FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Friday, March 26 2019

EcoPonto: Selective Waste Pickup

Conceção e Análise de Algoritmos
2018/2019

2MIEIC02_02:

Gonçalo Marantes Monteiro	up201706917@fe.up.pt
Gonçalo Fernandes Pereira	up201705971@fe.up.pt
João Renato Pinto	up201705547@fe.up.pt

Index

Introduction

From densely populated cities to smaller rural communities, waste management systems keep our homes and communities free from unwanted clutter.

Although these waste management services exist in nearly every community, the industry's current operating standards have proven inefficient and highly resource-intensive. This inefficiency is largely due to outdated manual collection methods and logistical processes which lack efficient data-driven solutions.

In this project, we aim to bridge the gap between the growing sensor network technology and trash pickup routes.

Project Specification

A waste management facility wants to develop an application that adaptively generates pickup routes in an effort to make the process the most resource saving as possible.

In the first place, there is the assumption that the garbage containers have a way to inform the waste management facility of their current capacity. This way it's possible to determine if the waste needs pickup or not, which is obviously an important factor when designing this application.

In the second place, there is the problem that a single truck may not have the capacity to load the garbage from all the containers, in that case it is not possible to go through every single container with one truck. We conclude there is the need for more trucks, and so the application will generate the best route for each truck individually.

Overall the application requires the following data:

- Garbage containers with information regarding their location and current waste levels;
- City streets network with road information and traffic rules (i.e, one way streets, bus-only streets, etc.)
- Garbage trucks parking location (starting point) and waste management facility location (final destination)
- Garbage truck information (i.e, capacity, average speed, etc.)

The final goal is to make this program applicable to real life scenarios, which is always a challenge. To achieve this we have decided to tackle this problem by separating it in 3 phases, each phase being more complex and requiring specifications from the previous ones.

Phase 1: Non-selective waste pickup with unlimited capacity trucks

In this first phase, the main goal is simply finding the best path from the truck parking lot to the waste management facility, going through all the valid waste containers with a single unlimited capacity truck.

It is important to point out that pickup can only be made if there are paths that connect all interest points (parking lot, garbage containers and waste facility), in pairs and both ways. So we easily conclude that a strongly connected graph is required for the best route plan. This need arises from the truck having to go through all interest points not in a specific order.

Phase 2: Selective waste pickup with unlimited capacity trucks

Although this phase was not requested, we have decided to add it to make the transition between phase 1 and phase 3 more accessible.

In this second phase, we begin to make selective trash pickup, which implies that there are at least four available trucks, one for each garbage container type: grey containers (regular garbage), green containers (glass), blue containers (paper) and yellow containers (plastic).

There is now the need to plan at least four routes, and to do that we will first require to process the graph from the first phase and generate four new graphs, one for each garbage container type. And now repeat phase one for each new graph.

Phase 3: Selective waste pickup with limited capacity trucks

In this third phase, there is now a new limitation in truck capacity. This means that a single truck can only collect a certain amount and type of garbage, so if the total amount of garbage to collect is higher than the capacity of a single truck, it will me necessary to deploy even more trucks that collects the same type of trash.

It is important to note that a truck cannot partially empty a garbage container, which means each container can only be fully emptied. So, if a truck still has some capacity left but it cannot empty a container in all, a new truck is needed to complete the pickup.

As a consequence of the capacity limitation we have two goals: minimize the total distance of truck routes and minimize the total number of trucks used to pick up waste.

