Homework 2
Planning and Graphical Models

15-381/781: Artificial Intelligence (Fall 2016)
OUT: Sept 21, 2016
DUE: Oct 3, 2016 at 11:59pm

Instructions

Homework Policy

Homework is due on autolab by the posted deadline. Assignments submitted past the deadline will incur the use of late days.

You have 6 late days, but cannot use more than 2 late days per homework. No credit will be given for homework submitted more than 2 days after the due date. After your 6 late days have been used you will receive 20% off for each additional day late.

You can discuss the exercises with your classmates, but you should write up your own solutions. If you find a solution in any source other than the material provided on the course website or the textbook, you must mention the source. You can work on the programming questions in pairs, but theoretical questions are always submitted individually. Make sure that you include a README file with your andrew id and your collaborator's andrew id.

Submission

Please create a tar archive of your answers and submit to Homework 2 on autolab. You should have three files in your archive: a completed `problems.py` for the programming portion, a PDF for your answers to the written component, and a README file with your Andrew ID and your collaborators' Andrew IDs. Your completed functions will be autograded by running through several test cases and their return values will be compared to the reference implementation. There is a `sample.txt` that contains sample inputs and outputs for reference.

1 Written [70 points]

1.1 Motion Planning [30 points]

Let S be a set of disjoint obstacles (simple polygons) in the plane, and let n denote the total number of their edges. Assume that we have a point robot moving on the plane that can touch the edges of the obstacles. (That is, we treat the obstacles as open sets.) The robot starts at the p_{start} position and must get to the p_{goal} position using the shortest collision-free path. In class, we proved that any shortest path between p_{start} and p_{goal} is a polygonal path whose inner vertices are the vertices of the obstacles. You may use this result in your answer to the following questions.

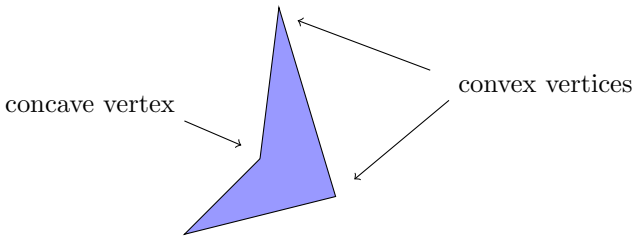

Figure 1: An example of a polygon with both convex and concave vertices.

1.1.1 [15 points]

We now consider two different kinds of vertices on obstacles.

i. A vertex whose internal angle formed by its two edges is less than 180° is called *convex*.

ii. A vertex whose internal angle formed by its two edges is greater than 180° is called *concave*.

See Fig. 1.

Extend the theorem from class by proving that a shortest path from p_{start} to p_{goal} is a polygonal path whose inner vertices (if they exist) correspond to obstacles' *convex* vertices.

1.1.2 [15 points]

Show that for any p_{start} and p_{goal} the number of segments on a shortest path is bounded by $O(n)$. Give an example where it is $\Theta(n)$.

1.2 Solving Special Cases of Classical Planning [20 points]

Suppose we are given a STRIPS planning instance in which all of the operators have no preconditions. Show that one can determine in polynomial time if it is satisfiable.

1.3 Graphical Models [20 points]

You are a basketball coach interested in determining the best way to spend your practice time in order to win your upcoming game. Specifically, you would like to know if you should spend your time practicing defence or practicing offence. If you practice defence, the probability the other team scores at least 100 points is 0.3, while if you don't practice defence the probability they score at least 100 points is 0.8. However, if you practice defence, the probability that your best player fouls out of the game is 0.5, while if you don't practice defence the probability is 0.2. Next, if you practice offence and your best player doesn't foul out, then the probability you score at least 100 points is 0.85; if you practice offence and your best player fouls out, the probability is 0.6; if you don't practice offence and your best player doesn't foul out, the probability is 0.5; while if you don't practice offence and your best player fouls out the probability is 0.2. Finally, if you score at least 100 points and so does the other team, you win with probability 0.6, while if both teams score less than 100 points than you win with probability 0.4. (Of course, if you score at least 100 points and the other team doesn't, you win with probability 1, and vice versa if you score at most 100 points and the other team scores at least 100.)

1.3.1 [10 points]

Provide a Bayes' Net describing all of the random variables and the dependencies. Add tables indicating all of the probabilities.

1.3.2 [10 points]

Should you practice offence or defence? (Assume that you can't practice both.)

