Homework 1: Search and SAT

CMU 15-780: Graduate AI (Spring 2023)

Out: Jan 30, 2023
Due: Feb 13, 2023 11:59 pm

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 third-party Python library, please let us know at least one full week before the due date so that we can ensure it is available to the autograder.

Submission

You should submit two files via Gradescope: 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 (not just the ones in `problems.py`) and their return values will be compared to the reference implementation.

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

1 Written: A* [30 pts]

For this question, assume search problems have non-negative step costs and finite branching. Give proofs or counterexamples for all parts.

1. [3 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. [4 points] Prove that if a heuristic $h(n)$ is monotonic, then it is also admissible.
3. [3 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?

The possibility of bi-directional uniform-cost and A* search. Imagine that we want to do UCS or A* bi-directionally, that is, by running two searches simultaneously, one from the source s and one from the target t. The algorithm terminates when the two searches meet "in the middle".

4. [3 points] If running a bidirectional UCS, what does $g(n)$ have to be for the reverse direction?
5. [8 points] Consider the following algorithm: Run UCS from both the source and target until both runs have dequeued the same node n. At that point, return the concatenation of the $s \to n$ path from the forward search and the $n \to t$ path from the reverse search.

   Is this algorithm correct? That is, does it always return the shortest $s \to t$ path? If so, provide a proof. If not, provide a counterexample and propose (with proof) a fix that would make the algorithm correct.

6. [3 points] Now consider a bi-directional A* search, implemented analogously to your answer to Part 5. Given a heuristic $h(n)$ for the forward direction, state a non-trivial heuristic that can be used with the reverse direction.
7. [3 points] If h is monotone, is the resulting bi-directional A* algorithm (implemented according to your answers to to Parts 5 and 6) always correct?
8. [3 points] What about if h is admissible?

2 Programming: SAT Solving [70 pts]

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

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

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,

$$\begin{aligned} x_0 &\equiv (0,0) \\ \neg x_0 &\equiv (1,0). \end{aligned}$$

A clause is a list of literals and a formula is a list of clauses. For example:

$$\begin{aligned} x_5 &\equiv (0,5) \\ (x_0 \vee \neg x_2) &\equiv [[(0,0),(1,2)]] \\ x_1 \wedge \neg x_0 &\equiv [[(0,1)],[(1,0)]] \\ (x_0 \vee x_1) \wedge (\neg x_1 \vee x_2) &\equiv [[(0,0),(0,1)],[(1,1),(0,2)]] \end{aligned}$$

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

$$\begin{aligned} a(0) &= 0 \\ a(1) &= 0 \end{aligned}$$

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. Let's 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. Assign variables in order of their index so x_0 is always assigned first. 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

$$\neg x_0 \wedge x_1.$$

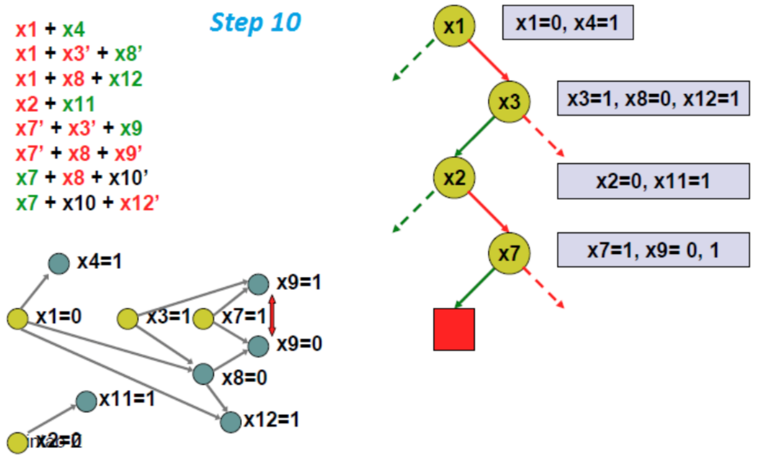
   The variable with the smallest index is x_0 so we first set $x_0 \to 0$. We perform a check to see if the assignment is not inconsistent with the formula and proceed. Next, we set $x_1 \to 0$. We perform the check and fail because the second clause is false. We backtrack and set $x_1 \to 1$. We end up 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 Propagation (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 one free variable left that hasn't been assigned, and all other literals in the clause are false. Boolean constraint propogation (BCP) involves repeating unit propogation until there are no more singleton clauses. 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. The return format is the same as Part 2. c should only count variable assignments made by braching, *not* variable assignments made by BCP.

4. **[10 pts] Clause Learning Identification:** 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.

   After any new assignments (whether it is from BCP or from branching), you should always check for unsatisfied clauses from left-to-right (*i.e.*, clauses where all literals have been assigned a value but the clause is false). After finding an unsatisfied clause, you should create the conflict node using the most-recently-assigned literal in the clause. Consider, for example, the case from lecture:



Here, when the unsatisfied clause $\neg x_7 \vee x_8 \vee \neg x_9$ was discovered, the node $x_9 = 0$ was added to the graph—not $x_7 = 0$ or $x_8 = 1$—because $x_9 = 0$ was the variable that was most recently assigned.

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 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 and 1-UIP. 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 conflict-induced clauses it encounters. It should return a tuple (a, c, ℓ) where a, c are as in Part 2 and ℓ is 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:

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

   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 backtracking search with BCP, clause learning, and conflict-directed backjumping. The return format is the same as Parts 2 and 3 (do *not* return the list of conflict-induced clauses).