Heuristics

CS3243: Introduction to Artificial Intelligence – Lecture 4

30 January 2023

Contents

1. Administrative Matters
2. Reviewing Informed Search
3. Heuristics & Dominance
4. Relaxing the Problem

Reference: AIMA 4th Edition, Section 3.6

Administrative Matters

Project 1

- Consultation Time Survey
  - Current results

Feb 2nd (Thursday) Morning (10:00 - 12:00)	3 respondents	**75 %**	✓
Feb 2nd (Thursday) Afternoon (12:00 - 16:00)		0 %	
Feb 3rd (Friday) Morning (10:00 - 12:00)	1 respondent	25 %	
Feb 3rd (Friday) Afternoon (12:00 - 16:00)	3 respondents	75 %	
I will not be able to attend any session		0 %	
I will review the recording (If made available)	3 respondents	75 %	

Deadline:
Today (30 JAN)
2359 hrs

- Live Coding Session
  - Friday (3 FEB) or Saturday (4 FEB)
  - Invitations will be sent to those with legitimate reasons
  - Email me a request by Thursday (2 FEB), 2359 hrs
    - Succinctly explain why you need to attend the Live Coding Session

Midterm Examination

- Schedule
  - Week 7 Lecture Slot
  - Monday (27 FEB), 1030-1130 hrs
  - Onsite Venue TBC
- Format
  - Duration = 1 hour
  - Closed-book + Cheat Sheet (1 × Double-sided A4 Sheet)
  - Total = 30 marks
- Practice Papers
  - Canvas > CS3244 > Files > Past Papers

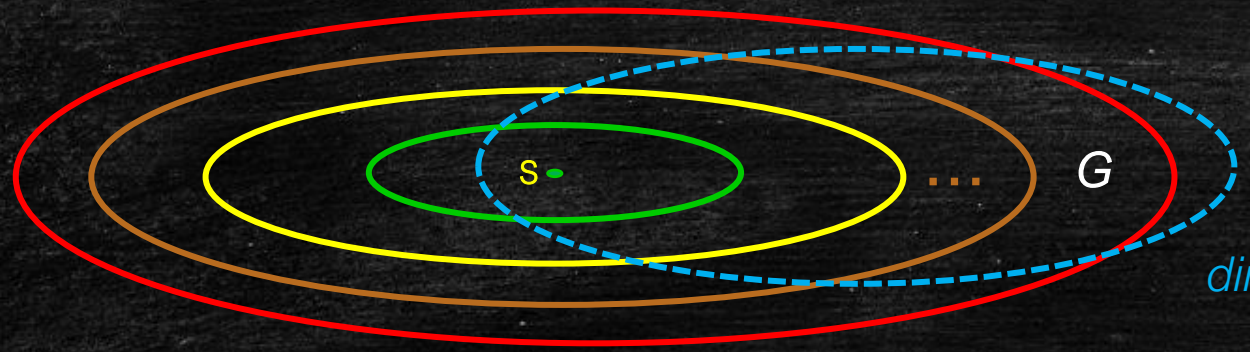
Upcoming…

- Deadlines
  - TA2 (released last week)
    - *Due in your Week 4 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 Informed Search

Informed Search Idea

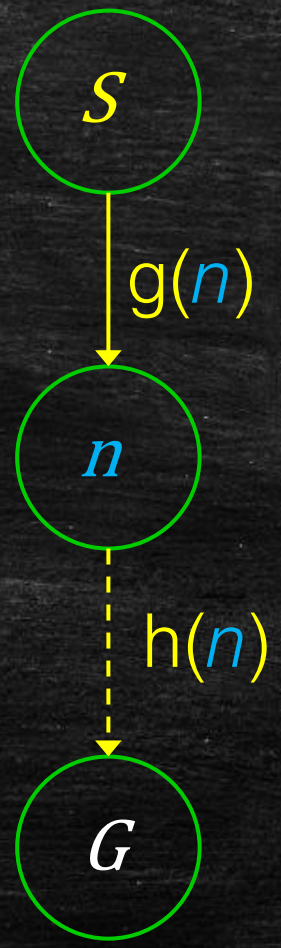
- Uninformed search algorithms systematically search the entire space
  - Search outward from the initial state
  - All directions
- Search spaces tend to be LARGE
- Informed search algorithms instead try to search more relevant paths



try to focus on "searching in the direction of the goal"

