600.325/425 — Declarative Methods
Assignment 1: Satisfiability[*]

Spring 2017
Prof. Jason Eisner

Due date: Monday, February 27, 2:00 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 (except as explained below), in accordance with University regulations at http://cs.jhu.edu/integrity-code.

Collaboration: ***You may work in pairs on the second half of this assignment,*** **which requires more programming than other assignments for this class. That is, if you choose, you may collaborate with one partner from the class on problems 7–12 only.**

Programming language: Let's make this as easy as we can. You may program in whatever language you feel comfortable in, so long as your program can be run from a Unix command line. If you are working in pairs, then the less experienced programmer should pick the language. You can ask anyone for help with basic programming stuff like I/O that is unrelated to the topic of the assignment. Make sure your code is well-documented.

How to hand in your work: You will hand in your work on Gradescope. The class entry code will be posted on Piazza.

For the **written** part: Please submit to **Assignment 1 Written** a PDF with each problem (**1, 2, 3, 5, 6, 7, 10**) on a new page in your submission. If you collaborate with a partner on questions 7 and 10, please submit only one writeup **for those two problems**. The other person's "solution" to those problems should merely indicate who they worked with. Each person must still complete and write up the first half of the assignment on their own.

For the **programming** part: Please submit to **Assignment 1 Programming** the following files: `encode`, `decode`, `wencode`, `wdecode`, `README` and any other (well-commented) programs they may call. `README` will be a **plaintext** file that contains your discussion of the code, in particular, answers to (**8, 9, 11, 12**). Please try submitting early: public tests are designed to ensure your code compiles on Gradescope machines. Please contact the TA if your code is not running correctly on Gradescope or if you have any concerns about your choice of language.

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 (not necessarily equal to the points indicated) for doing so.

[*]Thanks to Pandurang Nayak for the crossword puzzle idea, to John Blatz for co-writing an earlier version of the assignment, and to Jason Smith for improvements.

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 output ('`>`'), setting your `PATH` variable,[1] or running programs from a command line, please come see the TA as soon as possible. These short guides may help: `http://en.kioskea.net/contents/unix/unix-shell.php3`, `http://en.kioskea.net/contents/unix/unixcomm.php3`.

While late days are available for *emergencies* (`http://www.cs.jhu.edu/~jason/325/late-policy.html`), be wise and don't use them unless something unexpected happens. Plan ahead and start early!

CNF Encodings

We'll start off with some easy problems to get you used to thinking about CNF encodings. I've written a little tutorial to help: `http://cs.jhu.edu/~jason/tutorials/convert-to-CNF`.

1. [**12 points**] Suppose that you have logical variables A, B, C, etc. Show how you would encode each of the following constraints as a CNF formula.[2] For this problem, do *not* introduce any new variables.

   (a) [**0 points**] $A \vee (B \mathbin{\&} C)$

   (b) [**0 points**] $(A \mathbin{\&} B) \vee (C \mathbin{\&} D)$

   (c) [**2 points**] $(A \mathbin{\&} B) \vee (C \mathbin{\&} D) \vee (F \mathbin{\&} G \mathbin{\&} H)$

   (d) [**2 points**] $((A1 \vee A2) \mathbin{\&} (B1 \vee B2)) \vee (C \mathbin{\&} D)$

   (e) [**2 points**] $A \vee (B \mathbin{\&} (C \vee (D \mathbin{\&} E)))$

   (f) [**0 points**] $A \rightarrow B$

   (g) [**2 points**] $C \leftrightarrow D$

   (h) [**2 points**] $A \mathbin{\&} \neg(B \mathbin{\&} C)$

   (i) [**2 points**] $A \leftrightarrow ((B \mathbin{\&} C) \rightarrow \neg D)$

2. [**10 points**] Now solve (b)–(e) above by introducing "switching variables" (as shown in class). Where is this more efficient?

