $f(t) < f(n)$
- Assuming tree-search
  - All paths searched
  - All sub-paths* along a single path to a goal must be searched before that goal
    - For non-goal m, $f(m) \leq g(m) + h^*(m)$ (since admissible)
    - If goal m^* on path from m, $f(m) \leq f(m^*)$ (since before
  - Since t^* is goal on optimal path
    - $f(n) < f(t^*) < f(t)$
    - *CONTRADICTION*



* Consider a path, P, from an initial state s to a goal state t, to be $s > n_1 > n_2 > \ldots > n_k > t$
Let a sub-path to P, P' be any path $s > n_1 > n_2 > \ldots > n_i$, 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 goal
    - Optimal path will be found
  - Under graph-search we may skip some paths (due to no revisiting)
- Skipping only redundant paths
  - Graph-search checks and allows some revisits
    - As long as a path is cheaper, allow it onto the frontier even if must revisit
  - Still optimal since equivalent to tree-search

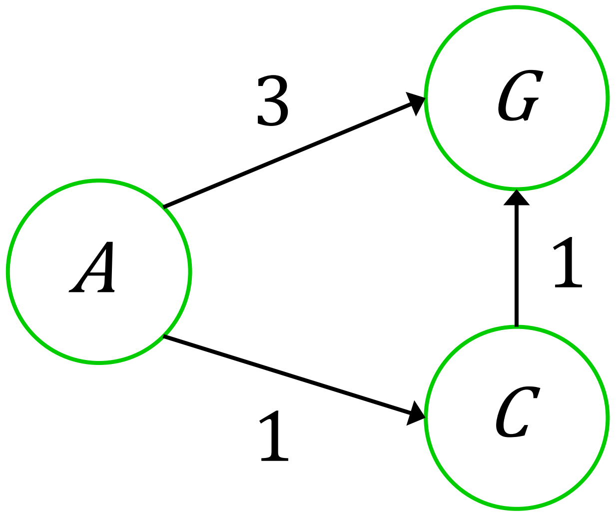
Limited-Graph-Search

- What if we just avoid revisits altogether without any exceptions?
  - i.e., as long as in reached, do not revisit, even if new paths are lower cost
- Limited-Graph-Search (version 1)
  - Just like graph-search, but no exceptions even on lower path costs
    - Uses **reached** hash table
    - Adds to reached on push to frontier
    - Only pushes to frontier when not in reached
      - Excludes **all** redundant paths, but may also exclude **some** non-redundant paths
- Limited-graph-search (version 2)
  - Similar to version 1
    - Except adds to **reached** on pop from frontier
      - Excludes **less** redundant paths than version 1*
      - Excludes **less*** non-redundant paths than version 1*

* Now allows revisits to states on the frontier, but not yet popped from the frontier

Limited-Graph-Search

- Consider limited-graph-search on UCS
- Recall UCS 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)
  - F = {G(2), G(3)}; R = {A, C, G}
    - pop G(2), path is A → C → G



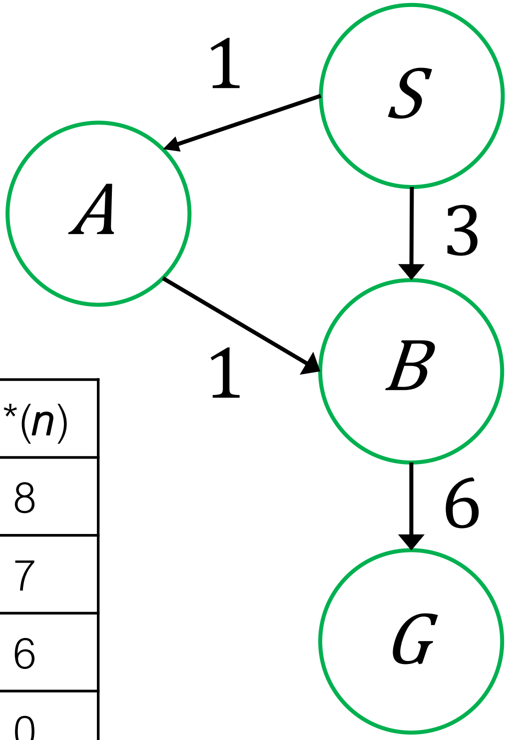
This works only under limited-graph-search version 2, and not version 1, for a similar reason to why an Early Goal Test would cause UCS to not return an optimal solution

From this point, let limited-graph-search imply limited-graph-search version 2. We will not study version 1 any further

Limited-Graph-Search Consistent Heuristics

- Does this mean A* is optimal under limited-graph-search?
- Example

T02 Q3b - construct an alternative example



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 popped before, do not revisit
ITR5 = DONE (S,B,G) <u>not</u> the optimal path!

Assume this admissible h:

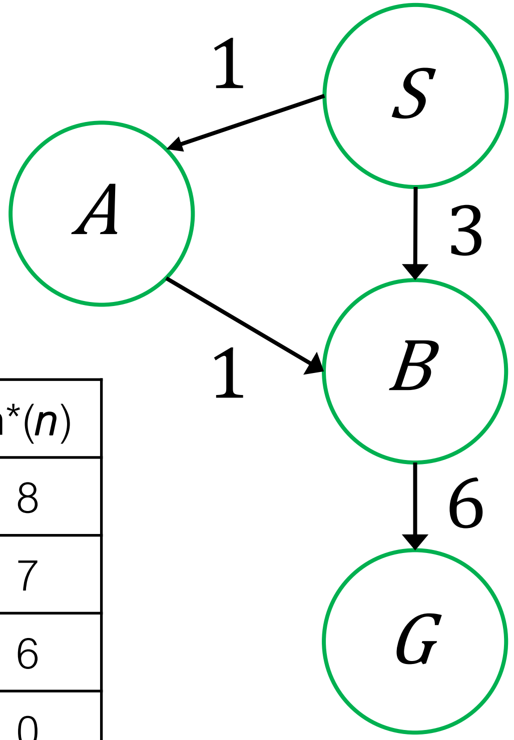
n	h(*n*)	h*(*n*)
S	8	8
A	7	7
B	0	6
G	0	0

We need a tighter constraint on h

Similar to what UCS offers → contours of search progression

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	
$S > A > B$	$\boxed{2+0}$	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	
$S > B$	$\boxed{3+0}$	3+6	Dip!
$S > B > G$	9+0	6	

Consistent Heuristics

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

$$h(n) \leq \text{cost}(n, a, n') + h(n')$$

Note that:
consistency ⇒ admissibility
Proof – T02 Q3a

- Theorem: If $h(n)$ is consistent, then A* using graph-search is optimal
- 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 some interpretations, dominance requires admissibility. We apply a more generic version that does not.

Questions on the Lecture?

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

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

OR https://archipelago.rocks/app/resend-invite/75289652625