Principles of Artificial Intelligence
Fall 2005
Handout #4
Constraint Satisfaction

Vasant Honavar
Artificial Intelligence Research Laboratory
Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011

1 Constraint Satisfaction Problems

Constraint satisfaction problems (CSP) are a subclass of state space search problems. Some examples of CSPs include:

- N-queens puzzle
- Job shop scheduling
- Scene labeling
- Circuit board layout

A constraint is simply a (logical) relation among several variables, each of which is allowed to take a value in a given domain. A constraint therefore restricts the possible values that variables can take. It thus represents some partial information about the variables of interest. For instance, in the cryptarithmetic puzzle, $SEND + MORE = MONEY$, the variables are letters of the English alphabet. Each variable is allowed to take integer values from 0 through 9, and the constraints are imposed by the laws of base-10 arithmetic.
Some properties of constraints are worth noting: Constraints may specify partial information, i.e., constraint need not uniquely specify the values of its variables. Constraints are declarative, i.e., they specify what relationship must hold without specifying *how* such a relationship is to be enforced. The effect of constraints are additive and independent of the order in which the constraints are enforced. Constraints are rarely independent, i.e., each constraint involves several variable, and each variable is subject to several constraints.

1.1 Formal Definition of CSP

The **domain** D_x of a variable x is the set of possible values that can be assigned to x. A **label** is a variable-value pair (x,v) which represents the assignment of value $v \in D_x$ to variable x. A k**-compound** label assigns values to variables: $(x_1, v_1)(x_2, v_2) \cdots (x_k, v_k)$. A **solution** to a CSP with N variables is an N-compound label satisfying all the problem constraints.
A **CSP** is specified by the 3-tuple (Z, C, D), where

- Z = set of variables.
- C = set of constraints.
- $D = \{D_x | x \in Z\}$.

Thus, constraint satisfaction problems can be viewed as state space search problems. It is possible to solves such problems or find that there is no solution using an algorithm that exhaustively generates all possible assignments to variables and tests whether the given assignment satisfies all constraints. However, this procedure is computationally not feasible except in the most trivial cases. Fortunately, however, many CSP share some special properties which lend themselves to (at least in practice) more efficient solution strategies.

- The search space is finite.
- The depth of the solution is typically fixed by the number of variables.
- Subtrees have similar topologies, so experience obtained in searching one subtree may be exploited when others are searched (e.g., if we find one pair of values incompatible for a given pair of variables, this information can help rule out all assignments that contain the incompatible assignment).
- The search space has minimal size under certain conditions (e.g., the size of the search space is minimized by ordering the variables from the most constrained to the least constrained).

2 Examples of CSP

In this section we will examine the 8-queens problem and the scene labeling problem in greater detail to illustrate how the general definition we have just given for CSPs can be applied to specific problems.

2.1 The 8-Queens Problem

This problem, first posed in a German chess magazine almost 150 years ago, requires that 8 queens be placed on an 8×8 chessboard in such a way that no two queens attack each other, i.e. no two queens occupy the same row, column, or diagonal.
This problem can be formally represented as a CSP in several different ways, and as we shall see, the choice of formal representation has a significant impact on the computational complexity of the problem.
Suppose, for example, that we choose to represent the problem by numbering the queens 1 through 8 and then defining a set Z of 64 variables, one for each square of the chessboard, where each variable has domain $D = \{0, 1, 2, \cdots, 8\}$ and a variable z having a value of i means that queen i occupies the corresponding square if $1 \leq i \leq 8$ or that the square is unoccupied if $i = 0$. This formulation of the problem yields a search space of cardinality $9^{64} \approx 1.18 \times 10^{61}$, making it impractical from a computational perspective.
Consider instead a representation with a set Z of 8 variables $Q_1, Q_2, \cdots Q_8$ representing the 8 rows of the board, with the domain of each variable Q_i being the set $D_i = \{1, 2, \cdots, 8\}$; in this case the value of the variable Q_i represents the column in the i^{th} row in which a queen is placed. The constraints can then be specified as logical expressions of the form

$$i \neq j \Rightarrow Q_i \neq Q_j$$

(i.e., no two queens are in the same column) and

$$i \neq j \Rightarrow |Q_i - Q_j| \neq |i - j|$$

(i.e., no two queens are in the same diagonal). Note that we have implicitly incorporated one of the problem constraints (that no two queens occupy the same row) as well as our reasoning about the consequences of this constraint (it implies that each row must have precisely one queen in it) into our representation. Note too that the cardinality of the search space is now reduced to $8^8 \approx 1.68 \times 10^7$ states.
This example clearly illustrates the need to carefully select a representation for a CSP from among the many formally valid possibilities, bearing in mind the computational complexity which each one entails.

