600.325/425 — Declarative Methods
Assignment 1: Satisfiability*

Spring 2005
Prof. J. Eisner
TA: John Blatz

Due date: Tuesday, February 22, 2 pm

In this assignment, you will gain familiarity with encoding real-world problems as instances of satisfiability,

Academic integrity: As always, the work you hand in should be your own, in accordance with University regulations at `http://cs.jhu.edu/integrity.html`

Programming language: You may program in whatever language you feel comfortable in, so long as your program can be run from a unix command line as described below. Make sure your code is well-documented.

How to hand in your work: Specific instructions will be announced before the due date. You must test that your programs run on `barley` with no problems before submitting them. You may prefer to develop them on `barley` in the first place, since the SAT software is already installed there. You can also work on `ugrad1`, ...`ugrad16`, which are on the same filesystem as `barley`.

Besides the comments you embed in your source code, include all other notes, as well as the answers to the questions in the assignment, into a file named `README`. Include directions for building the executables, either in your `README` or in a Makefile.

325 vs. 425: Problems marked "**425**" are only required for students taking the 400-level version of the course. Students in 325 are encouraged to try and solve them as well, and will get extra credit for doing so.

Unix help: You will need some very basic familiarity with operating on a Unix system to complete this assignment. If you don't feel comfortable using pipes ('`|`'), redirecting

*Thanks to Pandurang Nayak for the idea behind this assignment.

output (‘>’), or running programs from a command line, please come see the TA as soon as possible.

Data: All the files you will need for this project are or will soon be available on `barley` in `/usr/local/data/cs325/hw1/`.

We'll start off with some easy problems to get you used to thinking about CNF encodings.

1. Suppose that you have logical variables A, B, C, and D. Show how you would encode each of the following constraints as a CNF formula:[1]

   (a) A & $(B \vee C)$

   (b) $(A \vee B)$ & $(C \vee D)$

   (c) $A \Rightarrow B$

   (d) $C \Leftrightarrow D$

   (e) A & $\neg(B$ & $C)$

   (f) $A \Leftrightarrow ((B$ & $C) \Rightarrow \neg D)$

2. Now you will practice using a SAT solver by solving these puzzles. For this question, you don't have to hand anything in, just get a feel for using the software before we make real use of it in the later questions.

   (a) For each CNF formula you derived in question 1, write it in the DIMACS CNF format described in `satformat.ps`. You may find it helpful to use the script `convertToDIMACS.pl` that we have provided; this converts a file of the form

   ```
   foo v bar v ~baz &
   bang v ~foo &
   ...
   ```

   into the required format.

   (b) The two solvers that we will be using for this class are zChaff, which uses a constraint propagation algorithm based on the Davis-Putnam algorithm, and UBC-SAT, a collection of stochastic local search algorithms. They are already installed on `barley`. If you want to run them on your home computer, they are both

[1] TA's note: Some people use the symbol $\wedge$ to represent ‘and’, but I always get it confused with $\vee$ and so I use & instead.

available for free online from `http://www.princeton.edu/ chaff/zchaff/` and `http://www.satlib.org/ubcsat/`.

If you are running on `barley`, make sure that you have `/usr/local/data/cs325/hw1/zChaff/` and `/usr/local/data/cs325/hw1/ubcsat/` in your `PATH`.

(c) Run zChaff on each of these files to find a satisfying assignment using the command

```
./zchaff cnf_file
```

You may want to use the script `readOutput.pl` to recover the names you used for the variables.

(d) Run an algorithm or two from UBC-SAT using the command

```
./ubcsat -solve -i cnf_file -alg algorithm
```

where `algorithm` is an algorithm chosen from the options shown under `ubcsat -ha`. For help on the options allowed in UBC-SAT, run '`./ubcsat -h`'.

You don't need to include anything in your `README` for this part, just learn how to use the sat solver.

3. There is a famous class of puzzles which take place on an island inhabited by two different types of people: "knights," who speak only statements that are true, and "knaves", who speak only statements that are false. A typical puzzle will consist of a set of statements made by inhabitants of this island, and ask you to figure out who is a knight and who is a knave.[2] For each of the following situations, demonstrate how the problem of resolving who is a knight and who is a knave can be reduced to an instance of satisfiability by giving a CNF formula whose solution will enable you to solve the original problem. Explain how you derived that CNF formula. Use '$\sim$' to represent 'not', '`v`' to indicate 'or', and '`&`' to indicate 'and'.

Hint: For each person p, define a logical variable P corresponding to the proposition 'p is a knight'. Write each statement as a logical formula, then reduce that formula to CNF. You will probably find '$\Leftrightarrow$' useful.

(a) You meet two inhabitants, Amanda and Beth. Amanda says, "Beth is a knight". Beth says, "Well, at least one of us is a knight, anyway."

