Grad AI. 15-780 Fall, 2006

Homework 2

- *Homework deadline: 10:30am on Oct 16*
- *Please print your code and hand it in with the hard copy of your homework. Also send a copy of your code by e-mail to both TAs.*

1. **Planning**. This problem involves constructing a plan graph instance, reducing it to a satisfiability problem and solving it with a freely available SAT solver.

   The task that we will be planning in this problem is the *task of building a plan.* When you start building a plan you will start with knowledge of planning. To complete the task you must build a plan graph, build the corresponding SAT instance and solve it using a SAT solver. However, before you can use the SAT solver you must have downloaded it from the course web site, and in order to build a plan graph or build the corresponding SAT instance you must have knowledge of planning.

   (a) [**5 pts**] Describe the states and actions necessary to complete the task of building a plan. States should be of the form "have(x)" and "$\neg$ have(x)," where x is a string that describes one of the things needed for planning. Actions should be of the form "build(y)," where y is a string that describes one of the things that must be built/downloaded to execute the task of making a plan.

   (b) [**5 pts**] Describe the pre-conditions and post-conditions for each of the actions you listed in part (a).

   (c) [**5 pts**] Draw the plan graph that represents the task of building a plan as described above (you need only draw enough levels to reach the first appearance of the goal state). Draw "mutually exclusive" (mutex) edges with dashed lines between actions with contradictory post-conditions, actions that delete pre-conditions of each other, and actions with mutex pre-conditions. Also include mutex edges between literals with no non-mutex actions that can achieve both (you do not need to include mutex edges between contradictory literals).

   (d) [**8 pts**] Convert the plan graph to its corresponding satisfiability instance. Download the SAT solver linked from the course website and solve the plan. Provide your input to the SAT solver and the solution it returns. Our reccomended SAT solver is RSat, which takes input in the 2004 SAT competition format detailed on this site: `http://www.satcompetition.org/2004/format-solvers2004.html`. First your SAT formula must be in conjunctive normal and then each clause is given by a separate line in the file.

   (e) [**5 pts**] What can you conclude, if anything, about the worst-case complexity or complexity of checking a solution (i.e. the complexity class) of planning and/or SAT based *only* on the reduction you have performed in this problem?

2. **Constraint Satisfaction**. In this problem you will write code to generate and solve random constraint satisfaction problems. For simplicity we will consider only problems with *binary* constraints, which are constraints between two variables (note that problems with n-ary constraints can be converted to binary constraints by introducing new variables). By varying the parameters of the generator you will explore the hardness landscape and find the "phase transition" of these problems. A binary constraint satisfaction problem (CSP) is defined by the following,

   - n, the number of variables in the problem
   - $\{m_1, \ldots, m_n\}$, the number of values in each variable's domain
   - k, the number of constraints
   - $\{C_1, \ldots, C_k\}$, a 0-1 matrix for each constraint specifying which values it forbids. If C is a constraint between variables i and j, it has dimensions $m_i \times m_j$, such that each row corresponds to a value in i's domain and each column to a value in j's domain. A value of 1 at the intersection of a row and column indicates that the two corresponding values are consistent, and a 0 indicates the two values conflict and cannot occur together in the same solution.

   (a) **[3 pts]**: Show how a binary CSP that has variables with different domain sizes (i.e. there exists at least one pair i and j such that $m_i \neq m_j$) can be reduced to an equivalent problem where all variables have the same domain size (i.e. $m_i = m_j = m$ for all i and j) with $O(n)$ additional constraints.

   (b) **[5 pts]** For the remainder of this problem we will assume that all variables have the same domain size, $m_1 = \ldots = m_n = m$. A random binary CSP instance can be characterized by 4 parameters: i.) n, the number of variables; ii.) m, the number of values in each variable's domain; iii.) p_1, the probability that there is a constraint between any two variables (also called constraint density); and iv.) p_2, the probability that a pair of values is inconsistent for a pair of variables given that there is a constraint between them (also called constraint tightness). Using parameters $\langle n, m, p_1, p_2 \rangle$, write equations for the expected number of constraints and the expected number of inconsistent values per constraint of an instance generated with those parameters.

   (c) **[10 pts]** Write code that takes $\langle n, m, p_1, p_2 \rangle$ as input and generates a random binary CSP instance. Rather than interpreting p_1 and p_2 probabilistically (for example by flipping a coin with probability p_1 for each pair of variables to determine if they have a constraint), write your code so that every instance generated has the expected number of constraints and constrained values you calculated in part (a). (If this value turns out to be a non-integer please round it to the nearest integer using Matlab's `round()` procedure or an equivalent method). The reason we take the value of p_1 and p_2 to specify the problem precisely (rather than on average) is that our ultimate goal will be considering randomly-generated CSPs to make predictions about the behavior of problems with a particular numbers of constraints and numbers of inconsistent pairs of values.