2.2 The Scene Labeling Problem

This is a subproblem of the **vision problem**, which seeks to deduce facts about the physical world from visual representations of it. In the scene labeling problem we are given a 2-D line drawing of a 3-D scene and must label the components of the drawing in such a way as to make explicit the 3-D structure the drawing represents.
This problem has been studied extensively since the early 1970s; we will consider a simplified version which was studied by David Waltz as part of his Ph.D. thesis at M.I.T. during this period. The assumptions we make to achieve this simplification are:

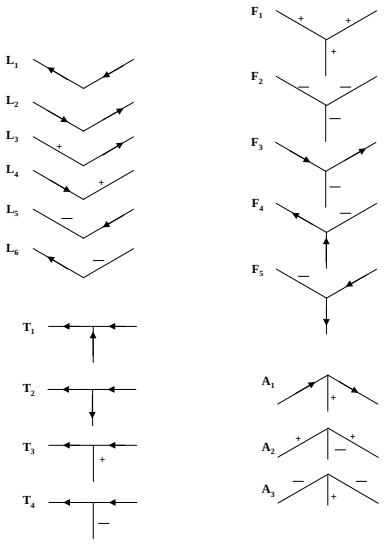
1. Every vertex is **trihedral** (formed by the junction of three planar surfaces; implicitly, we assume that all surfaces are planar).
2. Surfaces do not contain 0-width **cracks** (lines formed by the junction of coplanar surfaces).
3. **No shadows** (lines which represent projections of edges onto surfaces along the direction of light sources) are allowed.
4. The scene is viewed from a **general viewing position** (roughly, for some sufficiently small $\epsilon > 0$ we can move our viewing point in an arbitrary direction by a distance of ϵ without any topologically significant change in the resulting 2-D representation).

It actually turns out, as was revealed by later work on the problem, that assumption 3 is not really a simplification, since the information which can be deduced from the presence and positions of shadows in some sense outweighs the potential for confusion with actual edges they bring with them, but we will follow Waltz' path here and let the assumption stand.
Although a detailed analysis of the problem is beyond the scope of this lecture, one of the fundamental results of this analysis is that a relatively simple taxonomy of 4 classes of vertices and 4 classes of lines suffices to cover all physical possibilities, modulo the assumptions we have made. While we will not give formal definitions of the classes here, they may be intuitively described as follows:

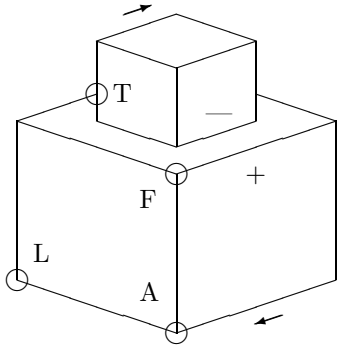
1. **L-Junction Vertices**: vertices at which only 2 visible lines meet.
2. **T-Junction Vertices**: vertices at which one line segment ends in the middle of another.
3. **Fork Vertices**: vertices at which three lines meet, with no angles greater than or equal to 180° between any pair of them.
4. **Arrow Vertices**: vertices at which three lines meet, with an angle greater than 180° between one pair of them.

1. **Convex Lines**: lines formed by the junction of two visible surfaces for which a line segment connecting the two surfaces would be inside the solids they bound.
2. **Concave Lines**: lines formed by the junction of two visible surfaces for which a line segment connecting the two surfaces would be outside the solids they bound.
3. **Right/Up Arrow Lines**: lines formed by the junction of one visible and one obscured surface, with the visible surface lying below/to the right of the line.
4. **Left/Down Arrow Lines**: lines formed by the junction of one visible and one obscured surface, with the visible surface lying above/to the left of the line.

Depending on the viewing location, these 4 classes of vertices and 4 classes of lines can be combined to give the eighteen types of vertices shown in the following figure.