Approximate Path Costs to Guide Search

- Frontier = Priority Queue
  - Ordered by evaluation function f(*n*)
    - Priority for path corresponding to ***node*** *n*
  - *Uniform Cost Search*: priority = f(*n*) = g(*n*)
  - *Greedy Best-First Search*: priority = f(*n*) = h(*n*)
  - *A* Search*: priority = f(*n*) = g(*n*) + h(*n*)
- General idea: use domain knowledge to design heuristic function h to estimate cost from *n.state* to *G*

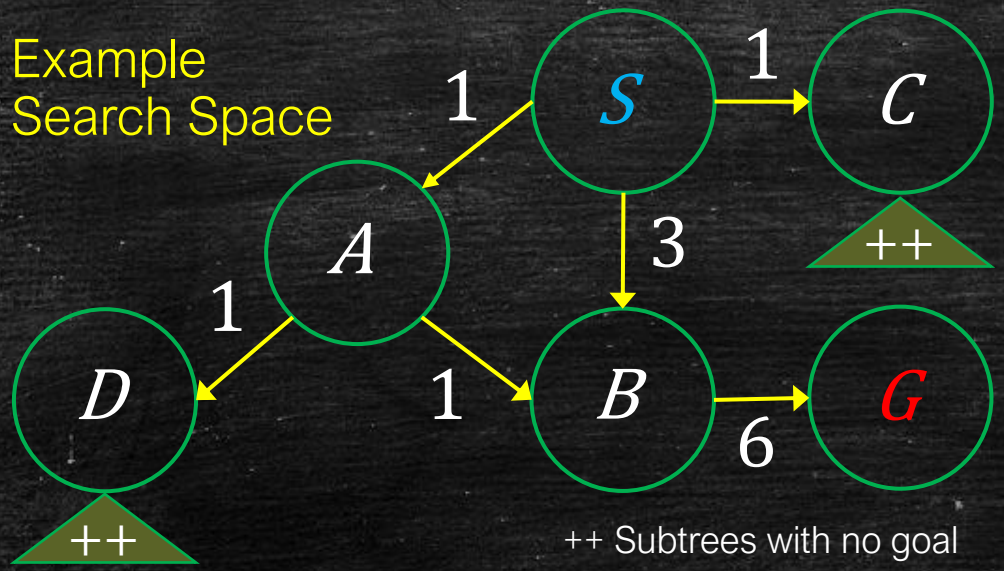


