Homework 1
Search and SATs

CMU 15-780: Graduate AI (Spring 2016)
OUT: Jan 26, 2016
DUE: Feb 4, 2016 11:59pm

Instructions

Collaboration Policy

You may discuss assignments with other students as you work through them, but writeups must be done alone. No downloading or copying of code or other answers is allowed. If you use a string of at least 5 words from some source, you must cite the source. If you want to use a 3rd party python library, please let us know before Friday, Jan 29, 2016 so that we can ensure it is available to the autograder on autolab.

Submission

Please create a tar archive of your answers and submit to Homework 1 on autolab. You should have two files in your archive: a completed `problems.py` for the programming portion, and a PDF for your answers to the written component. 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.py` that contains sample inputs and outputs for reference.

You have 8 late days for homeworks over the semester, and can use at most 3 for one homework.

TAs

If you need help, the names beside the questions are the names of the TAs who came up with them, and are more likely to be familiar with the topics.

1 Written: A* [Daniel; 30 points]

For this question, assume search problems have non-negative step costs and finite branching.

1. [6 points] How can we emulate breadth-first search, depth-first search, and uniform-cost search with A*, specifically what should $g(n)$ and $h(n)$ be?
2. [9 points] Prove that if a heuristic $h(n)$ is monotonic, then it is also admissible.
3. [5 points] Suppose we have an admissible, non-negative heuristic $h(n)$. Recall that for a general graph search, A* needs to take care of duplicates by keeping the duplicate with the lowest f value rather than the first duplicate it encounters. Does this also include the goal state?
4. [15 points] Can a bi-directional A* work? What do h and g have to be for the reverse direction A*? If both directions of A* have an admissible heuristic, and the returned solution is the path created from where the two directions meet for the first time, is this an optimal algorithm? What if both directions have a monotonic heuristic?

2 Programming: SAT Solving [Guillermo and Daniel; 70 pts]

We want to solve SAT problems. Let n be the number of variables. We denote the variables as $x_0, \ldots, x_{n-1}$. A literal consists of a variable or the negation of a variable, eg. x_0 or $\neg x_0$. A clause is a disjunction of literals, eg. $x_0 \vee \neg x_1 \vee x_2$. A conjunctive normal form (CNF) formula is a conjunction of clauses, eg.

$$(x_0 \vee \neg x_1) \wedge x_2 \wedge (\neg x_2 \vee x_1).$$

We deal with only CNF formulas in this problem set.

In python, we represent literals as a 2-tuple whose first coordinate is an indicator for whether the variable is negated and whose second coordinate is the variable's index. Thus,

$$x_0 \equiv (0, 0)$$
$$\neg x_0 \equiv (1, 0).$$

A clause is a list of literals and a formula is a list of clauses For example, the following formulas become

$$x_5 \equiv (0, 5)$$
$$(x_0 \vee \neg x_2) \equiv [[(0, 0), (1, 2)]]$$
$$x_1 \wedge \neg x_0 \equiv [[(0, 1)], [(1, 0)]]$$
$$(x_0 \vee x_1) \wedge (\neg x_1 \vee x_2) \equiv [[(0, 0), (0, 1)], [(1, 1), (0, 2)]].$$

An assignment is a mapping from the variables to 1 (true) or 0 (false) values. For a two variable formula, one assignment could be a where

$$a(0) = 0$$
$$a(1) = 0$$

which maps $x_0 \to 0$ and $x_1 \to 0$. A partial assignment is when not all of the variables are mapped to values. We will represent the assignments as dictionaries in python.

To avoid some corner cases, we will ensure all variables in a clause are distinct, so a variable does not appear more than once in a clause.

1. **[5 pts] Warm up**: Implement the check function in the handout. Given a formula and a partial assignment of variables, the function returns False if there is a clause that is false under that assignment. It returns True otherwise.

   Lets do an example. Consider the three variable CNF formula

   $$x_0 \wedge (x_1 \vee x_2).$$

   If we consider the assignment that just sets $x_0 \to 1$, then no clause is False so we return True. If we consider the assignment $x_1 \to 0, x_2 \to 0$, then the second clause is false so we return False. If we consider the assignment $x_1 \to 0$, none of the clauses are immediately False so we return True.

   Note: We consider the empty clause to be False and the empty formula to be True.

2. **[20 pts] Simple Backtracking Solver**: Implement the simpleSolver function in the handout. Given the number of variables and a formula, it output a tuple (a, c). a is a satisfying assignment if there is one. If there isn't a satisfying assignment then a is False. c is the total number of times we branched on a value for a variable in the search.

   The algorithm is a simple backtracking solver. We assign variables in order of their index so x_0 is always assigned first. We will always try to assign a variable to be 0 (False) first and then 1 (True) if that fails. The algorithm will keep assigning variables until either the formula because unsatisfiable and we backtrack, or the formula is satisfied.