Representative examples of each of these classes may be seen in the following drawing:



L (L-Junction Vertex)

T (T-Junction Vertex)

F (Fork Vertex)

A (Arrow Vertex)

\+ (Convex Line)

— (Concave Line)

↗ (Right/Up Arrow Line)

↙ (Left/Down Arrow Line)

As we shall see, the classification of vertices and lines in this way allows us to pose the scene labeling problem as a CSP, with vertices as variables to be labeled with values from the list of classes above; since each class of vertex can only have certain classes of lines meet to form it, the restrictions imposed by the classification of the two vertices at the endpoints of each line must be compatible, which then provides us with the constraints.
The general algorithm for scene-labeling is as follows.

1. Identify the boundary edges (optional but desirable).
2. Number the vertices (optionally, start with a boundary vertex) in some order.
3. Visit a vertex v (in the order of its numbering) and attempt to label it:
   (a) Attach to v all vertex labels compatible with its type.
   (b) Eliminate any labels that are not consistent with local constraints.

(c) For each vertex A that was visited in (b),

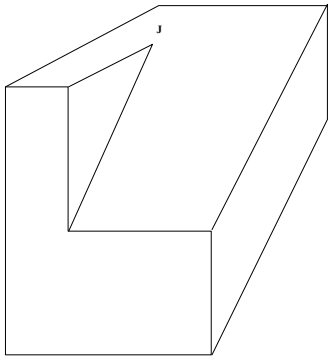
i. Eliminate every label of A that has no consistent label in v

ii. If any label of A was eliminated, continue propagating the constraints until no more labels are eliminated.

There are three possible outcomes of this algorithm:

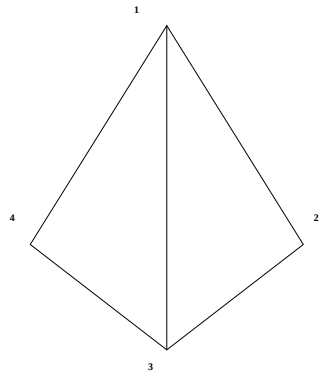
1. Every line has only one label left,
2. One or more lines have multiple labels left, or
3. One or more lines have no labels left.

If we are left with multiple labels on some lines, the scene can be interpreted in more than one way. An exhaustive search through the remaining possible combinations will identify the valid interpretations. This takes considerably less effort than an exhaustive search the entire search space. On the other hand, if some lines do not have any labels left, the scene violates one or more of the assumptions made earlier.
For instance, vertex J in the above figure violates the assumption that all vertices are trihedral. The algorithm will end with the lines at vertex J having no labels.

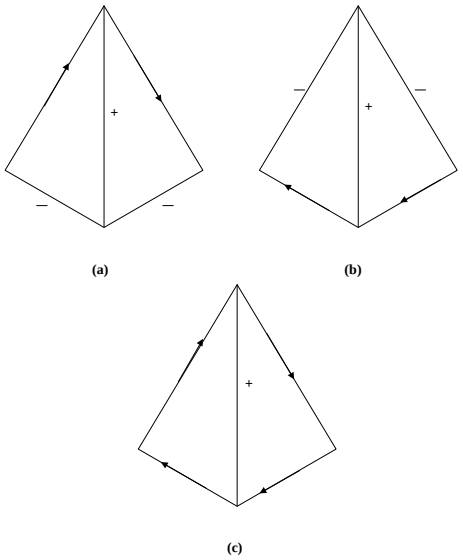


2.2.1 Example

As an example of the algorithm for scene-labeling, consider the scene shown in the following figure.



Vertex 1 is an example of arrow vertices and as such, has three possible labels: $\{A_1, A_2, A_3\}$. Thus, line (1,2) has three possible labels: $\{+, -, \downarrow\}$. These constrain the number of possible interpretations that vertex 2 (an L-vertex) can have. Specifically, vertex 2 can have only the following four labels: $\{L_1, L_4, L_5, L_6\}$. Call this constraint (i). Vertex 3 can also have three possible labels: $\{A_1, A_2, A_3\}$. These allow line (2,3) to have the the following labels: $\{\uparrow, \downarrow, -\}$. These act as an additional constraint on vertex 2 and together with comstraint (i), enable us to eliminate L_4 as a viable option. Propagating this back to vertex 1, we can then eliminate A_2 as one of the labels on Vertex 1. This process can be repeated till all possible constraints have been satisfied. We will be left with three interpretations of the scene. These are shown in the following figure.