3. [**25 points**] Suppose you want to solve *arbitrary* SAT problems, but all you have is a CNF-SAT solver. So suppose you implemented the pseudocode at `http://cs.jhu.edu/~jason/tutorials/convert-to-CNF`, including the switching variable trick, to get an encoder that converts *any* propositional formula ϕ to CNF. (In other words, it can answer the questions above!)

   To find a satisfying assignment for a formula ϕ, you'd then do the usual encode-solve-decode routine:

   - run ϕ through your encoder to get a CNF formula ϕ'
   - call the CNF-SAT solver on ϕ' to get an assignment $\mathcal{A}'$ that satisfies ϕ' (or perhaps $\mathcal{A}' = \text{UNSAT}$)
   - run $\mathcal{A}'$ through your decoder to get an assignment $\mathcal{A}$ that satisfies your original formula ϕ

[1] Try `export PATH=$PATH:/usr/local/data/cs325/hw1` or perhaps `setenv PATH ${PATH}:usr/local/data/cs325/hw1`.

[2] It is more common to represent "AND" with the symbol $\wedge$ (which looks kind of like the first letter of "AND"). But here we'll write "AND" as & to make it look more different from the "OR" symbol $\vee$.

(a) [**5 points**] Describe how your decoder should work. (*Hint:* Remember that your encoder may introduce switching variables. So what does $\mathcal{A}'$ look like? And what does $\mathcal{A}$ need to look like?)

(b) [**425**] [**10 points**] Let n denote the size of the input formula ϕ, and assume that ϕ does not contain $\leftrightarrow$ or xor.

Prove that the encoder produces a formula of size only $O(n^2)$, in time only $O(n^2)$.

(*Hint:* Use induction on n. The pseudocode considers several cases; show that the requirements are met in each case, assuming the inductive hypothesis.)

(*Note:* Formula size is measured as follows. n denotes the total number of variables and operators in the input formula ϕ. Equivalently, n is the total number of nodes in an expression tree that represents ϕ.)

(c) [**10 points**] Your lab partner, Prof. Fastidious, now modifies your encoder to eliminate duplicate clauses from its output. For example, if your encoder produced a CNF formula ϕ' that included both $(A \vee \neg B \vee \neg C)$ and $(\neg C \vee A \vee \neg B)$, then the final step, added by Prof. Fastidious would automatically delete the latter clause.

Let's ask whether that was a good idea:

i. **Correctness.** Can the modified encoder still be used to solve arbitrary SAT problems, as intended? If not, why not? If so, does the decoder need to be modified correspondingly, and how?

ii. **Asymptotic efficiency.** As noted above, your version of the encoder ran in time $O(n^2)$. Can Prof. Fastidious's modified version still run in time that is a polynomial function of n? How fast?

iii. **Practical efficiency.** Do you think Prof. Fastidious's modification is likely to be helpful in practice? Discuss thoughtfully.

Using a Solver

4. [**0 points**] Now you will practice using an actual CNF-SAT solver. 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.

Data: All the files you will need for this project are available on the `ugrad` machines in `/usr/local/data/cs325/hw1/`.

(a) In the early 1990's, some folks running a SAT solving competition at DIMACS (at Rutgers University) came up with a standard string encoding for CNF formulas. This DIMACS CNF format is described in section 2.1 of `satformat.pdf` in the above directory.

Use the DIMACS CNF format to encode several of the CNF formulas that you produced in questions 1–2. You will probably find it easiest to use the script `convertToDIMACS` that we have provided. Look at the script for documentation. It produces the DIMACS encoding automatically if you give it a CNF formula in the more user-friendly encoding

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

(where `foo`, `bar`, etc. are being used as variable names). Just create a file such as `myformula.enc` in the user-friendly format, and type

```
convertToDIMACS myformula.enc
```

As a side effect, this also produces a file `myformula.key` that records the correspondence between DIMACS variable numbers and your user-friendly variable names.

