

LINGI2261: Artificial Intelligence

Assignment 4: Local Search and Propositional Logic

Minh Thanh KHONG, François Aubry, Michael Saint-Guillain, Yves Deville
December 2017



Guidelines

- This assignment is due on **Wednesday 13 December, 8:00 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.
- **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. The online submission system will discover them anyway.
- Copying code or answers 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 in **a single PDF report file per group**, on Moodle, under Assignment 4-Report. Remember, the more concise the answers, the better. Indicate your group number in the PDF filename.
- Source code shall be submitted on the online **INGInious** system. Only programs submitted via this procedure will be graded. No report or program sent by email will be accepted.
- Respect carefully the **specifications** given for your program (arguments, input/output format, etc.) as the program testing system is **fully automated**.



Deliverables

- Answers to all the questions in a single report, named **report_a4_group_XX.pdf**, on Moodle. **Do not forget to put your group number on the front page.**
- The following files (encoded in **utf-8**) are to be submitted on **INGInious** inside the *Assignment 4* task(s):
 - `tsp_maxvalue.py`: your *maxvalue* local search for TSP problem, to submit on INGInious in the *Assignment4: TSP maxvalue* task.
 - `tsp_randomized_maxvalue.py`: your *randomized maxvalue* local search for TSP problem, to submit on INGInious in the *Assignment4: TSP randomized maxvalue* task.
 - `gc_sol.py`: which contains `get_clauses` method to solve the Graph Coloring problem, to submit on INGInious in the *Assignment4: Graph Coloring Problem* task.

1 The Travelling Salesman Problem (13 pts)

In this assignment, you will design algorithms to solve the a classical problem for a travelling salesperson, the Travelling Salesman Problem (TSP), which can be described as following. Given a list of cities and distances between them, the travelling salesman starts from home, **visits each city exactly once** to sell goods and returns to his home. In order to minimize the cost, the travelling salesman has to figure out a visiting order of all the cities that minimizes the sum of distances travelled when moving from one city to another. Figure 1 shows an example of a small TSP and a feasible visiting order.

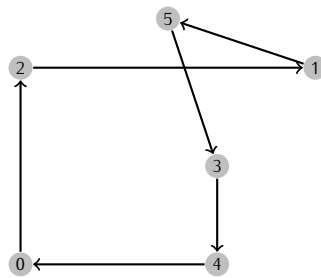



Figure 1: A TSP example. An possible visiting order is (0, 2, 1, 5, 3, 4).

For this problem, a list of n locations will be given, (x_i, y_i) is the location for the city i ($i = 1..n - 1$). The distance between each pair of cities will be computed by the Euclidean distance, $d(c_1, c_2)$ denotes the distance between two cities c_1, c_2 . Let v_i ($i = 0..n - 1$) be the i -th city in the visiting order. Then the goal is to minimize the sum of distances called the cost of the solution: $\sum_{i=0}^{n-2} d(v_i, v_{i+1}) + d(v_{n-1}, v_0)$.

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

 **Input format**

n	
x_0	y_0
x_1	y_1
...	
x_{n-1}	y_{n-1}

The first line contains the number of cities: n . It is followed by n lines, each line (x_i, y_i) represents the location of a city.

For this assignment, you will use *Local Search* to find good solutions to the TSP. The test instances can be found on Moodle. A template for your code is also provided. The output format **must** be the following:

Output format

V^i				
v_0^i	v_1^i	\dots	v_{n-1}^i	
V^f				
v_0^f	v_1^f	\dots	v_{n-1}^f	

Where V^i is the cost of the initial solution and the next line contains the initial visiting order. The next part is the same but contains the final solution. V^f is the cost of that solution and the next line is the visiting order of the final solution. Note that value of V^i and V^f should be 2-digit precision.

For example, the input and output for the TSP in Figure 1 could be:

Input format

6		
0	0	
3	2	
0	2	
2	1	
2	0	
1.5	2.5	

Output format

11.16					
0	2	1	5	3	4
9.58					
0	2	5	1	3	4

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.

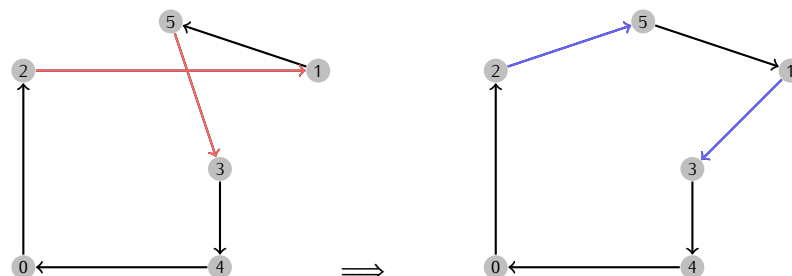


Figure 2: A possible 2-opt swap of the visiting tour (0, 2, 1, 5, 3, 4) is to take two edges (2, 1) and (5, 3) then replace them by (2, 5) and (1, 3). The new visiting order is (0, 2, 5, 1, 3, 4). Edges (3, 4) and (4, 0) should not be considered for 2-opt swap since they are adjacent (city 4 in common).