Actual path cost from *S* to *n* based on path taken by node *n*

Approximate path cost from *n* to nearest goal given by heuristic function h

Avoid Paths with Higher Path Costs

- Example Search Space

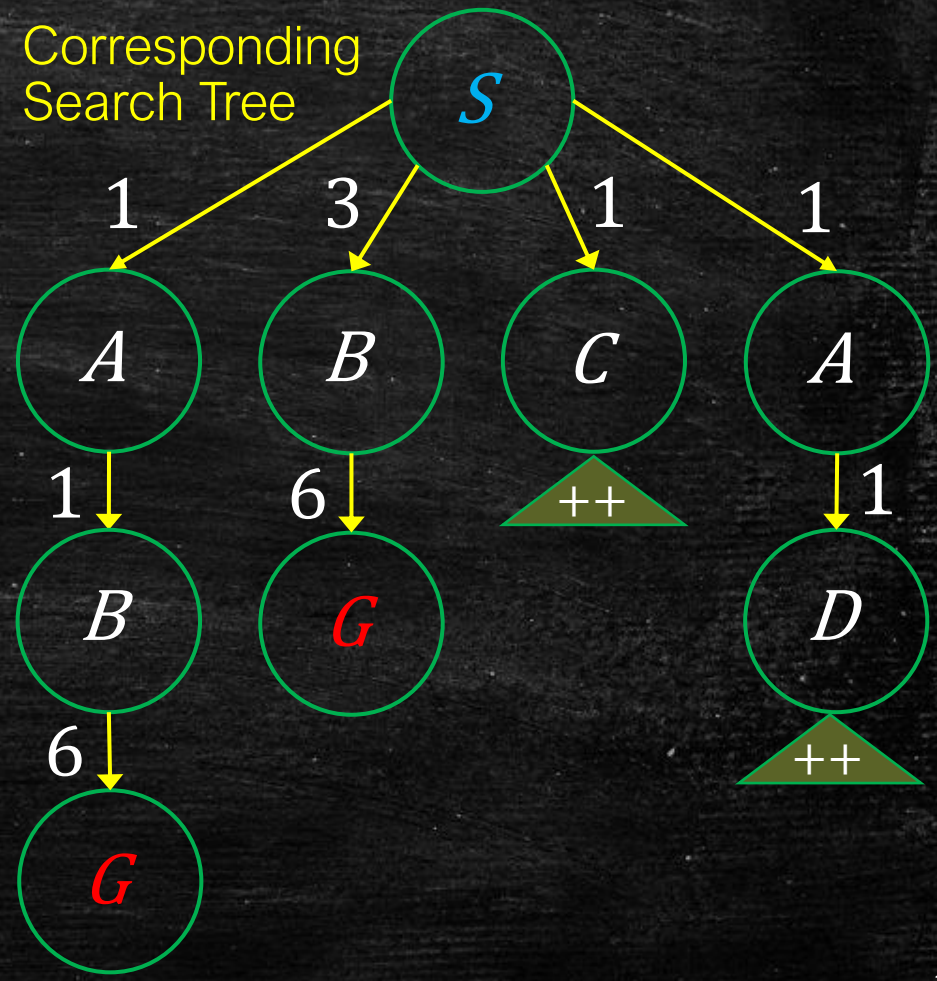


++ Subtrees with no goal

- Paths (and sub-path) lengths
  - *S-A-B-G* = 8 units to goal
  - *S-B-G* = 9 units to goal
  - *S-C++* = ∞ units to goal
  - *S-A-D++* = ∞ units to goal

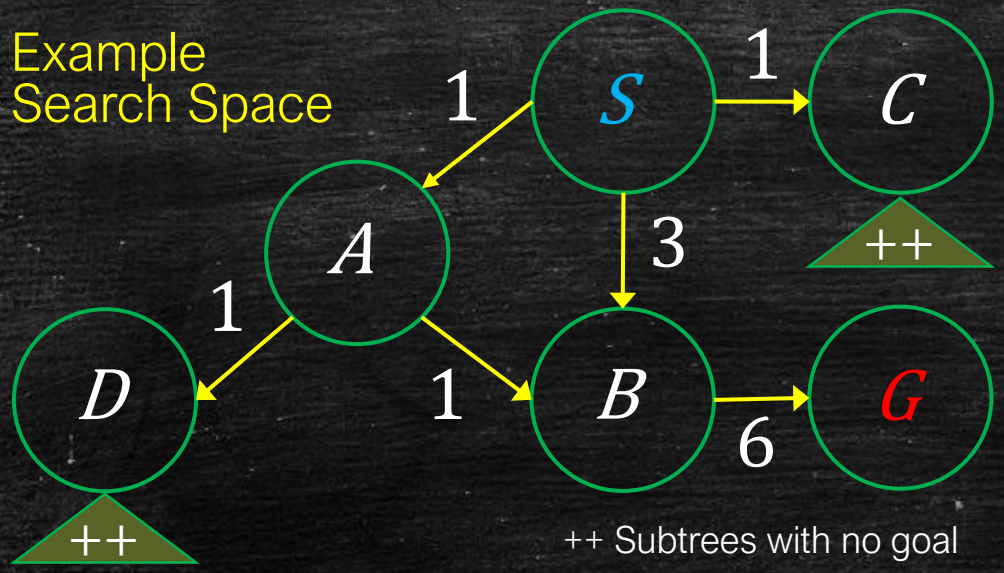
UCS explores ALL (sub)paths with cost ≤ 8

