

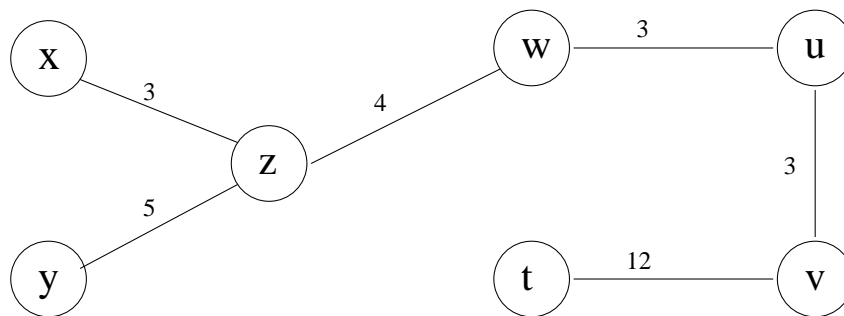
Problem assignment 3

Due: Thursday, October 8, 2009

Problem 1. Constraint satisfaction.

Constraint propagation procedures allow us to infer assignments of values to state variables that are consistent and inconsistent with constraints defining the goal configurations. These assignments are represented through equations and disequations.

Consider the following graph where each node is a variable and an arc is labeled with a number. Each variable can take on integer values from 0 to 9 (including 0 and 9). Each arc represent a constraint the two variables connected by the arc must satisfy. The constraint is that each variable must have the same value modulo the number on arc. For example, the arc connecting x, z with value 3 represents a constraint $x \bmod 3 = z \bmod 3$. This constraint can be satisfied by assignments $x = 2, z = 2$ or $x = 5, z = 2$, or $x = 7, z = 1$.



Assume we know the assignments $x = 2$, $y = 0$ and $t = 0$. Give variable values that would be inferred by:

- forward checking
- arc consistency

The procedures differ in terms of completeness of the inferences they make and their computational complexity. In the following we briefly summarize the two procedures in terms of inferences they make. Examples of the two procedures were given during the lecture.

Forward checking. Infers:

- Disequations: from equations and constraints.
- Equations: from disequations and constraints through the exhaustion of alternatives.

Arc consistency Infers:

- Disequations: from equations, disequations and constraints.
- Equations: from disequations and constraints through the exhaustion of alternatives.

Problem 2. Traveling Salesman Problem

The Traveling Salesman problem (TSP hereafter) is a classical graph-theoretical problem. It involves a traveling salesman who has to visit each of the cities in a given network before returning to his starting point, at which time his trip is complete. The objective is to find the cheapest tour, that is the shortest route that passes through each city exactly once and returns back to start.

The variants of TSP arise in the design of telephone networks and integrating circuits, in planning construction lines, and in the programming of industrial robots, to mention but a few applications. In all of these, the ability to find inexpensive tours of the graphs in question can be quite crucial.

In this assignment, we explore **simulated annealing** solution to the TSP problem. The TSP we use consists of n cities placed on a two dimensional map. The (x, y) coordinates define locations of cities. The objective is to find a tour with the shortest overall length. The distance between the two cities is the Manhattan distance of their locations:

$$d_M(A, B) = |x(A) - x(B)| + |y(A) - y(B)|.$$

To implement a TSP solver we need a means for representing a TSP, a tour and methods for computing the length (energy) of a tour. C/C++ routines supporting

these can be downloaded from the course web page. The files include the following functions:

- *generate_TSP(n)* - generates a random instance of the TSP with n cities;
- *generate_standard_TSP()* - generates the standard TSP you should experiment with in the assignment.
- *generate_random_tour()* - generates initial random tour for the problem;
- *generate_mutation(problem, a_tour)* - generates a mutation of a tour;
- *copy_tour(a_tour)* - makes a copy of *a_tour*;
- *delete_tour(a_tour)* - deletes a tour;
- *energy(problem, a_tour)* - computes the energy of a tour (note that we seek the tour with the lowest energy since energy reflects the distance)
- *show_tour(a_tour)* - prints the tour.
- *show_cities(a_problem)* shows the coordinates of cities used in the TSP.

Part a. Simulated annealing algorithm

Simulated annealing explores the space of all tours by generating random rearrangements of the current tour. The new tour is accepted when its energy (distance) is smaller than the energy of the current tour. The tour with a higher energy is accepted randomly with probability $e^{\Delta E/T}$, where ΔE is the energy difference between current and new energy, and T is the temperature parameter that is changed during the search. The probabilistic choice is a solution to the problem of local optima.