(d) **[14 pts]** Now you will find the parameter settings that create difficult random binary CSPs. Generate 100 random binary CSP instances using your generator with parameters $\langle n = 8, m = 5, p_1 = 1.0\rangle$ and vary p_2 from $[0.1, 1]$ at steps of 0.1 (by this scheme you will generate a total of 1000 instances, feel free to expand on this for more values of p_2 if you wish but this should be sufficient). Use your code for DFS from Homework 1 to solve the each of the CSP instances you generate and provide a plot of mean runtime in milliseconds versus p_2 with the phase transition clearly marked. The phase transition in this experiment[1] is the area where problems go from being under-constrained (always solvable) to over-constrained (hardly ever solvable). The Matlab function `clock()` will be useful for timing the run on each instance. (Note: running this entire experiment should not take longer than a few minutes on a desktop PC).

3. **Linear Programming and Duality**. Now we will focus on a traditional optimization problem called MAX FLOW. In a MAX FLOW instance we are given a directed network with weighted edges, a start node (called the source) labeled s and an end node (called the sink) labeled t. The weights on edges indicate their capacity to carry flow through the network. Our goal is to route as much flow as possible from the source node to the sink node, without exceeding the capacity of any edge. In addition, a valid solution should not waste any flow, that is any flow that enters a node other than the sink should leave it. Figure 1 shows a simple MAX FLOW instance.

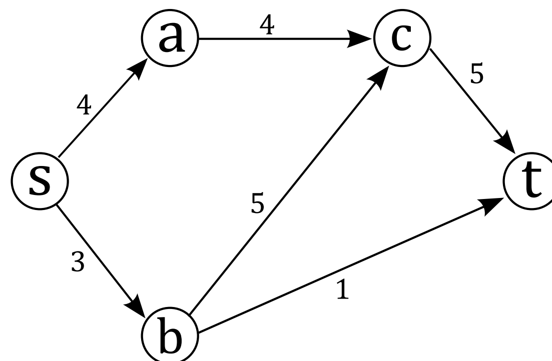

Figure 1: A simple MAX FLOW instance with source s, sink t, and edge weights indicating the capacity of each edge.

(a) **[10 pts]** Write the objective function and constraints of a linear program that solves the MAX FLOW instance in Figure 1 (hint: your LP should have one variable per edge). Your LP should be in its canonical form (shown below, note the min). Please provide the A, b, c, D, and e matrices for the instance in Figure 1.

$$\min_x c^T x$$

[1]This is a slightly different experiment than we saw in class, which looked at the phase transition of random *solvable* SAT instances.

$$\begin{aligned} Ax &\leq b \\ Dx &= e \\ x &\geq 0 \end{aligned}$$

(b) **[10 pts]** Write down the dual of your LP from part (a). What interpretation do the dual variables have in the original MAX FLOW problem?

(c) **[15 pts]** A MAX FLOW instance with n nodes can be represented as an $n \times n$ matrix, where the entry at row i and column j indicates the capacity of an edge from node i to j (this is sometimes called a directed weighted adjacency matrix). An entry of 0 indicates that the corresponding edge does not exist. Entries in the first row (with $i = 1$) represent edges leaving the source and entries in the last column ($j = n$) represent edges going to the sink. For example, the instance in Figure 1 is specified with the following matrix:

$$\begin{pmatrix} 0 & 4 & 3 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 5 & 1 \\ 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Write a program that reads in a comma separated value (CSV) file specifying the matrix form of a MAX FLOW problem and constructs the corresponding A, b, c, D, and e matrices. (A CSV file has one line for each row of a matrix with column entries separated by commas). Matlab provides the function `csvread(file)`, which reads a CSV file and returns the corresponding Matrix object.

(d) **[5 pts]** Read in the 3 MAX FLOW instances provided on the course web site, construct their corresponding LPs and solve them using Matlab's LP solver, which you can invoke with `linprog(c, A, b, D, e)`. You can use another solver if you wish, such as the one provided on Geoff's website or the GNU Linear Programming Kit (GLPK) linked from the course website, but be careful to get the input form correct. Record the maximum flow that reaches the sink in each instance.

Note: In most MAX FLOW instances a significant amount of space is wasted by the matrix representation to store 0 entries. We will look favorably on implementations that use Matlab's sparse matrix representation (or any sparse representation of your choice) and use the sparse instance files as input rather than the CSV files. The sparse files contain one row for each edge in the format: "i, j, c_{ij}", where c_{ij} is the capacity on the edge from i to j. You may find the Matlab command `spconvert(M)` useful for this task.