- Corresponding Search Tree



Greedy Best-first Search Review

- Example Search Space



++ Subtrees with no goal

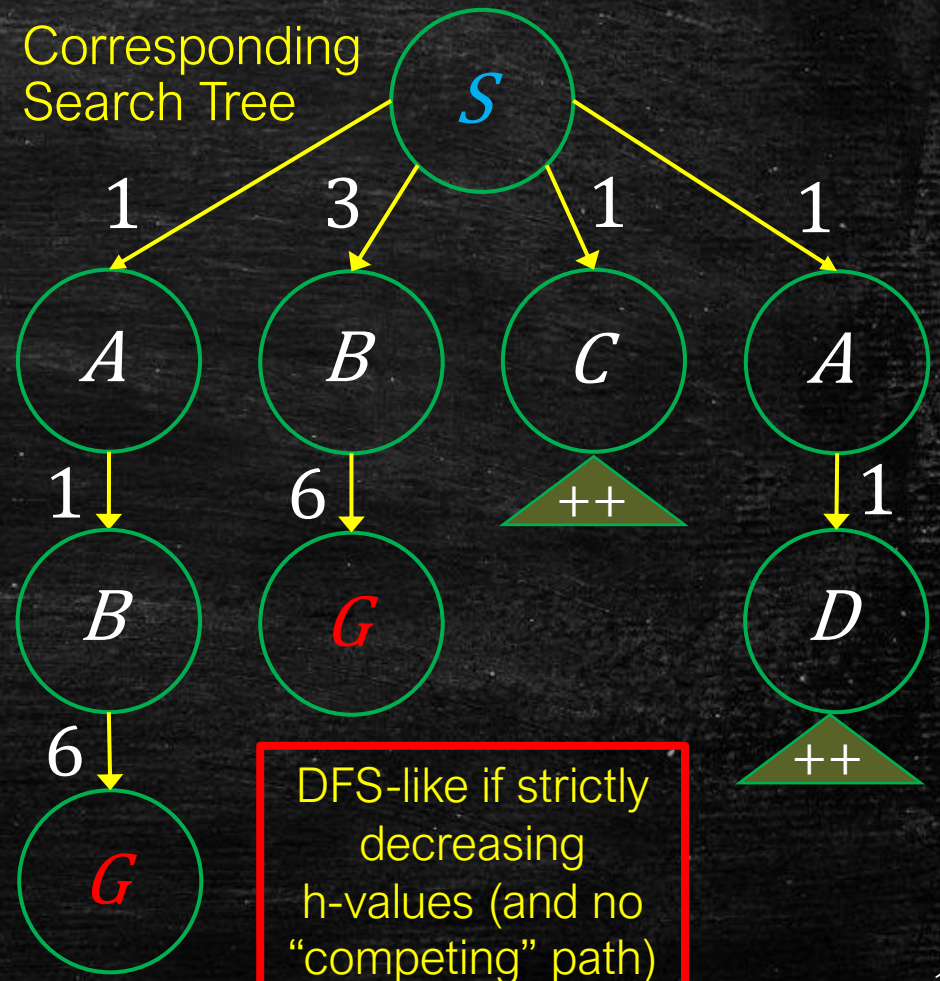
n	h*(*n*)
S	8
A	7
B	6

n	h*(*n*)
C	∞
D	∞
G	0

- Assuming h*
  - returns *S-B-G*
- Considered
  - *S*, *S-A*, *S-B*, *S-C*, *S-B-G*

