On the Computation of the Optimal Connecting Points in Road Networks

George Tsatsanifos
Computing Science Department
University of Alberta, Canada
george.tsatsanifos@ualberta.ca

January 8, 2016

Abstract

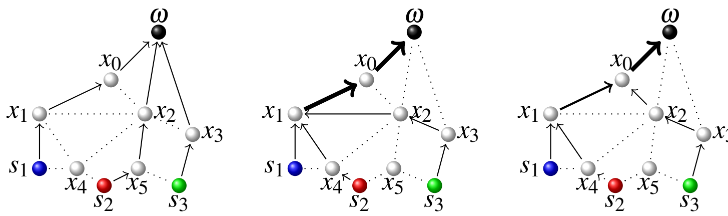
In this paper we consider a set of travelers, starting from likely different locations towards a common destination within a road network, and propose solutions to find the optimal connecting points for them. A connecting point is a vertex of the network where a subset of the travelers meet and continue traveling together towards the next connecting point or the destination. The notion of optimality is with regard to a given aggregated travel cost, e.g., travel distance or shared fuel cost. This problem by itself is new and we make it even more interesting (and complex) by considering affinity factors among the users, i.e., how much a user likes to travel together with another one. This plays a fundamental role in determining where the connecting points are and how subsets of travelers are formed. We propose three methods for addressing this problem, one that relies on a fast and greedy approach that finds a sub-optimal solution, and two others that yield globally optimal solution. We evaluate all proposed approaches through experiments, where collections of real datasets are used to assess the trade-offs, behavior and characteristics of each method.

1 Introduction

Consider a group of travelers starting from, likely different, points in a (city) road network (e.g., their work-places) towards a common destination (e.g., a restaurant) where traversing an edge of the network incurs a cost (e.g., distance, travel time or fuel cost). While they can each travel on their own following their respective most cost effective routes, it may be the case that they might prefer to travel alongside other travelers, e.g., "time flies when in good company." Then it is possible that travelers may accept to travel using less efficient paths in exchange for having company. This leads us to what we call the *Optimal Multiple Connecting Points* (OMCP) query. This query returns the optimal *connecting points* where subsets of travelers can connect and form groups of increasing sizes until they all reach the destination. The underlying goal in this query is to find routes that minimize an aggregate cost, e.g., the maximum or the average cost travelers have to incur in terms of the recommended route from their starting locations to the common destination.

This type of query, which one can classify as belonging to the class of Trip Planning Queries, becomes more relevant and applicable with the proliferation of geo-spatial applications, location- and mobility-aware handheld devices dictating a trend towards the development of efficient solutions for problems stated on actual road networks. Apart from portable devices, countless applications have been launched on the web, developed over the Google Maps and OpenStreetMap platforms. The provided APIs operate on official data for the former, or data gathered through crowdsourcing for the latter; so as to allow the implementation of web mapping services with cutting edge development tools, such as Javascript, Ajax and XML at the front-end. Our own flavor addressing the problem studied here relies on the Google Maps API and also employs Java Server Pages (JSPs). It is available at `http://connect.cs.ualberta.ca/omcp/` with an elegant and intuitive interface for interacting with the users.

Figure 1 illustrates three different scenarios for three travelers. First, let the starting locations s_1, s_2 and s_3 be where the travelers are initially located. Figure 1(a) depicts the case where all travelers follow their respective shortest path to the common destination independently of each other. In Figure 1(b), we depict the case where all travelers meet at an intermediate location (node x_1) before continuing their journey to the final destination ω. Depending on the affinity parameters the second route plan can be more preferable in terms of the total aggregated cost arising from the suitable cost function that provides a direct way to quantify this. Finally, in Figure 1(c), travelers t_1 and t_2 meet in x_1 before encountering traveler t_3 in x_0, and then all proceed to their common destination, node ω.



(a) shortest paths (b) single meetup (c) two meetups

Figure 1: Different scenarios for three travelers sharing a common destination.

We propose three different schemes to solve OMCP queries. The first one relies on a fast, greedy approach that finds a sub-optimal solution. The two other ones finds globally optimal solutions, the first one prunes candidate solutions based on cost bounds over the refinements of an examined route, whereas the second is more aggressive, it employs a best-first search approach so as to process the most promising routes first, thus aiming at enforcing stricter bounds. In summary, the main contributions we offer in this paper are the following:

- We define, for the first time, OMCP queries in road networks.
- We take the affinity between users into account in the computation of the aggregate travel cost.
- We explore several interesting properties of the problem in order to reduce the search space.
- We propose and evaluate three novel approaches for processing OMCP queries.
- We implement a framework leveraging the methods presented here, and make it available on-line at `http://connect.cs.ualberta.ca/omcp/`.

This paper is structured as follows: in Section 2, we formally pose the problem tackled in this paper; in Section 3, we propose a paradigm for processing the addressed query type; in Section 4, we evaluate our methods; in Section 5 we discuss the relevant literature, and finally, in Section 6, we summarize our contributions.

