PLD Agile

Introduction to the Long Duration Project (PLD)

Christine Solnon

INSA de Lyon - 4IF - 2022/2023

Overview

Introduction to the Long Duration Project

1. Description of the Application
2. Algorithms for computing tours
3. Organisation of the PLD

Description of the Application

Your mission:

- Design and implement an application for preparing delivery tours ...
  ... with bicycles

Use Case "Load a map"

- Read an XML file which contains lists of intersections and road sections:
  - Each intersection has a latitude and a longitude
  - Each section links 2 intersections and has a length and a name
- Display the map

Use Case "Enter a new request"

- The user selects an intersection, a courier, and a time-window
  $\rightsquigarrow$ time-window $\in \{[8, 9], [9, 10], [10, 11], [11, 12]\}$
- The system updates the tour of the selected courier:
  - Start from the warehouse at 8 a.m.
  - Visit each delivery during its time-window (service time = 5mn)
  - Minimise the arrival time back to the warehouse

Use Case "Save/load tours"

- Save the current tours in a file
- Restore a set of tours from a file

Overview

Introduction to the Long Duration Project

1. Description of the Application
2. **Algorithms for computing tours**
3. Organisation of the PLD

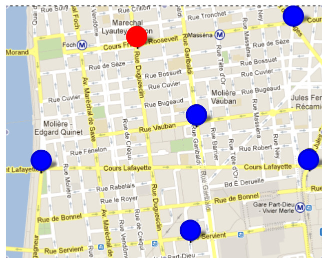
Two step solving process

Step 1: Computation of the shortest path graph

- Input: Set of delivery points + city map
- Output: Complete directed graph with 1 vertex per delivery point

Step 2: Solve the Asymetric Travelling Salesman Problem (ATSP)

- Input: Complete directed graph with 1 vertex per delivery point
- Output: Shortest Hamiltonian cycle



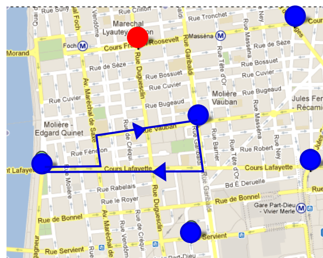
Two step solving process

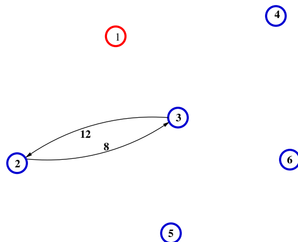
Step 1: Computation of the shortest path graph

- Input: Set of delivery points + city map
- Output: Complete directed graph with 1 vertex per delivery point

Step 2: Solve the Asymetric Travelling Salesman Problem (ATSP)

- Input: Complete directed graph with 1 vertex per delivery point
- Output: Shortest Hamiltonian cycle





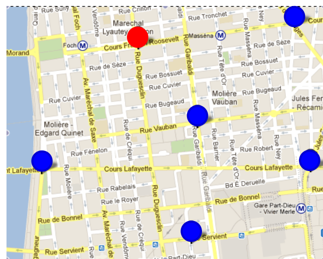
Two step solving process

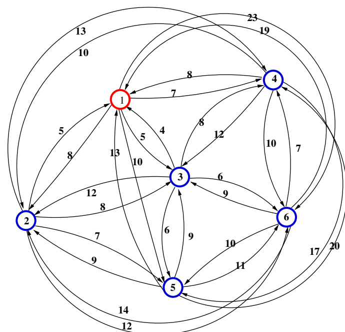
Step 1: Computation of the shortest path graph

- Input: Set of delivery points + city map
- Output: Complete directed graph with 1 vertex per delivery point

Step 2: Solve the Asymetric Travelling Salesman Problem (ATSP)

- Input: Complete directed graph with 1 vertex per delivery point
- Output: Shortest Hamiltonian cycle





