

LINGI2261: Artificial Intelligence

Assignment 4: Local Search and Propositional Logic

Jean-Baptiste Mairy, Cyrille Dejemeppe, Yves Deville
November 2012



Guidelines

- This assignment is due on **Wednesday 12 December, 6:00 pm**.
- **No delay** will be tolerated.
- Not making a **running implementation** in **Python 3** able to solve (some instances of) the problem is equivalent to fail. Writing some lines of code is easy but writing a correct program is much more difficult. (Remember B. Lecharlier's courses.)
- **Document** your source code (at least the difficult or more technical parts of your programs). Python docstrings for important classes, methods and functions are also welcome.
- Indicate clearly in your report if you have **bugs** or problems in your program. Do not let the instructor discover it.
- Copying code from other groups (or from the internet) is strictly forbidden. Each source of inspiration must be clearly indicated. The consequences of **plagiarism** is **0/20 for all assignments**.
- Answers to all the questions must be delivered at the INGI **secretary** (paper version). Put your names and your group number on it. Remember, the more concise the answers, the better.
- Source code shall be submitted via the **SVN** tool. Only programs submitted via this procedure will be corrected. No report or program sent by email will be corrected.
- Respect carefully the **specifications** given for your program (arguments, input/output format, etc.) as we will use fully-automated tests. When we provide a way to test your program, it **must** pass this test.



Deliverables

- The answers to all the questions in paper format **at the secretary** (do not forget to put your group number on the front page).
- The following files inside the milestone4 directory of your SVN repository:
 - randomwalk.py: TSP algorithm of section 1.1
 - maxvalue.py: TSP algorithm of section 1.1
 - randomized_maxvalue.py: TSP algorithm of section 1.1
 - simulated_annealing.py: TSP algorithm of section 1.2
 - tabu.py: TSP algorithm of section 1.2
 - install.py: program for the package installation problem
 - ubuntu.install: the solution to the Ubuntu installation problem


1 The Traveling Salesman Problem (13 pts)

The Traveling Salesman Problem (TSP) is the problem of finding a route visiting all the given cities with minimal travel distance. Each city must be visited once and only once. For this assignment, we assume that there is a road between each pair of cities. The positions of the cities on the map are not given. Rather, you have access to the distance matrix. This distance matrix contains, in row i column j , the distance between city i and j . The distances are here considered to be symmetric. This means that the distance between cities i and j is the same as the distance between cities j and i . As an example, Figure 1 represents an optimal route through the 15 largest cities in Germany, assuming a straight road between the cities.



Figure 1: Optimal TSP route for the 15 biggest cities in Germany

The format for describing the different instances on which you will have to test your programs is the following:

 **File format**

```
n
d0,0
d1,0 d1,1
d2,0 d2,1 d2,2
...
dn-1,0 ... dn-1,n-1
```

In those files, n is the number of cities and below, you can find the distance matrix. Since the distances are symmetric, only the lower part is given. In this sparse triangular matrix, $d_{i,j}$ represents the distance between cities i and j . All those distances are given as floats.

The goal of this assignment is to use local search to find good solutions for the Traveling Salesman Problem. In order to help you, some algorithms are already implemented in the `aima-python3` module on iCampus (still under Document > Assignments). The test instances can also be found on icampus under Document > Assignments > Assignment4.

Warning: the tests that you have to perform on the given instances of the TSP require time. Do not wait the last minute to design and run them.

1.1 Diversification versus Intensification

The two key principles of Local Search are intensification and diversification. Intensification is targeted at enhancing the current solution and diversification tries to avoid the search from falling into local optima. Good local search algorithms have a tradeoff between these two principles. For this part of the assignment, you will have to experiment this tradeoff.



Questions

1. Formulate the Traveling Salesman Problem as a local search problem. How do you represent a solution? How do you evaluate the solutions? The only move used to define the neighborhood is the swap of two cities in the route.
2. Implement the following strategies, each in their own file:
 - (a) `randomwalk.py` chooses the next node in the neighborhood randomly (you can use the `random_walk` function for that),
 - (b) `maxvalue.py` chooses the best node (i.e., the node with maximum value) in the neighborhood, even if it degrades the current solution,
 - (c) `randomized_maxvalue.py` chooses the next node randomly among the 5 best neighbors (again, even if it degrades the current solution).

To construct the initial solution, use a greedy algorithm: start with a random city and travel to the nearest non-visited city. The search shall stop after 100 steps and return the *best* solution found (which may be different from the last one). Each file is a Python program that takes the path to the instance to solve as only argument.

3. Compare the 3 strategies on the given TSP instances. Report in a table the results of the tests. Interesting metrics to report are: the computation time, the value of the best solution and the number of steps when the best result was reached (`Node.step` may be useful). A good way to eliminate the effect of the randomness of some of the strategies is to run the computation multiple times and take the mean value of the runs. For the first and the third strategy, each instance should be tested at least 5 times.
4. For the instance `att48.tsp` and for each strategy, plot a graph of the evolution of the value of the current solution with respect to the step number.
5. Answer the following questions:
 - (a) What is the best strategy?
 - (b) Why do you think the best strategy beats the other ones?
 - (c) What are the limitations of each strategy?
 - (d) What is the behavior of the `maxvalue` strategy when it falls in a local optimum?
 - (e) What do you conclude about the intensification versus diversification?

