Introduction to AI
Homework 1 solutions

1-

A1) practical, weak AI
A2) theoretical
A3) strong AI
A4) strong AI
A5) theoretical, strong AI
A6) theoretical, weak AI
A7) theoretical, strong AI
A8) practical, weak AI
A9) theoretical, strong AI

B1) student’s personal interests

2-
A) paper clip
performance measure: number of problems answered correctly
environment: computer interactions
actuators: screen, may be sound
sensors: keyboard, mouse

environment dimensions: Observable, Stochastic, Dynamic, Discrete (these vary depending on your assumptions and PEAS)

B) travel assistant
performance measure: getting to destination, time taken
environment: geographical area
actuators: screen, may be sound
sensors: location sensors

environment dimensions: Partially Observable, Stochastic, Dynamic, Continuous

C) rescue robot
performance measure: number of people found and time taken
environment: some type of field
actuators: navigation mechanism, pick up mechanism
sensors: location sensors, perceive sensors

environment dimensions: Partially Observable, Stochastic, Dynamic, Continuous

D) algebra tutor
performance measure: knowledge of students, time spent on tutoring
environment: physical class environment
actuators: speaking, writing
sensors: questioning (oral/written)

environment dimensions: Partially Observable, Stochastic, Dynamic, Continuous

E) robot soccer player
performance measure: number of goals, ball holding time
environment: some field or simulation
actuators: navigation mechanism, shooting mechanism
sensors: location sensors, perceive sensors

environment dimensions: Partially Observable, Stochastic, Dynamic, Continuous
A) Suppose we have disks 1, 2 and 3, with respective diameters and pegs A, B, C. We can represent a state as the set of disks on that peg – noting that there is only one legal ordering for the disks. So a state is a tuple ((A, \{1,2,3\}), (B, \{}), (C, \{})). This way we can avoid representing illegal states such as disk 3 on disk 1.

B) Initial state:
((A,\{1,2,3\}), (B, \{}), (C, \{}))

Goal state:
((A,\{}),(B, \{}), (C, \{1,2,3\}))

Successor function:
Idea: For each peg, move the top disk to either of the other two pegs, as long as the disks there are not smaller. So an operator is Move(from_peg,to_peg) and we need to double-check legality.

```python
legal_moves = {}
For from_peg = A .. C
    For to_peg = A .. C
        if (disk-on-top(from_peg) < disk-on-top(to_peg)) then
            add Move(from_peg,to_peg) to legal_moves

return legal_moves
```

Cost function:
number of moves

C) Move(A,C), Move(A,B),Move(C,B),Move(A,C),Move(B,A),Move(B,C),Move(A,C)

4- Search Space Formulation
We can formulate the class scheduling problem as a search problem by letting states represent partial schedules (schedules that do not necessarily include every class in C), and letting operators add classes to these schedules. Given this general idea, there are a number of reasonable ways to define the state space. We will discuss a formulation in which a state is a list of pairs \((c, q)\), where \(c\) is a class and \(q\) is a natural number representing the quarter in which the student will take that class. We will define our operators such that the quarters in each state are consecutive and each quarter has 1 to 4 classes assigned to it. One could also define a state as a list of pairs \((C_q, q)\), where \(C_q\) is the set of classes to be taken in quarter \(q\).

The initial state is an empty set; that is, a schedule in which no classes are assigned to any quarters.

We have two kinds of operators: one adds a class to the last quarter in the current schedule, and one adds a class to a new quarter that is appended to the end of the schedule. We are not allowed to add a class that is already in the schedule, or one whose prerequisites are not already scheduled for previous quarters. Also, we cannot have more than 4 classes scheduled for any one quarter.

We must also deal with the problem of repeated states. Now, because our operators only add classes to a schedule, we will never see the same state twice on a search path. However, we can still get repeated states; for example, we can get the schedule \{English, 1st quarter\}, \{Compilers, 1st quarter\} either by adding English first or by adding Compilers first.

There are a number of ways to deal with this problem. An easy way is to restrict the operators so that classes in the same quarter are scheduled in alphabetical order. That is, we cannot schedule a class in quarter \(q\) if a class that's alphabetically later has already been scheduled in quarter \(q\).

We could also avoid this problem by using the alternative state space definition mentioned above, and just having one kind of operator that schedules an entire quarter's worth of classes at once. This would result, in general, in a shallower search tree, but a larger branching factor.

