Graduate Artificial Intelligence 15-780

Homework #1: *Knowledge Representation, SAT, and CSPs*

Out on January 28
Due on February 11

Problem 1: Knowledge Representation [20 pts.]

In the not too distant future, a team of three CoBots is devising to split up delivering mail to NSH and GHC. Here are some possible rules for their deliveries:

a) CoBot A delivers mail to NSH and CoBot B does not deliver mail to NSH and either CoBot C delivers mail to GHC or CoBot B does not deliver mail to GHC.

b) If CoBot B delivers mail to NSH then CoBot C delivers mail to both NSH and GHC and CoBot A delivers mail to either NSH or GHC.

c) If CoBot B delivers mail to NSH, it is necessary that CoBot A delivers mail to GHC and CoBot C delivers mail to NSH.

d) If CoBot A delivers mail to NSH then CoBot C delivers mail to either NSH or GHC and either CoBot B does not deliver mail to GHC or CoBot A delivers mail to NSH.

Q1. Encode statements (a)-(d) in propositional logic using the following literals: **5 pts.**

A: Cobot A delivers mail to NSH
B: Cobot A delivers mail to GHC
C: Cobot B delivers mail to NSH
D: Cobot B delivers mail to GHC
E: Cobot C delivers mail to NSH
F: Cobot C delivers mail to GHC

Q2. Transform the statements you just wrote to Conjunctive Normal Form (CNF). **5 pts.**

Q3. Decide whether each rule is satisfiable, unsatisfiable and/or valid. Give informal proofs as necessary. **5 pts.**

1. $A \implies \neg A$
2. $(A \implies B) \implies (\neg B \implies \neg A)$
3. $(A \implies B) \implies ((A \wedge C) \implies B)$

Q4. In class you learned about parse trees and converting a formula into CNF. Consider the formula:

$$(A \wedge B) \vee (C \wedge D) \tag{1}$$

The naive way to convert this into CNF is to use distributivity to obtain:

$$(A \vee C) \wedge (A \vee D) \wedge (B \vee C) \wedge (B \vee D) \tag{2}$$

Notice that every time that distributivity is applied the formula size doubles. In fact, this distributivity algorithm for converting formulas to CNF leads to exponential blow up in size. However, in practice you can solve the problem of generating CNF with much less space using the Tseitin transformation. The Tseitin transformation produces an equation that grows only linearly with the size of the input. The idea is to transform your formula into a CNF formula that is equisatisfiable (equivalently satisfiable but not identical) to your original formula.

Here is the algorithm for the Tseitin Transformation:

(1) Introduce temporary variables for each non-atomic sub-formula of your original formula. If $X = Y?Z$ where Y and Z are literals and ? is a boolean connective, then X is a sub-formula of your original formula.

(2) Assert that $X = Y?Z$ is equivalent to $X \leftrightarrow Y?Z$. The $\leftrightarrow$ sign here represents logical equivalence: $(X \rightarrow Y?Z) \wedge (Y?Z \rightarrow X)$.

(3) Convert $X \leftrightarrow Y?Z$ to CNF.

(4) "And" all the $X \leftrightarrow Y?Z$ CNF formulas together, as well as your original formula transformed with the temporary variables, to come up with the final CNF statement.

Example: Apply the Tseitin Transformation to $(a \wedge b) \vee ((c \vee d) \wedge e)$.

First, introduce the temporary variables x, y, z for the sub-formulas.

$$x = (a \wedge b)$$
$$y = (c \vee d)$$
$$z = (y \wedge e)$$

Assert a logical equivalence statement for each temporary variable.

$$x \leftrightarrow (a \wedge b)$$
$$y \leftrightarrow (c \vee d)$$
$$z \leftrightarrow (y \wedge e)$$

Converting these equations to CNF yields:

$$(\neg x \vee a) \wedge (\neg x \vee b) \wedge (\neg a \vee \neg b \vee x)$$
$$(\neg y \vee c \vee d) \wedge (\neg c \vee y) \wedge (\neg d \vee y)$$
$$(\neg z \vee y) \wedge (\neg z \vee e) \wedge (\neg y \vee \neg e \vee z)$$

The original formula transformed into the temporary variables is:

$$(x \vee z)$$

Finally, "and" all of these CNF formulas and the original formula together to get the final equisatisfiable CNF:

$$(\neg x \vee a) \wedge (\neg x \vee b) \wedge (\neg a \vee \neg b \vee x) \wedge (\neg y \vee c \vee d) \wedge (\neg c \vee y) \wedge (\neg d \vee y) \wedge (\neg z \vee y) \wedge (\neg z \vee e) \wedge (\neg y \vee \neg e \vee z) \wedge (x \vee z)$$

Question: Perform the Tseitin transform on the following formulas. **Show your work step by step as in the example. First, define the temporary variables. Second, show the logical equivalences being asserted. Third, show the logical equivalences in CNF. Finally, show the final answer. Solutions that do not explicitly show these four steps will NOT receive full credit. 5 pts.**

1. $(a \vee b) \wedge ((c \vee d) \vee e)$
2. $(a \wedge b) \vee ((c \wedge d) \vee e)$

Problem 2: FOL [20 pts.]