Questions

1. (1 pts) Formulate the TSP problem as a Local Search problem (problem, cost function, feasible solutions, optimal solutions). You are given a template on Moodle: *tsp.py*. Implement your own extension of the *Problem* class from *aima-python3*. For this assignment, the moves considered in the algorithms are 2-opt swap i.e. two non-adjacent edges (i.e. do not have a common city) in the visiting tour will be replaced by two another edges to make a new feasible visiting tour. An example is illustrated in Figure 2.
2. (5 pts) Implement the maxvalue and randomized maxvalue strategies. To do so, you can get inspiration from the *randomwalk* function in *search.py*.
 - (a) maxvalue chooses the best node (i.e., the node with minimum value) in the neighborhood, even if it degrades the quality of the current solution. The maxvalue strategy should be defined in a function called *maxvalue* with the following signature: *maxvalue(problem, limit=100, callback=None)*.
 - (b) randomized maxvalue chooses the next node randomly among the 5 best neighbors (again, even if it degrades the quality of the current solution). The randomized maxvalue strategy should be defined in a function called *randomized_maxvalue* with the following signature: *randomized_maxvalue(problem, limit=100, callback=None)*.

To construct the initial solution, use the following greedy algorithm: the travelling salesman start with city 0 (i.e. $v_0 = 0$), then the next city to visit will be the nearest unvisited city of the current city until all cities are visited. In case of ties, assign the city with smallest index. Your search shall stop after 100 steps and return *the best solution found* (which may be different from the last one).
3. (4 pts) Compare the 2 strategies implemented in the previous question between each other and with randomwalk, defined in *search.py* on the given wedding instances. Report, in a table, 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). For the second and third strategies, each instance should be tested 10 times to eliminate the effects of the randomness on the result. When multiple runs of the same instance are executed, report the means of the quantities.
4. (3 pts) 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 in terms of diversification and intensification?
 - (d) What is the behavior of the different techniques when they fall in a local optimum?

2 Propositional Logic (7 pts)

2.1 Models and Logical Connectives (1 pts)

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

- $(\neg A \vee C) \wedge (\neg B \vee C)$
- $(C \Rightarrow \neg A) \wedge \neg(B \vee C)$
- $(\neg A \vee B) \wedge \neg(B \Rightarrow \neg C) \wedge \neg(\neg D \Rightarrow A)$



Questions

1. (1 pts) For each sentence, give its number of valid interpretations, i.e. the number of times the sentence is true (considering for each sentence **all the proposition variables** A, B, C and D).

2.2 Graph Coloring Problem (6 pts)

The Graph Coloring Problem can be defined as follow. Given an undirected graph $G = (V, E)$, a graph coloring assigns a color to each node, such that all adjacent nodes have a different color. A graph coloring using at most k colors is called a k -coloring. The Graph Coloring Problem asks whether a k -coloring for G exists. Figures 3 below shows an example of a valid coloring (left) and an invalid coloring (right).



Figure 3: Example of graph coloring

Your task is to model this problem with propositional logic. Given $N = |V|$ nodes and k colors, $N \times k$ variables: $v_{ij} = 1$ iff node i is assigned color j ; 0 otherwise.



Questions

1. (2 pts) Explain how you can express this problem with propositional logic. What are the relations and how do you translate them?
2. (2 pts) Translate your model into Conjunctive Normal Form (CNF) and write it in your report.

On the Moodle site, you will find several files to download:

- 1-FullIns_3.col, 1-FullIns_4.col, 2-FullIns_3.col, 3-FullIns_3.col, 4-FullIns_3.col, 5-FullIns_3.col are the graph instances for this problem.
- graph.py is a Python module to load and manipulate the graph instances described above.
- solve.py is a python file used to solve the Graph Coloring Problem.
- gc_sol.py is the skeleton of a Python module to formulate the problem into an CNF.
- minisat.py is a simple Python module to interact with MiniSat.

- `minisat.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 Moodle 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 the executable `minisat` from `minisat-ingilabs.tar.gz` in the same directory that `minisat.py` to be able to use the latter script.

To solve the Graph Coloring Problem, enter the following command in a terminal:

```
python3 solve.py GRAPH_INSTANCE K
```

where `GRAPH_INSTANCE` is the graph instance file and `K` is number of colors.



Questions

3. (2 pts) Modify the function `get_clauses(G, k)` in `gc_sol.py` such that they solve the Graph Coloring Problem. Submit your functions on INGIInious inside the *Assignment4: Graph Coloring Problem* task.