2 Problem Statement

Let us assume: (*i*) a road network, modeled as a graph $G(V,E)$, where d_{xy} denotes the cost to traverse the shortest path in G connecting the nodes x and y in V, (*ii*) a set $T = \{t_1, t_2, \cdots, t_n\}$ of n travelers and their respective starting locations (i.e., nodes in V) $S = \{s_1, s_2, \cdots, s_n\}$, (*iii*) a set of affinity factors λ_{ij} reflecting how much traveler i enjoys the company of traveler j, and (*iv*) a common destination ω. We also assume that λ_{ij}, with $0 \leq \lambda_{ij} \leq 1$ and $\sum_{j=1}^{n} \lambda_{ij} = 1, \forall i$. In particular, the lower the λ_{ij} value the more the enjoyment i has by traveling together with j. That is, we make the reasonable assumption that λ_{ij} serves as a relaxing factor for traveler i when accompanied by j, that is, when traveling together over a path $uv \in G$ the cost d_{uv} is *perceived* as $\lambda_{ij} d_{uv} \leq d_{uv}$.

Before we define the OMCP query, let us define a restricted version thereof, namely the Optimal *Single* Connecting Point (OSCP) query [4]. In fact, we will build upon a solution to the latter to solve the former. We illustrate a possible solution to an OSCP query in Figure 1(b), where the travelers from starting locations s_1, s_2 and s_3 meet at x_1 before proceeding to ω. The cost of that solution can be build as follows. First t_1, t_2 and t_3 travel by themselves from their starting locations to x_1, the connecting point where the group is formed before they continue together to their common destination ω. As such, the travel cost from the starting locations to the connection location are multiplied by a factor representing how much each traveler enjoys traveling by him/herself, i.e., λ_{11}, λ_{22} and λ_{33}, for each of the travelers t_1, t_2 and t_3, respectively, yielding $\lambda_{11}d_{s_1x_1} + \lambda_{22}d_{s_2x_1} + \lambda_{33}d_{s_3x_1}$. Next, we need to account for the average travel cost for all travelers from the connecting point x_1 towards ω, $\frac{1}{3}(\lambda_{12}\lambda_{13} + \lambda_{21}\lambda_{23} + \lambda_{31}\lambda_{32})d_{x_1\omega}$. Note that the multiplied affinity factors reflect how $d_{x_1\omega}$ is *perceived* by each traveler when travelling with the others. For instance, when traveler t_1 travels with t_2 and t_3, $d_{x_1\omega}$ "feels like" $\lambda_{12}\lambda_{13}d_{x_1\omega} < d_{x_1\omega}$ for that particular traveler. It follows that the average cost of the route plan shown in Figure 1(b) is equal to:

$$\begin{aligned}\sigma_{x_1} = \frac{1}{3}(&\lambda_{11}d_{s_1x_1} + \lambda_{22}d_{s_2x_1} + \lambda_{33}d_{s_3x_1} + \\ &(\lambda_{12}\lambda_{13} + \lambda_{21}\lambda_{23} + \lambda_{31}\lambda_{32})d_{x_1\omega})\end{aligned} \tag{1}$$

More formally, the result to an OSCP query is the single best location $\mu^* \in V$ where all travelers should meet so as to minimize the cost function σ_u, i.e., $\mu^* = \text{argmin}_{u\in V}\sigma_u$. If the goal is to minimize the *maximal travel distance* we define σ_u as:

$$\sigma_u^{max} = \max_i\{\lambda_{ii}d_{s_iu} + d_{u\omega}\prod_{j\neq i}\lambda_{ij}\} \tag{2}$$

Otherwise, if our target is to minimize the *average travel distance* we define σ_u as:

$$\sigma_u^{avg} = \frac{1}{n}\sum_i\{\lambda_{ii}d_{s_iu} + d_{u\omega}\prod_{j\neq i}\lambda_{ij}\} \tag{3}$$

The OMCP, however, allows for *multiple* connecting points. In order to understand how the cost of a solution to the OMCP is obtained, let us discuss the solution shown in Figure 1(c) in detail. First, traveler t_1 meets traveler t_2 at connecting point x_1 and they proceed together until x_0 (where traveler t_3 joins the group) at summed cost $d_{s_1x_1}\lambda_{11} + d_{s_2x_1}\lambda_{22} + (\lambda_{12} + \lambda_{21})d_{x_1x_0}$. In order to meet t_1 and t_2 at connecting point x_0 traveler t_3 needs to travel from s_3 to x_0 by him/herself at a cost equal to $d_{s_3x_0}\lambda_{33}$. Then, all three travelers proceed together from x_0 to the destination ω at a cost equal to $(\lambda_{12}\lambda_{13}+\lambda_{21}\lambda_{23}+\lambda_{31}\lambda_{32})d_{x_0\omega}$. It is noteworthy that x_0 is essentially the solution of another OSCP query, where the destination is ω, traveler t_3 leaves from s_3 and a "super-traveler" representing travelers t_1 and t_2 *together* leaving from x_0 (where they were connected). How to derive the affinity factor for such a "super-traveler" though? For *min-avg* ranking criteria λ_{12} + λ_{21} quantifies how much the "super-traveler" likes to travel alone as if a single traveler. This is intuitive as it combines how t_1 likes to travel with t_2 and vice-versa. Therefore, the average travel cost of the OMCP solution shown in Figure 1(c) is equal to $\frac{1}{3}(d_{s_1x_1}\lambda_{11} + d_{s_2x_1}\lambda_{22} + d_{s_3x_0}\lambda_{33} + (\lambda_{12} + \lambda_{21})d_{x_1x_0} + (\lambda_{12}\lambda_{13} + \lambda_{21}\lambda_{23} + \lambda_{31}\lambda_{32})d_{x_0\omega})$.