Consider the following knowledge base, where we show two statements in English and two statements in first-order logic (FOL).

a) All CatBot robots make electronic beeping noises at night.

b) $\forall x \forall y (Have(x,y) \wedge Real\_Cat(y) \Rightarrow \neg \exists z (Have(x,z) \wedge Mice(z))$.

c) Light sleepers do not have anything which makes electronic beeping noises at night.

d) Susie has either a real cat or a CatBot robot.

e) Conclusion: $Light\_Sleeper(Susie) \Rightarrow \neg \exists z (Have(Susie,z) \wedge Mice(z))$

(a) Write the statements 1, 3, 4 as well-formed formulas (wff) in FOL using predicates: $CatBot\_Robot(x)$, $Have(x,y)$, $Make\_Noise(x)$, $Real\_Cat(x)$, $Light\_Sleeper(x)$ and statement 2, and conclusion 5 in English. **5 pts.**

We will now prove the conclusion using the resolution inference rule, for which you need to carefully carry the following steps.

(b) Transform each wff by introducing variables instead of existential quantifiers (i.e., simple skolemize), and rewrite all the clauses in CNF (as shown on pg 345-346 of Russell/Norvig) **5 pts.**

(c) Prove the conclusion by resolution. At this time, you should have FIVE clauses as your CNF statements, and THREE CNF clauses as your conclusion. Please refer to the clause numbers when applying the resolution rule in your proof, and number the new clauses you get when as you proceed with your resolution proof. **10 pts.**

Problem 3: CSP Solver [60 pts.]

You have recently been employed as the Chief AI Officer at a company that builds social network games. You are working on the MMORPG game code-named *WarOfDoom*. The game consists of networks of players linked together on a graph. The nodes of the graph are players and two players share a link if they are friends. In the game:

- The server assigns each player a role of either `Warrior`, `Sorceress`, `Archer`, or `Blacksmith`.
- The server needs to assign roles to players on the graph such that *no friends have the same role*.
- The server uniformly randomly chooses N players to hold magical *tokens*. These players must be assigned the *same role*.

Luckily, you remember lectures from Graduate Artificial Intelligence on CSPs,[1] so you know exactly what to do!

1) Formulate the problem as a constraint satisfaction problem. **5 pts.**
   - What are the variables?
   - What are the values?
   - What are the constraints?

2) Implement a basic CSP solver (from scratch!) in the language of your choice. **25 pts.**
   - Implement a basic CSP solver with backtracking search. During the tree search, perform backtracking whenever at least one constraint is not satisfied, checking for this at every node in the search tree. Also, start your tree search with all variables' values unassigned, and assign values to variables one at a time from there.

     Required: Use the following *deterministic* ordering on the values: (1) `Blacksmith`, (2) `Archer`, (3) `Sorceress`, (4) `Warrior`. For variable selection, use the *deterministic* ordering of least to greatest vertex ID (that is, choose vertex 1 before 2, et cetera).
   - Implement the minimum remaining values (MRV)/most constrained variable heuristic. Tie-breaking is done via the deterministic ordering given above.
   - Implement the least constraining variable (LCV) heuristic. Tie-breaking is done via the deterministic ordering given above.
   - **Experimental Evaluation**: For each of the `easy_` test instances, run your solver in each of the following four configurations: just backtracking, backtracking and MRV (no LCV), backtracking and LCV (no MRV), and backtracking and LCV and MRV. As a table, report runtime and the number of nodes expanded.

	Basic		Basic+MRV		Basic+LCV		Basic+MRV+LCV	
Test Instance	Runtime (s)	Nodes	Runtime (s)	Nodes	Runtime (s)	Nodes	Runtime (s)	Nodes
`easy_1`	12.345	12345	…	…	…	…	…	…
`easy_2`	67.890	67890	…	…	…	…	…	…
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

3) Implement backjumping. **10 pts.**

   What is backjumping? Define the *conflict set* of a variable V to be all previously assigned variables that are also connected to V by at least one constraint. This conflict set can be maintained iteratively, as values are assigned to variables, or computed (at heavier cost) on the fly. Then, *backjumping* backtracks to the most recently assigned variable in the conflict set. You can think of backjumping as a more intelligent, accelerated form of backtracking. *For more information, see pp. 218–220 in R&N 3rd Ed., or pp. 148–149 in R&N 2nd Ed.*

[1] If you need a reminder, check out Chapter 6 of Russell & Norvig.

We would like you to implement backjumping in your solver. Please continue using the deterministic variable ordering, value ordering, and tie-breaking from above.

Experimental Evaluation: For each of the easy test instances, report the node count and runtime for your backjumping-enabled solver, with MRV and LCV turned on.

