Grad AI. 15-780 Fall, 2006

Homework 2 Solutions

- *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**] Necessary states:

   - have(knowledge)
   - have(plan graph)
   - have(SAT instance)
   - have(solver)
   - have(solution)

   Necessary actions are the following:

   - build(plan graph)
   - build(SAT instance)
   - build(solver)
   - build(solution)

   (b) [**5 pts**] Pre-conditions and post-conditions are the following:

   - build(plan graph)
     - **Pre-conditions**: ¬ have(plan graph), have(knowledge)
     - **Post-conditions**: have(plan graph)
   - build(SAT instance)
     - **Pre-conditions**: ¬ have(SAT instance), have(knowledge), have(plan graph)
     - **Post-conditions**: have(SAT instance)
   - build(solver)
     - **Pre-conditions**: ¬ have(solver)
     - **Post-conditions**: have(solver)

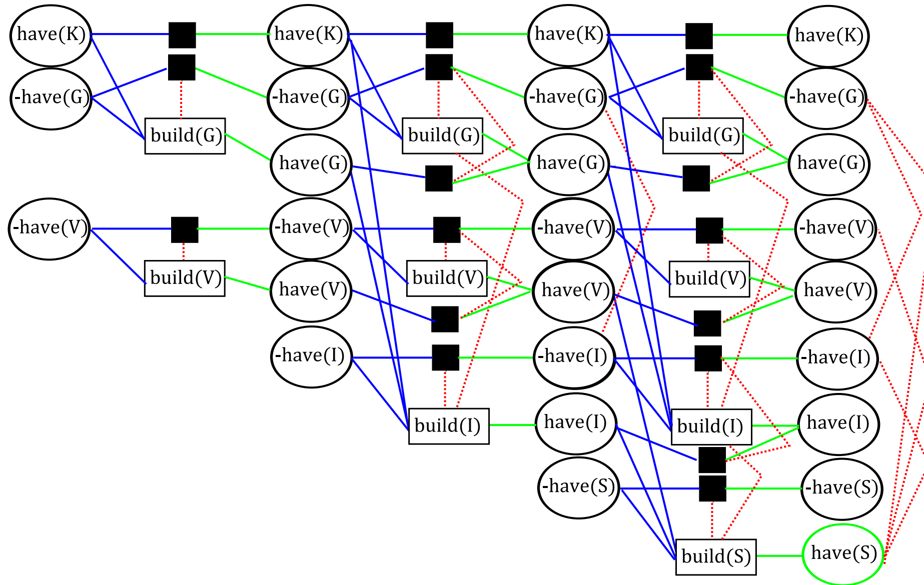

Figure 1: Plan graph solution. Knowledge of planning is abbreviated as "K", plan graph is abbreviated as "G", SAT instance is abbreviated as "I", Solver is abbreviated as "V", and solution is "S". Pre-condition edges are blue, post-condition edges are green, and mutex edges are red.

- build(solution)
  - **Pre-conditions**: ¬ have(solution), have(SAT instance), have(solver)
  - **Post-conditions**: have(solution)

  We include the pre-condition of ¬ have(x) when we build x since we can re-use each of the things we build and would not need to build them twice.

(c) [**5 pts**] The plan graph is shown in Figure 1.

(d) [**8 pts**] Coding problem.

(e) [**5 pts**] The technique used to construct a SAT instance in this problem showed that a graph plan instance can be reudced to an equivalent polynomial sized SAT instance (in polynomial time). We asked you to draw conclusions about the worst-case hardness of the SAT problem or planning problem based on this reduction. Since we have only showed that a SAT solver can solve planning problems, we can only conclude that in the worst case SAT is *at least as hard* as planning. In addition, a SAT instance can easily be checked in polynomial time, therefore a solution to planning can be checked in polynomial time (i.e. both problems are in $\mathcal{NP}$). To understand why this is true consider the following, if we have a solution to a planning instance we can convert it to a SAT instance in polynomial time (using our reduction from this problem) and then check the solution on the SAT instance in polynomial time.

2. **Constraint Satisfaction**. In this problem you will write code to generate and solve random constraint satisfaction problems. For simplicity we will consider only prob-

lems 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]**: Given a binary CSP with variables that have different domain sizes (i.e. there exists at least one pair i and j such that $m_i \neq m_j$) we can reduce it 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 in the following way. First let $m = \max(m_1, \ldots, m_n)$. For each variable i with domain $m_i < m$, pad i's domain with $m - m_i$ dummy values. Next add a constarint between pairs of padded variables (if there are an odd number pair the extra one with any other variable) with 1's for all original values, and 0's for all padded values. This will result in at most $\frac{n}{2}$ additional constraints and an instance that is equivalent to the original CSP.

(b) **[5 pts]**

- **Expected number of constraints:** $\frac{p_1 n(n-1)}{2}$
- **Expected number of inconsistent pairs:** $m^2 p_2$

(c) **[10 pts]** Coding problem.

(d) **[14 pts]** See Figure 2 for an example of what your phase transition graph should look like if you properly extended your DFS to implement CSP backtracking search.

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 3 shows a simple MAX FLOW instance.

