MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Department of Electrical Engineering and Computer Science

6.034 Artificial Intelligence, Fall 2003

**Recitation 5: Games & CSPs ("Can't get no satisfaction"),** October 2nd -3rd

The design and solution of constraint satisfaction problems (CSP): a CSP consists of 3 components: (1) a (finite or infinite) set of **variables** $V_1,\ldots,V_n$ with (2) **domains** $D_i$ of possible values. And (3) corresponding **constraints** $C_i$. A <u>**solution**</u> to a CSP is a complete assignment to all variables that satisfies the constraints (recall simultaneous eqns – they satisfy this dfn). The **representational** part of a CSP is figuring out how to map a problem into variables, domains (with values), and constraints – usually the trickiest part. We visualize the CSP as a **constraint graph** with nodes representing the variables, and the links as the constraints.

- The key goal of CSP algorithms is to **reduce the branching factor** to cut down search by eliminating combinations of variable values that can never participate in a solution. The whole idea is to **push for early failure** (ie, higher in the tree).
- Thus, constraint satisfaction problems can **take advantage of the problem structure** better than plain search to eliminate a large fraction of the search space, in the best case. (In the very best case: no search remains at all – just singleton values for each variable. If multiple values – still search left to do.)
- The principal source of structure in the problem space is that **the goal test is decomposed into a set of constraints on variables rather than being a unitary, black-box, 'goal'.** This is what allows us to do searching incrementally, and solve *part* of the problem first, like a jig-saw puzzle, before solving the *rest*. (In the Waltz world, we can label a pair of junctions perhaps uniquely before moving on – divide and conquer works.)

There are 3 main techniques we'll use to solve CSP problems (and you've seen the first), each using an increasing amount of 'propagation' of constraint. (Winston demo descriptions in brackets.) The key questions one asks to get different methods here are (1) which variable should be assigned next and in what order should the values be tried? (2) what are the implications of the current var assignments for other, unassigned variables? And (3) when a path fails, can the search avoid repeating this failure? In addition, we can divide up these methods **into two sorts**: arc-consistency, which only <u>reduces</u> the # of values that could be in a solution; and the other two, backtracking and forward-checking with backtracking, that can actually <u>find</u> solutions. So, arc-consistency, or "Waltz labeling" really is meant to reduce search. We still must do some search in the end (if only, as in the blocks world case, to easily find a single solution).

1. <u>**Backtracking**</u> (BT). No constraint checks are propagated ["Assignments only"]. One variable assigned a single value at a time, and then the next variable, etc.
2. <u>**Forward-checking**</u> with backtracking (BT-FC). A variable is assigned a unique value, and then its neighbors checked to see if their values are consistent with this assignment. [" Check neighbors only"].
3. <u>**Arc-consistency**</u> (AKA "Waltz labeling"). Constraints are propagated through all **reduced domains** of variable values, until closure. ["Propagate through reduced domains"]. (Variant: propagate if variable domain is reduced to one.) This gets rid of more inconsistencies than FC, but often doesn't settle on single assignments, because n arc from X to Y in the constraint graph is consistent if, for <u>*every*</u> value of X, there is <u>*some*</u> value of Y that is consistent with X.– it could be a <u>different</u> y for each choice of x.

We'll go through these three methods each in turn with the same example problem. We shall see that CSP search complexity may be affected by the following, often by nearly an order of magnitude:

- The order in which variables are assigned values (e.g., least restricted variable first; most-restricted first – think about the blocks labeling problem – (Waltz) – what is the trick that lets it work so well as a pre-filter, arriving at unambiguous solutions?
- The domain values chosen for assignment (e.g., the least-constraining value)

First, though, we'll take a brief look at the **hard part** of CSP (as usual!), namely, finding the right representation to map from the problem to the rows, domains/values, and constraints.

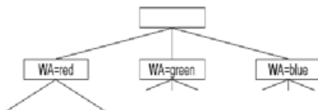
To give you some idea of the range of problems one can encode with CSP, a single 183-vertex graph (taken from the game *Diplomacy*) has been used here all for the following problems. The problems have in common

that they can be stated with variables that are a priori constrained to finite sets of nonnegative integers. Consequently, the problems could be solved by simply checking all possible combinations of the values of the variables. This naive generate and test method is infeasible for most realistic problems since there are just too many possible combinations. (For infinite domains, typically scheduling problems, we definitely need some additional constraint!)

1. DNA Sequencing: Graph vertices correspond to DNA fragments (GACGGAGTTGGACGGTA ...); find most likely DNA string: Some enzymes cut a DNA string into many fragments. If one knows the sequence of the fragments, one can guess the AGCT sequence of the whole string by sequencing the fragments to get as much overlap as possible.
2. Coloring: find four-coloring of the graph
3. Layout: graph vertices represents rectangular windows; find window configuration on the computer screen
4. Scheduling: Graph vertices represent sessions, the adjacent sessions must not be scheduled at the same time; find the shortest schedule
5. Control (N-queens problem): place N queens on NxN chessboard s.t. (at vertex) they control all vertices within the distance N;
    1. how few N-queens can be placed on Graph so all vertices are controlled
    2. how many independent N-queens can be placed on G
6. Spatio-temporal relations: the vertices represent regions in 2D space, for each vertex, the relative position to all other vertices is given; find the feasible solution, i.e., placing for the 182 regions
7. Warehouse: vertices represent regions in the world; find the best location of the manufacturer's warehouse

We give yet more examples at the end and in the text. OK, let's start with a 'canonical' problem, map coloring, with the following map, and backtracking search.