Problem Formalization

Input Data

type - type of waste to collect

R_{max} - percentage/rate of garbage level in a container, from which it becomes admissible for pickup

T_i - sequence of trucks, such that T_k is truck number k:

$capacity_{max}$ - maximum truck capacity (during phase 1 and 2 $capacity = \infty$)

$G_i = (N_i, E_i)$ - weighted directed graph:

N - Nodes (represents interest points):

W - amount of waste to collect (0 if node is not a container)

W_{max} - maximum amount of waste the container holds

R - current percentage/rate of garbage level in a container

Adj - adjacent edges that belong to the node ($Adj \subseteq E$)

E - Edges (represents paths between interest points)

ID - unique identifier

d - edge's weight (distance between two interest points)

$dest$ - edge's destiny ($dest \in N$)

I - initial node (truck parking lot) ($I \in N$)

F - final node (waste management facility) ($F \in N$)

Output Data

$G_o = (N_o, E_o)$ - weighted directed graph, with N_o and E_o having the same characteristics has N_i and E_i.

T_o - sorted sequence of used trucks, such that T_k is truck number k:

$capacity$ - occupied truck capacity

P_j - ordered subset of N, represents the nodes to visit ($P_j \subseteq N$ and $0 \leq j \leq |P|$)

d_{total} - total distance between all nodes in P_j

Restrictions

Note that some data represent real life values (i.e. distances or weights) which means they cannot have negative values.

Input Data Restrictions

type: only four types of waste are available (paper, plastic, glass, regular).

$0 < R_{max} \leq 1$, percentage/rate of garbage level in an admissible container.

T_i:

- $\forall\ t \in T_i : capacity_{max}(t) > 0$, a trucks maximum capacity has to be greater than 0
- $|T_i| > 0$, since we need at least one available truck

$G_i = (N_i,\ E_i)$:

- Container Capacity: $\forall\ n \in N_i,\ W(n) \geq 0,\ W_{max}(n) \geq 0,\ 0 \leq R(N_i) \leq 1$;
- Distances between interest points: $\forall\ r \in E_i,\ d(e) \geq 0$;
- $\forall\ e \in E_i$, e must be a truck accessible road, otherwise it is ignored during pre-processing and not added to G_i.

Let $L = \{n \in N_i \mid n = I \vee n = F \vee W(n) > 0\}$ (set of nodes representing the initial and final nodes and garbage containers). Like mentioned above, L must be a strongly connected graph to find the optimal solution. Which means for each $N \in L$, we can reach any other node in the same set L.

Output Data Restrictions

$G_o = (N_o,\ E_o)$:

- N_o is a subset of N_i, so the same rules of N_i apply to N_o;
- E_o is a subset of E_i, so the same rules of E_i apply to E_o.

T_o:

- $\forall\ t \in T_o : capacity = \sum_{p \in P} W(dest(p)) * u$, $u = 0$ if garbage was already collected by another truck or garbage type is incompatible with truck's garbage type. Note that the shortest path between two containers might include other containers that are not supposed to be picked up by the truck in question;
- $|T_o| \leq |T_i|$, since we cannot use more trucks than the ones available

P_j :

- Let p_1 be the first element of P , $p_1 = I$ since the truck starts its route from the parking lot
- Let $p_{|P|}$ be the last element of P , $p_{|P|} = F$ since the truck ends its route in the waste management facility

Objective Function

The optimal solution is minimizing resources, which means we want to find the shortest path between the truck parking lot to the waste management facility, going through all the valid waste containers. However, above is mentioned that during phase 3 we start using limited capacity trucks which introduces a new objective function.

We conclude that we have two functions to minimize:

- $f = |T|$, the total number of trucks used;
- $g = \sum_{t \in T} d_{total}$, the total distance of all trucks' routes.

Since we have to choose one of the functions, function f will have priority over function g .

Implemented Algorithms

Dijkstra's Algorithm

The original Dijkstra's Algorithm has the ability to calculate the best path between any two nodes, and given its greedy nature and the fact that it always guarantees the shortest path, this algorithm becomes efficient and very simple to implement.

In this algorithm, the nodes have information of the previous node in the best path to the current node, as well as the total weight of all the edges up to the current node. Dijkstra's Algorithm has a similar behaviour of a Breadth-first Search. However, instead of using a regular Queue has an auxiliary data structure to store the order of the vertices, it uses a Priority Queue, in which the priority nodes are the ones with the smallest edge weight of the shortest path.