Implement a simulated annealing function

sim_anneal(problem, no_of_steps, init_temperature)

for solving the TSP problem. The algorithm should:

- start from a random tour and should return the energy function of the tour found.
- Use the **mutation** mechanism to implement random rearrangements of the tour. A mutation of a tour consists of reversing the direction in which a section of the tour is traversed. To illustrate this, assume a tour: *ABCDEF* in the 6 city TSP. Reversal of *CDE* yields the new mutated tour *ABEDCF*. Function *generate_mutation(a_tour)* given to you accomplishes this task.

- Use a linear cooling schedule in which the temperature is decreased linearly in $k + 1$ steps, starting from the initial temperature T_{init} and ending up in the zero temperature. The temperature in the i th step is:

$$T_i = \frac{T_{init}}{k} * (k - i), \quad (1)$$

which gives $T_0 = T_{init}$ in the 0-th step, and $T_k = 0$ in the k -th step. The initial temperature T_{init} and the number of steps k are the parameters of the simulated annealing function (*init_temperature, no_of_steps*).

- Collect and print the following results and statistics:
 - Initial tour and its distance (energy);
 - Initial temperature T_{init} ;
 - Number of tours tried;
 - Number of tours accepted;
 - The best tour found and its distance (energy).

Hint: To write the annealing algorithm you must implement a probabilistic choice of configurations with higher path energies. To do this you can proceed as follows:

1. Compute $p = e^{\Delta E/T}$ (it must give the value $0 \leq p \leq 1$).
2. Choose randomly a number x from interval $[0, 1]$. In C/C++ you can implement the random choice using function *random()*. This function generates integers in some prespecified range, not reals in $[0, 1]$. To obtain random values in $[0, 1]$ compute *random()/RAND_MAX*, where *RAND_MAX* is the largest value *random()* can generate.
3. Accept if $x \leq p$, reject otherwise.

All of the above simulated annealing code should be included in the file *main3a.c*. In addition, the code should run a simulated annealing code on the standard TSP problem with the number of simulation steps = 100,000 and initial temperature = 100. The standard TSP is given in file *std_tsp.txt* and consists of 60 cities. Use function *generate_standard_TSP* to read in the problem.

Part b. Experiments with the simulated annealing program

Experiment with your simulated annealing algorithm on the standard TSP problem (file *std_tsp.txt*) while varying parameters of the simulated annealing procedure:

the initial temperature and the number of simulation steps. Choose values of the parameters such that your algorithm is able to find the solution with the path cost at least as small as 120. Submit the solution you have found, its distance (energy) and collected statistics in your report. See if you can beat our best solution of 72.892862. You do not have to submit programs you use in experiments in this part.

Part c. Cooling schedule competition

In part (a) you were asked to implement a linear cooling schedule. However the linear schedule may not be the best option. Propose a new cooling schedule which you think performs better. Describe it briefly in the report and write a C code implementing the corresponding procedure.

Write program *main3c.c* that calls your cooling schedule procedure and applies it to accept or reject 20,000 candidate tours. Your program should randomly restart the annealing 10 times and report the average energy of the resulting tour. The authors of the three best programs (in terms of average best tour energy) shall receive extra credit of up to 20 points.

Part d. Genetic algorithm

The disadvantage of the simulated annealing algorithm is that at any point in time it keeps only one current configuration and that next step configurations are obtained using “local” changes. Thus, it may take a number of steps till one gets to explore good configurations. The genetic algorithm attempts to alleviate the problems by keeping a limited number of “current” configurations and by combining (more radically) two good quality solutions hoping that the combination will lead to a larger improvement in the quality.

Please familiarize yourself with the code in *main3d.c* and the methods *next_gen()* and *pick_a_pair_random()* in *Population.cpp*. You will find potentially useful comments there.

Collect results (fitness of the best individual in the last generation) for different settings of parameters of the genetic algorithm procedure. The procedure is included in *main3d.cpp* and uses the following five parameters:

1. Country – describes the problem (you do not have to change this one);
2. Number of generations to simulate;

3. Population size (max 500 or play with defs.h);
4. Mutation probability (a value between 0 and 1);
5. Survival rate (a value between 0 and 1). It determines how much of the old population is kept in the new population.

The default configuration is: Number of generations 300, Population size 300, Mutation probability 0.03, Survival rate 0.2.

Please report best configuration(s) found by varying:

- **(a)** Number of Generations in the range between 50 and 500 in increments of 50 while keeping all other parameters fixed at the default setting.
- **(b)** Population size in the range between 50 and 500 in increments of 50 with the rest of the parameters set to the defaults.
- **(c)** Mutation probability in the range between 0.00 and 0.25 in increments of 0.05 with the remaining parameters set to the default values.
- **(d)** Survival rate in between 0 and 1 in increments of 0.2 with the remaining parameters fixed at the defaults.

Please report your findings either in tables (one for each experiment) or graphs. Analyze the results and draw the conclusions. No submission of code is required for this part.