Potentially efficient

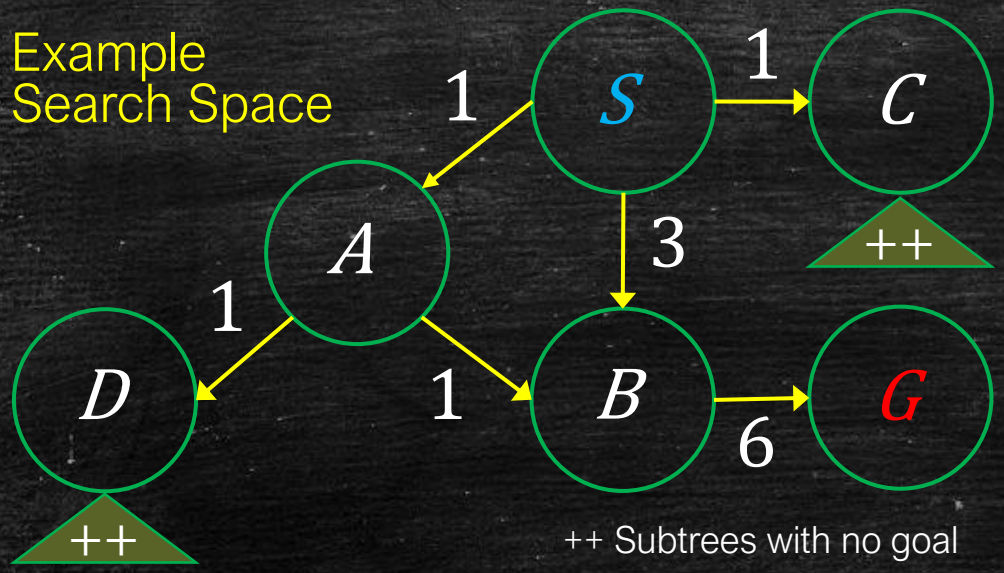
- Corresponding Search Tree



DFS-like if strictly decreasing h-values (and no "competing" path)

