The Algorithm for Constrained Shortest Path Problem Based on Incremental Lagrangian Dual Solution

Boris Novikov and Roman Guralnik(✉)

Saint Petersburg State University, Universitetskaya emb. 7/9, 199034 Saint-Petersburg, Russia
romasha_nar@mail.ru

Abstract. Most of the systems that rely on the solution of shortest path problem or constrained shortest demand real-time response to unexpected real world events that affect the input graph of the problem such as car accidents, road repair works or simply dense traffic. We developed new incremental algorithm that uses data already present in the system in order to quickly update a solution under new conditions. We conducted experiments on real data sets represented by road graphs of the cities of Oldenburg and San Joaquin. We test the algorithm against that of Muhandiramge and Boland [1] and show that it provides up to 50% decrease in computation time compared to solving the problem from scratch.

Keywords: Incremental · Constrained shortest path · Road graphs

1 Introduction

With graph databases being one of the central representations of big data, we focus our attention on their dynamic version in the form of dynamic road graphs. Calculating a pathway between two points is an essential combinatorial optimization problem not only as a stand-alone problem, but also as a subtask for a series of other more complex optimization problems. The list of examples includes the travelling salesman problem and its counterpart – vehicle routing problem [2, 3]. In the latter problem a route is constructed from several points of interest. In case these points are represented by some graph nodes, it is almost always assumed that the distance between two points is designated by the graph shortest path between them. Other problems include detecting arbitrage opportunities in currency markets, pathfinding problems that are used, for instance, by AI (artificial intelligence) to plot routes or by video game engines to assist users in plotting route.

Several complex applications of shortest path problem require to approach real life conditions and to consider the resource constrained shortest path problem (RCSPP), where graph edges, besides edge cost, are also associated with the resource, which is consumed upon travelling through this edge. The solution path summary resource consumption should fall within the range of $[0, W_L]$. A wide set of problems which require solving constrained shortest path as a subtask include problems of long-haul aircraft and truck routing [4], military path planning under resource constraints [5], crew scheduling problems [6], pipeline and valve location [7], designing telecommunication network with relay nodes [8]. RSCPP was extended by Smith et al. [9], introducing replenishment edges, which reset consumed resource to zero upon travelling a replenishment edge. Such setting is convenient and used in aircraft routing.

To get even closer to real life conditions we decided to consider constrained shortest path problem applied to real traffic events that may occur on the road. These events may include regular traffic jams, car accidents, road line repair works, road closures etc. We assume that we have calculated (by using an algorithm that we will refer to as *the baseline* algorithm, which implies that we will propose another one that in some way improves this baseline algorithm) a constrained shortest path from desired a source to a desired destination and then one (or several) of the mentioned events take place. All of these events imply that car flow through this road is impaired, that is the number of cars that pass through this segment of road per particular unit of time as well as average speed of travelling through the segment are reduced.

To model this behavior of the system we use resource constrained shortest path setting, where the cost of the edge denotes L_2 distance of the road segment and the weight of the edge represents traffic flow with inverse proportion (the greater the weight of the edge – the less is the flow through this edge). This means that we have to recalculate our solution path from the source to the destination but now taking into account that some of the edges have increased their weight. In cases where the changes in graph edge weights are not heavy (i.e. only a small amount of edges have changed their weights) but still affect our optimal solution it may prove highly inefficient to perform all calculations from scratch. With that in mind, we developed new incremental algorithm to utilize data obtained during the initial run of the baseline algorithm.

Our algorithm is influenced by the algorithm of Muhandiramge and Boland [1] and utilizes the similar preprocessing method, when the solving of Lagrangian dual is integrated with network reduction. Moreover, we combine the above-mentioned algorithm with modified version of the algorithm of Pallottino and Scutella [10] for reoptimizing shortest path trees. This will allow us to use already calculated trees to find new trees with changed edge weights.

2 Related Work

Resource constrained shortest path problem has been extensively studied for several past decades. In most of the works the optimal solution for RCSPP is obtained in three step approach: preprocessing, network reduction and gap closing. The work by Aneja et al. [11] was published in 1983 and is considered to be the first to utilize preprocessing. Its main idea lies in removing nodes and edges that cannot be a part of optimal solution by checking for feasibility every path through considered node or edge.

Several more recent papers use the above-mentioned three step approach. Beasly and Cristofides [12] apply the similar preprocessing procedure but also perform node checks with reduced cost and best lower bound obtained by solving the Lagrangian dual. Dumitrescu and Boland [13] used the same preprocessing but considered regular costs instead of Lagrangian relaxed costs. They conducted testing on sparse network and achieved heavy reductions in the size of the graph. Mehlhorn and Ziegelmann [14] suggest an algorithm with more effective preprocessing due to obtaining better upper bound.

Muhandiramge and Boland [1] were the first to suggest an algorithm that combines preprocessing and Lagrangian dual solving in one step, thus offering a two-step approach. To solve Lagrangian dual Kelley's Cutting Plane Method (KCPM) is used on the set of all Lagrange multipliers. In the gap closing phase a modified version of Carlyle's and Wood's [15] enumeration is utilized and represents a depth-first branch and bound search that uses shortest path trees for optimal Lagrange multiplier to perform tests and fathom branches. Muhandiramge and Boland [1] also offer an SPT reoptimization algorithm that recalculates SPT if it is necessary. Their reoptimization algorithm employs Dijkstra's shortest path algorithm, starting from some advanced point and does not consider changes in edge weights.