1.4 Grad Problem: Boolean Satisfiability and the Lovász Local Lemma [20 points]

Perhaps the most famous CSP is *boolean satisfiability*, wherein an algorithm is given a boolean formula and must output whether or not it is satisfiable. We assume all bolean formulas are given in *conjunctive normal form* (CNF). Thus, a boolean formula ϕ on *variables* $x_1, \ldots, x_n$ can be written as $\phi = C_1 \wedge \cdots \wedge C_m$, where each C_j is a *clause* of the form $z_{i_1} \vee \cdots \vee z_{i_k}$, $1 \leq i_1 < \cdots < i_k \leq n$, where each z_{i_t} is a *literal*, i.e. it is either x_{i_t} or $\bar{x}_{i_t}$ (i.e., the variable x_{i_t} negated). An *assignment* is simply a mapping $a : \{1, \ldots, n\} \to \{T, F\}$, and an assignment a satisfies the formula ϕ if ϕ evaluates to true (T) when each variable x_i is replaced by the boolean value $a(i)$. More concretely, for our purposes, this means that for each clause $C_j = z_{i_1} \vee \cdots \vee z_{i_k}$, at least one literal z_{i_t} evaluates to true under the assignment a, i.e., $z_{i_t} = x_{i_t}$ and $a(i) = T$ or $z_{i_t} = \bar{x}_{i_t}$ and $a(i) = F$. We will focus on *k-satisfiability*, which is the boolean satisfiability problem with the additional promise that each clause contains exactly k variables.

The *probabilistic method* is a strategy for proving the existence of satisfying assignment for CSPs. Essentially, one defines a way of randomly generating assignments for a CSP, and then argues that such a randomly generated assignment has a nonzero probability of satisfying the CSP. This would then imply that there must be some satisfying assignment to the CSP, as otherwise the probability that a randomly generated assignment satisfies the CSP is necessarily 0.

1.4.1 [5 points]

Suppose $\phi = C_1 \wedge \cdots \wedge C_m$ is a k-CNF with $m < 2^k$ clauses. Use the probabilistic method to prove that it has a satisfying assignment.

Hint: Consider setting each variable to true or false with equal probability. Determine the probability that a clause evaluates to false, and then use a union bound (see: https://en.wikipedia.org/wiki/Boole's_inequality) to upper bound the probability that ϕ is not satisfied.

1.4.2 [5 points]

Show that the result in (a) is tight. Namely, give a k-CNF ϕ with 2^k clauses that is *not* satisfiable.

1.4.3 [5 points]

The *Lovász Local Lemma (LLL)* is a beautiful result that intuitively states the following. Suppose you have a bunch of "bad" events that you want to avoid, say when trying to prove the existence of something via the probabilistic method. If you have some amount of independence between the events, you can take many "localized" union bounds to prove that it is possible to avoid all the bad events. Here is a formal statement:[1]

[1] This is just one of the simplest versions of the lemma. For more information/more general versions, wikipedia's a good source: https://en.wikipedia.org/wiki/Lovasz_local_lemma.

Lemma 1. *Let $A_1, \ldots, A_m$ be a sequence of events such that each event occurs with probability at most p and such that each event is independent of all the other events except for at most d of them. If*

$$ep(d+1) \leq 1,$$

where $e = 2.718\ldots$ is Euler's number, then there is a nonzero probability that none of the events occurs. That is,

$$\Pr[\neg A_1 \wedge \cdots \wedge \neg A_m] > 0 \ .$$

Using the LLL, prove the following. Suppose ϕ is a k-CNF in which any clause shares a variable with at most $d := 2^k/e - 1$ other clauses. Prove that ϕ has a satisfying assignment.

1.4.4 [5 points]

Conclude that any k-CNF in which each variable appears in at most $\ell := \frac{1}{k} \cdot (2^k/e - 1) + 1$ formulas is satisfiable.

2 Programming [30 points]

Amayon, the online shopping conglomerate, has decided to change their delivery strategy. Instead of using a single drone to deliver to multiple residences, they now use many drones, each of which delivers to a unique residence. This way, there is no need to visit residences that haven't ordered new packages, and drones can be summoned on demand for every order. However, now the problem is that drones may end up colliding if their paths cross. Your goal is to plan paths for drones such that they each visit their designated residences and no paths ever cross.

Given a map of a neighborhood, along with the starting positions of the drones, you must construct a plan for all drones so that every residence is visited by its designated drone, and none of the paths cross at any point. The map will be a 2D rectangular grid divided into discrete squares. Each square is either free, an obstacle, or a residence.