Notation	Description
$t_1, t_2, \cdots, t_n$	t_i stands for the i-th traveler.
$s_1, s_2, \cdots, s_n$	s_i stands for the starting location of traveler t_i.
ω	common destination shared by all travelers.
d_{uv}	minimum cost for traveling from node u to v.
λ_{ij}	captures how much traveler t_i enjoys t_j's company.
p	a partitioning of travelers into groups and subgroups.
μ_p	connecting point where the elements of group p meet.
σ_u	aggregated score for point u of the road network.
σ_p	aggregated score that grouping p achieves.

Table 1: Notation and terminology.

We can generalize the reasoning above as follows. Let S_{ik} the k-th connecting group where traveler t_i appears; and let us assume that there are G such groups. Further, we denote as $\mu_{S_{ik}}$ the location (node) in the network, where this connection takes place. Note that $\mu_{S_{i0}} = s_i, \forall t_i$, i.e., t_i's starting position. By construction, we have that $S_{i_0} \subset S_{i1} \subset S_{i2} \subset \cdots \subset S_{im_i}, \forall t_i$. Once a traveler joins a group it never leaves it, and at each connection point at least one new traveler or possibly a subgroup of travelers, joins a group. With this in mind the cost of the OMCP query can be broken down into the following components. We know that all travelers will travel by themselves from their starting location to their first connecting point, thus contributing each with cost $d_{s_i \mu_{S_{i1}}} \lambda_{ii}$. After traveler t_i joins its k-th connecting group, it moves to the next connecting point, i.e., $S_{i(k+1)}$. However, we need to take into account that all travelers in that group will be traveling together, therefore, such group movement will yield for each traveler $t_i \in S_{i(k+1)}$ a cost equal to $d_{\mu_{S_{ik}} \mu_{S_{i(k+1)}}} \prod_{\substack{t_j \in S_{ik} \\ t_i \neq t_j}} \lambda_{ij}$.

It leads that a route plan costs on average for each traveler:

$$\sigma^{(avg)} = \frac{1}{n} \sum_{t_i \in T} \Big(\sum_{k=0}^{m_i - 1} d_{\mu_{S_{ik}} \mu_{S_{i(k+1)}}} \prod_{\substack{t_j \in S_{ik} \\ t_i \neq t_j}} \lambda_{ij} \Big) \tag{4}$$

We can now generalize Equation 2 as well so as to account for the increased number of meeting locations. The respective cost function now takes the following form:

$$\sigma^{(max)} = \max_{t_i} \Big(\sum_{k=0}^{m_i - 1} d_{\mu_{S_{ik}} \mu_{S_{i(k+1)}}} \prod_{\substack{t_j \in S_{ik} \\ t_i \neq t_j}} \lambda_{ij} \Big) \tag{5}$$

The challenge we shall address for the remaining of this paper is to find the best such partitioning of the travelers' set into subgroups and their respective meeting locations $\mu_{S_{ik}}$ where each traveler joins a group so as to minimize the aggregated travel cost.

3 Processing Route Combinations

In this section, we propose a paradigm consisting of three concrete algorithms for combining paths rooted at n sources and led to a single destination. The proposed scheme involves computing the best sequence of locations where each time two or more groups of travelers following different paths are joined together in such a way that any other combination of joined routes would yield a suboptimal score overall. However, we first need an effective algorithm for solving the simplest form of the problem, where we seek the optimal connecting point for a single group. We will use consistently for this purpose the method from [4] for smaller subgroups by the methods for computing the best combination of multiple connecting points.

3.1 Structuring Routes Plans

The most fundamental concept of our paradigm that we need to delineate first is that of the *grouping*. We define a *grouping* to be any *disjoint partitioning* of the travelers. Therewith, a grouping is essentially nothing more than a collection of groups of travelers having a recursive structure of arbitrary depth with the following properties: (i) a number of contained subgroups with possible nested deeper group structure, (ii) a meeting location where all contained subgroups meet, and of course, (iii) is associated with a specific destination node of the road network shared by all travelers.

Given a number of groups, we provide three different options on how to combine a collection of groups from all available choices, so as to form a valid grouping, and therewith, select the most beneficial combination each time. Namely: (i) leave them intact as each group arrives at the destination point (or next connecting point) independently, as in Fig. 1(a) where three travelers follow completely different paths and each one constitutes a group of its own, as in the partitioning $\{t_1\}, \{t_2\}, \{t_3\}$, (ii) *join* them by establishing a meeting location where those groups meet and from then on form a single solid group, as in Fig. 1(c) where at x_0 a group of two travelers is joined with a third traveler from s_3, a route plan that corresponds to partitioning $\{\{t_1, t_2\}, \{t_3\}\}$, (iii) *merge* them altogether by rejecting and dissolving their previous structure, for example the route plan of Fig. 1(c) becomes the one in Fig. 1(b) as we split the group of two travelers into its element and compute the meeting location for all three, corresponding to partitioning $\{t_1, t_2, t_3\}$. Likewise, the route plan produced by our framework in Fig. 2 was produced by joining the user D with a larger group consisting of users A, B and C. Each of those two subgroups has its own meeting location where its members are met, that also serves as a "starting location" for computing where those two groups should be joined. Conversely, we could merge those two groups into a single one where all four users would meet at a single location before proceeding to their common destination. Thereby, we would discard the previous formation of the two separate groups. Which one of the two choices is more preferable depends on the aggregated cost of the two route plans.