As for incremental approaches in regular and constrained shortest path, several algorithms have been suggested in the past 30 years. Gallo [16] proposed the first algorithm to recalculate shortest paths but only for particular cases when the source vertex has changed or exactly one edge has lowered its cost. Ramalingam and Reps [17], King and Thorup [18], Demetrescu [19] provided several reoptimization algorithms for cases when exactly one edge change its weight (in any direction). Pallotino and Scutella [10] devised a method to reoptimize single-source shortest path tree in two-phase approach, dealing with edges that increase and decrease their weights separately. The work by Zhu [20] provides reoptimization algorithm for shortest path tree in an acyclic graph. They consider a case where RCSPP is a subproblem in column generation, i.e. edges change their costs instead of weights as in our problem definition. Our algorithm also deals with graph reductions that is when we need to update shortest path tree on the graph with different number of vertices.

3 Problem Definition

Let $G = (V, A)$ be a directed graph, where V is the set of nodes and A is the set of edges. We give every node a label $i \in Z^+$ - a set of positive integers, so every edge by default gets labeled as (i, j) with i being the source and j being the target of the edge. We denote with s and t the source and target for the whole problem.

Every edge of the graph is associated with two real non-negative values: cost and weight. This can be described with functions $C : A \to R^+$ and $W : A \to R^+$. We note here that our algorithm can be easily applied to the case with $W : A \to (R^+)^n$.

Path p from node i_1 to node i_k is a sequence of edges $(i_1, i_2), \ldots, (i_{k-1}, i_k)$ s.t. $(i_{l-1}, i_l) \in A, \forall l = 1, \ldots, k$. We refer to the cost and weight of the path p as $C(p)$ and

$W(p)$ respectively. If we denote with P_G the set of all paths from s to t then the regular RCSPP would imply the search for such path p^* that

$$C(p^*) = \min_{p \in P_G} C(p) \tag{1}$$

$$\text{s.t. } W(p) \leq W_L \tag{2}$$

where W_L is the global weight limit for the path, designated by problem input.

Next, we suppose that constrained shortest path problem was solved with weight function W and we have to apply now the new weights to graph edges. Due to the way the definitions are set, we don't have to apply changes to the graph itself in order to introduce incrementality, but can rather say that we still have the same graph G and new weight function $W^{'} : A \rightarrow R^{+}$. Considering real life everyday traffic events, we utilize the following statement:

$$\forall e \in AW^{'}(e) \geq W(e). \tag{3}$$

The main idea of incremental RCSPP is formulated as follows. Suppose we solved the RCSPP problem stated with weight function W with some baseline algorithm. We propose a new incremental algorithm for RCSPP to solve the problem under new weight function W' using data obtained with the baseline algorithm. The proposed algorithm is faster than the one that solves the new problem from scratch.

4 Lagrangian Dual and Baseline Algorithm

Our incremental algorithm utilizes the data obtained by the initial run of the base algorithm of Muhandiramge and Boland [1] to give preprocessing a head start. It initializes the problem from advanced point and then continues with solving the Lagrangian dual. This section will provide a brief overview of the Muhandiramge and Boland algorithm in order to explain what data from baseline algorithm run we use and how we do it. For more detailed exposition the reader is referred to [1]. As we mentioned earlier, the algorithm of [1] consists of only two phases – solving the Lagrangian dual and gap closing – since preprocessing (or network reduction) is integrated in the first phase.

4.1 Simple Node Elimination

As for the first phase of [1], there are several options for network reduction. We decided to dwell on the basic part of it called Simple Node Elimination (SNE) and to create its incremental counterpart.

To set up a Lagrangian dual problem for RCSPP a relaxed weight constraint function is introduced

$$\mathcal{L}(\lambda, p) = C(p) + \lambda(W(p) - W_L), \tag{4}$$

where coefficient $\lambda \geq 0$ is known as Lagrange multiplier. For the sake of brevity the *reduced cost function* is sometimes introduced as

$$\Lambda(\lambda, a) = (C + \lambda W)(a) \tag{5}$$

for every edge $a \in A$ and

$$\Lambda(\lambda, p) = \sum\nolimits_{a \in p} (C + \lambda W)(a) \tag{6}$$

for every path p.

Next $\mathcal{L}(\lambda, p)$ is minimized over $p \in P_G$ to obtain Lagrangian dual function

$$\Phi_G(\lambda) = \min\nolimits_{p \in P_G} \Lambda(\lambda, p) - \lambda W_L. \tag{7}$$

It is known that for every $\lambda \geq 0$ Lagrangian dual function provides a lower bound for the value of the target primal function, hence we want to maximize $\Phi_G(\lambda)$ over all non-negative lambdas to obtain greatest lower bound. For a particular value of λ the value of $\Phi_G(\lambda)$ can be calculated by solving unconstrained shortest path problem. By construction, $\Phi_G(\lambda)$ is a piecewise concave function so the authors of [1] rely on modified Kelley's cutting plane method (KCPM). They introduce the convenience notation of minimum cost path form i to j with respect to reduced cost function Λ, found by shortest path calculator, and denote it as Q_{ij}^{λ}. Q_{ij}^{∞} denotes the minimum weight path. Detailed discussion on KCPM is provided in [1].

To combine preprocessing and solving of the Lagrangian dual Muhandiramge and Boland [1] integrate KCPM with network reduction in the following way.

