University of Pittsburgh
CS 2710 Foundations of Artificial Intelligence — Handout 4
Professor Milos Hauskrecht — Wednesday, September 18, 2019

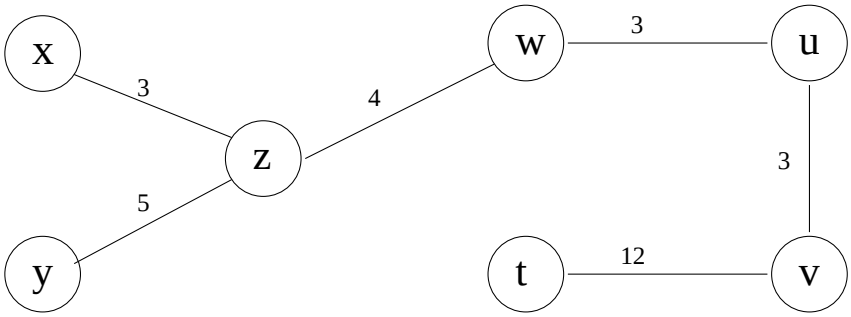
Problem assignment 3

Due: Wednesday, September 25, 2019

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 (assignments of values to variables) and disequations (invalid assignments).

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 two 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 (invalid assignments): from equations (assignments) and constraints.
- Equations: from disequations and constraints through the exhaustion of alternatives.

Arc consistency Infers:

- Disequations: from equations, disequations and constraints. (Variables and their remaining values in a constraint are rechecked for consistency in both directions accross the constraint that is represented by an arc.)
- 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** and **genetic algorithms** 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 of a tour. Python code supporting the definition of the TSP problem is given to you in $TSP.py$ file. The TSP file supports methods that let you generate a random tour, calculate its length, mutate a tour locally, as well as, combine (crossover) two tours.

Part a. Simulated annealing algorithm

Simulated annealing explores the space of all tours by generating random local 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

$$sim\_anneal(TSP\_problem, no\_of\_steps, init\_temperature)$$

for solving the TSP problem, and include it in the $SA.py$. 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 $ABEDCF$. Method *permute_tour* given to you in $TSP.py$ 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 ith step is:

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

  which gives $T_0 = T_{init}$ in the 0-th step, and $T_k = 0$ in the k-th step. The number of steps k and the initial temperature T_{init} are the parameters of the simulated annealing function $(no\_of\_steps, init\_temperature)$.
- 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 Python you can implement the random choice using function *random.random*() in *random.py* module.
3. Accept if $x \leq p$, reject otherwise.

All of the above simulated annealing code should be included in the file *SA.py*. In addition, the code should run simulated annealing algorithm on the standard TSP problem with the number of simulation steps = 100,000 and initial temperature = 100. The standard TSP is given in file *TSP.py* (in var *Standard_Cities*) and consists of 60 cities.

Part b. Experiments with the simulated annealing program

Experiment with your simulated annealing algorithm on the standard TSP problem 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 to generate results for this part (part b) of the assignment.

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 Python code implementing the corresponding procedure.

Write program *mySA.py* 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.

Problem 3. 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 (GA) 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 $Population_py$, *GA.py*, that implements the GA algorithm with crossover, mutation, elite selection and culling. The parameters are defined in *GA.py* file. The default configuration is: Number of generations 500, Population size 500, Mutation probability 0.05, Elite selection 0.05 and culling 0.05.

Collect the statistics and fitness of the best individual in the last generation for the different settings of parameters of the genetic algorithm procedure. Please report results found by varying:

- **(a)** Number of Generations in the range between 200 and 1000 in increments of 100 while keeping all other parameters fixed at the default setting.
- **(b)** Population size in the range between 250 to 750 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)** Culling in between 0 to 0.25 in increments of 0.05
- **(e)** Elite selection in 0 to 0.25 in increments of 0.05 with the remaining parameters fixed at the defaults.

Please analyze the results and report your findings either in tables (one for each experiment) or graphs. Draw conclusions based on the experiments for each of the parameters and the effect the parameters have on the solution.