[2] If you find these puzzles enjoyable, Raymond Smullyan has published a series of books of these puzzles, of which the most famous is *What is the Name of this Book?*. Philip Lang, a grad student in the philosophy department at the University of Wisconsin, has put up a webpage with a knight-knave puzzle generator at `http://philosophy.wisc.edu/lang/211/knightknave.htm`, as well as an immense list of them.

(b) You meet three inhabitants, Charles, David, and Eric. Charles says, "If David is a knight, then Eric is too." David says, "I could tell you that Eric is a knave." Eric says, "Either Charles or I is a knight."

(c) **[425]** You meet two inhabitants, Faith and Gary. Faith says, "Either I am a knight or Irene is a knave." Gary says, "Only a knave would call Hal a knave."

(d) **[425]** You meet two inhabitants, Hal and Irene. Hal says, "Both of us are knights." You turn to Irene and ask, "Is that true?" She replies, and based on her reply, you know the classes of both characters.

4. Use one of the SAT solvers to find the solution to the puzzles from the previous question. Include the answers in your writeup.

5. **[Extra credit]** Just for fun, solve this knight-knave puzzle. You don't need to try and give a reduction to CNF, just an answer and an explanation.

> One time, I encountered two people on this island. I asked the first a question, and he answered. I couldn't yet figure out which types they were, so I then asked the same question to the second, to which he replied "yes". I still couldn't figure out who was what. Eventually, one of them said something or other, from which I was able to conclude that the second guy was a knave, though I still couldn't figure out what the first guy was.
>
> What was my original question?

6. Now that we have gained some familiarity with industrial-grade SAT solvers, we will use them to tackle a more difficult problem. In this section you will write code that will generate crossword puzzles by expressing the constraints as a CNF formula and passing the problem off to a SAT solver.

   Your input will be in the form of a `.puzzle` file, which will take the following form:

```
7 7
...#...
.#aorta
.u.#...
#n###.#
.i.#...
.t...#.
.e.#...
```

   The first line contains the number of rows and columns, and the subsequent lines represent the puzzle grid. The '`.`' character represents a blank space that must be

filled in with a letter, the '`#`' character represents a black space, and letters occurring in the `.puzzle` file must appear at the same location in the filled-in puzzle.

Your job is to find a way to fill in the blank spaces with letters so that every set of adjacent characters separated by '`#`' characters in a row or column of the puzzle is an English word.

You must write the following two programs:

(a) A program `encode` that generates a cnf from a `.puzzle` file. This will be run as:

```
./encode foo.puzzle /usr/dict/words > foo.cnf
```

where the second argument is the a of English words that will be allowed in the puzzle. `/usr/dict/words`, which is a word list distributed with Unix,[3] is suggested.

(b) A program `decode` that prints out the puzzle from the SAT solver output. It should be run as:

```
./decode foo.puzzle foo.output
```

and its output should mimic the format of the `.puzzle` file:

```
7 7
hot#oaf
a#aorta
mud#cid
#n###l#
aid#ate
steam#r
hen#usa
```

After you have written these programs, you can run

```
./crossword foo
```

which will read a puzzle description from `foo.puzzle` and print out a filled-in version to the file `foo.ans` by making calls to your encoder and decoder and to zChaff.

You may use whatever method you choose to do the encoding, however, you should be sure to describe how you did it in your writeup.

Some hints to help you out:

[3] Note: Versions of this file seem to vary from system to sytsem. For example, the file `/usr/share/dict/words` distributed with MacOS X is not the same file; it is the entire Webster's Dictionary, and has about 10 times as many words as the unix version, most of which you have never heard of. You should use the version found on `barley`.

- Most likely, you will want to have one variable for each square/letter pair; i.e. your variables will represent propositions like "There is an 's' in square (3,5)".
- Think carefully about how you are going to encode the constraint that all words in the puzzle must be in the dictionary. A naïve approach to express that there is a three letter word from (1,1) to (1,3) might look something like this:

  ```
  (C_11 & A_12 & T_13) v (B_11 & A_12 & G_13) v ...
  ```

  i.e. either the word is 'CAT' or it is 'BAG' or it is .... Unfortunately, this is not in CNF, and the most direct way to convert it into CNF will require a number of clauses that is exponential in the length of the word to encode when converted to CNF (Why?), Even for small words, this memory requirement quickly becomes prohibitive.

  However, we can also express the same constraints using statements like:

  - first letter is in the set {A, B, C, D, ..., Z}
  - if first letter is J, then second letter is in the set {A, E, I, O, U}
  - if first two letters are JI, then third letter is in the set {B,F,G,L,M,N,T,V}
  - if first three letters are JIN, then fourth letter is in the set {G, X}
  - if first four letters are JING, then fifth letter is L
  - if first five letters are JINGL, then sixth letter is E
  - if the first six letters are JINGLE, we must be at the end of the word

  You will also of course need constraints that prevent words from ending in the middle; 'JING' is not a valid word.