First, an initialization procedure is performed that sets $\lambda^{+} = 0$ and $\lambda^{-} = \infty$ (values, maintained by KCPM) and for each value calculates its forward and reverse shortest path tree. Forward shortest path tree (SPT) is a tree that stores a shortest path from source vertex (vertex s) to every other vertex in the network, reverse SPT stores paths from every vertex to target vertex (vertex t). This procedure not only verifies that the problem is feasible and non-trivial but finds a feasible solution (if there are any) represented by minimum weight path. Since every feasible solution provides an upper bound whereas Lagrangian dual value provides a lower bound, network reduction can be performed for every node independently. Suppose we have $P_{G,k}$ – a set of all paths from s to t through vertex k. Now let us consider current problem upper bound U and a Lagrangian dual

$$\Phi_{G,k}(\lambda) = \min\nolimits_{p \in P_{G,k}} \Lambda(\lambda, p) - \lambda W_L. \tag{8}$$

If for some $\lambda \geq 0$ it appears that $\Phi_{G,k}(\lambda) \geq U$ then node k can be eliminated from the graph with all of its adjacent edges. It is important to note that for a certain value of λ its forward and reverse SPTs provide all shortest paths through each vertex of the graph (to get such a path through k we can concatenate the one from s to k. obtained from forward SPT with that from k to t obtained from reverse SPT).

The algorithm inserts the elimination checks for every vertex after calculation of λ_{new} and its SPTs. To increase elimination efficiency, this step is preceded by traversing the graph in search of feasible paths at every node in an attempt to improve current upper bound. Then lower bound is calculated for every node and the elimination checks are performed.

We state here full pseudo-code of SNE as provided by Muhandiramge and Boland. We use steps from 3 to 8 (as well as incremental SPT update) to continue solving KCPM for our incremental preprocessing, described in Sect. 7.

```
1.  Do $initialization\ procedure$ (check if the problem is feasible and
    non-trivial and of so: set $U = C(Q_{st}^{\infty})$ and $pU = Q_{st}^{\infty}$ where $pU$ is the
    upper bound path)
2.  $\lambda^{+} = 0$
    $\lambda^{-} = \infty$
    $L = (\lambda^{+}, \lambda^{-})$
3.  $\lambda_{new} = \frac{C(Q_{st}^{\lambda^{+}}) - C(Q_{st}^{\lambda^{-}})}{W(Q_{st}^{\lambda^{-}}) - W(Q_{st}^{\lambda^{+}})}$
    $\mathcal{L}_{new} = C(Q_{st}^{\lambda^{+}}) + \lambda_{new}(W(Q_{st}^{\lambda^{+}}) - W_L)$

$Calculate\ forward\ and\ reverse\ SPTs, \Phi_G, \Phi'_G\ for\ \lambda_{new}, add\ \lambda_{new}\ to\ L$
4.  For $each\ node\ k \in V,$
        If $C(Q_{sk}^{\lambda_{new}} + Q_{kt}^{\lambda_{new}}) < U\ and\ W(Q_{sk}^{\lambda_{new}} + Q_{kt}^{\lambda_{new}}) \leq W_L$
            $U = C(Q_{sk}^{\lambda_{new}} + Q_{kt}^{\lambda_{new}})$
            $pU = Q_{sk}^{\lambda_{new}} + Q_{kt}^{\lambda_{new}}$ (new upper bound has been found)
5.  $V' = \emptyset\ and\ A' = \emptyset$ (vertices and edges that will be deleted)
    For $each\ node\ k \in V$
        If $\Phi_{G,k}(\lambda) \geq U,$
            $V' = V' \cup \{k\}$
            $A' = A' \cup \{all\ edges\ adjacent\ to\ k\}$
      $V = V \backslash V' and\ A = A \backslash A'$
      $G = (V, A)$
6.  If $V = \emptyset$, STOP (the current upper bound is the optimal solution to
    RCSPP)
    If $\Phi_G(\lambda_{new}) = \mathcal{L}_{new}$, STOP (OLM found)
7.  If $\Phi'_G(\lambda_{new}) \leq 0$
        $\lambda^{-} = \lambda_{new}$
        If $SPTs\ for\ \lambda^{+}\ are\ out\ of\ date, recalculate\ SPTs\ for\ \lambda^{+}$
        If $\Phi'_G(\lambda^{+}) \leq 0\ and\ \lambda^{+} \neq 0,$
```

```
                **Repeat**
                    $Set\ \lambda^+ to\ the\ next\ lowest\ value\ of\ \lambda\ in\ L$
                    $Recalculate\ its\ SPTs\ if\ out\ of\ date$
                **Until** $either\ \Phi'_G\ (\lambda^+) > 0\ or\ \lambda^+ = 0$
            **If** $\lambda^+ = 0\ and\ \Phi'_G\ (0) \leq 0,$ **STOP** (current upper bound is optimal)
        **Else**
            $\lambda^+ = \lambda_{new}$
            **If** $SPTs\ for\ \lambda^-\ are\ out\ of\ date, recalculate\ SPTs\ for\ \lambda^-$
            **If** $\Phi'_G\ (\lambda^-) > 0\ and\ \lambda^- \neq \infty,$
                **Repeat**
                    $Set\ \lambda^- to\ the\ next\ highest\ value\ of\ \lambda\ in\ L$
                    $Recalculate\ its\ SPTs\ if\ out\ of\ date$
                **Until** $either\ \Phi'_G\ (\lambda^-) \leq 0\ or\ \lambda^- = \infty$
            **If** $\lambda^- = \infty\ and\ \Phi'_G\ (\lambda^-) > 0,$ **STOP** (no feasible path in current
    network, current UB is optimal)
    8.  **Goto** $step$ **3**
```

Step 7 is necessary since all of the calculated SPTs (except λ_{new}) become obsolete after the reduction of the network in step 4, which for the new graph may result in $\Phi'_G(\lambda^-) > 0$ or $\Phi'_G(\lambda^+) \leq 0$. To address this issue the authors of [1] make use of the ordered set L of all lambdas for which SPTs were calculated. This allows to keep λ^+ and λ^- correct throughout the algorithm. For additional clarifications on Simple Node Elimination the reader is referred to the original article [1].

