**Optimized Delivery Simulation**

Author: Ha Dinh Duc Huy

# Problem

Delivery now is one of the most important part in trading industry. The delivers take packages from depot and deliver to customers. To minimize the cost of transportation, routes must be optimized.

# Solution

This application simulates the algorithm that solve the problem.

Include:

* Generate a random map
* Calculate the shorted distance from a depot to others
* Execute implemented algorithm to find the optimized route
* Output map and route to screen

# Algorithm

## Map Generator

First, this function creates a matrix hold units with x is number of column and y is number of row, this matrix called **Map**. A random pair of i and j ( 0 ≤ i < y, 0 ≤ j < x) is generated called **StartPoint**; mark the unit, which coordinate is **StartPoint,** as a **crossroads**. This **crossroads** then is sent to **BuildMap** function.

function **MapGenerator** (*rows*, *columns*):  
 **Map** ←Initialize Matrix with *rows* and *columns*  
 **StartPoint** ← Initialize Point with random x,y

**StartCrossroads** ← Mark Map[**StartPoint**] as a crossroads  
 **BuildMap** (**StartCrossroads**, **Map**)

**BuildMap** is a recursive function that will help in constructing roads. **BuildMap** recieve 2 value: ***a crossroads*** and ***a matrix(map)***.

For each direction(left, up, right, down) of the **current crossroads** that hasn’t connected to any other **crossroads**, mark the unit apart from **current crossroads** a valid random distance as a **crossroads** and connect it with **current crossroads**, mark units between those two crossroads as **road**; if there’s **road** on the way to **newly mark crossroads** then mark that road as a crossroad and connect those two crossroads and return the function or if there’s **crossroads** on the way to **newly mark crossroads** then just connect them and return the function; if there isn’t any **road** or **crossroads** on the way to **newly mark crossroads**, there’s chance for **BuildMap** to be called with **newly mark crossroads.**

function **BuildMap** (*crossroads*, *matrix*):

for each direction(left, up, right, down) of input *crossroads*:

if current direction is connected to other crossroads:

continue

**i** ← current position

distance ← **i** plus random number in current direction

while **i** less than distance and still in territory of *matrix*:  
 **i** ← **i** plus 1 in current direction

if there’s crossroads at **i**:

connect input *crossroads* with this crossroads

return

else if there’s road at **i**:

mark this road as crossroads

connect input *crossroads* with this crossroads

return

**NextCrossMap** ← mark unit at **i** as crossroads

connect input *crossroads* with this crossroads

if random() equal true:

**BuildMap** (**NextCrossMap**, *matrix*)

There are **properties** between two connected **crossroad**:

* **Weight**: the distance between two crossroads.
* **Traffic**: represented by an integer, it affects the weight of road linearly. In this project, there are 3 level of traffic: 1, 2, 3.
* **Road’s flow direction** (oneway-road or bothway-road): sometimes there are a pair of connected crossroads that not link to each other but only one of them links to other.

**Time’s complexity:** The worst case happen when BuildMap always is call itself again at the end of function and random distance between crossroads is always equal 1. The **Map** would be completely fill up with crossroads. **O(m\*n)**

**Space’s complexity:** It would take up to the size of the Map to store road, crossroads. **O(m\*n)**

## Create Lookup Distance

From depot positions create a lookup table store shortest distance and trail for between each pair of depot.

Create a two-dimension dictionary called **LookupPath** which’s each key is a **Point** and value hold a **Trail**. A **Trail** consist of two components: an **integer** and a **Point**. An **Integer** is the shorted distance between two key Point, **Point** hold coordinate of next depot that lead to depot has coordinate of first-key from depot has second-key’s coordinate.

For each depot, run Dijkstra and store value archived in **LookupPath.** In this project, Dijkstra is implemented using **priority queue**, there’re three parameters: **source**, **depots**, **storageTrail**; **source** is destination **Point**, **depots** is an array of all depot **Points**, **storageTrail** is a first-dimension of **LookupPath**.

function **Dijkstra** (*source*, *depots*, *storageTrail*):

*storageTrail*[*source*].Distance ← 0

**pQueue** ← Initialize priority queue

for each **Point** in depots:

if **Point** is not *source*:

*storageTrail*[**Point**].Distance ← INFINITY

*storageTrail*[**Point**].Next ← UNDEFINED

**pQueue**.add\_with\_priority(**Point**, *storageTrail*[**Point**].Distance)

while **pQueue** is not empty:

**u** ← **pQueue**.pop()

for each neighbor **v** of **u**:

**alt** ← *storageTrail*[**u**].Distance + distance(**u**,**v**) \* traffic(u,v)

if **alt** less than *storageTrail*[**v**].Distance:

*storageTrail*[**v**].Distance ← **alt**

*storageTrail*[**v**].Next ← **u**

**pQueue**.update\_priority(**v**, **alt**)

**Time’s complexity:** The expected time’s complexity for a Dijkstra running using priority queue is O(e + vlog(v)). e is number of edges and v is number of vertex. e can be up to m\*n and v can be up to (m+1)\*(n+1). Running Dijkstra for each depot would take **O((m\*n + (m\*n)log(m\*n))\*N)**. N is number of depot

**Space’s complexity:** it takes up to O(m\*n) to store shortest paths to a depot. For N depots it would take **O(N\*m\*n)**



## Create Optimized Route

This function take in two value: **depots** and **lookupPath**, **depots** is an array of depot point, **lookupPath** is a two-dimension dictionary. Create an array called **OptimizedRoute** store depots in optimized order.