(b) The three solvers that we will be using are

- zChaff, which uses a backtracking algorithm with constraint propagation (DPLL) plus learning, and which handles only SAT problems;
- UBC-SAT, a collection of stochastic local search algorithms, which can also handle weighted MAX-SAT problems;
- SAT4J, which also exploits some linear programming techniques that we will study later in the course, and which can also handle weighted and partial MAX-SAT problems (see question 11b).

These are already installed on the `ugrad` machines and other CS network machines. If you want to run them on your home computer, they are available for free online (`http://www.princeton.edu/~chaff/zchaff.html`, `http://www.satlib.org/ubcsat/`, `http://www.sat4j.org`).

(c) Run zChaff on each of your encoded files from part (a) to find a satisfying assignment. If you don't want to work with DIMACS format directly, you'll do

```
convertToDIMACS myformula.enc > myformula.cnf
zchaff myformula.cnf > myformula.output
readOutput myformula.output myformula.key > myformula.ans
cat myformula.ans
```

(d) Run an algorithm or two from UBC-SAT. The commands are the same as above, except that you should replace the word `zchaff` with

```
ubcsat -solve -alg ALGORITHM -i
```

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

You don't need to include anything in your writeup for this part. The point is just to learn how to use the SAT solvers and the scripts `convertToDIMACS` and `readOutput`. Feel free to play around!

5. [**18 points**] 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.[3]

In general, you can solve these puzzles if necessary by a generate-and-test approach: If X,Y,Z are knight,knight,knight, could they make these statements? What if they're knight,knight,knave? and so on. However, there is usually a more elegant way of getting to the solution.

For each of the following puzzles, we are not asking *you* to solve the puzzle of determining who is a knight and who is a knave (although you're welcome to do that too). Instead, demonstrate

[3]If you find these puzzles enjoyable, there are some great puzzle books along these lines by Raymond Smullyan, starting with *What is the Name of this Book?*. You can find a huge collection of knight-knave puzzles in `/usr/local/data/cs325/hw1/knights.pdf`. Philip Lang used to have an automatic knight-knave puzzle generator running at `http://philosophy.wisc.edu/lang/211/knightknave.htm`.

how that puzzle can be *reduced to an instance of satisfiability* that a machine could solve, by giving a CNF formula whose solution will enable you to solve the original puzzle. (An automatic SAT solver might attack that CNF formula using either generate-and-test or something more elegant.) Explain how you derived your CNF formula.

LaTeX tip: Use `\sim` for $\sim$ (or `\neg` for $\neg$), `\lor` for $\vee$, and `\land` for $\wedge$.

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) [**3 points**] 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."

(b) [**5 points**] You meet three inhabitants, Charles, David, and Eric. Charles says, "If David is a knight, then Eric is too." David says, "I am capable of telling you that Eric is a knave." Eric says, "Either Charles or I is a knight."

[**Note:** When knights and knaves use the English word "either," the intended meaning is always "either or both," corresponding to the logical operator $\vee$. (Ordinary English speakers sometimes use "either" differently, to mean "either but not both," corresponding to the logical operator xor.)]

[**Hint:** David did not simply claim "Eric is a knave;" he claimed that he had the ability to say that.]

(c) [**425**] [**5 points**] You meet two inhabitants, Faith and Gary. Faith says, "Either I am a knight or Gary is a knave." Gary says, "Only a knave would call Faith a knave."

(d) [**425**] [**5 points**] 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 yes or no, and based on her reply, you know the classes of both characters.

(In this puzzle, you will not simply be able to call a SAT solver on a single CNF formula. What should you do instead?)

6. [**5 points**] Use one of the SAT solvers to find the solution to the puzzles from the previous question. What would the SAT solver do if a puzzle did not give enough information to determine the knighthood or knavehood of all characters?

