6

ARTIFICIAL INTELLIGENCE COM S 472/572 LECTURE NOTES WEEK 6

Author: Vasant Honavar
Scribe(s): Jeff Yakey, Xin Yu, Hao Wang

Computer Science Department,
Iowa State University, Ames, IA 50011

1 SCENE LABELING PROBLEM (CONTINUED)

In the previous week's notes, the taxonomy of four classes of vertices and four classes of lines was presented. From this taxonomy, we can now calculate the possible ways in which a junction may be labeled. An **L** may be labeled in 4^2 ways. A **Fork, Arrow or T** may be labeled in 4^3 ways. This gives a total of 208 possibilities.

Out of these combinations, only 18 are physically realizable. These are shown in the figure 1. Thus, a problem which appeared to have an exponential number of possible labelings is reduced to one with a constant number by the use of constraints.

The reduction in the number of options shown above was done intuitively. This reduction then allows the Scene Labeling Problem to be specified as a constraint satisfaction problem. So the next question that we can ask is whether there exists an algorithm that can perform this job.

Waltz's Scene Labeling Algorithm is given in figure 2:

Example

The line object to be labeled and interpreted is given in the figure 3. Assume that the first two steps of the above algorithm have been carried out (however,

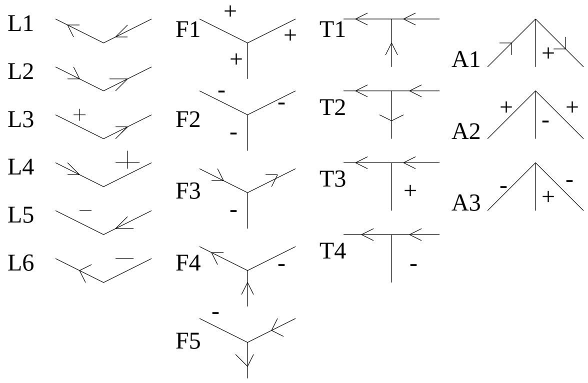


Figure 1 Valid Labels for the Junctions

in this case, without identifying the boundary lines just to make the labeling task more interesting).

The third step in the algorithm is enumerated below.

Start with vertex 1. This is an arrow. The possible labels are (A1,A2,A3) from figure 1. Thus, the possible labels for the edges are:
<1,2> = (D,+,-)
<1,3> = (+,-)
<1,4> = (U,+,-)
where <i,j> denotes the arc connecting vertices i and j in figure 3 and D and U denote the up,Right and down,Left arrows.

Go on to vertex 2. This is an L. The possible labels are (L1,L2,L3,L4,L5,L6). (as identified in figure 1) Out of these (L2,L3) are eliminated because of vertex 1. Backtrack to vertex 1. No change is required to vertex 1, as no additional restrictions are added by vertex 2. Thus the possible labels for edges are:
<1,2> = (D,+,-)
<1,3> = (+,-)
<1,4> = (U,+,-)
<2,3> = (U,D,-)

Go on to vertex 3. This is an arrow. The possible labels are (A1,A2,A3). Going back to vertex 2, we note that U is no longer valid for arc <2,3>. Hence it is eliminated. Since, vertex 2 has changed, we go back to vertex 1 and note that

```
1.  Identify the boundary edges (optional, but desirable, to
    reduce future effort).
2.  Number the vertices.
    Start from a boundary vertex (optional, but desirable) and
    number the other boundary vertices in clockwise direction.
    Number the interior vertices starting from a numbered
    neighbor.
3.  Visit a vertex  V (in the order of its numbering) and
   attempt to label it:
    a.  Attach to  V a set of all candidate labels
        compatible with its type.
    b.  Eliminate any label not consistent with local
        constraints.
    c.  For each vertex  A affected by (b):
       1.  Eliminate every label of  A that has no
           consistent label in  V.
       2.  If any label of  A was eliminated, continue
           propagating constraints to  A's labeled neighbors
   until no more labels are eliminated.
```

Figure 2 Waltz's Labeling Algorithm

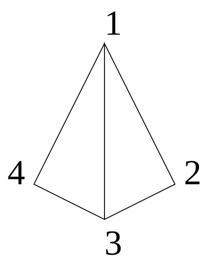


Figure 3 Example Line drawing: Vertices numbered

(+) is no longer valid on the arc <1,2>. Hence it is removed. But now that (+) is removed from <1,2>, arc <1,4> becomes (U,-) and arc <1,3> becomes

(+) as the (-) is removed.

Go on to vertex 4. It is an L. The possible labels are (L1,L2,L3,L4,L5,L6). Hence arc (4,3) is (U,-) No backtracking is required in this case.