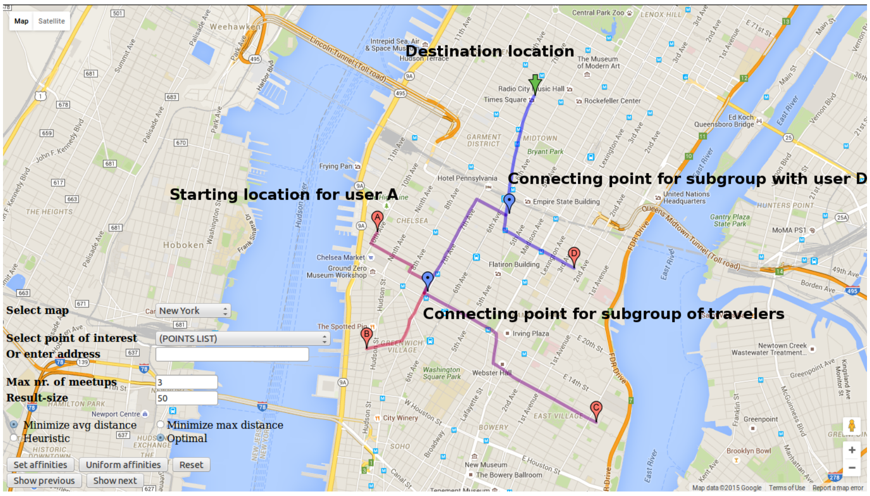


Figure 2: Route plan derived by merging users A, B, C into a single group, joined with user D later on.

These are the three fundamental operations we consider when computing incrementally more complex combinations of larger groups, starting from trivial groups initially, each consisting of just one traveler. Moreover, any route plan can be represented with a tree hierarchy that corresponds to a multi-level recursive group, where the n travelers constitute the bottom layer (level 0), and each intermediate level corresponds to a separate grouping. Thereby, the route plan of Fig. 2 has four travelers as its leaves, three of them (A, B, C) merged together to an internal node, which is then joined with leaf node D to form an another internal node that is directly connected to the root corresponding to the destination. Each internal node is associated with a meeting location, or a single connecting point that is computed by taking into consideration the locations of the members of that group only. More importantly, changing the location associated with an internal node does not affect the meeting locations of the lower layers of the hierarchy. In other words, by choosing a different location where the two groups (one group for users A, B, C, and another containing user D only) are joined in Fig. 2, has no effect on where users A, B, C meet, as the same location would serve as the optimal connecting point for this specific subgroup in spite of any change in the rest of the route plan. We could generalize this with the statement that the best location where any subgroup meets does not change with what happens in another subgroup, e.g. if we change its meeting location, or merge it with another subgroup. This very simple concept allows for the incremental bottom-up computation of the most preferable route plans.

Now, what it remains is to find the best way to combine these travelers until only one group remains in the end, encompassing all travelers' routes that are joined in combinations arranged in such a way that the aggregated travel cost is minimized. Towards this end, we propose three different algorithms, namely Heuristic Processing (HP), Extensive Processing (EP), and Clustered Processing (CP), that leverage similar processing principles. The aforementioned algorithms are incremental and with each iteration we obtain another more complex grouping in terms of its nested structure that might replace the previous candidate solution, only if better. In principle, all methods start with an initial tentative solution, which we try to refine successively. Thereby, a naive grouping solution would involve all individual travelers, which we represent with groups having no deeper structure. The sequential scheme we propose tries to put together only the groups whose either joining, or merging yields a better score overall, while the rest remain intact for the same iteration. Moreover, users' preferences expressed with their respective λ-values play a prominent role and affect two major aspects: (i) which groups belong together, and (ii) how much a route should be expanded (starting from the shortest path between two nodes), so as to allow for a more preferable journey when combined with others. In practice, we start from an initial solution of completely independent travelers that are grouped together in more complex structures in a bottom-up approach that leverages agglomerative clustering in an effort to construct more efficient route plans.