**Method 1.** Simple backtracking 'solution' to CSP map problem:




What's the problem with this method?

**Method 2.** Let's now try the same thing via forward checking (with Backtracking) i.e., (BT-FC). The idea is simple: before assigning values to successor nodes from domains with unique values, check to see which ar ecompatible with the current value. We'll start off a table representation of this for you to fill out, as before. Let's see how much work this saves.

| | WA | NT | Q | NSW | V | SA | T |
|---|---|---|---|---|---|---|---|
| Initial domains | R G B | R G B | R G B | R G B | R G B | R G B | R G B |
| After WA=red | (R) | G B | R G B | R G B | R G B | G B | R G B |
| After Q=green | (R) | B | (G) | R B | R G B | B | R G B |
| After V=blue | (R) | B | (G) | R | (B) | | R G B |

**Note** that the domain for SA becomes NULL – this is always the sign that we have no valid solution yet, so we would have to now BACKTRACK to a previous color assignment, and try another. In this case – we assigned blue to V, so we can go back to that and try, say, green. That would be **chronological backtracking** – choose the last decision point to go back to. Usually, there are smarter ways to backtrack – the online tutor discusses some of them.

**3. Method 3:** arc consistency (AC). Now we propagate constraints not to just neighbors, but as far as we can go, and not just when the value at a node is reduced to one, but if it is **reduced at all.** Thus AC…
•Can detect more inconsistencies than forward checking.
• Can be applied as a preprocessing step before search or as a propagation step after each assignment during search.
• Process must be applied repeatedly until no more inconsistencies remain. Why?
But…because AC does not wait until a unique value is found at a node, it can result in no solutions or ambiguous solutions. We'll see this in the map example.

All these methods are sensitive to the order in which variables are selected for checking (when we have a choice). Consider what happens if at a node we pick the most constrained variable first (This is why the Waltz method works – why?). There are lots of tricks here, some of which are explored in the online tutor. Similarly, there are various backtracking techniques that can help.

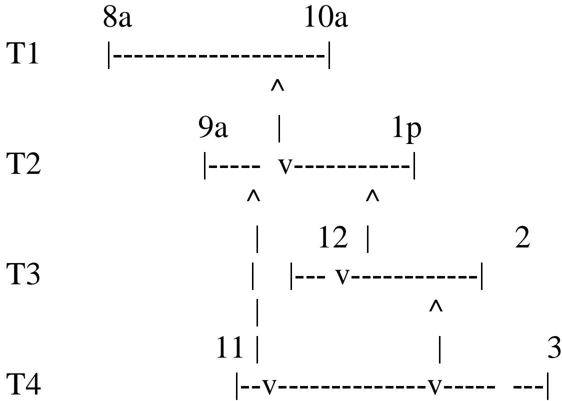
```
function AC3(csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ... Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   loop while queue is not empty do
      (Xi, Xj) ← REMOVE-FRONT(queue)
      if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue
end
--------------------------------------------------------------------
function REMOVE-INCONSISTENT-VALUES(Xi, Xj) returns true iff we remove a value
   removed ← false
   loop for each x in DOMAIN[Xi] do
      if (x,y) satisfies the constraint for some value y in DOMAIN[Xj]
         then delete x from DOMAIN[Xi]; removed ← true
   end
   return removed
end
```

As a final example to practice converting a problem to a CSP, we'll consider train scheduling.
.
There are 4 trains (T1 T2 T3 T4) and three locomotives.(L1 L2 L3). What assignments of locomotives to trains will satisfy the following schedule:

| Train | in use |
|---|---|
| T1 | 8am to 10am |
| T2 | 9am to 1pm |
| T3 | noon to 2pm |
| T4 | 11am to 3pm |

Make the following assumptions:
Each locomotive can only pull one train at a time.
Each locomotive has time to get to the station where its next train is located.
L3 is too small to pull T3.
L2 and L3 are too small to pull T4.

Draw timelines representing when each train is in use, and show constraints by drawing arrows between timelines representing trains that cannot use the same locomotive. (These are the constraints.)

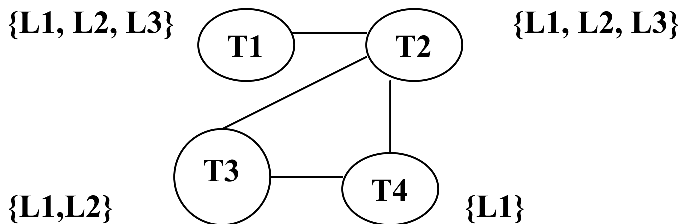



Lines between the time lines represent the **constraint** that a single locomotive cannot pull more than 1 train at a time. (There are other constraints that we'll have to encode too.)