The final set of labels possible for the arcs are:

<1,2> = (D,-)
<1,3> = (+)
<1,4> = (U,-)
<2,3> = (-,D)
<3,4> = (-,U)

This gives rise to three possibilities, which are shown in the figure below. To resolve amibuity of interpretation, additional information is needed. Suppose we knew that the vertices 1, 2, 3, and 4 were boundary vertices, then we have a single unique interpretation (the third possibility in figure 4). The other two interpretations are also possible and physically realizable.

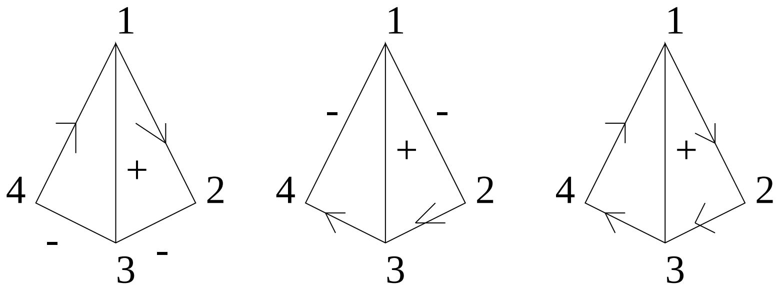


Figure 4 The three possible alternatives

Performance of this algorithm

It can be shown rigorously that this algorithm takes O(N^3) where N is the number of vertices.

On a concluding note we may add that this algorithm has been extended to other vertex types and non-planar objects.

In general these algorithms are called **arc consistency problems** and form a subset of problems called **path consistency problems**. Most such problems have efficient polynomial time algorithms.

2 RANDOMIZED(STOCHASTIC) SEARCH ALGORITHMS

Randomized search algorithms have generated a lot of interest in the last 5-6 years.

Exhaustive search is a practical method for search if the solution is uniformally distributed in the search space. However, if the search space is large then it is not possible to have exhaustive search. If good heuristics are unavailable, systematic heuristic search is not feasible. Random search, which involves taking a small sample of the search space and examining it for a solution is feasible only in problems in which the search space is very densely populated by candidate solution points.

A compromise to both of these is Directed Randomized Search algorithms. The basic idea is to sample a part of the search space, get some information from it about the space being sampled, and use the information to guide further sampling.

2.1 Simulated Annealing

Simulated annealing is an example of randomized search algorithms. It is non-deterministic and primarily for application to NP-hard (exponential) problems. A reference is given to Kirkpatrick et al. 1983.

The simulated annealing algorithm draws from the analogy of cooling a molten material slowly, so that the material reaches a state of minimal energy. A pair of simple examples showing $h(x)$ as it goes from high-energy states to low-energy states follows; note that the second example has "local minimums"where a search could think it had found a minimum value of the $h(x)$ function:

Simulated annealing works well when we only need to find a goal with minimal $h(n)$ and don't care about the path from the start to the goal node.

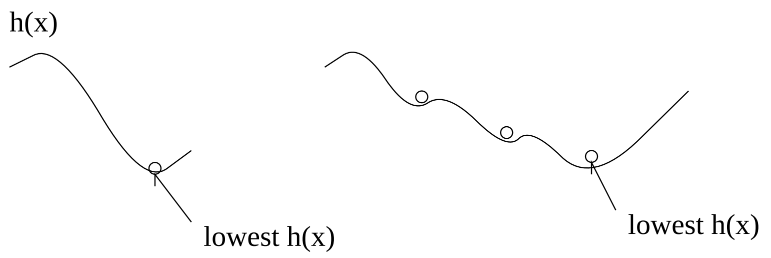


Figure 5 Two Examples of $h(x)$

Definition 1 *Annealing refers to 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.*

Algorithm We define a suitable representation of "states" and a quality meaure h which measures the quality of x as a solution to the problem being solved.

For simplicity, let us assume states are represented using binary vectors of some fixed length.

The **metropolis** algorithm:

1. We start with a random state x (a candidate solution) in the search space.
2. Assuming that the candidate solutions are binary vectors, pick a bit at random and tentatively flip it.
   Compute the quality of the resulting solution (y): let $\Delta h = h(y) - h(x)$.
3. If $\Delta h \leq 0$, replace x by y (so y becomes a candidate solution).
   If $\Delta h > 0$, we accept the state change with the probability that is proportional to $e^{\frac{-\Delta h}{T}}$ where T is "temperature" ($0 \leq e^{\frac{-\Delta h}{T}}$).