1.2 Some well-known strategies

Simulated annealing

A technique you saw during the course to escape from local minima is *simulated annealing*. The most important parameter of this technique is the temperature schedule. To get you started, an exponential time schedule is provided.

Tabu Search

A disadvantage of the randomized_maxvalue strategy is that the neighborhood of the neighbor of a node includes the node itself. As the result of applying two times the same swap to a state is the same state, we would like to prohibit an action (i.e., the swap) for a certain number of steps after using it.



Questions

1. Write a program, called `simulated_annealing.py`, that uses the provided `simulated_annealing` function to solve the TSP. The search shall use the default temperature schedule. The program takes the path to the TSP instance as single argument.
2. Write a function `tabu_search(problem, length, limit)` implementing a tabu search. At every step, it shall select an action randomly among the 5 non-tabu actions that yield the best neighbors. A chosen action is then tabu for 'length' steps. The search shall stop after 'limit' steps and return the best solution found. Use this function in a program `tabu.py` taking two arguments: the path to a TSP instance to solve and the length of the tabu list. The step limit is 100.
3. Compare the simulated annealing and tabu strategies on the given TSP instances. For tabu search, use lengths 3, 5, 10 and 20. As always, beware of the randomness. Also compare the results to the strategies of the previous section.
4. For the instance `att48.tsp` and for the two strategies, plot a graph of the evolution of the value of the current solution with respect to the step number.
5. The principal parameter of simulated annealing is the temperature schedule. What is its effect on the search?
6. The principal parameter of tabu search is the length of the tabu list. How does this parameter impact the search?

2 Propositional Logic (7 pts)

2.1 Models and logical connectives

Consider the vocabulary with four propositions A , B , C and D and the following sentences:

- $(A \vee B) \wedge (B \vee C)$
- $A \wedge B$
- $(A \Rightarrow B) \Leftrightarrow C$



Questions

1. For each sentence, give the number of models that satisfy it (considering the whole vocabulary).
2. Give **two valid interpretations** of the following sentence: $(A \wedge B) \Rightarrow C$.

2.2 Package installation problem

A modern open-source OS distribution has over 30,000 packages that can be installed by the user. However, not all combinations are possible. Three kinds of relations impose constraints on what can and what cannot be installed together:

Depends: when package A depends on package B and the user wants to install A, then B must also be installed.

Conflicts: packages that must not be installed together.

Provides: when package A provides package B, every dependence constraint on B is satisfied whenever A (or any other package providing B) is installed. Package B is said to be a virtual package.

Given a list of packages selected by the user, the problem consists in finding a minimal superset of packages that need to be installed in order to satisfy all the constraints. For this assignment, we will only search for a satisfying superset without bothering about the minimal part.



Questions

1. Explain how you can express this problem with propositional logic. What are the variables and how do you translate the relations and the query?
2. Translate your model into CNF.
3. An extension of the problem deals with disjunctive dependencies. For example, package A may depend on B or C, meaning A may be installed if either B or C (or both) is also installed. How can you transform this problem into the basic problem without disjunctive dependencies?

On the iCampus site, you will find a number of files to download:

- `Packages.gz` is a simplified version of the Ubuntu 12.10 repositories (main and universe),
- `Packages_tiny.gz` is a small test repository,
- `packages.py` is a Python module to load and manipulate the packages in the above repositories,
- `minisat.py` is a simple Python module to interact with MiniSat
- `minisat-ingilabs.tar.gz` is a pre-compiled MiniSat binary that should run on the machines in the computer labs.

MiniSat is a small and efficient SAT-solver we will use. Either use the pre-compiled binary from iCampus or download the sources from <http://minisat.se>. A quick user guide can

be found at <http://www.dwheeler.com/essays/minisat-user-guide.html>, but that should not be needed if you use the `minisat.py` module. Extract `minisat` from `minisat-ingilabs.tar.gz` in the same directory that `minisat.py` to be able to use the latter script.



Questions

4. Create a program `install.py` which takes as first argument the repository's file-name (e.g., `Packages.gz` or `Packages_tiny.gz`). All following arguments are the packages that the user wants to install. Your program shall print the packages that need to be installed.
5. What is the output of your program when asking to install "aima-python3" using the `Packages_tiny.gz` repository? How many variables and how many clauses did you generate to get this result?
6. Using the `Packages.gz` repository, what do you need to install if you want the Ubuntu desktop (package "ubuntu-desktop")? Put your output in a file called `ubuntu.install` in your svn repository (one package name per line).
7. To reduce the search space and avoid some unneeded packages in the solution, can you think of lower and upper bounds? Given a list of packages to install, the lower bound are packages that will certainly need to be installed. Every package outside the upper bound is certainly not needed.

We have simplified the repositories by removing the version numbers. In a real system, there can exist different versions of a same package, but only one can be installed at the same time. The "Depends" and "Conflicts" relations may then have an additional information for each package stating that the constraint only applies to versions below or above a given number.



Questions

8. How would you modify your model to take the version constraints into account? You do not need to program anything. But explain the impact on the variables and the clause generation.