First, push depot points into **OptimizedRoute** randomly and calculate total distance. Then, for each pair of depot, if by inversing their route giving smaller total distance, swap them and start over the iteration. This is technique is called **2-opt.**

Using this technique may cause the result stuck at greedy point called **local minimum**. To reduce the affection of this problem to the result, use a **temperature variable**. The higher temperature is, the the higher chance the swapping happen when new total distance is not smaller than the old total distance.

function **CreateOptimizedRoute** (*depots*, *lookupPath*):

**totalDis** ← 0

**temp** ← 100

**OptimizedRoute** ← Initialze array

for **i** ← 2 until **i** equal *depots* size:

**totalDis** ← **totalDis** + *lookupPath*[*depots*[**i**]][*depots*[**i**-1]].Distance

start:

for each pair of point in *depots*:

**newTotalDist** ← calculate new total distance after swap

if **newTotalDist** < **totalDist** or random(1,100) < **temp**:

swap route

**temp** ← max(**temp** - 1, 1)

go to start

return **OptimizedRoute**

**Time’s complexity:** What appears to be the most intuitive and common answer for this procedure is O(N!). At each iteration the algorithm can apply at most O(N^2) inversions but the number of overall iterations is weakly bounded by O(N!) since by removing a crossing the algorithm can consequently create new ones leading to the worst case scenario where all possibilities are tested

**Space’s complexity:** Only take space to store optimized route. O(N), N is number of depot

# Result

Note:

Green road: traffic level = 1

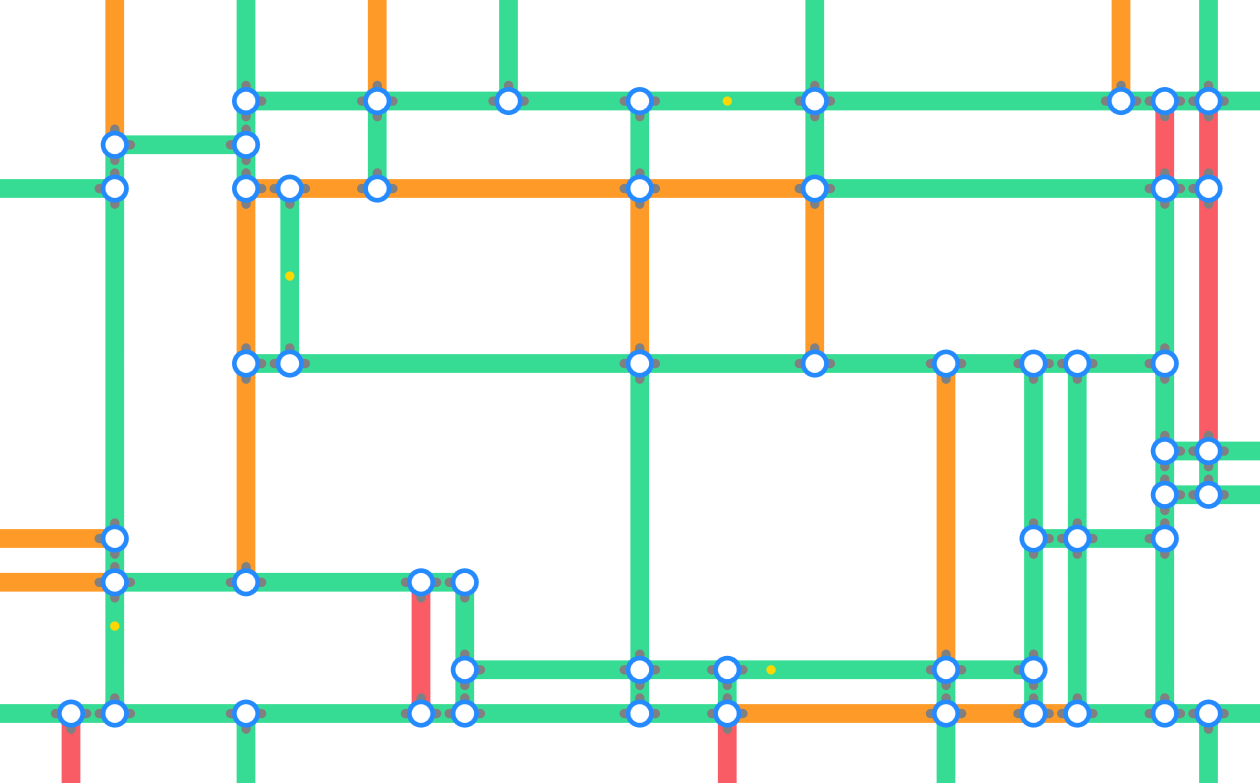
Orange road: traffic level = 2

Red road: traffic level = 3

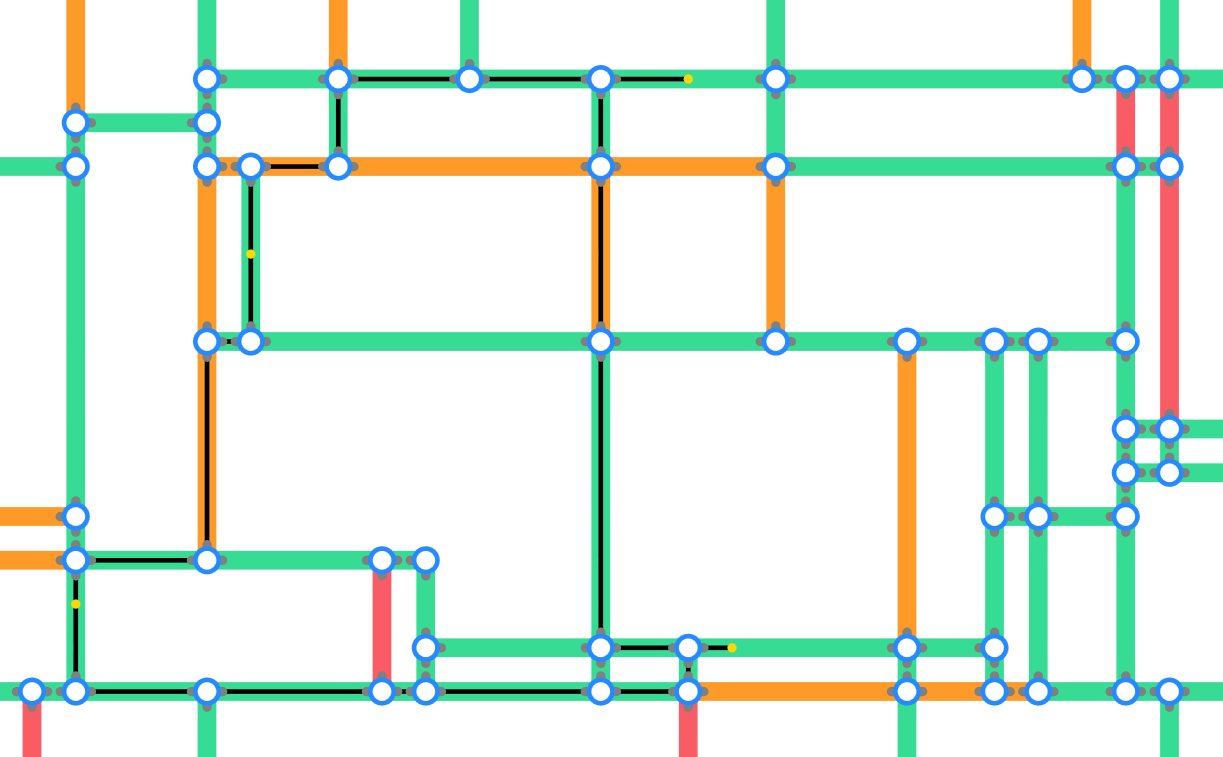
White node with blue stroke: crossroads

Grey node around crossroads: flow direction of road

**Three input depot (yellow dot)**



**Output route**



Test on Map has width of 50 and height of 50:

* 10 depots → take 0.091s
* 20 depots → take 0.098s
* 50 depots → take 0.0113s

Test on Map has width of 100 and height of 100:

* 10 depots → take 0.129s
* 20 depots → take 0.146s
* 50 depots → take 0.192s

# Project Visual

There’re a lot of aspects that need to take care in vehicle routing problem in real life.

For example, the maximum amount of packet that vehicle are carrying also affect the result. So vehicle might have to go back to station to refill packets and then continue to deliver it.

Also, in order to optimized profit, at the same time every vehicle should carry as much packet as possible, which would affect the result too.

Specific type of vehicle is considered a factor could affect the route because there’re roads only allow specific vehicle to go in.

# References

<https://en.wikipedia.org/wiki/Vehicle_routing_problem>

<https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm>

<https://en.wikipedia.org/wiki/2-opt>

<https://en.wikipedia.org/wiki/Travelling_salesman_problem>

<https://towardsdatascience.com/improving-operations-with-route-optimization-4b8a3701ca39>

<http://pedrohfsd.com/2017/08/09/2opt-part1.html>