Simulated annealing(due to Kirkpatrick, Gelatt, and Vecchi) essentially adds an outer loop to the metropolis algorithm which entails starting with some $T = T_{max}$ and reducing T gradually according to some suitable cooling schedule.

Theorem 1 *If the temperature T_k that is used in the k^{th} iteration of simmulated annealing satisfies the condition $T_k \geq \frac{T_0}{\ln(1+k)}$ for every k, where T_0 is a* *"sufficiently large" initial temperature, then the probability that the algorithm finds a solution corresponds to optimal value of k approaches 1 asymptotically. (This is sufficient, but not necessary.)*

What does this imply in terms of the cooling schedule? Solve $T_k \geq \frac{T_0}{\ln(1+k)}$ for k:
$\ln(1+k) \geq \frac{T_0}{T_k}$
$1+k \geq e^{\frac{T_0}{T_k}}$

So the number of iterations goes exponential if we lower probability of randomly replacing a node.

Note that k is the iteration number, so this implies an exponentially slow cooling schedule (impractical using digital computers!). This is not surprising, since if it were otherwise, we may have suspected P = NP!

In practice, we use $T_k = \alpha T_{k-1}$ with $\alpha \leq 1$, say 0.9, to give a linear cooling schedule, which yields "near-optimal" solutions "most" of the time.

The choice of the probability of state transition $\propto e^{\frac{-\triangle E}{T}}$ ensures that the stable states of the network are characterized by the Boltzmann distribution of statistical physics.

The probability P_α of a physical system being in state α with corresponding energy E_α is given by

$$P_\alpha = \frac{1}{Z} e^{-\frac{E_\alpha}{K_\beta \tau}}$$

$$\begin{aligned} \text{where } K_\beta &= \text{Boltzmann constant,} \\ \tau &= \text{absolute temperature,} \\ Z &= \textstyle\sum_\gamma e^{\frac{-E_\gamma}{K_\beta \tau}} \end{aligned}$$

Z is called the sum over all states of the energy values (zustandsumme).

At thermal equilibrium

$$\frac{P_\alpha}{P_\beta} = e^{\frac{-[E_\alpha - E_\beta]}{K_\beta \tau}}$$

In our case, we replace $K_\beta \tau$ by T which is the (pseudo) temperature and E_α by h_α (the evolution of the state α)

Question: What happens to P_α as $T \to \infty$?

(Assuming that states are represented by N-bit binary vectors)

$$P_\alpha \to \frac{1}{Z}$$
$$Z = \sum_\alpha e^{\frac{-h_\alpha}{T}} = 2^N$$

This means that at very high values of T, almost every state is equally likely to be visited by the algorithm. As temperature is lowered, "better" states become more likely to be visited.

In practice, we start with a large T and lower it slowly (but not exponentially slowly) and hope for a near optimal solution relatively fast.

2.2 Genetic Algorithms (GA)

GAs are a class of randomized algorithms for searching high-dimensional spaces "efficiently" for "good" (hopefully near-optimal) solutions especially in the absence of any knowledge of the search space other than a function that can be used to "evaluate" candidate solutions (and partial solutions).

Not surprisingly, they are computationally intensive. Yet they have some attractive properties that make them a viable alternative for certain classes of problems.

Representation

In standard genetic algorithms, states are represented by fixed length (typically binary) strings over a finite alphabet. For example, the set of all binary strings (i.e., strings over the alphabet $\{0, 1\}$) of length 4 can represent a state-space of size 2^4. Each such string (e.g., 1101) represents a particular state (or a candidate solution) in the state-space.

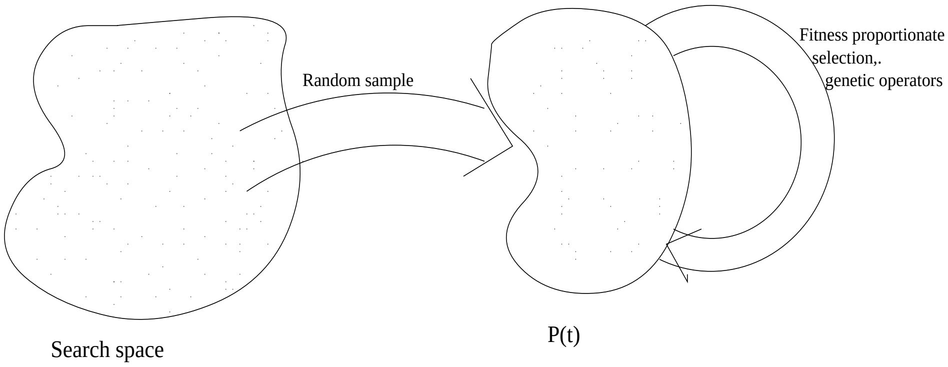
Fitness function (Evaluation function)

$$f(x_i) = fitness\ of\ individual\ (state)\ x_i$$

is used by the GA to evaluate candidate solutions. Note that f is analogous to heuristic functions used in more traditional search algorithms.