What are the variables? – think about what objects there are in this world Answer: the **trains**

What are the values? Answer: the **Locomotives**

What are the possible values for each variable? (this is where we put in the **other** constraints mentioned in the text): {L1, L2, L3} can be assigned to T1 and T2. {L1, L2, L3} can be assigned to T3. T4 can only have L1 assigned. So, this gives us our CSP graph: note there are three elts, the nodes (vertices) – the variables; the links, the constraints; and the labels – the variable values, the set of these at each node, the domain of the variable at each node.
Draw the constraint graph for the first choice of variables and values: we connect T1, T2, T3, T4 with a line iff there is a line between the two trains in the time constraint graph:




Now we can run AC (arc consistency), or BT-FC on this graph. Let's use AC. A good place to start is with the most constrained node, T4 – it has a single value, so this is a lot like the Waltz labeling situation where the boundary edges provide a lot of constraint. If we propagate using AC, we put the neighbors of T4, T2 and T3, on the queue. We now look at T4-T2. The constraint eliminates L1 from the domain of T2. So, using the AC algorithm above, we put the node T2 on the queue, and look at **its** links, T1-T3 and T2-T3 (note that we haven't specified this order – we are going depth-first, but the algorithm above doesn't say **where** new nodes go in the queue, so we could use any method – BFS, for example, or even A*).
Continuing with a DFS expansion, we consider T1-T3 and see if this domain can be reduced. Nope.

So we next examine T2-T3. Since T2's domain is now only L2, L3, this eliminates L1 from T3.
Since T3's domain is reduced, now just L2, we put it on the queue to examine.
T3 links to T4 and T2. There is no further reduction in T3-T2, but of course, the constraint on T4 reduces T3's domain now to L2. So, we put T3 on the queue **again,** and examine its links: T3-T2 and T3-T4. The new domain on T3, just L2, that reduces the domain of T2 to just L3. So, we put T2 on the queue **again.** Finally, we consider T2-T1 and T2-T4. The first link now can reduce the domain of T1 to just L1, L2. The second causes no new reduction. We must put T1 on the queue, to round things off, but this prompts no more domain reductions, and AC finishes. **Note** that there are two solutions: T1 still has L1, L2 as possibilities. This is quite typical.

**Other examples of CSP situations**

**1. Safe** The code of Professor Smart's safe is a sequence of 9 distinct nonzero digits $C_1, \dots, C_9$ such that the following equations and inequations are satisfied. Can you determine the code?

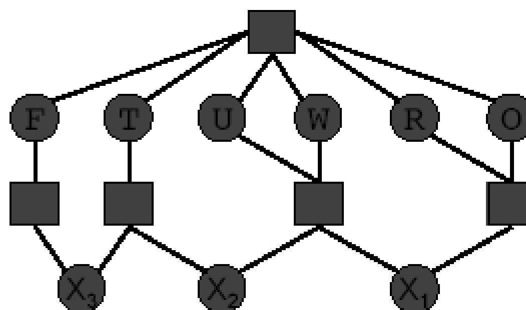
$$C_4 - C_6 = C_7$$
$$C_1 * C_2 * C_3 = C_8 + C_9$$
$$C_2 + C_3 + C_6 < C_8$$
$$C_9 < C_8$$
$$C_1 \neq 1, \dots, C_9 \neq 9$$

**2.** Grocery. A kid goes into a grocery store and buys four items. The cashier charges $7.11, the kid pays and is about to leave when the cashier calls the kid back, and says ``Hold on, I multiplied the four items instead of adding them; I'll try again; Hah, with adding them the price still comes to $7.11". What were the prices of the four items?

Here is the solution to the cryptarithmetic problem in the problem handout…at least, one way of encoding the variables via the graph shown in the problem sheet.

```
  TWO
+ TWO
-----
 FOUR
```

- $O + O = R + 10 \cdot X_1$
- $X_1 + W + W = U + 10 \cdot X_2$
- $X_2 + T + T = O + 10 \cdot X_3$
- $X_3 = F$
- *alldiff*(F,T,U,W,R,O)




Here is the solution to the two 'zany' waltz diagrams. **NOTE** that I failed to copy over a crucial line on the diagram that was in initial handout on the lefthand side – the straight line showing that it is like an empty box.

In any case – the interesting point is that the lefthand diagram CANNOT be labeled with the original Waltz labels – the failure is at the intersection of that straight 'inner' box line with the front top edge of the box. The second figure can be labeled. The point is that the Waltz labels make the IMPLICIT assumption that no edges have infinite thinness. To correctly label the diagram, you'd have to modify the primitive vocabulary. Finally, the Waltz system used just arc-consistency yet seemed to give unique answers almost all the time – no search needed at all. We have seen that AC doesn't usually give answers – it just cuts down the AMOUNT of search. So why did the Waltz system work? The 'trick' is in using the boundary edges – the figure is imagined to be suspended in mid-air, and the boundary edges can be always drawn first. That gives enough constraint to reduce many vertex labels to just 1.