4.2 Gap Closing

For the second phase, which is supposed to close the gap between lower and upper bounds, Muhandiramge and Boland chose to utilize the depth-first branch and bound approach facilitated by fathom tests. This method starts from the source vertex and iterates through graph with a depth-first approach, building a path from visited edges. The fathom test is applied to every branch (the branch is represented by a certain edge) under consideration and depending on the outcome of the test the branch is pruned, i.e. is removed from further considerations since it cannot be in the optimal path. If dfbb (depth-first branch and bound) managed to reach the target vertex the algorithm checks the cost of the path, which was used to reach the target, and, if necessary (i.e. the cost of this path is less than current upper bound), updates upper bound and the result path.

The fathom test uses a procedure similar to KCPM's search for optimal Lagrange multiplier. Keeping in mind that $\Phi(\lambda)$ is concave and having the ordered set $\overline{L} = (\lambda_0, \ldots, \lambda_1)$ of lambda values for which we have SPTs the algorithm performs a bisection search to find the best available lambda (for which $\Phi(\lambda)$ is the greatest). The explicit pseudo-code for fathom test as well as the gap closing phase is given in [1].

5 Incremental SPT Update

As a starting point we assume that the baseline algorithm finished its work, reduced the network and found an optimal path. As the problem statement claims, we need to apply now the new weight function to our graph (that is, to increase the weights of some edges). Because of this, the reduction of some nodes and edges may prove invalid. That means we cannot use the current graph and have to perform new network reduction on the initial full graph. For the reasons stated above in this research we focus our attention on the first phase in an attempt to perform fast network reduction and optimal Lagrange multiplier search based on the data computed in the course of the baseline algorithm. Due to the article size limitations we omit detailed discussions on the gap closing phase.

As we may infer, the largest computational overhead of SNE lies in computing and recomputing shortest path trees. A* (or Dijkstra's) is suggested as a default SPT calculator, but, for example, Dijkstra's $O(|A| + |\mathrm{V}| \log|V|)$ for a single SPT calculation can prove inefficient for big graphs, considering the fact that all calculations have to be performed in real time and provide solutions as fast as possible. It is important to notice that the authors of [1] also provide their own algorithm for updating SPT. However, this algorithm is designed to only recalculate SPTs after network reduction. This can be regarded as the special case of changing edge weight, when edge weight is set to be equal to ∞ (i.e. delete the edge). For that reason such algorithm cannot be applied to general case of edge weight increase. To address this issue we decided to use slightly modified version of Pallottino and Scutella [10] algorithm for reoptimizing shortest path tree. Below, we will provide a brief overview of the algorithm.

5.1 Incremental SPT Update Algorithm

Since the algorithm operates only with unconstrained edge costs, it is convenient to denote a cost of an edge (i,j) as c_{ij} keeping in mind that every SPT is tied to a certain value of λ, so $c_{ij} = C(i,j) + \lambda W(i,j)$. Dijkstra algorithm also provides us with the potential π_i of each node (the weight of the shortest path from SPT root r to this node) as well as reduced cost $\overline{c}_{ij} = c_{ij} + \pi_i - \pi_j$.

Algorithm also receives as input an SPT $T_r = (V, A_r)$ as well as new costs c'_{ij}. For T_r to be a shortest path tree it has to satisfy a complementary slackness condition (CSC), that is $\overline{c}_{ij} = 0, \forall (i,j) \in A_r$. If the new edge costs $\overline{c}'_{ij}$ satisfy CSC for every $(i,j) \in A_r$ then the problem is trivial and T_r is returned as an updated SPT.

Otherwise, forest $F_r = (N, A_F)$ is obtained from T_r by removing all edges that do not satisfy CSC ($\overline{c}'_{ij} > 0$). That means that $A_F = \{(i,j) \in A_r : \overline{c}_{ij} = 0\}$. Root subtree $T(r) = (N(r), A(r))$ contains SPT root r and does not need reoptimization. Other subtrees are reconnected to $T(r)$ through a series of "hanging operations" described below.

Consider a cut $\left(N(r), \overline{N}(r)\right)$ of the vertex set, where $\overline{N}(r) = N \backslash N(r)$. Hanging operation defines two sets:

$$A^+ = \left\{(i,j) \in A : i \in N(r), j \in \overline{N}(r)\right\}, \tag{9}$$

known as the set of *border edges*, and

$$A^E = \left\{(i,j) \in A : i \in \overline{N}(r), j \in \overline{N}(r)\right\}, \tag{10}$$

known as the set of *external edges*.
For every node $j \in \overline{N}(r)$, the following values are calculated

$$\delta_j = \min\left\{\overline{c}'_{ij} : (i,j) \in In(j) \ \cap A^+\right\}\alpha_j = \min\left\{\overline{c}'_{ij} : (i,j) \in In(j) \ \cap A^E\right\}$$
$$\delta_j^+ = \min\{\delta_i : i \in N^+(j)\}\delta_j^- = \min\{\delta_i : i \in N^-(j)\}$$
$$\alpha_j^+ = \min\{\alpha_i : i \in N^+(j)\}\alpha_j^- = \min\{\alpha_i : i \in N^-(j)\},$$

where $In(j)$ denotes a set of all incoming edges of j, $N^+(j)$ and $N^-(j)$ denote sets of nodes which are ancestors and descendants of j respectively. These values are assumed to be ∞ if the corresponding set is empty.