3.2 OMCP Query Complexity

We can show that the problem we study is NP-hard with a reduction to the well-known weighted set cover problem. We assume a set of elements called the universe, and a set S of n sets whose union equals the universe. The set cover problem is to identify the min weight subset of S whose union equals the universe. We have for our reduction that S corresponds to a set which is larger than the power-set of the set of travelers. The main reason behind this claim is that a group of 3 travelers, like the one in the example of Fig. 1 may consist of other smaller groups. For instance, in Fig. 1(c), the group that is formed at x_0 congregates two smaller groups. The first consists of two travelers that met at x_1, while the remaining traveler constitutes a group of his own. This peculiarity increases the complexity and the perplexity of the problem we study here by a great deal. Therefore, since we can have groups that are nested within other groups recursively at arbitrary depth (we can have no smaller granularity than one traveler each), we will form S as the power-set of the travelers' set and for each element of that set we will compute its own power-set and congregate its elements appropriately to produce new more complex subsets to add to S. This is also repeated for each newly created subset until no sets can be further produced. For instance, let travelers t_1, t_2 and t_3. Then, as discussed earlier, we have the following set $S = \{\{t_1\}, \{t_2\}, \{t_3\}, \{t_1,t_2\}, \{t_1,t_3\}, \{t_2,t_3\}, \{\{t_2\}, \{t_1,t_3\}\}, \{\{t_1,t_2\}, \{t_3\}\}, \{\{t_1\}, \{t_2,t_3\}\}, \{t_1,t_2,t_3\}\}$, a set which is clearly superset of the power-set of the travelers' set. Therefore, it is at least as difficult to find the optimal route plan than it is to solve the set cover problem with up to 2^n sets to select from, so as to find the minimum weight sets required to form a valid partitioning that comprises all travelers, whereas the search space of the problem we tackle here comprises $\Theta(n^n)$ possible route plans*. We also note that each element of S corresponds to a grouping associated with a meeting location. Likewise, each nested subgroup corresponds to a group of travelers which is associated with a meeting location where those travelers met in advance. Also, we should add another rule to this variation of the set-cover problem, according to which we do not allow to select elements with common elements. In terms of the weights of each element of S, we want the arising cost for S to be minimized, and hence, we set this by convention according the cost functions we discussed earlier.

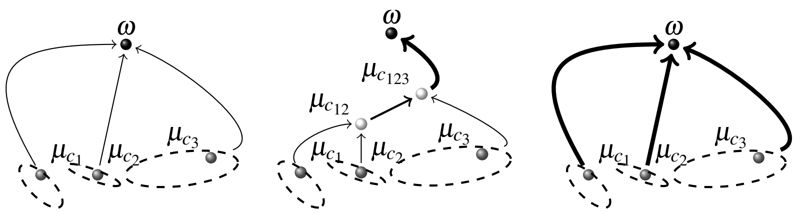
After showing the intractability of the problem we tackle in this paper, we will shortly discuss the courses of actions we undertake. First, we propose a method that computes an approximate solution by incrementally refining a naive solution where each traveler sets his journey separately towards ω, by putting together the travelers that yield the greatest gain at the time. Second, two approaches are presented for finding the optimal solution. Even though they are of exponential complexity with regard to the number of travelers by a worst case analysis, the pruning policy we discuss in later sections reduces the search space so as to empower the computation of the solution in an effective way.

3.3 Bounding and Pruning

Given a grouping, say p, we can make some safe assumptions with regard to the minimum and maximum score any other more complex grouping that uses p as a basis can achieve. These bounds can be used to safely discard the derived groupings, according to some sequence of the three aforementioned operations, that are dominated in terms of their possibility to produce the best solution. In effect, we can disregard a route plan and stop processing it any further when the lower bound of the score of the dominated route plan is greater than the upper bound of the better one. Otherwise, unless an effective pruning policy is adopted, the total number of processed groupings would rise dramatically and the computation cost would be immense to constitute such methods unable to respond in real-time. Towards this end, we start initially from a naive route plan where each individual traveler follows independently the shortest path to the destination. We are in search next for a lower bound for the optimal score σ^* of any route plan that can be derived from this naive route plan where the travelers meet nowhere else but at the destination. Since there is no way of knowing this beforehand, we would like to have a lower bound, so that we can approximate and quantify how promising a route plan is and whether processing it any further could yield better and improved route plans subsequently. One could expect that a reasonable lower bound would be the minimum individual traveler cost. However, this is not the case, as a route plan of lower cost could still be feasible when other travelers join him early on from his starting post, for a given setting of the affinity parameters of course. More specifically, we have then that, $\text{aggr}_i d_{i\omega} \prod_j \lambda_{ij} \leq \sigma^* \leq \text{aggr}_i d_{i\omega} \lambda_{ii}$, where aggr_i is replaced with either $\sum_i$ or $\max_i$, accordingly.

*Due to lack of space the computation of the number of possible route plans has been omitted

In other words, we have on the right side the aggregated scores from the individual travelers serving as an upper bound for the score any subsequent route combination could achieve, whereas on the left we relax each individual cost at the maximum possible degree, as if all travelers could join right from the starting post, without considering the score of the other travelers moving to that post from their own, though. Clearly, any possible route plan that can be derived would yield a higher cost because of the overhead arising from the other travelers' costs to get there.



(a) bound $\tau_p \geq \sigma_p^*$ (b) best group over p (c) bound $\ell_p \leq \sigma_p^*$

Figure 3: Bounds for σ^* over grouping p at processing level k.

In Figure 3, we illustrate the case where a number of groups has already been formed. Each of these groups is naturally associated with a location where its comprised members meet. From then on, each group follows the shortest path to the common destination ω. In Fig. 3(a), we show the shortest paths from the point where each group meets before proceeding to ω. However, a more preferable route plan might be found when two or more of these groups meet before they reach their final destination. This is due to the affinity parameters that affect the aggregated cost according to the combinations of travelers that are formed. This case is shown in Fig. 3(b), where thicker lines depict paths followed by more travelers. Let's see now how the formula that we use as a lower bound here relates to this particular route plan. The cost value of the route plan (does not correspond to a realistic route plan) in Fig. 3(b) would diminish if the travelers that participate in each of the groups where accompanied by all their fellow travelers, something that is infeasible of course. In other words, if we could see only thick lines in Fig. 3(b), as an insight. But still, we can find an even lower bound that corresponds to the unrealistic route plan that we illustrate in Fig. 3(c), where each group follows the shortest path to the destination at the cost of being accompanied by all travelers in the same time. Still an unrealistic query plan but devoid of the additional costs arising from the groups meeting at intermediate locations that detour from the shortest paths. To sum up with, the lower bound for the cost of the route plan that corresponds to Fig. 3(a) is calculated by multiplying the respective cost of each group to the destination by the affinity parameters that correspond to all travelers, an unrealistic route plan but extremely useful as a lower bound for pruning.