Constructing Crosswords

Collaboration: *You may collaborate with another student on the programming problems below.* Of course, the two of you should observe academic integrity and not claim any work by third parties as your own.

Now that we have gained some familiarity with industrial-grade SAT solvers, we will use them to tackle a difficult real problem. In this section you will write code that will generate crossword puzzles, by

- encoding the constraints on a valid puzzle as a CNF formula,
- passing that formula to a SAT solver, and
- decoding the satisfying assignment to obtain the puzzle.

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

```
9 9
#########
#...#...#
#.#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 square that must be filled in with a letter; the '`#`' character represents a black square; and (lowercase) letters occurring in the `.puzzle` file must appear at the same location in the filled-in puzzle. You may assume that the grid will be surrounded by a border of '`#`' characters—this convention makes your job easier.

Your job is to find a way to fill in all blank squares '`.`' with letters so that every horizontal or vertical string of letters bordered by '`#`' characters is an English word. The example input already has 4 such strings completely filled in, and fortunately they are English words: `aorta`, `unite`, `n`, and `o`. (For our simple definition of a valid crossword to work, we must consider single letters such as `n` and `o` to be "words.")

In short, your program is going to complete a crossword puzzle without any clues! Another perspective is that it will *create* a crossword puzzle—or rather, it will create the solution; then you would have to remove all the letters and write some clues ("blood carrier" ⇒ `aorta`) to create a puzzle that humans would be able to solve by hand.

Of course, your program will simply harness the magic of a SAT solver. All *you* need to do is to figure out how to write an encoder and a decoder:

- A program `encode`, whose input is a `.puzzle` file, and whose output is an encoding in the format of problem 4a:

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

  Your `encode` program should be runnable as:

  ```
  ./encode foo.puzzle words > foo.enc
  ```

  where the second argument is the list of English words to allow in the solution. This list should ordinarily include all the single-letter words (`a`, `b`, ...`z`), as discussed above. For your final answer you should use the `words` file that we gave you in the `hw1/puzzles` directory.[4] While getting your program working, however, you might want to try working with a list of only 3 or 4 words so that you can check your output by hand.

[4]We obtained this list of 20,068 words by typing `grep -v '[^a-z]' /usr/dict/words > words` on the machine `barley`. This extracts just the all-lowercase words from a standard Unix list of English words, `/usr/dict/words`. If you'd like to try a much larger English dictionary, try `/usr/share/dict/words` on one of the `ugrad` machines; that will give you 355,545 10 times as many words, most of which you have never heard of.

- A program `decode` that turns the SAT solver output back into a filled-in puzzle. If `foo.ans` holds the solver output—or more precisely, the cleaned-up version of that output produced by `readOutput`—then typing

  ```
  ./decode foo.puzzle foo.ans
  ```

  should print a filled-in grid in the same format as the `.puzzle` file:

  ```
  9 9
  #########
  #hot#oaf#
  #a#aorta#
  #mud#cid#
  ##n###l##
  #aid#ate#
  #steam#r#
  #hen#usa#
  #########
  ```

After you have written the `encode` and `decode` programs, you'll be able to run the `crossword` script that we've given you:

```
crossword foo words
```

That will read a puzzle description from `foo.puzzle` (along with a word list from `words`) and print out a completed puzzle to the file `foo.ans`. It does this just by calling your encoder, the zChaff SAT solver, and your decoder (in that order).

7. (a) [**5 points**] Why should you expect that an encode-SAT-decode strategy is capable of constructing crosswords? Well, the Cook-Levin Theorem says that a SAT solver is capable of solving any problem in NP. That is, if a problem is in NP, there *exist* polynomial-time `encode` and `decode` functions that can be used to reduce the problem to SAT. (Those functions may be tricky to describe, but at least you know they do exist!)

   So, check that the crossword problem is in fact in NP. That is, explain why it is a generate-and-test problem, whose test can be implemented in time that is polynomial in the length of the original input. (The input consists of the partially filled in grid + the dictionary.)

   (b) [**5 points**] Although a SAT solver *can* be used to solve any problem in NP, that doesn't mean it *should* be. A SAT solver might be overkill if the problem is easy, i.e., if there's a much faster way to construct crosswords.

   What kind of argument would convince you that the crossword problem is hard enough to need a SAT solver? (Don't give such an argument, unless you're ambitious, but say a little about how it would be structured. That is, what's a *general* strategy for proving that a problem is hard enough to need a SAT solver?)

Now it is time to actually write `encode` and `decode`. You may use whatever method you choose. However, you should be sure to describe *how* you did it in your writeup—a plaintext `README` file that you should submit along with the code.

You can choose any programming language (or languages) that will make your life easy. (Maybe a quick scripting language like Python or Perl?[5]) However, make sure your programs can be run exactly as shown above (else the `crossword` script won't work). For example, if you are using Java, you'd need to create a one-line executable file called `encode` that just contains something like

[5] I used Perl. My encoder and decoder were 51 and 13 lines of code, respectively, as counted by the `cloc` program.