Next, *gap* Δ is defined for the cut $\left(N(r), \overline{N}(r)\right)$ as

$$\Delta = \min\left\{\delta_j : j \in \overline{N}(r)\right\} \tag{11}$$

and node w for which $\delta_w = \Delta$.

Finally, node v is considered *hangable* if it satisfies the following inequalities:

$$\delta_v \leq \min\left\{\delta_v^+, \delta_v^-\right\} \tag{12}$$

and

$$\delta_v \leq \Delta + \min\left\{\alpha_v^+, \alpha_v^-\right\} \tag{13}$$

If the conditions are true, node v and its subtree $T(v) = (N(v), A(v))$ are hanged to the root subtree $T(r)$ through the edge $(u, v) \in A^+$, such that $\overline{c}'_{uv} = \delta_v$. After the hanging we have:

$$N(r) := N(r) \ \cup N(v) \tag{14}$$

$$A(r) := A(r) \ \cup A(v) \ \cup (u, v) \tag{15}$$

$$\pi_i := \pi_i + \delta_v, \forall i \in T(v) \tag{16}$$

Not only a set of hangable nodes is never empty, it can also contain multiple vertices and their subtrees can be hanged in parallel.

After hanging all currently hangable vertices, the values of alphas and deltas are updated and new hanging operations can be performed. The algorithm stops when all vertices are in the root subtree. More detailed discussions on SPT update and hanging operations can be found in [21, 22].

6 Incremental SPT Update Complexity in the Scope of RSCPP

Now if we return to RSCP problem and try to apply SPT update algorithm in the form stated above, we may face the following difficulty. The current graph on which we want to have an updated SPT can have different set of vertices, compared to the graph on which input SPT has been calculated. This can happen if either of the graph has undergone reduction in the course of KCPM. Since it will cause problems in the entire algorithm logic, we have to address this issue by considering two cases. First case represents an event when new graph does not contain all the vertices from the old graph, that is $\exists i \in V_{old} : i \notin V_{new}$. Second case represents the opposite event when $\exists i \in V_{new} : i \notin V_{old}$. There is actually a third option for the case when both graphs were reduced and now contain different sets of vertices. However, this case is just a combination of the first two and does not require to be handled independently.

First case does not require much attention since it is rather trivial. During initial forest F_r and root subtree $T(r)$ construction our algorithm deletes not only nodes that do not satisfy CSC but also nodes that are not contained in new graph vertex set V_{new}. The latter nodes (which are not in V_{new}) and there adjacent edges also don't participate in hanging operations.

As for the second case, let us consider nodes that are in V_{new} but not in V_{old}. Let us call them e-nodes (from extra nodes). Obviously such nodes fall into $\overline{N}(r)$ and require to be hanged. Moreover, if a node i in $N(r)$ has an adjacent edge (k, i) and node k is an e-node (node i has an edge incoming from e-node), then node i is deleted from $T(r)$ and is also required to be hanged. The latter action is required to keep optimality of the SPT, since that adjacent e-node can offer a better shortest path, than the current one for the node in question.

It is obvious, that the great number of e-nodes can seriously impair computational efficiency of SPT updated. Default complexity for SPT update through hanging operations is $O(m + Rn)$, where m and n are the numbers of edges and vertices respectively and R is the number of hanging iterations (remember that each hanging iteration can hang several subtrees in parallel). For our modified version this would mean a complexity of $O(m_{new} + Rn_{new})$. If there is many e-nodes the number R will be great as well and in the worst case can be equal to n, making worst case complexity of $O\left(m_{new} + n_{new}^2\right)$ which is worse than computing SPT from scratch with Dijkstra's algorithm. However, in practice, Dijkstra's algorithm outperforms incremental SPT update only when we are trying to update SPT calculated on heavily reduced graph, and the new graph being from almost full to non-reduced at all. Of course, we can think of at least one example when this could happen. As soon as the edge weights have changed, the first step of preprocessing performs a test for the problem being trivial. This test is usually done by calculating SPTs for $\lambda = \infty$ (testing if minimum weight path is feasible, otherwise – the problem is infeasible) and for $\lambda = 0$ (testing if minimum cost path is feasible, and if so – the optimal solution for the problem is the minimum cost path). The difficulty may arise if during baseline algorithm SPTs for $\lambda = \infty$ and $\lambda = 0$ were recalculated for heavily reduced graph. So, to avoid recalculation of SPTs calculated on heavily reduced graph and to facilitate the initialization of the incremental RCSPP our algorithm stores two additional SPTs for $\lambda = \infty$ and $\lambda = 0$ and in particular the ones that were computed on non-reduced graph and performs updates using these SPTs.

Now we can discuss the case, were no e-nodes are present. Of course, in this case we know that $m_{new} \leq m_{old}$, $n_{new} \leq n_{old}$ and complexity of incremental SPT update (the number of hanging operations R) will depend on the number of edges that has changed their weight. Worst case complexity is still $O\left(m_{new} + n_{new}^2\right)$, but we have to remember here that we consider real life events such as car accidents, road line repair works or road closing etc. Hence, it can be inferred that edge weight changes are not scattered randomly across the graph but rather congregated around particular edges. For incremental SPT update that means that lots of previously calculated SPT subtrees remain intact which allows not only to hang subtrees with many nodes to the root subtree but also to hang many subtrees at once (in parallel) and in the end outperform Dijkstra's algorithm. Indeed, if, for instance in some SPT a path from i to j contains three edges with changed weight and these edges are adjacent, then everything before and after this "congregation" does not need to be recomputed and will be present as is in the updated SPT.

