Heuristics

CS3243: Introduction to Artificial Intelligence – Lecture 4

31 January 2022

Contents

Administrative Matters

Tutorial Assignment Submission Deadline Poll

Would it be more helpful for your learning to complete the tutorial sessions before the tutorial assignments are due?

<u>**Keep**</u> the existing tutorial assignment deadline as it is - i.e., due on Sundays at 2359 hrs <u>**before**</u> the week of the tutorial. You prefer to attempt the assignment first and then have the tutor review all tutorial questions, including the assignment questions, during the tutorial.	41%
<u>**Change**</u> the existing tutorial assignment deadline - i.e., move it to Sunday at 2359 hrs <u>**in the same**</u> week of the tutorial. You prefer to have tutors review all tutorial questions apart from the tutorial assignment during tutorials before you attempt and submit the tutorial assignment questions. There will be no review of tutorial assignment questions during your tutorial sessions.	59%

Version 1:
Submit before week of tutorials 41%

133 responses

Version 2:
Submit same week of tutorials $\boxed{59\%}$

CNY Alternate Tutorial Arrangements

- Affected Tutorials
  - Monday (T02, T03, T04)
  - Tuesday (T05, T06, T07, T08)
  - Wednesday (T09, T10, T11, T12)

- Alternative Sessions
  - Refer to announcement on LumiNUS
  - Only attend the session conducted by your tutor
  - If you are unable, confer with your tutor

Upcoming…

- Deadlines
  - DQ4 (released today)
    - *Due this Friday (4 February), 2359 hrs*
  - TA3 (released today)
    - *Due next Sunday (13 February), 2359 hrs*
    - *Refer to the tutorial assignment instructions document on LumiNUS*

Reviewing Uninformed Search

Tree-Search

```
frontier = {initial state}
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
```

- Implications
  - No restrictions on revisiting states
  - Does not try to avoid redundant paths, including cycles

Tree-Search Implementations

- Performance under tree-search

Criterion	BFS	UCS	DFS	DLS	IDS
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]
Optimal Cost?	Yes[3]	Yes	No	No	Yes[3]
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 state space either finite or has a solution
2. Complete if all actions costs are > ε > 0
3. Cost optimal if action costs are all identical (and several other cases)

- Recall that an Early Goal Test on BFS may improve runtime practically
- 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 it might get caught in a cycle
- DFS space complexity may be improved to O(m) with backtracking (similar for DLS and IDS)

Graph-Search

```
frontier = {initial state}
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
```

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

Graph-Search Implementations

- Performance under graph-search

<table>
<tr><th>Criterion</th><th>BFS</th><th>UCS</th><th>DFS</th><th>DLS</th><th>IDS</th></tr>
<tr><td>Complete?</td><td>Yes[1]</td><td>Yes[1,2]</td><td>Yes[1]</td><td>No</td><td>Yes[1]</td></tr>
<tr><td>Optimal Cost?</td><td>Yes[3]</td><td>Yes</td><td>No</td><td>No</td><td>Yes[3]</td></tr>
<tr><td>Time</td><td rowspan="2" colspan="5">$O(|V| + |E|)$</td></tr>
<tr><td>Space</td></tr>
</table>

1. Complete if b finite and state space either finite or has a solution
2. Complete if all actions costs are > ε > 0
3. Cost optimal if action costs are all identical (and several other cases)

- DFS under graph search is complete, assuming a finite state space
- Time and space complexities are now bounded by the size of the state space
  - i.e., the number of vertices and edges, $|V| + |E|$
- Note that we ***do not*** need to allow cheaper paths under graph-search for BFS and DFS since costs play no part in algorithm and they cannot guarantee an optimal solution anyway

Reviewing Informed Search

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

Utilises search problem definitions

Search Variations

- Tree-search
  - No check to avoid revisits
    - Will evaluate redundant paths and can get stuck in cycles
- Graph-search
  - Uses reached hash table to avoid revisits
  - Adds to reached on push to frontier
  - Only pushes to frontier when not in reached or new path is cheaper
    - Will only evaluate non-redundant paths

