UMass CMPSCI 383 (AI) HW1: Chapters 1-3

Tony Gao

Assigned: Sep 6 2017; Due: Sep 17 2017 @ Midnight

Abstract

Submit a (.zip) file to Moodle containing your latex (.tex) file, rendered pdf, and code. All written HW responses should be done in latex (use sharelatex.com or overleaf.com). All code should be written in Python and should run on the edlab machines ([yourumassid]@elnux[1,2,...].cs.umass.edu).

1 Search Problem Formulation

Consider an ambitious freshman who is interested in pursuing a double major in college. The college has a number of course requirements for completing a double major and this student would like to fulfill the requirements in as few classes as possible.

1.1 Formulation & Search Strategy

Frame this as a search problem. Give the initial state, goal test, available actions, transition model (a.k.a. successor function), and step cost function. A detailed description of these components in words or using pseudocode is acceptable.

Note that your choice of state space affects your set of available actions, which then affects your branching factor (b) and possible solution depth (d). Keep in mind when designing your search problem that many uniformed search algorithms require $\mathcal{O}(b^d)$ space or time complexity, so your formulation can have a significant impact on search. For example, introduce conditions or constraints that can reduce your action space (reduce b). Points will be deducted for naive, wasteful approaches.

Initial State **(1pt)**:
Student(Freshman, Double Major, n, 0, 0, 0): Student is initially a Freshman with a Double Major declared which has a set n remaining requirements and 0 credits. Student has taken 0 classes total. Student is on the 0^{th} semester

In general, Student(a, Double Major, n, m, o, p): Student is an a with a Double Major. Student has set n remaining requirements. Student has m credits. Student has taken o classes. Student is on the p^{th} semester

Goal Test **(1pt)**:
Test(State) : State is Student(Graduated, Double Major, $\emptyset$, m $\geq$ 128, o, p)

Available Actions **(6pts)**:
Enroll(Class) : Student Enrolls in Class for the current semester, adds to list of current classes. Student cannot enroll in more than 19 credits per semester. Student must enroll in at least 16 credits per semester. Student must meet prerequisites of class being enrolled in. Student cannot enroll in same class more than once. Classes fulfill 1 to 4 credits.
EndSemester(): End of Semester, student passes all classes. Semester can only be ended if class credit constraint is met.

The set of enroll-able classes is a subset of all available classes, specifically limited to those which fulfill at least 1 requirement with prerequisites fulfilled. If a class only fulfills 1 requirement, and that requirement is already met, it is not included in the set of enroll-able classes.

I assume the requirements are some arbitrary set, unique per double major.

Transition Model **(1pt)**:
Result(Student(a, Double Major, n, m, o, p), Enroll(Class)) = Student(a, Double Major, n \ Requirements(Class), m + Credits(Class), o + 1, p)
Result(Student(a, Double Major, n, m, o, p), EndSemester()) = Student(a', Double Major, n, m, o, p + 1)

def Requirements(Class): returns set of requirements fulfilled by given class

def Credits(Class): returns number of credits fulfilled by given class

if (m + Credits(Class) between 0 and 31) a' = Freshman
if (m + Credits(Class) between 32 and 63) a' = Sophomore
if (m + Credits(Class) between 64 and 95) a' = Junior
if (m + Credits(Class) between 96 and 127) a' = Senior
if (m + Credits(Class) $\geq$ 128) a' = Graduated

Example from initial state:
Result(Student(Freshman, Double Major, n, 0, 0, 0), Complete(Class)) = Student(Freshman, Double Major, n \ Requirements(Class), 0 + Credits(Class), 1, 0)

Example Freshman to Sophomore transition: Result(Student(Freshman, Double Major, n, 32, o, 1), EndSemester()) = Student(Sophomore, Double Major, n, 32, o, 2)

Step Cost **(1pt)**:
Cost(s, Enroll(Class), s') = 1
Cost(s, EndSemester(), s') = 0

1.2 Repeated States

Will you need to consider repeated states when performing search? In other words, are there multiple ways to arrive at the same state? Explain. **(2pts)**

Yes, there are multiple ways to arrive at the same state. For example, a student could either take 4 classes of 2 credits each which fulfill 1 requirement each for 8 credits and 4 requirements, or take 2 classes of 4 credits each which fulfill 2 requirements each for 8 credits and 4 requirements. Assuming the same start state, this would be two different paths to the same state.

1.3 Complexity

As a toy example, consider a 4-year college that

- offers 10 distinct classes,
- requires a minimum of 8 classes for a double major,
- and requires students take exactly 1 class per semester.

Given your formulation,

- What is the maximum branching factor? Explain. **(2pts)** The maximum branching factor is 10. There is no information given besides the existence of 10 classes so I assume the 10 classes are specifically for the double major. From the initial state of a student haven taken 0 classes, a student can choose to take 1 of 10 distinct classes, giving a branching factor of 10 from the initial state.
- What is the depth of the shallowest goal node? Explain. **(2pts)** The depth of the shallowest goal node is 8. Since exactly 1 class must be taken per step, and 8 classes must be taken, 8 steps must be taken to reach the goal of 8 classes giving a depth of 8

Consider the search strategies in the Big-O comparison table in Figure 3.21 of the book. Based on your branching factor and depth, is time or memory more of an issue in choosing a search strategy? **(5pts)** Which search algorithm would you use? **(5pts)** Which would you not use? **(4pts)**

Memory is more an issue given a branching factor of 10 and depth of 8, which would use $\mathcal{O}(10^8)$ space with BFS.

I would use the Graph-Search version of Depth First Search for the given example. Any class that is taken always results in a state that is on the optimal path. Since the goal state only requires 8 classes, any path that contains 8 classes is an optimal path so the completeness and optimality of DFS is a non-issue.