As experiments show, there are certain thresholds on the amount of edges that change their weights. These thresholds signify when it is more efficient to use incremental SPT update than to recalculate SPT from scratch. Due to the lack of article space we shall not dwell on the discussions about exact values for the above-mentioned thresholds and move to the discussion of the algorithm itself.

7 Incremental Preprocessing Pipeline

In this section we will provide a pseudo-code for the preprocessing step and explain its general steps.

To perform efficient preprocessing using the data obtained by baseline algorithm, we would like to start from some advanced point, that is, we would like to use some good upper and lower bounds and use them in network reduction.

We start with checking a problem for being feasible or trivial. Normally we would have to do that by updating SPTs for $\lambda = \infty$ and $\lambda = 0$ using SPTs calculated on the full graph. We can notice though, that an update for $\lambda = 0$ does not have to happen here, if the global weight limit stays the same. This is true because minimum cost path stays the same even after weight changes. Since we work with weight increases, we can infer that new problem is non-trivial if the old problem is non-trivial. The reverse statement, however, is not true. Looking for an upper bound we can always rely on minimum weight path, provided it is feasible, if we have recalculated SPTs for $\lambda = \infty$. However, to obtain better upper bound and to possibly avoid recalculation of SPTs we suggest the following procedure.

During the gap closing phase of baseline algorithm, the depth-first branch and bound (dfbb) approach is used to traverse the graph in search of feasible solutions. Every feasible solution found by dfbb is stored for the use in incremental preprocessing. After applying new weight function to edges, we can check every solution for feasibility. We assume that the path with minimum cost out of all feasible solutions (the one that was optimal for the old graph) is infeasible, because the problem would be trivial otherwise, if this path would retain its optimality. However, if any of the remaining feasible solutions appears to be feasible under new weight function, then we can use it as an upper bound for preprocessing and as an indication of problem feasibility. This will allow us to avoid recalculation of SPTs for $\lambda = \infty$ during initialization.

To perform incremental preprocessing we need to find new optimal Lagrange multiplier, since it could change after applying new weights to the edges, and then perform reduction. However, since we have all the information obtained in the course of baseline algorithm, we have a list of values for λ and their, albeit invalid now, SPTs. For that reason we don't need to start KCPM from scratch, that is with $\lambda^{+} = 0$ and $\lambda^{-} = \infty$. To give our preprocessing a head start we can recalculate SPTs for a particular value of λ_{inc} from the list of available lambdas, perform network reductions (since recalculated SPTs will provide us with a lower bound) and continue KCPM setting $\lambda^{+} = \lambda_{inc}, \lambda^{-} = \infty$ or $\lambda^{-} = \lambda_{inc}, \lambda^{+} = 0$ depending on the sign of $\Phi^{'}(\lambda_{inc})$.

The question may arise: what value for λ_{inc} may be optimal? One seemingly logical option would be to choose OLM obtained by baseline's KCPM. However, it may turn out that SPTs for this value might have been calculated on the graph already heavily reduced, which renders recalculations of such SPTs highly inefficient due to the large number of e-nodes. For that reason in our algorithm we were not using OLM obtained in baseline's KCPM. In the course of our experiments on SNE, we made an interesting observation related to values of λ and graph reductions. Let us denote as q_r the number of nodes that were removed from the graph during r-th iteration of KCPM. We noticed that during several initial iterations of KCPM, network reductions can be trivial, i.e. no nodes and edges are deleted. Next, let us assume that first non-trivial reduction happened at r_{nt}-th iteration and we removed $q_{r_{nt}}$ nodes. We observed, that in large majority of cases after r_{nt}-th iteration the amount of nodes removed during network reduction is always less than $q_{r_{nt}}$, that is

$$\forall r > r_{nt}, q_r < q_{r_{nt}} \tag{17}$$

It means that the largest number of nodes is deleted during first non-trivial reduction. Now if we denote as λ_{nt} the value of lambda, which was used to perform first non-trivial graph reduction, we can be certain, that for λ_{nt} baseline algorithm calculates SPTs on the non-reduced graph. We store these SPTs, as well as the value of λ_{nt} right after first non-trivial reduction takes place, since we cannot be sure that these SPTs would not be recalculated on some reduced version of the graph in the course of KCPM.

Thus, in our incremental preprocessing we set $\lambda_{inc} = \lambda_{tr}$ and recalculate SPTs using specific stored ones. After that network reduction is performed using the lower bound obtained from $\Phi(\lambda_{inc})$, set $\lambda^{+} = \lambda_{inc}, \lambda^{-} = \infty$ or $\lambda^{-} = \lambda_{inc}, \lambda^{+} = 0$ depending on the sign of $\Phi^{'}(\lambda_{inc})$, recalculate the necessary invalid SPTs (recalculate SPTs for $\lambda = 0$ if $\Phi^{'}(\lambda_{inc}) \leq 0$ or recalculate SPTs for $\lambda = \infty$ otherwise) and continue KCPM, using incremental SPT updates where necessary.

The initial network reduction with λ_{inc} gives incremental preprocessing a head start and allows to avoid recalculating SPTs for the graphs with a great number of e-nodes.