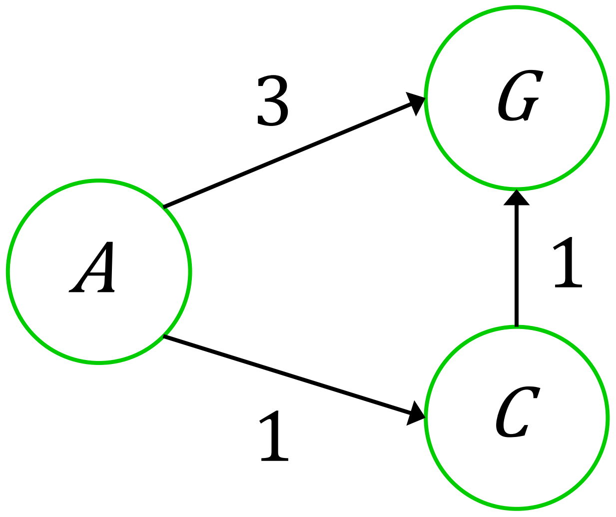
Search Variations

- 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

Summary of UCS & A*

- UCS
  - On popping node *n*, optimal path to *n* found
  - Optimal under
    - Tree-search
    - Graph-search
    - Limited-graph-search

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

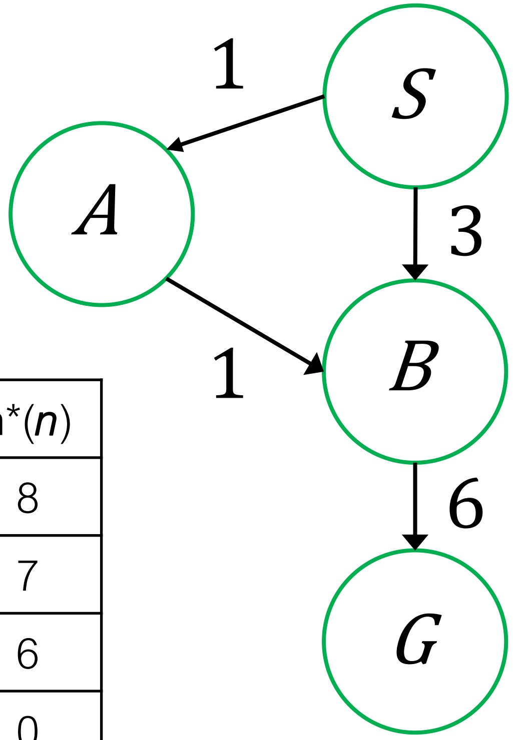
Limited-graph-search refers to version 2
Version 1 not optimal for UCS or A*

- A*
  - Assuming h admissible
  - Traversal not monotonically increasing with path cost
  - Optimal under
    - Tree-search
    - Graph-search

  - Assuming h consistent
  - On popping node *n*, optimal path to *n* found
  - Optimal under
    - Tree-search
    - Graph-search
    - Limited-graph-search

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
$S > A > B > G$	8+0	8+0

Dip!

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
$S > B > G$	9+0	6

Dip!

We need h to be consistent

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/75289652625

Heuristics

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

Note: with some interpretations, dominance requires admissibility. We apply a more generic version that does not.

- How do we define a heuristic?

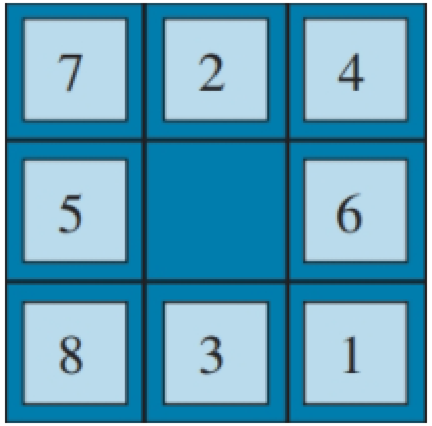
How To Craft Heuristics

- Goal is to identify a function that approximates h*(*n*)
  - Cost from n to the nearest goal