For example, consider the following two variable formula

$$\neg x_0 \wedge x_1$$

The variable with the smallest index is x_0 so we first set $x_0 \to 0$. We perform a check to see if the assignment is not inconsistent with the formula and proceed. Next, we set $x_1 \to 0$. We perform the check and fail because the second clause is false. We backtrack and set $x_1 \to 1$. We end up with a complete satisfying assignment so we return it. We made 3 assignments ($x_0 \to 0$, $x_1 \to 0$, and $x_1 \to 1$) here so we return that as well.

3. **[25 pts] Boolean Constraint Propogation (BCP)**: Unit propagation is the process of assigning variables in singleton clauses so that those clauses are True. Singleton clauses are clauses where there is only only one free variable left that hasn't been assigned. Boolean Constraint Propogation (BCP) involves repeating unit propogation until there are no more singleton clauses in each iteration. See wikipedia `https://en.wikipedia.org/wiki/Unit_propagation` for a more precise explanation of BCP. Make sure to scan the clauses from left to right when trying to find the next singleton clause to assign.

   Implement the unitSolver function. The function performs backtracking search with the BCP heuristic. The algorithm should first perform BCP before starting the backtracking procedure. After starting the backtracking procedure, the algorithm should perform BCP after every branch (i.e. trying to assign a value to the next lowest indexed, unassigned variable).

4. **[10 pts] Clause Learning Identificaton**: Clause learning involves maintaining an implication graph as described in lecture (under the slide for conflict graph). The implication graph at every level of backtracking should be constructed from the implications resulting from BCP. When there is a conflict, a new clause is induced that should be added to the formula.

   Below is an example of how to add new implications. Suppose we have the following clause:

   $$(\neg x_0 \vee x_1 \vee x_2)$$

   Suppose we have already assigned $x_0 \to 1$ and $x_2 \to 0$. Thus this clause is a singleton clause with free variable x_1. BCP would then assign $x_1 \to 1$. Then in the implication graph, we create a new node for that assignment $x_1 \to 1$, and we add two directed edges from the previously assigned x_0 and x_1 nodes to this new node for x_1. We also record the current backtracking level in which we made an assignment to x_1.

   Also, after any new assignments (whether it is from BCP or from branching), you should first check for unsatisfied clauses from left-to-right (i.e. clauses where all literals have been assigned a value but the clause is False a.k.a. a conflict). After finding an unsatisfied clause, you should create a new special conflict node, whose parents are the variables in the unsatisfied clause. Then you should stop BCP, since you have found a conflict, and continue on with identifying the conflict-induced clause.

   We will use the 1-UIP heuristic to construct a conflict-induced clause. See the lecture and `http://www.cs.tau.ac.il/research/alexander.nadel/SAT-05_CBH_2.pdf` for precise details on 1-UIP.

   For details in how to create the directed implication graph, refer to section 2.4 in "GRASP: A Search Algorithm for Propositional Satisfiability", Marques-Silva and Sakallah, IEEE Trans. Computers, C-48, 5:506-521,1999.

   In this problem, implement the implication graph as well as identifying the conflict-induced clause. However, just keep track of the conflict-induced clauses instead of actually adding them to the formula. In other words, the algorithm should proceed in the same way as in the previous question, except it should also keep a list of the clauses it would have added if it were to try to do clause learning. The literals in the conflict-induced clauses should be in variable index order.

   Implement the clauseLearningSolver function. The function performs backtracking search with BCP, and keeps track of all conflicted-induced clauses it encounters. It will return a potential satisfying

assignment, the number of assignments that were made during the branching part of backtracking (not BCP), and a list of the conflict-induced clauses that it encountered.

5. **[10 pts] Conflict-Directed Backjumping**: Conflict-directed backjumping is when we backtrack potentially multiple levels when we detect a conflict. For this problem, we will use the conflict-induced clause that we identify from clause learning in the previous question to perform a backjump.

   After reaching a conflict and computing the conflict-induced clause, we will add the the new clause to the formula. Then, we will backtrack up the search tree until we reach the first level that coincides with the level of one of the variables in the new clause. After backjumping, we will perform BCP again since we have added a new clause to the formula, and continue the search.

   For example, suppose we computed the following conflict-induced clause:

   $$(\neg x_0 \vee x_1 \vee x_2)$$

   where x_0 was assigned at our current level of 5, $x_1 \to 0$ at level 3, and $x_2 \to 0$ at level 2. We would first add the clause to the formula, and then backjump to level 3. After backjumping, recall that x_1 is still assigned the value 0. Then we perform BCP, which will discover that the newly added clause is a singleton clause, and assign $x_0 \to 0$. Note that it is possible BCP will again find a conflict, so we must handle that case and construct another conflict-induced clause, and backjump again.

   Implement the backjumpSolver function. The function performs bactracking search with BCP, clause learning, and conflict-directed backjumping.