Next, we provide the pseudo-code for our incremental preprocessing.

```
1. $Set\ U = \infty$
2. For $every\ path\ p\ in\ the\ list\ of\ feasible\ solutions\ obtained\ in$
    $the\ gap\ closing\ of\ baseline\ algorithm$
            If $W(p) \le W_L$
                If $C(p) < U$
                    $U = C(p)$
                    $pU = p$
3. If $U = \infty$ (no path remained feasible and $U$ was not updated)
        $recalculate\ SPTs\ for\ \lambda = \infty$
        $U = C(Q_{st}^{\infty})$
        $pU = Q_{st}^{\infty}$
4. $Use\ stored\ full\ SPTs\ to\ recalculate\ SPTs\ for\ \lambda_{inc} = \lambda_{nt}$
5. For $each\ node\ k \in V,$
       If $C\left(Q_{sk}^{\lambda_{inc}} + Q_{kt}^{\lambda_{inc}}\right) < U\ and\ W\left(Q_{sk}^{\lambda_{inc}} + Q_{kt}^{\lambda_{inc}}\right) \le W_L$
           $U = C\left(Q_{sk}^{\lambda_{inc}} + Q_{kt}^{\lambda_{inc}}\right)$
           $pU = Q_{sk}^{\lambda_{inc}} + Q_{kt}^{\lambda_{inc}}$ (new upper bound has been found)
6. $V' = \emptyset\ \ and\ A' = \emptyset$ (vertices and edges that will be deleted)
   For $each\ node\ k \in V$
       If $\Phi_{G,k}(\lambda) \ge U,$
           $V' = V' \cup \{k\}$
           $A' = A' \cup \{all\ edges\ adjacent\ to\ k\}$
     $V = V \backslash V' and\ A = A \backslash A'$
     $G = (V, A)$
7. If $V = \emptyset$, STOP (the current upper bound is the optimal solution to
RCSPP)
8. If $\Phi_G'\ (\lambda_{inc}) \le 0$
       $recalculate\ SPTs\ for\ \lambda = 0$
       $continue\ KCPM\ with\ \lambda^{+} = 0\ and\ \lambda^{-} = \lambda_{inc}$
   Else
       $recalculate\ SPTs\ for\ \lambda = \infty$ (if it was not recalculated in STEP 3)
       $continue\ KCPM\ with\ \lambda^{+} = \lambda_{inc}\ and\ \lambda^{-} = \infty$
```

Thus, our preprocessing algorithm starts from advanced points, performs necessary network reductions and stops when either found optimal solution or optimal Lagrange multiplier for the new problem.

8 Experiments

We conducted extensive experiments on real road graphs of the city of Oldenburg containing 6000 nodes and 14000 edges and the city of San Joaquin [23] containing 18000 nodes and 46000 edges. We used L2 distance of the edge as its cost. To remain close to real life cases we decided to apply the edge weights according to real traffic distribution. We set the edge weight according to the load it is likely to have, which is determined by the number of incoming and outgoing edges of the source vertex of the edge as well as the weights of the outgoing edges, and the number of outgoing edges of the target vertex of the edge. The rationale behind this is simple. Let us denote as $In(i)$ and $Out(i)$ the sets of incoming and outgoing edges of the node i. Now let us consider the edge (i,j). If there is a lot of edges incoming in i, then there is a lot of traffic congregated at that vertex i. This traffic is then distributed across outgoing edges. However, each edge from $e \in In(i)$ "receives" only portion of traffic, determined by the number of outgoing edges of the target vertex of e. Therefore, for the result formula of the edge weight we have

$$W(i,j) = |In(i)| \frac{|Out(j)|}{\sum_{e \in Out(i)} |Out(e_t)|} \tag{18}$$

where e_t denotes the target vertex of the edge e and $|\cdot|$ denotes cardinality of the set. Note that $Out(j)$ is never empty, since we consider two-way roads, so we can always make a U-turn. It is possible to implement the edge weight setting in a way to compute all edge weights in $O(m)$.

We tested our algorithm against that of Muhandiramge and Boland after applying different amount of changes to the edge weights. The only restriction for weight changes was keeping the problem non-trivial, that is changed edges were to affect the solution path. The results are shown in Table 1. Each number represents an average computation time of several runs with different source and target vertices.

Our incremental version provides up to 50% decrease in computation time compared to Simple Node Elimination combined with gap closing. It is important to mention that this time decrease can help in keeping the computations in real-time, which is the purpose of this algorithm. During the day in the big after numerous traffic events this saved time can accumulate in even more impactful difference. Time save comes from faster initialization and recalculations of SPTs and depends on the number of edges that changed their weights.

Table 1. Experimental results with various numbers of e-nodes

Number of e-nodes	Algorithm used	Roads of Oldenburg, s	Roads of San Joaquin, s
Small	Muhandiramge and Boland	1.34	2.13
	Incremental RCSP	0.72	1.18
Medium	Muhandiramge and Boland	1.29	2.20
	Incremental RCSP	1.14	1.96
Large	Muhandiramge and Boland	1.36	1.94
	Incremental RCSP	1.88	2.31

9 Conclusion

In this paper we described a new incremental approach to solving the resource constrained shortest path problem by utilizing the data from the baseline algorithm and starting Lagrangian dual solving from advanced point. We introduced a modified version of unconstrained SPT update for the cases when the graph has either more or less vertices as well as different edge weights. We conducted experiments on real road graphs and achieved decreased computation time compared to recalculating a solution from scratch.

As a part of future work we intend to focus on the gap closing phase and its incremental version and publish already existing progress on the topic. We also plan on analyzing the amount of changes in the graph edge weights. Such analysis can yield important threshold which would signify whether it is more effective to use incremental version, recalculate from scratch or maybe even skip preprocessing phase if the number of changed edges is very low.