```
java Encode $*
```

which will call `Encode.class`'s `main()` on the command-line arguments to `Encode`.

Some hints to help you out:

- 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 $(3, 5)$."
- It would be cool if we required different locations in the puzzle to have words in particular languages. That could be still done as a SAT problem. However, you are being asked to do only a boring case of that—each location requires an *English* word. So it's just English, English, English everywhere, not English, Arabic, Japanese ... Since all the locations are the same, your encoding will be quite repetitive!
- Think carefully about how you are going to encode the constraint that all words in the puzzle must be in the English 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 formula will get much bigger if we convert it to CNF. (How much bigger? Is it worse to have a long dictionary of short words or a short dictionary of long words?)

  A smarter encoding of the puzzle uses constraints like these, which do exactly the same job but more efficiently:[6]

  - following `#`, you must have one of {`#`, `a`, `b`, `c`, `d`, ..., `z`}
  - following `#j`, you must have one of {`#`, `a`, `e`, `i`, `o`, `u`}
  - following `#ji`, you must have one of {`b`, `f`, `g`, `l`, `m`, `n`, `t`, `v`}
  - following `#jig`, you must have one of {`#`, `g`, `s`}
  - following `#jigs`, you must have `a`
  - following `#jigsa`, you must have `w`
  - following `#jigsaw`, you must have `#`

  Each of these implications can be easily expressed as a fairly short disjunctive clause, so this description is much easier to represent in CNF. Furthermore, it avoids duplicating work for words that share prefixes. We only need to say once that `ji` is a valid start of a word, rather than stating this separately for `jitterbug` and `jigsaw`.

  (You may recognize this idea as related to a *trie* data structure. You are not expected to implement a trie, though. Using built-in basic hash tables and vectors should be enough to let you construct the formula pretty efficiently. Or you could use someone else's trie class if you prefer.)
- If your program is not working, try finding the smallest example that you can that reproduces the problem. Then track down what is going wrong on that example.
- If you are considering extending your program later for extra credit, you may want to read ahead to look at those extra credit problems now, in case this affects your design.

[6] Notice that this dictionary contains `j`, `jig`, and `jigsaw`, but not `jigs` or `jigsaws` or `jigsawed`.

8. [**40 points**] The following questions are intended to get you going in the right direction, so you probably want to answer them **before** you write any code. If we have problems understanding your code or getting it to work, we'll consult your answers to these questions.

   (a) In the `jigsaw` constraints above, why is `#` included in only some of the sets? (*Note:* the `words` file omits most plural words.)

   (b) What finished puzzle should be printed by `crossword micro microwords`? (Note that a puzzle can use a word more than once.)

   (c) What CNF formula should be printed by `encode micro.puzzle microwords`? *Hint:* The `jigsaw`-type constraints above are not the only clauses you need to include. If this isn't obvious yet, thinking about question 8f might make it more obvious.

   (d) How would these constraints differ if some of the inner squares of `micro.puzzle` had been filled in by letters or `#` symbols?

       (*Hint:* They shouldn't have to differ very much! You should be able to generate most of your encoding by knowing only the *size* of the grid. The fact that parts of the grid may already be filled in does *not* have to muck up your encoding of the dictionary.)

   (e) The dictionary is a big file. If `encode` has to read it over and over, it may be extremely slow. You want to read the dictionary file only once, using a data structure in memory to record which symbols are allowed to follow `#jig`.

       What kind of data structure will you use? (There's more than one reasonable option. Pick something that's pretty fast but doesn't require you to write much code of your own.)

   (f) How will you decode the output of the SAT solver to get your filled-in puzzle?

   When you write the code, you might want to try it first on `micro.puzzle` and `microwords`.

   Turning in your code: Explain your solution in your `README`, 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 on the `ugrad` machines.