As a visual reference, here is an ASCII rendition of a neighborhood map with 2 residences and drones and some walls:

```
0#1
.#.
A.B
```

The 0 and 1 represent drones, and the A and B represent residences. 0 is matched with A, and 1 is matched with B. This will be represented in code as 5 parameters: the number of rows, the number of columns, a list of wall positions, a list of residence positions, and a list of drone starting positions. For this example, there are 3 rows and 3 columns. The wall positions are

```
[
(0,1),
(1,1)
]
```

The residence positions are

```
[
(2,0),
(2,2)
]
```

and are also indexed in this order. The drone start positions are

```
[
(0,0),
(0,2)
]
```

and are also indexed in this order. A drone is matched up with the residence with the same index (there are always the same number of drones and residences).

There are multiple types of facts associated with every position `(row,col)` of the grid. There is fact `wall(row,col)` that represents a wall at that position. For every residence i, there is a fact `residence(i,row,col)` that represents residence i to be visited at that position. For every drone i, there is a fact `drone(i,row,col)` that represents drone i being at `(row,col)`. There is also a fact `occupied(row,col)` to indicate that some drone has been at `(row,col)` and is used to ensure paths don't cross.

There are two types of actions associated with every position `(row,col)` of the grid: movement actions and visit actions. There are up to 4 movement action for every drone i: `up(i,row,col)`, `down(i,row,col)` ,`left(i,row,col)`, `right(i,row,col)`. Only the actions that do not attempt to move off the grid are present e.g. `up(i,0,0)`,`left(i,0,0)` don't exist because those would try to move off the grid. The preconditions for a movement action is for a drone i to be at the position and for the corresponding adjacent square to be free of walls or other drones or was not visited by a previous drone e.g. the preconditions for `right(i,0,0)` is `drone(i,0,0)` $\wedge$ $\neg$`occupied(0,1)` $\wedge$ $\neg$`wall(0,1)`. The postcondition is an update of the drone's position e.g. the postcondition for `right(i,0,0)` is `occupied(0,1)` $\wedge$ `drone(i,0,1)` $\wedge$ $\neg$`drone(i,0,0)`. Apart from movement, there is also a visit action for every i, `visit(i,row,col)`, that makes the drone i visit residence i on the same square. The preconditions are `drone(i,row,col)` $\wedge$ `residence(i,row,col)`. The postcondition is $\neg$`residence(i,row,col)` $\wedge$ $\neg$`drone(i,row,col)`, which removes the residence from further consideration and also prevents the drone from further actions.

The goal is for every residence to be visited by its corresponding drone, and all the paths don't cross, which translates to $\neg$`residence(i,row,col)` for all residences and their corresponding positions. The initial state should have `drone(i,row,col)` facts placing each drone at its starting position. The starting drone positions should also be covered by `occupied(row,col)` to indicate that those spots have been visited by drones. The `wall(row,col)` facts should place the walls. The `residence(i,row,col)` facts should place the residences. Finally, you should also include all the negative facts that are true as well (e.g. $\neg$`residence(i,row,col)` for every position that does not have residence i, $\neg$`drone(i,row,col)` for every position that does not have drone i, $\neg$`occupied(row,col)` for every position that has not been visited by a drone, and $\neg$`wall(row,col)` for every position with no walls).

Note that you are free to use your own data structures to represent facts and you do not need to use the same names. We will not be explicitly checking how you keep track of facts.

For example inputs and outputs, see `sample.txt`.

2.1 Set Level Heuristic [15 points]

Implement the `set_level_heuristic(nrow,ncol,wall,res,start)` function by constructing a planning graph and returning the first level of facts where all the goal facts are present and no two goal facts are mutex. You should start from the initial state of the problem.

2.2 Planning [15 points]

Implement `plan(nrow,ncol,wall,res,start)`, either by using A* with the set level heuristic, or Graphplan and Extract-Solution. Note that for A* a node is a complete set of facts that are true (including negative facts), and you will need to reconstruct the planning graph every time you compute the heuristic; also, the initial set of facts of the planning graph should start from the current node's set of facts. You should return

the solution in the form of a filled in grid. For the example given above, the solution would look something like this:

```
0#1
0#1
0.1
```

and be represented in code as

```
[
[0,-1,1],
[0,-1,1],
[0,None,1]
]
```

The path that drone 0 took is filled in with its index of 0, and same with drone 1. The corresponding residences, since they have been visited, are also just filled in with the index of the drone. Walls are represented by -1. Free spaces should be represented by the python value `None`.