The previous example was about establishing lower and upper bounds for the score any subsequent route plan could achieve after processing further the naive solution where each traveler follows an independent route to the destination. Let's generalize and examine a grouping p that consists of a finite number of groups, say $c_1, c_2, \cdots, c_m$. Now, even if this is not the overall perfect grouping assigning travelers to groups, and consequently to subgroups, and the structure within each c_i group of the route plan expressed within p is not optimal, it is however definitely known, and hence, the respective score can be calculated. We denote the score of grouping p as σ_p from either Eq. 5 or 4. Accordingly, we define τ_p the upper bound that can be achieved, where $\tau_p = \sum_{c \in p} \tau_c + d_{\mu_c \mu_p} \sum_{x \in c} \prod_{y \in c, y \neq x} \lambda_{xy}$ for *min-avg* ranking, or $\tau_p = \max_{c \in p} \tau_c + d_{\mu_c \mu_p} \prod_{y \in c, y \neq x} \lambda_{xy}$, with $x = \operatorname{argmax}_z \tau_z + d_{z \mu_c} \prod_{y \in c, y \neq x} \lambda_{zy}$. Then, for the globally optimum score that involves all groupings that can be built over p, in other words the different ways that the groups of p can be combined together to form more complex and preferable route plans, it follows that $\sigma_{p^*} \leq \tau_p$, where we note with τ_p the maximum score any grouping built over the p can achieve, and with p^* the best (not known at the time) grouping that can be built over the structure of p and σ_{p^*} its optimal respective score. Analogously, we can establish a lower bound ℓ_p, which is extremely useful for processing all these collections of groups effectively and prune from early on the ones that cannot contribute to the result for being dominated by other more successful route plans. In particular, we have $\ell_p = \sum_{c \in p} \ell_c + d_{\mu_c \mu_p} \sum_{x \in c} \prod_{y \in p, y \neq x} \lambda_{xy}$ for *min-avg* ranking, while for *min-max* ranking it follows, $\ell_p = \max_{c \in p} \ell_c + d_{\mu_c \mu_p} \prod_{y \in p, y \neq x} \lambda_{xy}$, where $x = \operatorname{argmax}_z \ell_z + d_{z \mu_c} \prod_{y \in p, y \neq z} \lambda_{zy}$. Therefore, it follows that $\sigma_p^* \geq \ell_p$. In effect, both bounds will be used to prevent from expanding the groupings that cannot lead to good route plans, and also prioritize the most promising ones. Besides, processing those first sets stricter constraints for all others, and allows for even more effective pruning, and thus, faster response-times for our schemes. We reckon that even though the derived bounds from early processing stages would be quite far from the optimal route plan, since they consist of individual travelers or rudimentary groups, as we proceed with building more complex but more cost-efficient groupings and filtering-out the ones that are guaranteed to be outranked eventually, both bounds are getting closer and closer to the optimal value σ_p^*, the best possible score that a route plan that is built over p can achieve.

Algorithm 1: Heuristic Processing (HP) algorithm.

```
1  Θ ← +∞;
2  p̂ ← new Group (ω);
3  foreach s in {s_1, s_2, ··· , s_n} do
4      g ← new Group(s);
5      g.addSubbgroup (new Group(t));
6      p̂.addSubgroup (g);
7  hasImprovedGroup ← true;
8  while hasImprovedGroup do
9      hasImprovedGroup ← false;
10     for outer in p̂.subgroups() do
11         for inner in p̂.subgroups( do
12             joined ← p̂.join(outer,inner);
13             merged ← p̂.merge(outer,inner);
14             if σ_joined < σ_merged then
15                 if σ_joined < Θ then
16                     hasImprovedGroup ← true;
17                     p̂ ← joined;
18                     Θ ← σ_joined;
19             else
20                 if σ_merged < Θ then
21                     hasImprovedGroup ← true;
22                     p̂ ← merged;
23                     Θ ← σ_merged;
24 return p̂;
```

3.4 Heuristic Processing

The first method we propose in this paper relies on greedy heuristics. Towards this end, we start from a certain grouping and we iteratively refine it until no other change can occur. The best improvement each time builds up incrementally to the previous candidate solution that we have constructed so far. More specifically, Algorithm 1 starts from the input group, and dissolves it to its constituents in lines 10 and 11 before it combines them in lines 12 and 13 by trying to either join them, where the two subgroups will meet at an intermediate location, or merge them, and by that we mean that all the elements of the two groups will meet at a new single meeting location to form a new bigger group. This last operation resembles merging the containment of the two into one that meets at a new intermediate location. Another way to put it, we revoked the structure of the previous groups and formed a new one that congregates their constituents. On the other hand, we formed a new more complex group for the former case by joining appropriately the examined parts at an intermediate location while preserving their previous structure at the same time. The derived groups are then compared with each other in line 14, and then in lines 15 and 20, whether the best derived group outranks the previous form of the ingredient elements that produced this more preferable group combination.