A* Search Review

- Example Search Space



++ Subtrees with no goal

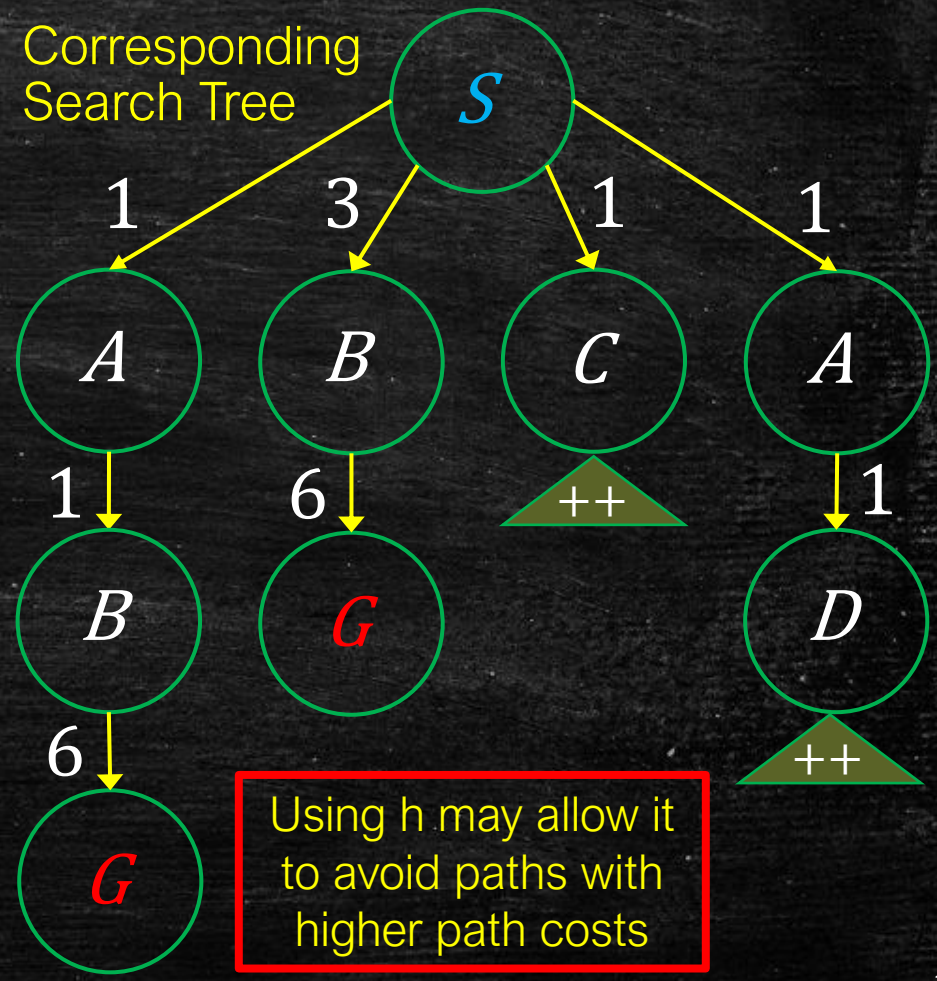
- Assuming h*
  - returns *S-A-B-G*
- Considered
  - *S*, *S-A*, *S-B*, *S-C*, *S-A-B*, *S-A-B-G*

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

n	h*(*n*)
C	∞
D	∞
G	0

More accurate h will result in higher efficiency

- Corresponding Search Tree



Using h may allow it to avoid paths with higher path costs

Admissible & Consistent Heuristics

- h(n) is ***admissible*** if ∀n, h(n) ≤ h*(n)
  - h(n) never overestimates the cost

Main idea, by the time we visit a path to a goal, P, all paths with actual costs less than P must be searched

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

- h(n) is ***consistent*** if ∀n, and successor of n, n', h(n) ≤ cost(n, a, n') + h(n')

Ensure that f costs are monotonically increasing along a path

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

Under Version 2 (but not Version 1)