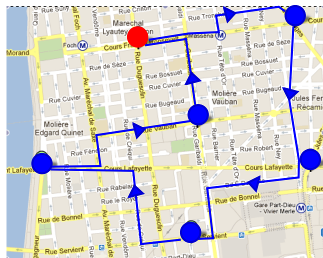
Two step solving process

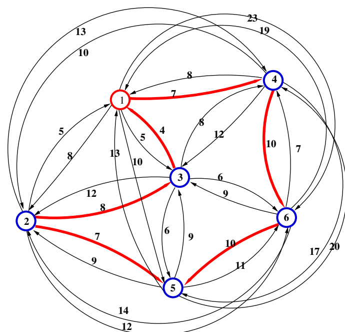
Step 1: Computation of the shortest path graph

- Input: Set of delivery points + city map
- Output: Complete directed graph with 1 vertex per delivery point

Step 2: Solve the Asymetric Travelling Salesman Problem (ATSP)

- Input: Complete directed graph with 1 vertex per delivery point
- Output: Shortest Hamiltonian cycle





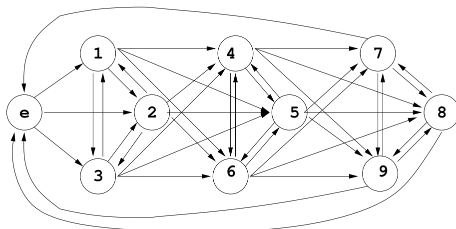
How to handle time-windows?

By removing edges from the directed graph:

Given two deliveries i and j with time-windows $[e_i, l_i]$ and $[e_j, l_j]$

- If $e_i = e_j$: Don't remove edges between i and j
- If $l_i = e_j$: Remove (j, i)
- If $l_j = e_i$: Remove (i, j)
- Otherwise: Remove both (i, j) and (j, i)

Example when $\{1, 2, 3\} < \{4, 5, 6\} < \{7, 8, 9\}$:



$\rightsquigarrow$ Check that each non visited vertex can be visited when building a tour

Approaches for solving the TSP (Recalls from 3IF)

The TSP is NP-hard!

$\leadsto$ Use appropriate approaches to explore the search space

Complete approaches (Dynamic Programming, Branch & Bound, ...)

- Exhaustive exploration of the search space
  $\leadsto$ Proof of optimality but exponential time complexity
- Use mechanisms to prune branches
- Use heuristics to explore first the most promising branches

Incomplete approaches (Local search, Ant Colony Optimisation, ...)

- Heuristic exploration of the search space
  $\leadsto$ May not find the optimal solution, but polynomial time-complexities
- Use mechanisms to intensify the search towards promising areas
- Use exploration mechanisms to guide the search towards new areas

For the PLD, you are free to choose your favorite approach/library

... but we provide you a very basic implementation

Enumeration of all Hamiltonian Tours (Recalls from 3IF)

```
public void allTours(Graph g){
    Collection<Integer> visited = new ArrayList<Integer>(g.getNbVertices());
    visited.add(0);
    Collection<Integer> unvisited = new ArrayList<Integer>(g.getNbVertices()-1);
    for (int i=1; i<g.getNbVertices(); i++) unvisited.add(i);
    allTours(0, unvisited, visited);
}
public void allTours(int currentVertex,
        Collection<Integer> unvisited,
        Collection<Integer> visited){
    if (unvisited.size() == 0){
        if (g.isArc(currentVertex,0)){
            // visited is an hamiltonian tour
        }
    } else {
        for (Integer nextVertex : unvisited){
            if (g.isArc(currentVertex,nextVertex)){
                visited.add(nextVertex);
                unvisited.remove(nextVertex);
                allTours(nextVertex, unvisited, visited);
                visited.remove(nextVertex);
                unvisited.add(nextVertex);
            }
        }
    }
}
```