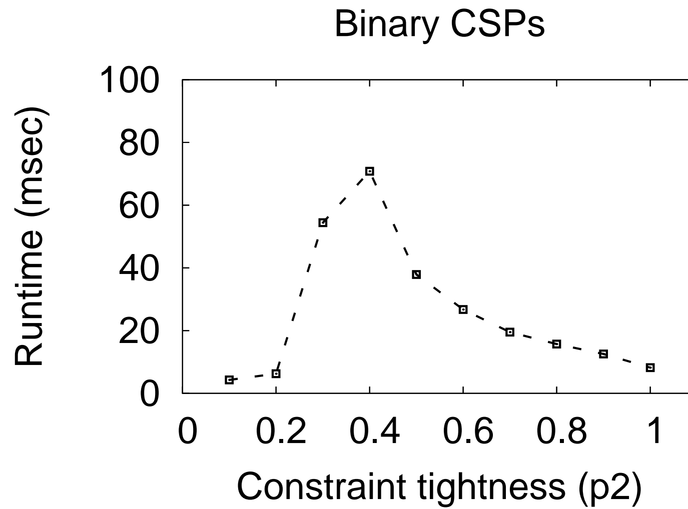

Figure 2: Phase transition for Binary CSPs occurs when $p_2 = 0.4$.

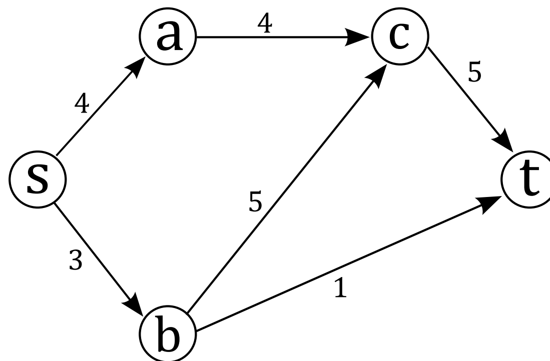

Figure 3: A simple MAX FLOW instance with source s, sink t, and edge weights indicating the capacity of each edge.

(a) **[10 pts]** The matrices describing an LP that solves the MAX FLOW instance in Figure 3 are described below. Each variable indicates the amount of flow placed on an edge. Variables appear in the following order, $s \rightarrow a$, $s \rightarrow b$, $a \rightarrow c$, $b \rightarrow c$, $b \rightarrow t$, $c \rightarrow t$. The A matrix encodes the upper bounds on the flow for each edge, and the equality constraints enforce the flow conservation. The objective function maximizes the flow entering the sink node.

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad b = \begin{pmatrix} 4 \\ 3 \\ 4 \\ 5 \\ 1 \\ 5 \end{pmatrix} \quad c = \begin{pmatrix} 0 & 0 & 0 & 0 & -1 & -1 \end{pmatrix}$$

$$D = \begin{pmatrix} 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & -1 & 0 \\ 0 & 0 & 1 & 1 & 0 & -1 \end{pmatrix} \quad e = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

(b) **[10 pts]** The dual of the MAX FLOW problem is a MIN CUT problem. In the MIN CUT problem the goal is to "cut" a set of edges with the smallest weight that disconnects s and t. The y variables in the dual can be interpreted as whether ($y_i = 1$) or not ($y_i = 0$) each edge appears in the cut found by the LP solver. If $y_i \in (0, 1)$, then some min-cuts include i and others do not. Different solvers will return different answers in this case: some may still give an integral solution, but others may give a fractional solution that corresponds to a weighted average over some arbitrary subset of the set of minimum cuts. There is one z variable for each node other than s and t in the graph and it will be set to 1 if the corresponding node appears in the partition with s and 0 otherwise.

$$\min_{y,z} b^T y + e^T z = \min_{y,z} b^T y$$

subject to:

$$\begin{aligned} A^T y + D^T z &\geq -c \\ y &\geq 0 \\ z & \quad \text{unbounded} \end{aligned}$$

(c) **[15 pts]** Many of you lost a few points because you did not realize that there could be edges leaving the sink and you did not maximizing its net-flow (flow in minus flow out). To understand why this is incorrect consider the following example: some amount of flow enters the sink (it is counted in your objective function because you were maximizing flow in to the sink), it then leaves on one of the sink's outgoing edges (this flow is not deducted from the objective if you did not maximize net-flow), finally the flow returns to the sink through a different edge: this flow has been encouraged to leave the sink only to return and double counted by your objective function! There were two ways to address this problem, you could have either deducted flow leaving the sink from your objective function or ensured with a constraint that flow did not leave the sink. Coding problem.

(d) **[5 pts]**

- **Instance 1**: This was the example instance described above, the max flow is 6.
- **Instance 2**: Max flow is ≈ 1.3 if you maximized net flow (e.g. by subtracting flow that left the sink). The flow was ≈ 6.4 if you maximized only flow going into the sink (this will result in a minor deduction).
- **Instance 3**: Max flow is ≈ 212 if you maximized net flow (e.g. by subtracting flow that left the sink). The flow was ≈ 978 if you maximized only the flow going into the sink (this will result in a minor deduction).