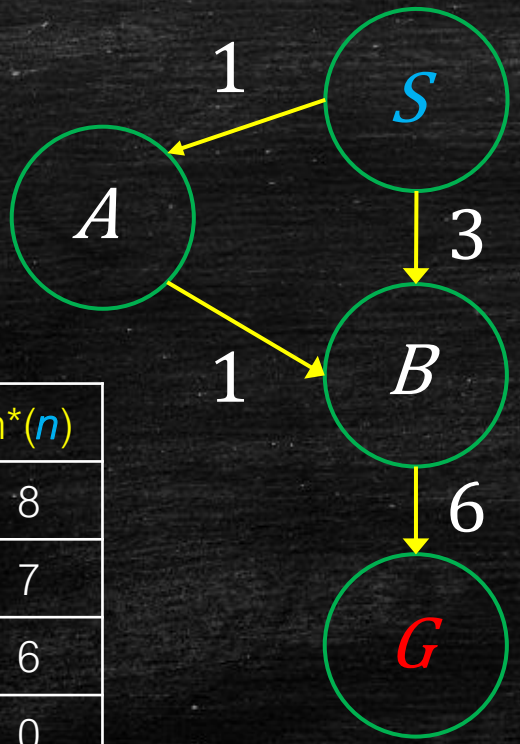
Graph Search Algorithm (Version 3)

```
frontier = {Node(initial state)}
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

Is A* Optimal Under Graph Search Version 3?

- Will A* be optimal under graph search Version 3 using admissible h?
- 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)] note that B added to reached!
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!

Requires consistent h

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 version 2

Utilises search problem definitions

Summary

- Greedy Best-first Search
  - Not optimal and not complete
- UCS
  - On popping node *n*, optimal path to *n* found
  - Optimal under
    - Tree search
    - Graph search (version 2 and 3)
- A*
  - Assuming h admissible
  - Traversal not monotonically increasing with path cost
  - Optimal under
    - Tree search
    - Graph search (version 2)
  - Assuming h consistent
  - On popping node *n*, optimal path to *n* found
  - Optimal under
    - Tree search
    - Graph search (version 2 and 3)

Completeness assumptions:
- b finite AND (m finite OR has a solution)
- All action costs are $> \varepsilon > 0$

Assume graph search version 1 and late goal testing (unless otherwise stated)

Heuristics & Dominance

Efficiency & Dominance

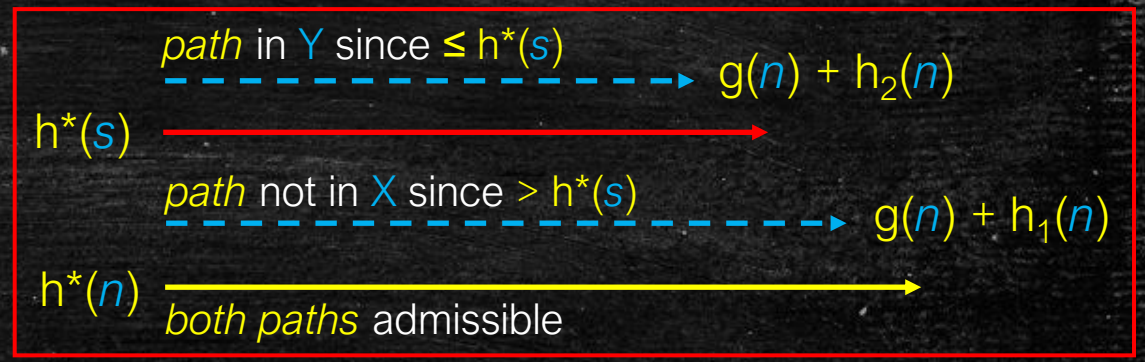
- Efficiency of A* depends on the accuracy of its heuristics
  - Higher heuristic accuracy means we need to try fewer paths

- 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

Recall that in CS3243, we assume a dominant heuristic is an admissible one.

Efficiency & Dominance

- Proving a dominant heuristic improves efficiency?
  - All paths with approximated path cost ≤ optimal path cost are explored
  - A* is more efficient under h_1 since $|X| \leq |Y|$, where
    - X denotes the set of paths with costs below $h^*(s)$ based on h_1
    - Y denotes the set of paths with costs below $h^*(s)$ based on h_2
    - And s is the initial state
  - Consider example path, node n ; assume
    - $h_1(n)$, $h_1(n) < h^*(n)$
    - $g(n) + h_2(n) \leq h^*(s)$
    - $g(n) + h_1(n) > h^*(s)$
  - Some paths in Y have higher approximations under h_1 and may not exist in X



Gauging Improvement with Effective Branching Factor

- How do we measure the improvement when using dominant h?

- Measure effective branching factor, b*
  - Given empirical results
    - N nodes explored
    - Solution path at depth d
  - Solve for b* using
    - $N + 1 = (b^*)^0 + (b^*)^1 + \ldots + (b^*)^d$
    - i.e., $N + 1 = ((b^*)^{d+1} - 1) / ((b^*) - 1)$

How To Craft Heuristics?

- Goal is to identify a function that approximates h*(*n*)
  - Cost from *n* to the nearest goal
- Can we implement another search to give us this?
  - E.g., use UCS as h since it will give us h*(*n*)
    - h(*n*) called on each node → need UCS to find the optimal path from the start state to ALL *n*
    - Defeats the purpose of using h, which is to improve A*
    - Better to just use h(*n*) = 0 and just run UCS once
- We want efficient h
  - Ideally, h is O(1), or something reasonably cheap
  - Set our objective to admissible heuristics (since consistent is much more difficult)

Example Problem: 8-Puzzle

Example Initial State

Goal State

- Puzzle requires the player to shift the numbered squares into the empty cell until the final pattern is obtained
- Search Problem Specification