Branch & Bound (Recalls from 3IF)

```
private void branchAndBound(int currentVertex, Collection<Integer> unvisited,
        Collection<Integer> visited, int currentCost){
    if (System.currentTimeMillis() - startTime > timeLimit) return;
    if (unvisited.size() == 0){
        if (g.isArc(currentVertex,0)){
            if (currentCost+g.getCost(currentVertex,0) < bestSolCost){
                visited.toArray(bestSol);
                bestSolCost = currentCost+g.getCost(currentVertex,0);
            }
        }
    } else if (currentCost+bound(currentVertex,unvisited) < bestSolCost){
        Iterator<Integer> it = iterator(currentVertex, unvisited, g);
        while (it.hasNext()){
            Integer nextVertex = it.next();
            visited.add(nextVertex);
            unvisited.remove(nextVertex);
            branchAndBound(nextVertex, unvisited, visited,
                    currentCost+g.getCost(currentVertex, nextVertex));
            visited.remove(nextVertex);
            unvisited.add(nextVertex);
        }
    }
}
```

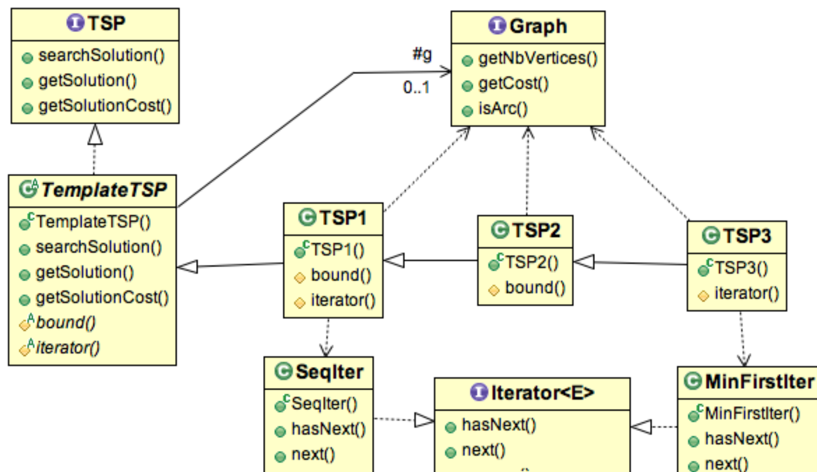
Different instantiations may be obtained by changing:

- The order vertices of *unvisited* are visited (*iterator*)
- The function used to compute a lower bound of the cost (*bound*)

How to avoid duplicating code?

GoF Pattern: Template

- Template method (*branchAndBound*) defines the sequence of steps
- Steps that may change (*bound, iterator*) = Abstract methods defined in sub-classes (*TSP1, TSP2, TSP3*)



Overview

1. Description of the Application
2. Algorithms for computing tours
3. **Organisation of the PLD**

Organisation of the PLD

Teams of 5 to 7 students:

- You are free to choose your organisation
  - Projet manager? Quality manager?
  - Product owner ? SCRUM manager ?
  - Daily stand-ups ?
  - ...
- But we'll ask you to take stock at the end of the project

Implementation of an Agile Iterative development process

Iteration 1: Inception

- Duration: 4 sessions of 4 hours
- Goals:
  - Identify the main use cases
  - Analyse the most important use cases
  - Design and implement a first version of your application

  $\leadsto$ Demo with the client at the end of the fourth session

Next iterations: from 1 to 4 iterations

For each iteration:

- Choose some use case scenarios
- Analyse, implement and integrate them to your application

$\leadsto$ Compare previsional and effective plannings at the end of each iteration

Test Driven Development:

Experiment it on at least one class...

Technical Environment

Some tools that you may use:

- Version Control System: Git
- Language: Java $\leadsto$ JavaDoc + Oracle Style guide (`http://www.oracle.com/technetwork/java/codeconventions-150003.pdf`)
- GUI: Swing (example with PlaCo) or Java FX
- IDE: Eclipse
- Unit Tests: JUnit4 (`http://www.junit.org/`)
- Reverse engineering: ObjectAid (`http://www.objectaid.com/`)
- UML diagram edition: Paper and pencil or StarUML (`http://staruml.io/`)

You may use other tools...

...But this must be discussed with us before!