  Each of these statements is expressible as a disjunction quite naturally, so this description is much easier to represent in CNF. Furthermore, it avoids unnecessary constraints for words that share prefixes– we only need to say once that "JIN" is a valid start of a word, rather than to state it for both "JINX" and "JINGLE". Those of you who have taken data structures will recognize this as a *trie*.

What to hand in: Include in your `README` a description of the algorithms you used, and turn in filled-in puzzles `foo.ans` for as many of the puzzles as you can. You should also submit source code for all programs you wrote for this part, with a Makefile or directions on how to compile them.

Don't worry if you aren't able to solve all the puzzles–they're just included to illustrate the limitations of this method.[4]

[4] I was unable to solve `big.puzzle` and `huge.puzzle`

7. [**Extra credit**] Algorithms like Chaff try to find structure in the CNF formula they're trying to solve in hope of using it to help find a solution quickly. Thus, it may be possible to improve the performance of the solver by adding extra constraints that will help "guide" it toward a correct solution. By doing this we're blurring the line between the "declarative" and "procedural" ways of solving the problem. We're saying a little more than just the form of the solution, in hope of having a little more control over the implementation strategy.

   If you have time, see if you can find a way to improve performance of your encoding. You are not required to do this, however, extra credit will be available for extra effort. Here are a couple suggestions for speedups:

   - *Length constraints.* The range of letters that can follow"BAB" is limited to {B,O} if we know that the word has only 6 letters. The full trie would allow {B,E,O,Y}.
   - *Reverse trie constraints.* Add trie constraints that work backward from the end of the word, e.g. "if the last letter is W, the second-to-last letter is in { A, E, O }", "if the last two letters are EW, the third-to-last letter is in ...". We haven't tested this–it may or may not prove helpful.
   - *Arbitrary subset or substring constraints.* Add other constraints on valid word-formation based on your observations of the dictionary. "If the 3rd letter is Q, then the 4th letter is U." "If the 4th letter is U, then the 3rd letter is in {...}." "If the 2nd letter is X and the 5th letter is Y, then ..." The engineering trick is to decide how many of these to put in.

   Include in your `README` a description of any speedups you implemented in this section, and discuss what effect they had, if any. Turn in any source code you wrote for this part.

8. Modify a copy of `crossword` so that your program uses some of the random algorithms from UBC-SAT. How does this affect your program's runtime? Give a couple observations, and make a guess as to why they might hold. You don't need to turn in the modified `crossword` script.

9. In addition to solving SAT problems, UBC-SAT is designed to solve MAX-SAT problems. By passing the `-r best` option to `ubcsat`, it will return the assignment that satisfies the greatest number of clauses that it found while searching for a solution. This will work for any algorithm, however, SAMD and IRoTS are designed particularly for the MAX-SAT case.

   UBC-SAT is designed to solve weighted MAX-SAT problems as well, in which each clause has associated with it a weight, and the goal is to find an assignment that maximizes the total weight of all satisfied clauses.

We can make our crossword generator more powerful by encoding it as a MAX-SAT problem instead. In this problem, we will allow squares marked as blank (‘`.`’) to be filled in as black (‘`#`’) in addition to being filled by a letter.

The file format for weighted CNF formulas is identical to the unweighted case, except that

- The line that says how many variables and clauses are in the CNF formula begins `p wcnf` instead of `p cnf`.
- The first number in each clause is its weight, which is a real-valued number.

To run UBC-SAT on a weighted CNF formula, use the option `-w`. Since it won't be able to find an exact solution, you no longer need `-solve`, but since you want it to print out the best it can do, use `-r best`.

Modify the programs you've written so they allow you to place black squares on the board. Include the constraint that the board must have 180° rotational symmetry; i.e. if in an $n \times n$ puzzle the square at (x, y) is black, then the square at $(n-x, n-y)$ must be black too.

Your trie constraints shouldn't have to change much: For the 4th cell in a row or column, you'll want rules such as

- If the preceding 3 letters are BAG, must be in {#,A,E,G,P}
- If the preceding 3 letters are #BA, must be in {B,C,D,F,G,...}
- If the preceding 2 letters are #B, must be in {A,E,H,I,L,O,R,U,Y}
- If the preceding 1 letter is #, must be in {A,B,...Z}

You should put weights on your constraints so that the generator will include as few additional black squares as possible. Weight your constraints so that every square is filled in either as black or with a letter, and so that as many squares are filled by letters as possible, and so that it is still essential that all the words in the puzzle be in the dictionary. It may make it easier for you to surround the entire puzzle with black squares.

Run your solver on the blank puzzles provided. What are the densest (fewest black squares) puzzles you can generate?

Hand in your source code for this part, and describe your algorithm in your `README`. Include also some sample output on the blank puzzles.