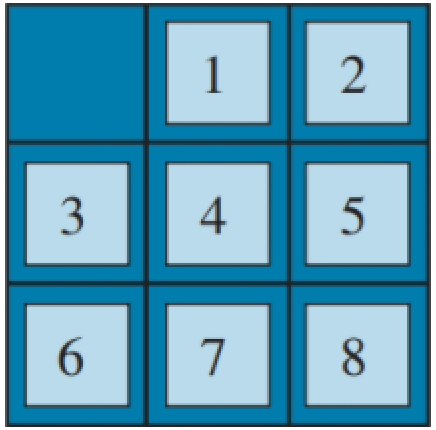
- Can we implement another search to give us this?
  - E.g., use UCS as h since it will give us h*(*n*)
    - Since h(*n*) is call on each node we encounter, we need UCS to find the optimal path from the start state to all n
    - This defeat the purpose of using h, which is to improve A*
    - Better to just use h(*n*) = 0 and just run UCS once

- We want efficient h
  - Ideally, h is O(1), or else something else reasonably cheap
  - Set our objective to admissible heuristics (since consistent is much more difficult)

Example Problem: 8-Puzzle



Example Initial State



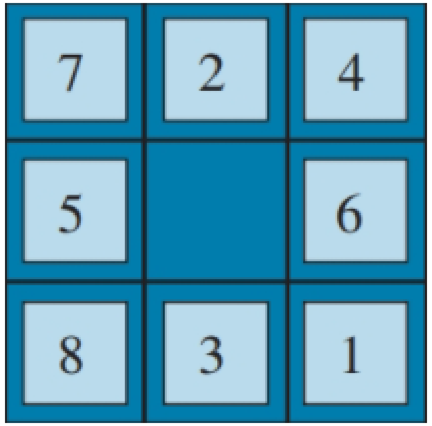
Goal State

- 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

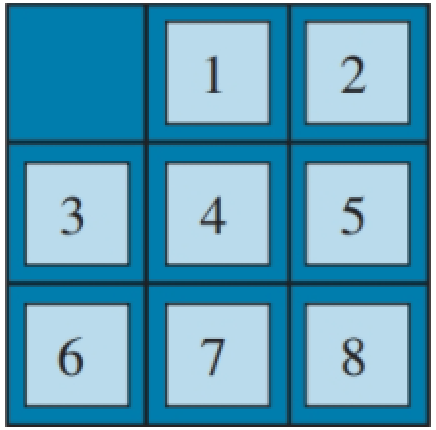
How do we get an admissible heuristic out of this puzzle?

Relaxing the Problem

Heuristic Generation by Relaxing the Problem



Example Initial State



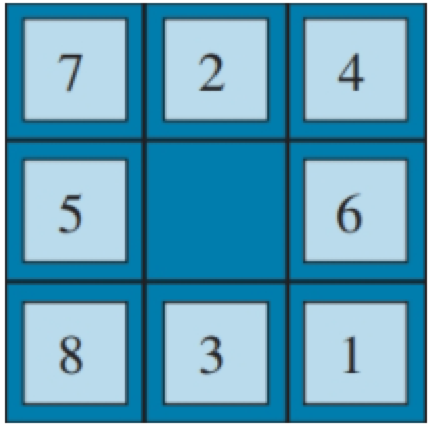
Goal State

- 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

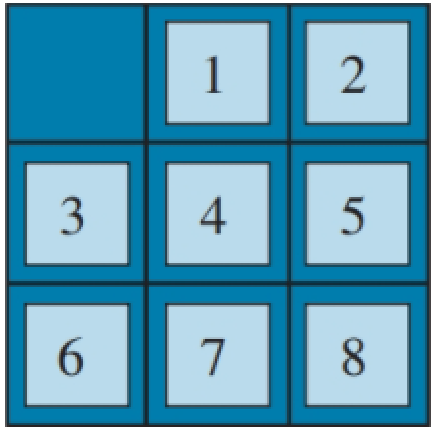
Test admissibility by trying to work backwards from the goal state

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

Heuristic Generation by Relaxing the Problem



Example Initial State



Goal 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

- 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 distances 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)

Refer to AIMA Figure 3.26 (pp. 117)

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?

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