15-780 Homework 2
Deadline: 10:30 am on February 17 (Tuesday)
There are 240 total points: point values are listed with each question.

1) Suppose we are given a search problem whose search space is a tree T with branching factor b and total depth m. Assume that the shallowest goal g has depth d, and that no other goals have depth d. Also assume that g is the last node of depth d added to the search queue by both Breadth First Search (BFS) and Iterative Deepening Search (IDS). Please justify your answers to the following questions, evaluating all summations (70 pts total: 10 pts each part).

a) For $i \in [0, m]$ how many nodes are there in T of depth i?

b) How much space does BFS require in this search problem?

c) How much space does IDS require in this search problem?

d) How many nodes does BFS visit before finding the first goal?

e) How many nodes does IDS visit at iteration i (where iteration 0 is the iteration where we remove the start node from the queue for the first time)?

f) How many total nodes does IDS visit before finding the first goal?

g) Which algorithm visits more nodes—BFS or IDS? What are the asymptotic running times (i.e., in big O notation) of the two algorithms, treating b, d, and m as parameters? Please justify your answers using the prior parts of this question—do not just cite results from lecture.

2) Consider the map k-coloring problem discussed in class: we are given a map, represented as a list of regions $R = \{r_i\}$ and a list of pairs of adjacent regions $A = \{(r_i, r_j)\}$, and must assign one of k colors to each region without ever assigning the same color to two adjacent regions. In this question you will describe how to set this problem up as a formal search problem in *two* distinct ways. Interestingly, while both formulations are quite different, it is not clear which one is better, and each formulation appears to outperform the other on a subset of the problem instances (70 pts total: 5 pts each part).

We will provide informal English-language descriptions of the two formulations. Note that there may be more than one way to translate our descriptions into more-formal pseudocode. Your answers to each part will depend on the exact details of your particular translation. We will therefore weight your justifications and internal consistency more heavily than whether you got the same final numbers we did.

Recall that a search problem consists of the following elements: a data structure node, a function initialNode(), a function getSuccessors(node), and a function testGoal(node).

In the first formulation, a node will represent a partial coloring of the map, and we will color the entire map by successively coloring one additional region.

a) Describe the node data structure for this formulation. How many total nodes are there?

b) Give pseudocode for the initialNode() function of this formulation.

c) Give pseudocode for the getSuccessors(node) function of this formulation. What is the maximal number of successors of a node?

d) Give pseudocode for the testGoal(node) function of this formulation.

e) What is the branching factor of the search tree T induced by the search problem?

f) What is the total depth of T?

g) What can you conclude about the depth of the shallowest goal in T?

In the second formulation, a node will represent a complete coloring of the map, and we will color the entire map by repeatedly changing one of the regions to be a new color.

h) Describe the node data structure for this formulation. How many total nodes are there?

i) Give pseudocode for the initialNode() function of this formulation.

j) Give pseudocode for the getSuccessors(node) function of this formulation. What is the maximal number of successors of a node?

k) Give pseudocode for the testGoal(node) function of this formulation.

l) What is the branching factor of the search tree T induced by the search problem?

m) What is the total depth of T?

n) What can you conclude about the depth of the shallowest goal in T?

3) (100 pts total) In this question you will implement a boolean satisfiability (SAT) solver that takes a set of variables and constraints in conjunctive normal form (CNF) and returns either a satisfying assignment or determines that no satisfying assignment is possible.

In class we discussed the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, which, after almost 50 years, is still the basis for some of the world's fastest SAT solvers. Although SAT is an NP-complete problem, DPLL is able to efficiently exploit CNF clause structure with heuristic techniques like backtracking and unit propagation to solve problems with thousands of variables (see the lecture notes for details). As we will see later in the semester, many NP-complete problems are easily translated into SAT problems, so developing efficient SAT solvers has long been an active area of research with SAT dedicated conferences and annual competitions.

Satisfiability problems are often written in a standard text format called "DIMACS CNF" format which the program that you are about to write will have to be able to parse. The format is defined

as follows:

- Each line that begins with a '`c`' is a comment.
- The first non-comment line must be of the form: '`p cnf numberOfVariables numberOfClauses`'
- Each of the non-comment lines after the first line defines a clause. These lines are space separated lists of variables with a positive value indicating the variable and a negative value indicating the negation of the variable. The line is always terminated with a space followed by a '`0`'.

For example, the first few lines of this .cnf file

```
c SAT07-Contest Parameters: unif k=3 r=4.2 v=19000 c=79800 seed=49237390
c Uniform UNKNOWN KSAT Instance k=3, nbc=79800, nbv=19000, seed=49237390
p cnf 19000 79800
1987 452 3709 0
-9366 -9214 -1319 0
-17825 8919 4446 0
-16248 11524 -13331 0
4850 -908 -7624 0
```

indicate that this is a 19,000 variable SAT problem with 79,800 clauses. The first clause is $(x_{1987} \vee x_{452} \vee x_{3709})$, the second clause is $(\neg x_{9366} \vee \neg x_{9214} \vee \neg x_{1319})$, etc.

The SAT problems required for the following questions can be downloaded from:
http://www.cs.cmu.edu/~sandholm/www/cs15-780S09/hws.html
under Homework 2. The archive also contains a verifier in C that will allow you to check your solutions for each SAT problem.

a) (60 pts) Implement DPLL. When solving SAT problems, speed is paramount and you will be graded on program correctness *and* speed of execution. You should use the heuristics discussed in class to make your implementation as fast as possible. These include:

- Backtracking.
- Unit propagation.
- Variable ordering heuristics.
- Value ordering heuristics.
- No branching on variables that only occur in Horn clauses.

You should *not* implement clause learning or conflict-directed backjumping. You will get to implement those in a future assignment.

Your implementation should be in either Java or C/C++. Note, however, that if you choose Java you will need to find creative ways to speed up your program in order to compete with programs that are written in C/C++. If you are writing your program in C or C++, be sure to compile with level 3 optimization (e.g. `gcc -O3 satsolver.c -o satsolver`).

Do *not* attach your code to your writeup. Instead, email your source code to Byron by the due date and time.

b) (10 pts) Run your program on each of the following satisfiable SAT problems:

problem1.cnf
problem2.cnf
problem3.cnf
problem4.cnf
problem5.cnf

The solutions to each sat problem should be recorded in separate solution files, each named `problem_NUMBER.sol`. Each line of the `.sol` format contains a variable name followed by a space and the Boolean assignment. For example:

```
1 1
2 0
3 0
4 0
```

Please send your solution files by email to Byron by the due date and time.

c) (10 pts) For each of the following SAT problems, use your SAT solver to determine whether each problem is satisfiable or unsatisfiable. You *do not* have to report the actual satisfying assignments for this part.

problem6.cnf
problem7.cnf
problem8.cnf
problem9.cnf
problem10.cnf
problem11.cnf
problem12.cnf

d) (20 pts) Each of the problems in part c was generated from the same random CNF generator, but with different numbers of variables and clauses and different ratios of clauses to variables. Report the amount of time that it takes to solve each SAT problem in part c.

Do these times make sense? Specifically: How does the time it takes to solve each problem increase as the number of variables increases (with the same ratio of variables to clauses)? How does the changing the ratio of variables to clauses change the time it takes to solve these randomly generated problems?

Will these timing results hold for real-world SAT problems? What other factors might influence the time it takes DPLL to solve a SAT problem?