References

1. Muhandiramge, R., Boland, N.: Simultaneous solution of Lagrangean dual problems interleaved with preprocessing for the weight constrained shortest path problem. Networks **53**(4), 358–381 (2009). https://doi.org/10.1002/net.20292
2. Guralnik, R.: Incremental rerouting algorithm for single-vehicle VRPPD. In: Proceedings of the 18th International Conference on Computer Systems and Technologies, pp. 44–51. ACM (2017). https://doi.org/10.1145/3134302.3134326
3. Jaw, J.J., Odoni, A.R., Psaraftis, H.N., Wilson, N.H.: A heuristic algorithm for the multi-vehicle advance request dial-a-ride problem with time windows. Transp. Res. Part B: Methodol. **20**(3), 243–257 (1986). https://doi.org/10.1016/0191-2615(86)90020-2
4. Barnhart, C., et al.: Flight string models for aircraft fleeting and routing. Transp. Sci. **32**(3), 208–220 (1998). https://doi.org/10.1287/trsc.32.3.208
5. Carlyle, W.M., Royset, J.O., Wood, R.K.: Routing military aircraft with a constrained shortest-path algorithm. Naval Postgraduate School, Monterey CA, Department of Operations Research (2007)
6. Desrochers, M., Soumis, F.: A column generation approach to the urban transit crew scheduling problem. Transp. Sci. **23**(1), 1–3 (1989). https://doi.org/10.1287/trsc.23.1.1
7. Laporte, G., Pascoal, M.M.: The pipeline and valve location problem. Eur. J. Ind. Eng. **6**(3), 301–321 (2012). https://doi.org/10.1504/EJIE.2012.046669
8. Cabral, E.A., Erkut, E., Laporte, G., Patterson, R.A.: The network design problem with relays. Eur. J. Oper. Res. **80**(2), 834–844 (2007). https://doi.org/10.1016/j.ejor.2006.04.030
9. Smith, O.J., Boland, N., Waterer, H.: Solving shortest path problems with a weight constraint and replenishment arcs. Comput. Oper. Res. **39**(5), 964–984 (2012)
10. Pallottino, S., Scutella, M.G.: A new algorithm for reoptimizing shortest paths when the arc costs change. Oper. Res. Lett. **31**(2), 149–160 (2003). https://doi.org/10.1016/S0167-6377(02)00192-X
11. Aneja, Y.P., Aggarwal, V., Nair, K.P.: Shortest chain subject to side constraints. Networks **13**(2), 295–302 (1983). https://doi.org/10.1002/net.3230130212
12. Beasley, J.E., Christofides, N.: An algorithm for the resource constrained shortest path problem. Networks **19**(4), 379–394 (1989). https://doi.org/10.1002/net.3230190402

13. Dumitrescu, I., Boland, N.: Improved preprocessing, labeling and scaling algorithms for the weight constrained shortest path problem. Networks **42**(3), 135–153 (2003). https://doi.org/10.1002/net.10090
14. Mehlhorn, K., Ziegelmann, M.: Resource constrained shortest paths. In: Paterson, Mike S. (ed.) ESA 2000. LNCS, vol. 1879, pp. 326–337. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45253-2_30
15. Carlyle, W.M., Royset, J.O., Wood, R.K.: Lagrangian relaxation and enumeration for solving constrained shortest path problems. Networks **52**(4), 256–270 (2008). https://doi.org/10.1002/net.20247
16. Gallo, G.: Reoptimization procedures in shortest path problem. Rivista di matematica per le scienze economiche e sociali **3**(1), 3–13 (1980). https://doi.org/10.1007/BF02092136
17. Ramalingam, G., Reps, T.: An incremental algorithm for a generalization of the shortest-path problem. J. Algorithms **21**(2), 267–305 (1996). https://doi.org/10.1006/jagm.1996.0046
18. King, V., Thorup, M.: A space saving trick for directed dynamic transitive closure and shortest path algorithms. In: Wang, J. (ed.) COCOON 2001. LNCS, vol. 2108, pp. 268–277. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44679-6_30
19. Demetrescu, C.: Fully Dynamic Algorithms for Path Problems on Directed Graphs. Ph.D. thesis, Department of Computer and Systems Science, University of Rome "LaSapienza" (2001)
20. Zhu, X.: The Dynamic, Resource-Constrained Shortest Path Problem on an Acyclic Graph with Application in Column Generation and Literature Review on Sequence-Dependent Scheduling. Doctoral dissertation, Texas A&M University (2007)
21. Nguyen, S., Pallottino, S., Scutellà, M.G.: A New Dual Algorithm for Shortest Path Reoptimization. In: Gendreau, M., Marcotte, P. (eds.) Transportation and Network Analysis: Current Trends. Applied Optimization, vol. 63, pp. 221–235. Springer, Boston (2002). https://doi.org/10.1007/978-1-4757-6871-8_14
22. Pallottino, S., Scutellà, M.G.: Dual algorithms for the shortest path tree problem. Networks **29**(2), 125–133 (1997). https://doi.org/10.1002/(SICI)1097-0037(199703)29:2<125::AID-NET7>3.0.CO;2-L
23. Li, F., Cheng, D., Hadjieleftheriou, M., Kollios, G., Teng, S.-H.: On trip planning queries in spatial databases. In: Bauzer Medeiros, C., Egenhofer, Max J., Bertino, E. (eds.) SSTD 2005. LNCS, vol. 3633, pp. 273–290. Springer, Heidelberg (2005). https://doi.org/10.1007/11535331_16