Here is a formal definition of the operators, using alphabetical order to avoid repeated states. Let \(q\) be the highest number quarter in the schedule corresponding to a state. The legal successors of this state are those that add a pair \((c, q)\) such that:

* \(c\) is not in the schedule,
* all of \(c\)'s prerequisites are paired with quarters less than \(q\) in the schedule,
* \(q\) is either \(q\) or \(q+1\),
* if \(q = q\) then the total number of pairs \((c, q)\) in the new schedule is at most 4, and
* no alphabetically later class is already scheduled in quarter \(q\).

The goal test checks that the list contains all the required classes. The definition of the state space and operators precludes a schedule which contains all the classes but does not satisfy the prerequisite constraints.

As an alternative search space formulation, each state will be a list of tuples quarter, classSet corresponding to consecutive quarters. The list represents a partial schedule. For example, one state in the search space might be
Quarter1 \{cs109, cs140, cs157\} , Quarter2, \{cs248, cs145\}

The constraints for each tuple are:
* The size of classSet is at least 1 and at most 4.
* Each class in classSet is drawn from the initial list of required classes, and only appears once.
* For each class c in classSet, all of c's prerequisites must be in the classSet of a tuple with an earlier quarter.

The constraints for the entire state are the constraints for each tuple, plus:
* The quarter of the ith tuple in the list must have quarter = Quarteri. (Otherwise, the search space will have repeated states when the same set of tuples is added to the list in different orders.)
* Each class does not appear in more than one classSet.

The operator adds a new tuple to a state's list, subject to the above constraints. The initial state is an empty list of tuples. The goal test checks that all classes appear in the list, or, more formally, that the union of classSets in all tuples of the list is equal to the set of all required classes. Note that the goal test does not need to check any of the constraints listed above, because the initial state satisfies all constraints, and we have defined the operator so that no successor state can violate them.

B)

We assumed that tuition is charged per quarter, not per unit. Since the cost function must reflect the student's real-world cost, the cost of a schedule should correspond to the number of quarters in it. Thus, the operators above can have one of two costs. If the pair added is \( (c, q +1) \), then we have added a new quarter, and the cost of the operator is 1. Otherwise, we have scheduled a class in an already existing quarter, and the cost of the operator is 0.

For the alternative formulation, the operator always adds one quarter to the schedule. So the cost of the operator is 1.

C)

The heuristic function is an estimate of the remaining cost until the goal (graduation) is reached. In class, we discussed one approach for designing admissible heuristic functions: removing constraints from the operators, and estimating the cost to the goal using those relaxed operators. Here, there are two main constraints on the operators:
(a) you can't schedule more than 4 classes a quarter;
(b) you can't schedule a class before its prerequisites.

We can eliminate (a) or (b). (We can also eliminate both, but there's no need.) It turns out that both of these options lead to very intuitive admissible heuristics.

If we eliminate (b), our operators allow us to schedule any class in an empty slot, regardless of the prerequisites. Therefore, the cost is essentially the number of remaining classes, divided by 4 (since we can still schedule only 4 classes per quarter). More formally, the heuristic function:

\[ h = \text{the number of unscheduled classes minus the number of empty slots in the current quarter q} \div 4 \]
This heuristic is very easy to compute: we know the length of the current list of scheduled classes, so we can subtract this from |C| to get the number of unscheduled classes. Figuring out the number of empty slots in the current quarter is easy, since classes scheduled for the current quarter are at the end of the list; then we just have to do a division.

If we eliminate (a), our operators allow us to schedule as many classes a quarter as we want, so long as we satisfy the prerequisite requirements. Therefore, the cost is the longest chain of prerequisites among the unscheduled classes. More formally (and accounting for the quarter currently being scheduled), the heuristic function:

\[ h = \text{the length of the longest chain of prerequisites among the set of unscheduled classes and the classes scheduled for } q \text{ minus 1.} \]

Computing this heuristic is a bit more difficult: we need to iterate over the unscheduled classes and the classes scheduled for \( q \), and find the length of the chain of prerequisites leading to each one. This can be done with a recursive algorithm, augmented with a lookup table so we don't have to compute the length of the prerequisite chain for a given class more than once.

Both of the above heuristics will also work for the alternative formulation, with minor changes. For the first one, we ignore the number of empty slots in the current quarter, since the operator never adds more classes to current quarter. For the second one, we only look at the length of the longest chain of prerequisites among the set of unscheduled classes.

5- Termination of A*: When the algorithm first reaches the goal state, it might not be the optimal solution, that is a shorter solution might later be found. When a goal node has been selected for expansion, A* has found the shortest path to it. One example is provided in the text book, on page 98. In step e, Bucharest is reached but the cost is not optimal, and we need to continue until Bucharest is selected for expansion.