  - State Representation (Initial State):
    - Matrix representing the grid, with each (r, c) ∈ {0-8}
    - 0 is the blank cell
  - Actions:
    - Move a chosen cell adjacent to the blank, (r, c) into the blank (r', c')
  - Goal Test:
    - Current state matrix = goal state matrix
  - Transition Model:
    - Swap the contents of (r, c) and (r', c')
  - Cost Function:
    - Each action cost 1 unit

How do we get an admissible heuristic out of this puzzle?

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/45015042986

Relaxing the Problem

Heuristic Generation by Relaxing the Problem

Example Initial State

Goal State

- Define an easier problem based on the same context
  - Let h be a function that counts actions required in the easier problem
- The puzzle is constrained by this rule
  - A tile can move from square X to square Y if X is adjacent to Y and Y is blank
- Relaxed 8-Puzzle: Version A
  - Remove all tiles (1 move)
  - Place them in the correct positions (8 moves)
  - h = 9, for any problem
  - h overestimates the moves required on some problems

Test admissibility by trying to work backwards from the goal state

We did not properly relax the rules and instead just defined new ones

Heuristic Generation by Relaxing the Problem

Example Initial State

- Relaxed 8-Puzzle: Version B
  - A tile can move from square X to square Y ~~if X is adjacent to Y and Y is blank~~
  - h = number of cells in the wrong position - O(n), n is the size of the grid
  - h is now admissible!

By properly relaxing the rule, we got an admissible heuristic: misplaced tiles – h_1

Can we do better? Find an admissible h that dominates this one

Goal State

- Relaxed 8-Puzzle: Version C
  - A tile can move from square X to square Y if X is adjacent to Y ~~and Y is blank~~
  - h = sum over each Manhattan distance between a square and its goal location - O(n), where n is the size of the grid
  - h is admissible and dominates the previous version

New admissible heuristic: Manhattan distance – h_2
- h_2 dominates h_1 (h_1 is a relaxation of h_2)
- h_2 is admissible (h_2 is a relaxation of the original rule)

Affects of Dominance Under 8-Puzzle

	Search Cost (nodes generated)			Effective Branching Factor		
d	BFS	A*(h_1)	A*(h_2)	BFS	A*(h_1)	A*(h_2)
6	128	24	19	2.01	1.42	1.34
8	368	48	31	1.91	1.40	1.30
10	1033	116	48	1.85	1.43	1.27
12	2672	279	84	1.80	1.45	1.28
14	6783	678	174	1.77	1.47	1.31
16	17270	1683	364	1.74	1.48	1.32
18	41558	4102	751	1.72	1.49	1.34
20	91493	9905	1318	1.69	1.50	1.34
22	175921	22955	2548	1.66	1.50	1.34
24	290082	53039	5733	1.62	1.50	1.36
26	395355	110372	10080	1.58	1.50	1.35
28	463234	202565	22055	1.53	1.49	1.36

Data are averaged over 100 puzzles for each solution length d from 6 to 28

AIMA Figure 3.26 (pp. 117)

Rules to Functions

- Able to define functions h_1 and h_2 to match the relaxed rules (or even the original rules)?
- Can we always define such functions?

- Models and approximations
  - Finding functions that model or approximate the quantity you want (efficiently)
  - Constructing models
    - Bottom-up
      - What variables can you efficiently calculate?
      - What can these variables model?
    - Top-down
      - What (dependent) variables do I want to model / approximate?
      - What are the (independent) variables that help to calculate these?

Project 1 Heuristics?

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/45015042986