General Idea

Genetic algorithms are inspired by processes that occur in biological evolution: fitness-proportionate selection and transfer of traits from parents to offspring through genetic operations such as crossover that take place during sexual reproduction. The following example illustrates how the basic operations of a GA.



Example

Suppose that you want to find an optimal business strategy for running a coffee shop. Suppose also that you have no idea what such a strategy might be. However, you have some variables that might affect the success of your strategy. You can run an experiment to evaluate any given strategy.

To keep things simple, consider only 3 binary variables that could affect your success. Therefore, the size of the search space is $2^3 = 8$.

For the sake of this example, suppose that the objective function $f(x_i)$ gives me the decimal value of string x_i. For example,

$$f(011) = 3$$

$$f(101) = 5$$

In that search space, the best individual (state) is 111. However, in practice, such an explicit definition of the fitness function may not be possible. In such a case, the candidate solutions (states in the state space) are evaluated in terms of their relative success at solving the problem at hand. In our example, evaluation of a business strategy might require that we use it to run a coffee shop and calculate the profits.

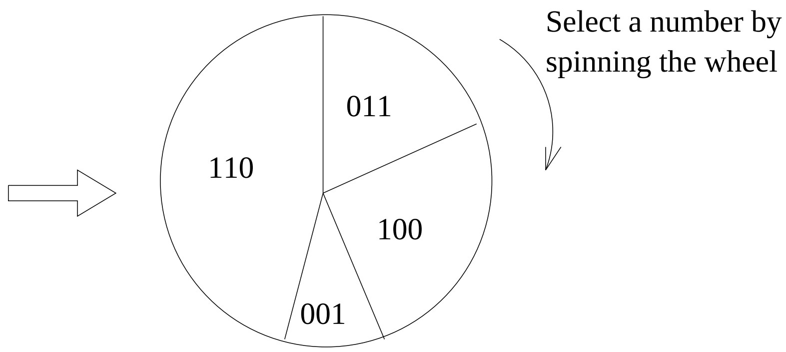
GA starts with a population of candidate solutions. (The size of this population is usually much smaller than the size of the search space.)

The initial population of size M is obtained by picking M points from the search space according to the uniform probability distribution. Thus if we have 8 points in the search space, each point has a probability $\frac{1}{8}$ of being selected.

Initial population

j	x_j	$f(x_j)$	$\sum_{i=1}^{j} f(x_i)$	$P(x_j)$
1	001	1	1	1/14
2	100	4	5	4/14
3	011	3	8	3/14
4	110	6	14	6/14

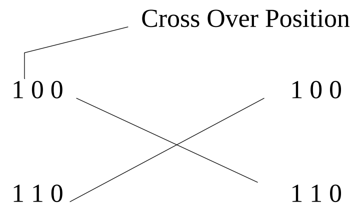
Fitness-Proportionate Selection
We use a wheel of fortune to ensure fitness-proportionate selection. That is, for a population size of M, there are M independent trials or spins of the wheel. On each trial, an individual x_i has a probability of being selected (on average) given by $P(x_i)$.

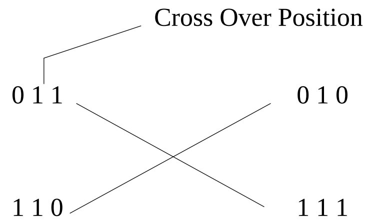


Suppose the selected individuals are 001, 011, 100, 110.

Mating

Some fraction of the selected individuals (determined by the probability of crossover p_c – more on this later) are mated pairwise at random. In this example, we will assume that $P_c = 1$.





---

1 0 0, 1 1 0, 0 1 0, 1 1 1

Mutation

Mutation is neglected in our example since the probability P_m of an arbitrary bit being mutated is generally very small and the size of the population in our case is also very small.

Note that crossover may (but not necessarily) produce offsprings that are radically different from their parents.

Of course, in this example, GA was able to find the optimal solution 111 within one generation. However, in larger search spaces with more complex fitness landscapes, finding an acceptable solution may take a large number of generations. This example, although suggestive of the general effectiveness of GA, is hardly conclusive evidence for its reliability as a search algorithm. In order to understand why GA works (to the extent that it does) we need to examine its theoretical foundations.