If so, then it is replaced. This process is repeated until the candidate solution cannot be improved any further, when no other possible change would yield a more preferable solution.

Algorithm 2: The Extensive Processing (EP) algorithm.

```
Θ ← +∞;
p̂ ← new Group (ω);
foreach s in {s1, s2, ··· , sn} do
    g ← new Group(ω);
    g.addSubbgroup (new Group(s));
    p̂.addSubgroup (g);
processed ← new Set();
pool ← new Queue();
processed.add (p̂);
pool.add (p̂);
while not pool.isEmpty() do
    p ← pool.remove();
    if ℓp > Θ then
        continue;
    else if σp < Θ then
        p̂ ← p;
    for outer in p.subgroups() do
        for inner in p.subgroups() do
            if outer = inner then
                break;
            else
                merged ← p.merge(outer,inner);
                if ℓmerged < Θ and not processed.contains(merged) then
                    processed.add(merged);
                    pool.add(merged);
                joined ← p.join(outer,inner);
                if ℓjoined < Θ and not processed.contains(merged) then
                    processed.add(joined);
                    pool.add(joined);
return p̂;
```

3.5 Extensive Processing

The second method we present resembles a brute-force algorithm (with a twist) that examines a route plan and produces all possible combinations that can possibly outrank the best solution found so far with a further refinement over the route plan that we examine. Those refinements are performed using the aforementioned operations over the congregated groups of the grouping, namely merging and joining. In addition, whenever we construct a subsequent route plan two comparisons take place that decide whether it will be qualified for further processing, or even whether should replace the candidate solution we have found at the time.

First, an eligible subsequent grouping that comprises a group that corresponds to a construct of two groups of the route plan, should constitute a more preferable option than the aggregation of its ingredient groups from which it was formed. If this is not the case, then we can safely discard the derived element and rely on the previous combination of the two separate elements instead. In other words, drop the derived grouping for the sake of the previous route plan. Second, another check in terms of the lower bound of the score the newly derived group can achieve is performed. More specifically, the resulting route plan containing the newly derived group should have a lower bound that outranks the upper bound of the best solution we have found so far. Then, the groups that pass the test are appended at the end of a queue for further processing later on. To elaborate, while there are remaining groupings to be processed and further refined (Alg. 2, line 11), we probe the next one in line 12, and test whether it can produce more preferable route combinations in line 13. If this is the case, then we check in line 15 if the examined group outranks the candidate solution and replace it accordingly in line 16. From then on, we produce groups and compute the next possible meeting locations for the travelers to meet up by combining in pairs the elements of the examined group in lines 17–29. Initially, in line 22, we try to merge the two elements by rejecting their previous structure and combine their encapsulated subgroups in such a way that a new group is formed. Then, in line 26, we construct an additional combination when the two subgroups have to be joined at an intermediate location while maintaining their initial structure before reaching their target location. In lines 23 and 27, we perform checks to ascertain whether the derived combinations can be further refined into better route plans. If this is possible, then they are appended to the appropriate queues in lines 24, 25, 28 and 29, accordingly.

3.6 Clustered Processing

For the remainder of this section, we shall discuss the intricate details of our last method. More specifically, in Algorithm 3, we maintain all promising groupings in a heap, and process them according to their rank. More importantly, only groupings of the next processing stage whose lower bound is less or equal than the smaller upper bound found so far are processed any further, with Θ serving as a threshold to filter-out all groupings with higher lower bounds. In addition, the algorithm terminates when the heap is either empty, or the next popped element is outranked in terms of its bound by the current candidate solution $\hat{p}$, which is then returned to the user.

Algorithm 3: ClusteredProcessing $(\omega, \{s_1, s_2, \cdots, s_n\})$

$\Theta \leftarrow +\infty$;
$\hat{p} \leftarrow$ **new** Group (ω);
foreach s ***in*** $\{s_1, s_2, \cdots, s_n\}$ **do**
    $g \leftarrow$ **new** Group(ω);
    g.addSubbgroup (**new** Group(s));
    $\hat{p}$.addSubgroup (g);
groupings $\leftarrow$ **new** MinHeap ();
groupings.push (**new** HeapEntry$(\hat{p}, \sigma_{\hat{p}})$);
while ***not*** *groupings.isEmpty()* **do**
    top $\leftarrow$ groupings.pop();
    if $\sigma_{\hat{p}} \geq \ell_{top}$ **then**
        break;
    $\mathscr{P} \leftarrow$ generateNextPhaseGroupings (top);
    $\theta \leftarrow \min_{p \in \mathscr{P}} \sigma_p$;
    if $\theta < \sigma_{\hat{p}}$ **then**
        $\hat{p} \leftarrow \mathrm{argmin}_{p \in \mathscr{P}} \sigma_p$;
    if $\theta < \Theta$ **then**
        $\Theta \leftarrow \theta$;
    foreach p ***in*** $\mathscr{P}$ **do**
        if $\ell_p \leq \Theta$ **then**
            groupings.push (**new** HeapEntry(p, σ_p));
return $\hat{p}$;

Furthermore, with each iteration of Alg. 3, we invoke Alg. 4 in order to compute the groupings of the next phase. In essence, Alg. 4 takes as input a sequence of groups constituting a valid grouping, and detects which of them can be merged together for they yield a better score. In Alg. 4, lines 9 and 15, the elements of two distinct groups are put together to form a more preferable option when qualified in lines 11, 12, 17 and 18. All constituents of the newly formed group (either smaller subgroups, or individual travelers) will be met at the location that is returned by the method proposed in [4].