After finding the final node, the algorithm is complete and we proceed to a path reconstruction, starting from the Node from which the initial Node comes from and repeating this process until the initial node is reached.

Pseudocode

Next Page

Time Efficiency

This algorithm is divided in two phases: finding the best route from the initial to the final node and reconstructing that path through the information in the Nodes.

The first phase can be divided into two subphases. The first subphase consists in setting up the nodes for the algorithm execution, which has a time complexity of $O(|N|)$. The second subphase is the actual algorithm itself. As mentioned before this part is much like a Breadth-first Search with a change in the auxiliary data structure. This change however is big enough to alter its time complexity from $O(1)$ to $O(log\ N)$. This happens because insertions in a priority queue is not constant as opposed to a regular queue.

The second phase consists in going through all Nodes that build up the shortest path, which is completed in linear time (worst case) relative to the number of Nodes: $O(|N|)$

For these reasons, the final time efficiency of this algorithm is:

$$O((|N| + |E|) \times log\ |N|)$$

Space Efficiency

To run this algorithm we need to create a single container that amounts to a size of $|N|$, this way the space efficiency of this version of the algorithm is:

$$O(|N|)$$

```
Dijkstra(G, Ni, Nf)
    nodeQueue = {}  // priority node queue

    // populate nodeQueue
    for each Node n : G do
        if equals(n, Ni) then // initial node has a total shortest path edge weight of 0
            distance(n) = 0
        else
            distance(n) = infinite
        path(n) = null
        nodeQueue.insert(n)

    // Perform Search
    while notEmpty(nodeQueue) do
        n = nodeQueue.pop() // extract minimum (greedy)

        // Check for finish node
        if equals(n, Nf) then
            return buildPath(G, Ni, Nf)

        // check all adjacent nodes
        for each w : adjacent(n) do
            if distance(w) > distance(n) + weight(n, w) then
                distance(w) = distance(n) + weight(n, w)
                path(w) = n
                updateNodeQueue(nodeQueue, w)

    // loop ended with no solution found
    return null


buildPath(G, Ni, Nf)
    sortestPath = {}   // Node set that contains the nodes in shortest path order
    w = Nf      // distance(Nf) contains the total shortest path weight

    while w != Ni do
        shortestPath.pushFront(w)
        w = path(w)

    shortestPath.pushFront(Ni)

    return shortestPath
```

Nearest Neighbour

This algorithm is a constructive heuristic used to find a path through a set of nodes using a simple behavior: In each node we calculate the next on by finding the closest one from the set of unvisited nodes.

Pseudocode

```
NearestNeighbour(Node V)
    queue nodesSorted   // the remaining nodes sorted by distance relative to V
    result = []
    Vt ← V              // copy set of nodes into a temporary container

    while |Vt| > 0 do
        nodesSorted.resortRelativeTo(N)
        result.append(N)
        N ← result[0]
        nodesSorted.removeTop()

    return result
```

Time Efficiency

Given that the pseudo code above uses a sort algorithm for each nove V, and let $O(S)$ be the resulting time efficiency of the sort algorithm, we conclude that the resulting time efficiency will be:

$$O(V \times S)$$

Space Efficiency

To run this algorithm we need to create a single container that amounts to a size of $|V|$, this way the space efficiency of this version of the algorithm is:

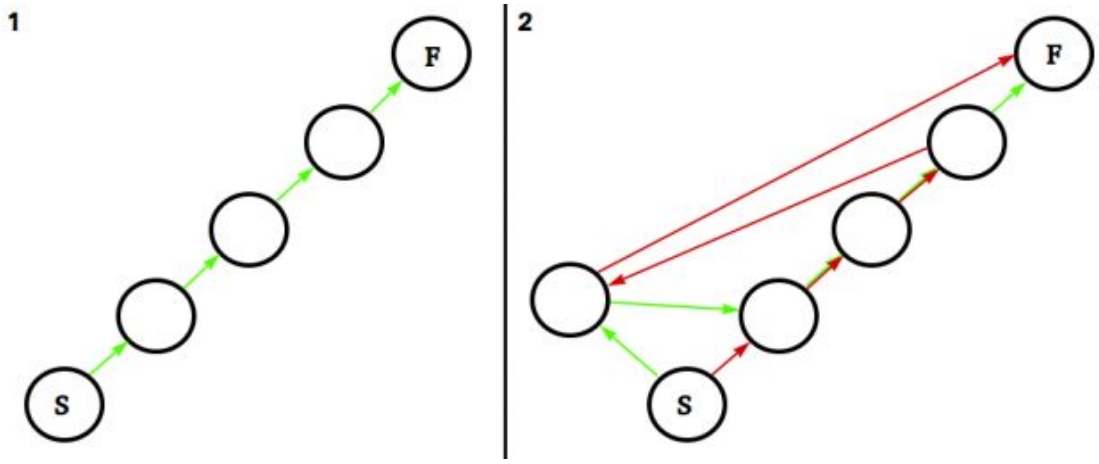
$$O(|V|)$$

Implemented Solution

This problem is very similar to an instance of the Travelling Salesman problem, with the restriction of having a starting and final node.

However, the only method to find the optimal solution is using *brute-force*, which means we need to test every single possibility and pick the best one. For this reason, this type of algorithm is very time complex.

However, there exists various heuristics such has the Nearest Neighbour that help with finding a solution that is linear in time. This algorithm consists in finding the closest Node, given a set of Nodes, from a starting node. However, this solution does not guarantee an optimal solution.



As we can see, in diagram 1, the Nearest Neighbour algorithm finds the solution in linear time. However, when looking at diagram 2 we see that a new node was added and the algorithm gives us the answer in red, however the shortest path is in green.

We also have to consider that in a directed graph, this algorithm does not always find a possible solution. This happens because the closest Node may not be able to access other interest nodes or the final node.

In an attempt to find the optimal solution, the implemented algorithm is recursive, progressively making the problem smaller, making use of backtracking strategies when the current node cannot access other nodes.

The pseudocode for this algorithm can be found in the next page.

```
// G - Graph, Ni - initial node
// Nf - final node, containers - set of containers (interest points)

TravellingSalesmanProblem(G, Ni, Nf, containers)
    visitOrder = {}
    calcVisitOrder(G, Ni, Nf, containers, visitOrder)

    // Check if visitOrder contains all containers
    if isComplete(visitOrder) then
        return buildPath(visitOrder, G)
    else
        return null

calcVisitOrder(G, Ni, Nf, containers, visitOrder)
    visitOrder.insert(Ni)

    // Check if it is possible to build a path from the given nodes
    if canAccess(Ni, containers) and canAccess(Ni, Nf) then
        nextVisit = findNearestContainer(Ni, containers, G)
        containers.remove(Ni)
        calcVisitOrder(G, nextVisit, Vf, containers, visitOrder)

        // check if the recursive call was successful
        if isComplete(visitOrder) then
            return
        else
            // unwanted Node, backtrack
            visitOrder.remove(Ni)
            return

    else
        // unwanted Node, backtrack
        visitOrder.remove(Ni)
        return

buildPath(visitOrder, G)
    finalPath = {}
    // find the shortest path from N0 to N1, N1 to N2, ..., Nn-1 to Nn
    for each Node in  visitOrder do
        // bestPath can be Dijkstra Algorithm
        finalPath.append(bestPath(Node, nextNode))

    return finalPath
```

As we can see from the pseudocode, this algorithm is composed of two main phases.

During the first phase we calculate, if possible, the order of all containers to visit using the Nearest Neighbour heuristic. To calculate this order we use a recursive algorithm that requires the use of backtracking algorithms to communicate eventual impossible routes.