I would not use Breadth First Search due to the space issue. It is also redundant to expand all possible steps at a certain depth when it is known that any 8 steps results in reaching the goal state through the optimal path. I would not use Iterative Deepening or Depth Limited Search because both would be redundant compared to a plain DFS given the goal state

as portrayed is guaranteed to exist at depth 8. I would not use Bidirectional Search, because it is nonsensical to "untake" a class starting from the goal. I would not use Uniform Cost Search, because the cost of reaching the goal node through the optimal path is always going to be 8, and so the UCS would incur unnecessary time overhead exploring paths that are cost 1 less than the current path.

2 Search on the 8-puzzle

In the tasks below, you will demonstrate various search algorithms on the 8-puzzle domain and then implement A* search in python. To simplify the task, we are providing you with python source code that implements an 8-puzzle simulator as well as Node class. ***Assume nodes are always expanded to generate children in the order: 'Up', 'Down', 'Left', 'Right'.

2.1 Demonstrations

Draw the search tree to a depth of 2 for the 8-puzzle given the start state and `8Puzzle-SearchTree-Template` provided at HWs Public/HW1; make a copy and move the copy to your own Google Drive to edit it **(5pts)**.

See Figure 1.

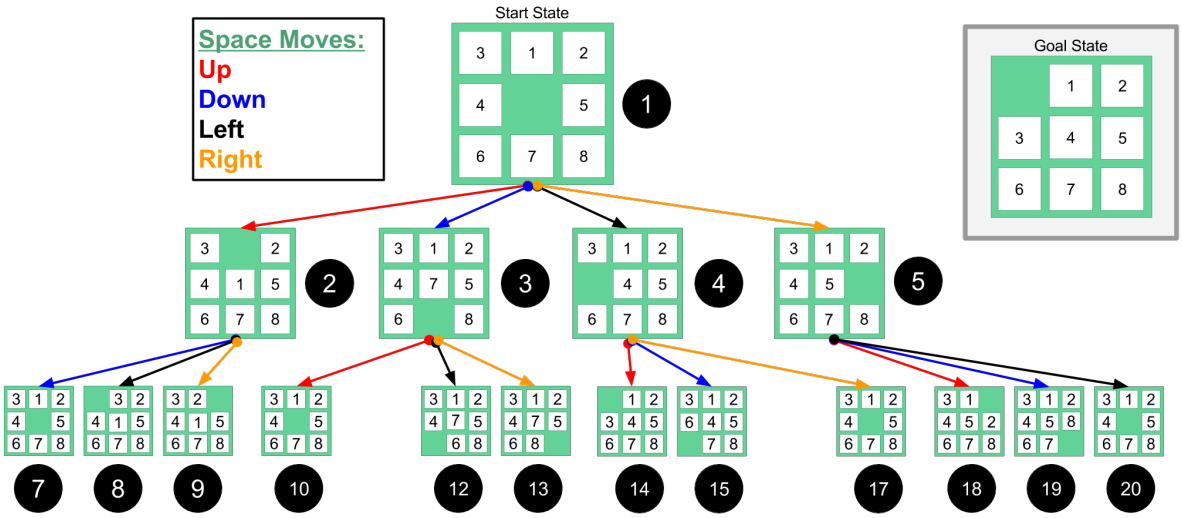

Figure 1: 8-puzzle Search Tree

List the order in which the nodes will be visited (up until the goal node) for

2.1.1 Breadth-first search (10pts),

1, 2, 3, 4, 5, 7, 8, 9, 10, 12, 13, 14

2.1.2 Depth-limited search with depth limit 2 (10pts),

1, 2, 7, 8, 9, 3, 10, 12, 13, 4, 14

2.1.3 and Iterative deepening search (10pts).

$l = 0$:
1
$l = 1$:
1, 2, 3, 4, 5
$l = 2$:
1, 2, 7, 8, 9, 3, 10, 12, 13, 4, 14

2.2 A* Search

Consider A* with the Manhattan distance heuristic as described in Section 3.6 of the text. Using the same start state as above (Subsection 2.1), fill in the table below with $g(n)$, $h(n)$, and $f(n) = g(n) + h(n)$ and the nodes ordered according to when they were generated by A*. Remember, $g(n)$ is the path cost from start (root) to n, and $h(n)$ is the heuristic function **(5pts)**.

Node	g(n)	h(n)	f(n)
1	0	2	2
2	1	3	4
3	1	3	4
4	1	1	2
5	1	3	4
14	2	0	2
15	2	2	4
17	2	2	4

2.2.1 Implementation

Implement A* with the Manhattan distance heuristic in Python 3.5.2. As a sanity check for your code, make sure your implementation takes the correct steps for the search problem above (Subsection 2.1) **(5pts)**. As an additional check, make sure your implementation returns a solution with 26 steps for the start state given in Figure 3.28 of the text **(10pts)**.

Using the start state given in Figure 3.28 of the text, generate 100 random start states (use the `Puzzle.shuffle` method). For each start state, compute an upper bound for the effective branching factor and calculate the average branching factor over the 100 start states **(15pts)**. The branching factor should be strictly less than 1.48, which is approximately the branching factor for the *misplaced tiles* heuristic (h_1) on 26-step puzzles (see Figure 3.29).

We have provided template python code, `my_hw1.py`, with input-output signatures. Fill in the missing code and verify your implementation produces the correct outputs to achieve full credit (5+10+15=30pts). You will need to make use of the `Puzzle` class we have provided. You can read the doc string for the class with `help(Puzzle)`. Also, you may find the queue package helpful.