   Don't worry if you aren't able to solve all the puzzles–the bigger ones just illustrate the limitations of this method.[7]

9. [**Extra credit, but there are more regular problems below**] [**variable points**] It may be possible to speed up the solver by adding extra, *redundant* constraints that reveal the problem structure and help "guide" the solver toward a correct solution.

   By doing this we're blurring the line between the "declarative" and "procedural" ways of solving the problem. In hopes of having a little more control over the implementation strategy, we're giving more than a minimal description of the solution.

[7]For reference, the TA's reference solution encoded `normal.puzzle` into a SAT problem with about 6,000 variables and 750,000 clauses, which zChaff solved in 7 minutes. But solving `big.puzzle` and `huge.puzzle` by the due date would probably require the extra credit extensions below. Here are some suggestions that may help if you are running out of disk space:

(a) You don't have to keep `big.enc` around once you have `big.cnf`.

(b) You can create your big files under the `/tmp` directory if they do not fit in your home directory. The `/tmp` directory is meant for temporary files. Files there are not subject to quota. On the other hand, they are not backed up and could be automatically deleted if you do not access them.

    I suggest making a subdirectory called `/tmp/`*`your_user_id`* as a convenient place to put your own temporary files, so that your `resolv_conf` file does not conflict with someone else's.

Please delete large temporary files when you are done with them, no matter where they are stored.

If you have time, see if you can find an encoding that improves performance. You are not required to do this; however, extra credit will be available for extra effort.

Here are a few suggestions for speedups. We don't know ourselves which ones will help! (Extra constraints might just bog the solver down without helping it pin down letters any faster; even some of the useful extra constraints might get discovered automatically anyway by a solver like zChaff that has clause learning.)

- *Reverse trie constraints.* Add trie constraints that work backward from the end of the word: e.g., "preceding `w#`, you must have one of {`a`, `e`, `o`}"; "preceding `ew#`, you must have one of . . . "

  You can imagine that these would be helpful if the solver has already guessed the end of the word.
- *Length constraints.* The range of letters that can follow `#bab` is limited to {`b`, `o`} if we know that the word has exactly 6 letters. The full trie would allow {`b`, `e`, `o`, `y`}.
- *Arbitrary substring or subset constraints.* Add other constraints on valid word-formation based on your observations of the dictionary. For example, what symbols (either letters or `#`) can precede or follow `ue`? This might be useful if the solver has already guessed the middle of the word. How about symbols at greater distance from `ue`? Most generally, how about symbols that can fall at the `?` positions in `#??u?e?`—that is, if the 3rd letter is `u` and the 5th letter is `e`, what can we say about the other letters?

  The engineering trick is to decide how many of these kinds of constraints to put in.

You are also certainly free to develop your own ideas. Include in your `README` a description of any speedups you implemented in this section, and provide some data comparing the performance of your original algorithm to the improved version. A (tasty) prize will be provided to anyone able to solve `big.puzzle` (or prove it unsatisfiable) in a reasonable amount of time. Be sure to turn in any source code you wrote for this part.

10. [**10 points**] Modify a copy of `crossword` so that your program uses some of the randomized 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. More thorough and more thoughtful answers will get more credit.

11. [**20 points**] In addition to SAT problems, UBC-SAT is designed to attempt MAX-SAT problems. If you include 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 (remember that `ubcsat -ha` lists the available algorithms), but SAMD and IRoTS are designed particularly for the MAX-SAT case.

    More generally, UBC-SAT can attempt *weighted* MAX-SAT problems, in which each clause has associated with it a weight, and the goal is to find an assignment that maximizes the total weight of the satisfied clauses. To run UBC-SAT on a weighted CNF formula, use the option `-w`. Again, to make it print out the best it can do, use `-r best` instead of `-solve`:

```
convertToDIMACS myformula.enc > myformula.cnf
zchaff myformula.cnf > myformula.output
ubcsat -w -r best -alg ALGORITHM -i myformula.cnf > myformula.ans
cat myformula.ans
```

We can make our crossword generator more powerful by using a a weighted MAX-SAT solver. In this version of the task, we will now allow empty squares (‘`.`’) to be filled in as black (‘`#`’) instead of a letter. Thus, you can start with a completely blank $n \times n$ grid, and try to construct a decent crossword out of it. For most n and most dictionaries, such a grid would be completely unsatisfiable if we weren't allowed to add black squares (we'd need $2n$ perfectly interlocking n-letter words).

(a) **[15 points]** The DIMACS 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.
- The encoding is traditionally stored in a file with the extension `.wcnf` rather than `.cnf`.

We have again provided a more user-friendly encoding scheme for you: look at the script `convertToWeightedDIMACS` for details. Basically, it's the same encoding as before, but you can put something like `w:5` at the start of each line, to indicate that the clause has weight 5. If you leave this part off, a very large weight will be used, indicating a hard constraint.

Modify the programs you've written so that the solver is allowed to place new black squares on the board. You should weight your constraints so they ask to have as few new black squares as possible. You will still need some "hard constraints" to ensure that the solution is a legal English crossword, just as before. These hard constraints should be given a sufficiently high weight that they will not be violated.

Overall, there are three kinds of constraints:

- **Decodability constraints**, which ensure that each square contains exactly one symbol (not < 1 and not > 1). These must be *hard* constraints, to ensure that any assignment produced by the solver can be decoded and printed out as a filled-in crossword grid.
- **Dictionary constraints**, which ensure that each word on the completed grid is in the dictionary. The `jigsaw` example shows that the partially correct `jigxyz` would violate just one of these constraints (saying that `#jig` must be followed by an appropriate letter). These constraints should probably be hard constraints, but see the discussion below.
- **Density constraints**, which attempt to rule out black (`#`) cells. These should be *soft* constraints since they will be enforced only to the extent possible.

Because UBC-SAT is a stochastic solver, it is possible that UBC-SAT will fail to satisfy the hard constraints—even though a trivial solution does exist, since the decodability and dictionary constraints are satisfied if you just fill all cells with `#`.[8] If the decodability constraints are violated, then your decoder will not be able to turn the resulting assignment into a grid, and should print an error message instead.

Your new encoder should be called `wencode` (for "weighted encode"). *Hint:* It shouldn't need to be very different from `encode`. In fact, one possible strategy is just to run `encode` to produce an unweighted formula, and then run that formula through a Perl script or

[8]It would therefore be reasonable to start UBC-SAT at this trivial solution and let it stochastically explore from there. However, we haven't found an option to pick an initial soluion in this way.

something to turn it into the weighted formula you want, which grudgingly allows using `#` in the `.` squares.

We've provided a script `wcrossword` that will do the wencode-MAXSAT-decode sequence for you. Run your solver on `blank6.puzzle`, `blank8.puzzle`, `blank10.puzzle`, and `blank20.puzzle`. What are the densest (fewest black squares) puzzles you can generate? How long does it take?

You might have to run the solver for a long time. You may want to experiment with some of the options: type `ubcsat` with no arguments to get help.

You might also want to experiment with the weight of the dictionary constraints, rather than simply letting `convertToWeightedDIMACS` pick an extremely large weight for you.

- *Low:* If you give a low enough weight to the dictionary constraints, then they will be soft constraints. In other words, optimal solution might place a small number of nonsense words in the puzzle, in order to avoid black squares. This is probably a bad idea (although it might be acceptable in practice if you can manage to write crossword clues for these nonsense words).
- *Moderate:* By raising the weight of the dictionary constraints, then you can force the optimal solution to avoid nonsense words.
  But note that a stochastic solver will not necessarily find the optimal solution. For example, GSAT may get stuck in local maxima, and these may have nonsense words.[9]
- *High:* By raising the weight of the dictionary constraints (or other hard constraints) too high, you will discourage some of the stochastic algorithms from violating them even temporarily. This may make it harder for the local search to break out of its current configuration and wander randomly into a useful new part of the search space.

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

(b) [**5 points**] So far, we have simulated hard constraints in a weighted MAX-SAT solver by giving them a sufficiently high weight. In principle, that should work perfectly. But there are also solvers that let you explicitly specify hard constraints (weight ∞). These are known as *partial MAX-SAT solvers*. They may be faster or more successful in such settings, since they usually treat hard and soft constraints differently in the solving algorithm.

We've installed one of these solvers, SAT4J, on the Ugrad machines. (By the way, SAT4J also provides an efficient library of SAT solvers that can be easily called from Java.)

You can use the same user-friendly format as in the previous question (that is, just omit the weight to indicate a hard constraint). However, there is yet another variant of the DIMACS format that must be used here. (The `p wcnf` line must specify one extra number—a large weight that will be interpreted as ∞ whenever it is used in the rest of the file.)

To convert to from your user-friendy encoding to the partial MAX-SAT DIMACS format, run `convertToPWeightedDIMACS` on it to create a file like `blank10.puzzle.wcnf`. Then run SAT4J on this `.wcnf` file, and interpret its output via `readSAT4JOutput`. We have provided a script `pwcrossword` to automate this wencode-MAXSAT-decode process.

Try using SAT4J to solve the blank crossword construction problems. Does it find better or worse solutions than UBC-SAT? Give examples. Note that we won't learn about the methods used in this solver until later in the course.

[9]By being clever about the way you set weights or add extra constraints, you may be able to ensure that even a GSAT solver will be able to avoid violating the dictionary constraints. Basically, you'd have to make it possible for GSAT to "climb" step by step out of a solution that violates the dictionary constraints—greedily improving its score at each step until the nonsense word has been shortened into a dictionary word or completely eliminated.

12. [**Extra credit**] [**variable points**] Now let's try to construct prettier puzzles from the blank grids. You can try either or both of these improvements.

   (a) [**easy**] Improve `wencode` so that the puzzles it creates will be symmetric. That is, if in an $n \times n$ puzzle the square at (i, j) is black, then certain other squares must be black too. You could experiment with reflectional or 180°-rotational symmetry.

   How much does this hurt the density of the puzzles generated? How about runtime? As for the previous problem, hand in your source code, a description, and some sample output.

   (b) [**harder, especially if you want the solver to handle it efficiently**] As we've stated the problem, UBC-SAT or SAT4 is free to make a crossword that basically falls apart into two crosswords—two white regions that are completely separated by a black region.

   Improve `wencode` so that the puzzles it creates will be connected. That is, from any white (letter) square, you should be able to get to any other white square by horizontal and vertical moves, without ever stepping on a black square.

   Answer as before.