In the second phase we calculate the best route between the vertices of *visitOrder*, in the order they appear in the set using a shortest path between two nodes, like Dijkstra's Algorithm. That is, if *visitOrder* is composed of nodes N_i, N_1, N_2, N_3, ..., N_n, N_f, the calculated best route between N_i and N_f is composed of the best path between N_i and N_1, between N_1 and N_2 and so on until we reach N_f.

Time Complexity

During the first phase, it might be necessary to do a Depth-first search in all the graph's Nodes (in the case that all of the nodes are containers). For this reason and in the worst case scenario, the search can be done $|N|$ times. Since the time complexity of a Depth-first search is $O(|N| + |E|)$, the time complexity of this phase is $O((|N| + |E|) \times |N|)$ in the worst case.

Regarding the second phase of this algorithm, in the worst case the set *visitOrder* contains all the graph's Nodes, so it is necessary to use Dijkstra's Algorithm $|N| - 1$ times, that has a time complexity of $O((|N| + |E|) \times log\ |N|)$. So the time complexity of this phase is $O((|N| + |E|) \times log\ |N| \times |N|)$.

Final Time Complexity: $O((|N| + |E|) \times log\ |N| \times |N|)$

Space Complexity

This algorithm uses as an auxiliary data structure a vector containing the order of containers to visit. For this reason, the algorithm's space complexity is equal to $O(|N|)$.

Phase 1: Non-selective waste pickup with unlimited capacity trucks

This problem is the simplest one. Essentially, since the prioritized objective function is to minimize the number of trucks used and, during these early phases, the trucks have unlimited capacity we shall use only a single truck.

This approach can be tested using the implemented solution's algorithm such that the containers vector is made up of all containers which have a garbage level (R) greater or equal to the stipulated max garbage level (R_{max}) not paying attention to its type

Phase 2: Selective waste pickup with unlimited capacity trucks

This phase is very similar to phase 1, only now we have four types of waste that can only be picked up by the same type trucks. So essentially, we are repeating phase 1 four times, one for each waste and truck type.

Phase 3: Selective waste pickup with limited capacity trucks

Finally, this is the most complex phase, such that each truck has a maximum volume and will, at times, be insufficient to collect every container.

To achieve the optimal route for a truck such that it travels the shortest distance possible and at the same time collects as many containers as possible we need to update our *backtracking* algorithm. That is, after we have checked if a Node N can access other containers (line 18 *if canAccess(*N_i*, containers)* ...) we also need to check if the truck can still pick up any leftover containers. If so we continue with the algorithm, if not we dispatch it to the waste management facility and repeat the process for another truck of the same waste type until all valid containers have been collected.

Data Structures

Container

The container can have 2 types, a Waste Container or a Recycling Container, and it has the information of its Maximum Capacity and its Current Capacity, so the algorithms can be more efficient about the amount of Containers it can pass without exceeding the Truck Maximum Capacity.

Truck

The Truck also has the same 2 types, its Maximum Capacity and Current Capacity but it also holds the information about the Containers where it has to collect trash, and the path of maximum efficiency that has all those Containers. It also has the path color in which the edges will be colored, the Waste Path is colored in Orange and the Recycling Path is in green.

Edge

The Edge has its Id, the Destination Node's Id and the distance between the nodes it connects.

Node

The Node holds its location (x and y) an Id, a Type from a enum that has the nodes tags. Also contains the information about the edges it is connected to.

DijNode

The DijNode extends the Node, so it has all the information that the Node has, and also the total Weight (the distance since the start node) and it also has the Id of the last node it is connected to.

Graph

The maps that were provided were not the best to test the algorithms because of their connectivity, a lot of containers and other nodes were dead ends or isolated from the rest, so we decided to use all edges as bidirectional, and we created our own simple map, so it makes testing easier.

Conclusion

In conclusion, the aim of this paper was to discuss a possible approach to the problem of route planning much similar to the famous Travelling Salesman Problem. This problem was reduced to shortest paths algorithms, backtracking algorithms and greedy algorithms.