2.3 Other Applications

Beginning with the pioneering work of Mackworth, Waltz, and others, in the 1970s, constraint satisfaction problems have been studied extensively in AI. They have found applications in diverse problem domains including temporal reasoning (pioneered by James Allen), scheduling, planning, natural language processing, computational biology, among others. This has led to the development of a variety of languages and systems for specification and solution of CSP.

3 Randomized Search Algorithms

In some problems, the state description contains all the information required for a solution. In these cases, the path taken to reach the solution is irrelavent and randomized search algorithms can be used to find the solution. Random search algorithms explore the state space sequentially in a seemingly random fashion to the optimal point. The optimal point in the state space maximizes (or minimizes) an evaluation function. Several variations on this basic scheme are possible. These include simultaneously searching from several points (as opposed to one start point), hill-climbing and simulated annealing.
Simulated annealing was derived from physical characteristics of spin glasses. The principle is analogous to what happens when metals are cooled at a controlled rate. The slowly falling temperature allows the atoms in the molten metal to form a regular crystalline structure that has high density and low energy. But if the temperature goes down too quickly, the atoms do not have time to orient themselves into a regular structure and the result is an amorphous material with higher energy. It can be proved that if the temperature is lowered sufficiently slowly, the material will attain a lowest-energy (perfectly ordered) configuration.

Definition (Annealing): Annealing is defined as the process of cooling molten metal or glass from a high temperature to a low temperature slowly so that the resulting material ends up in a stable energy state.
Definition (Cooling Schedule): The cooling schedule or annealing schedule specifies how rapidly the temperature is lowered from high to low values.
In simulated annealing, we define a suitable representation that allows us to encode candidate solutions to the problem and a quality measure. The quality measure is analogous to the heuristic function and lets us measure the quality of a candidate solution. We also define operators that correspond to state transitions. The algorithm is based on the Metropolis algorithm which is shown below.

3.1 Metropolis Algorithm

Assume that, for simplicity, the states and candidate solutions are represented by binary vectors of a fixed length. Also assume that state transitions correspond to flipping individual bits. The Metropolis algorithm is as follows.

1. Start with a random state x drawn from the solution space.
2. Pick a bit at random and tentatively flip it to obtain y. Compute the difference in quality of solution y relative to the previous state x. Let $\Delta h = h(y) - h(x)$.
3. If $\Delta h \leq 0$, replace x by y and y becomes the new candidate solution. Else, accept state change with probability proportional to $\exp(-\frac{\Delta h}{T})$ where T is the current temperature.

Simulated annealing essentially adds an outer loop around the Metropolis algorithm so that it executes for different values of T, say from T_{max} to T_{min}.

3.2 Simulated Annealing

The complete algorithm for simulated annealing is shown below.

1. Initialize the temperature T to T_{max}. Start with a random state x.
2. Select a random successor of the current state. Call it y.
3. Compute the difference in quality of solution y relative to that of x. Call this difference Δh.
4. If $\Delta h \leq 0$, replace x by y. y becomes the new solution. Else, accept this state change with probability proportional to $\exp(-\frac{\Delta h}{T})$.
5. Set $T = T_{next}$. If $T = T_{min}$, Stop else go to step 2.

3.3 Optimality of Simulated Annealing

Simulated annealing is optimal (i.e., it finds an optimal solution if one exists) only when the cooling schedule is exponentially slow.
Theorem If the temperature T_k used in the k^{th} iteration of simulated annealing satisfies the condition

$$T_k \geq \frac{T_0}{\ln(1+k)} \quad \forall k \tag{1}$$

where T_0 is a sufficiently large initial temperature, then the probability that simulated annealing finds an optimal solution approaches one asymptotically.

Note:

1. This condition is sufficient but not necessary.
2. The theorem states that at the k^{th} iteration, $1 + k \geq \exp(\frac{T_0}{T_k})$. This is an exponentially slow cooling schedule and usually will not terminate in a reasonable amount of time. In practice, $T_k = \alpha T_{k-1}$ with $\alpha < 1$ is used.