4) Implement forward checking. **5 pts.**

What is forward checking? Forward checking is a way to cut off unpromising search paths early. When assigning a value v_x to variable X, for all unassigned variables Y connected to X, remove all values from Y's domain that are inconsistent with X's value v_x. *For more information, see pp. 217 in R&N 3rd Ed.*

We would like you to implement forward checking in your solver. Please continue using the deterministic variable ordering, value ordering, and tie-breaking from above.

Experimental Evaluation: For each of the easy test instances, report the node count and runtime for your forward checking-enabled solver, with MRV, LCV, and backjumping turned on. Is there anything interesting about the node counts for your solver with all of {forward checking, backjumping, MRV, and LCV} enabled versus your solver with all of {backjumping, MRV, and LCV} enabled?

5) Implement conflict-directed backjumping (CDBJ). **5 pts.**

What is CDBJ? We will go over conflict-directed backjumping in the SAT context in lecture, then discuss CDBJ in a CSP context in recitation on Friday! *For more information, see pp. 219-220 in R&N 3rd Ed.*

We would like you to implement conflict-directed backjumping in your solver. Please continue using the deterministic variable ordering, value ordering, and tie-breaking from above.

Experimental Evaluation: For each of the easy test instances, report the node count and runtime for your CDBJ-enabled solver, with MRV, LCV, and forward checking turned on.

6) Improve your CSP solver! **10 pts.**

You are no longer required to use the deterministic variable and value ordering from above!

- Improve your solver! Here is a (non-exhaustive) list of potential improvements:
  - Improve your variable or value selection heuristics beyond LCV/MRV
  - Leverage CDBJ to implement no-good learning
  - Take the problem instances' structure into account
  - Consider using arc-consistency (see, e.g., the AC-3 algorithm in the book)
  - Anything else! Be creative!
- **Experimental Evaluation:** For each of the test instances, run your improved solver against the full determinstic solver (that is, the determinstic ordering on variables and values with backtracking, LCV, and MRV activated). Report your runtime and node count results in tabular form.

	Basic+MRV+LCV		My Awesome Solver	
Test Instance	Runtime (s)	Nodes	Runtime (s)	Nodes
`easy_1`	12.345	12345	…	…
`easy_2`	67.890	67890	…	…
⋮	⋮	⋮	⋮	⋮

Also, make sure to tell us about your solver! What did you implement? What worked? What didn't work?

7) Turn in your code! We'd like a tarball of (a) everything needed to run your code and (b) a README file telling us how to run your code. Please name the tarball `<your-andrew-id>.tgz`. Submission instructions to come (we'll set up a read-only Dropbox for students).

There is a good chance we will test your code on some undisclosed instances. Please make sure your code outputs the following (on a single line to standard output):

```
<andrew-id> <file-input-name> <number-of-nodes> <runtime> <feasibility>
```

As an example, Andrew ID `jqbovik`'s solver, run on instances `test_instance.graph`, would output

```
jqbovik test_instance.graph 1401 2.33 TRUE
```

if it found a successful solution to the instance in 2.33 seconds, expanding 1401 nodes. Conversely, it would output

```
jqbovik test_instance.graph 14 0.02 FALSE
```

if it proved no solution could exist for the instance, using 14 nodes and 0.02 seconds.

The Test Instances

Along with this document, we provide test instances—of varying difficulty—and a graph generator. You can download the generator and test instances from the homework section of the class website, or directly from `www.cs.cmu.edu/~15780/hw/<your Andrew ID>.tgz`. For example, the student with Andrew ID "jqbovik" would download from `www.cs.cmu.edu/~15780/hw/jqbovik.tgz`

A sample instance is formatted as follows:

```
<num-vertices>   <num-edges>   <num-tokens>
    <src1>         <dst1>
    <src2>         <dst3>
    <src3>         <dst3>
      ⋮              ⋮
    <srcN>         <dstN>
   <srcID>
   <srcID>
      ⋮
   <srcID>
```

The first line is information about the graph $G = (V, E)$ with K tokens. The next $|E|$ lines describe each edge. The last K lines mark the vertices with tokens. For example, the (tiny!) sample file with three vertices and two edges—from vertex 1 to vertex 2, and vertex 1 to vertex 3—as well as a single token placed on vertex 1 is encoded as follows:

```
3   2   1
1   2
1   3
1
```

You're only required to test on the instances we provide. Still, if you'd like to use our generator (which is just a little Python script), run it from the command line as follows:

```
$> python generator.py <num-vertices> <prob-of-edge> <num-tokens> <output-file> [<seed>]
```

Here, "prob-of-edge" is the probability of an edge existing between two vertices, and must be between 0 and 1. The generator uses the Erdös-Rényi model for random graph generation. The (optional) seed value can be included to set the random seed value for the graph generator.

Please let the TAs know if there are any glaring problems with the homework!