Algorithm 4: nextPhaseGroupings (grouping)

1 pool $\leftarrow$ **new** Set ();
2 list $\leftarrow$ **new** ArrayList ();
3 **foreach** *group* ***in*** *grouping* **do**
4 list.add (group);
5 pool.insert (group);
6 **foreach** *over* ***in*** *list* **do**
7 **foreach** *under* ***in*** *list* **do**
8   **if** *over* $\neq$ *under* **then**
9    $\zeta \leftarrow$ join (over,under);
10   **if** *useMinMaxRank* ***and*** $\sigma_\zeta < \sigma_{group}$ ***and*** $\sigma_\zeta < \sigma_{top}$
11   ***or*** *useMinAvgRank* ***and*** $\sigma_\zeta \leq \frac{|over|\sigma_{over}+|under|\sigma_{under}}{|over|+|under|}$ **then**
12   **if** ***not*** *pool.contains (ζ)* **then**
13   list.add (ζ);
14   pool.insert (ζ);
15    $\xi \leftarrow$ merge (over,under);
16   **if** *useMinMaxRank* ***and*** $\sigma_\xi < \sigma_{group}$ ***and*** $\sigma_\xi < \sigma_{top}$
17   ***or*** *useMinAvgRank* ***and*** $\sigma_\xi \leq \frac{|over|\sigma_{over}+|under|\sigma_{under}}{|over|+|under|}$ **then**
18   **if** ***not*** *pool.contains (ξ)* **then**
19   list.add (ξ);
20   pool.insert (ξ);
21 groupingtree $\leftarrow$ **new** GroupingTree ();
22 **foreach** *c* ***in*** *pool.reverse()* **do**
23 groupingtree.addGroup (c, **not** grouping.contains(c));
24 newgroupings $\leftarrow$ groupingtree.getGroupings();
25 $\theta \leftarrow \min_{p \in \text{newgroupings}} \sigma_p$;
26 qualified $\leftarrow$ **new** List();
27 **foreach** *p* ***in*** *newgroupings* **do**
28 **if** $\ell_p \leq \theta$ **then**
29   qualified.insert (p);
30 **return** qualified;

As a result, this incremental operation improves the traveling experience for all the involved parties. What is more, this is repeated successively until we are left with a collection of groups that cannot be improved so as to give another more influential group. This is accomplished within the while-loop of Alg. 4, where with each iteration we check whether the rudimentary groups of the previous levels can either be merged or joined with any of the other groups so as to yield a better grouping. Since the final number of groups is not known in advance in the general case, we will have to design an auxiliary structure that concentrates all required information and allow us to process effectively the numerous derived groups, but also prioritize them according to the bounds thereof. As a result, we would be in position of filtering-out any groupings that lead to suboptimal route combinations. More importantly, as the more complex qualified groups achieve a better score usually, we shall often place them at the top of the hierarchy, since it is very likely that they will participate in the most successful route plans, and thus, probably in the final solution, as well.

To elaborate, the root of this purpose-specific hierarchy that we call *grouping-tree* corresponds to an empty group containing no subgroups, and is therefore, disjoint with any other group. By disjoint we mean

Algorithm 5: GroupingTree.addGroup (newgroup, isDerived) : adds a new group in the appropriate places of the hierarchy built for retrieving the best groupings of a specific level.

```
stack ← new Stack();
stack.push (root);
while not stack.isEmpty() do
    top ← stack.pop();
    if top.isLeaf() then
        top.addChild (newgroup);
    else
        hasDisjointChild ← false;
        foreach child in top.getChildren() do
            if child.isDisjoint(newgroup) then
                hasDisjointChild ← true;
                stack.push (child);
        if isDerived and not hasDisjointChild then
            top.addChild (newgroup);
```

that the travelers each group contains do not overlap, in a sense that they congregate no common travelers, and thus, correspond to disjoint travelers' sets and partitions thereof. Next, since larger groups were constructed last on the basis of the ones formed previously, we will examine them in reverse order to insert them at the top levels of the tree. For each group we will execute Alg. 5, which in essence, performs a DFS-like traversal of the hierarchy we have constructed so far, by expanding the encountered tree-nodes, and inserting their children to a stack, if and only if, they correspond to groups that are disjoint with the currently examined group. Otherwise, we proceed with the next tree-node we pop from the stack. Leaves are expanded naturally by appending a new leaf-node associated with the examined group. Alternatively, if no branch of the encountered node accepts the examined group, then we will insert a new child-node for this node. As a final note, we generalize and claim that each path of the grouping-tree is a more complex expression of a list of trivial groups, where two or more groups have been funneled together through a sequence of merging or joining operations so as to form each node of the tracked path.

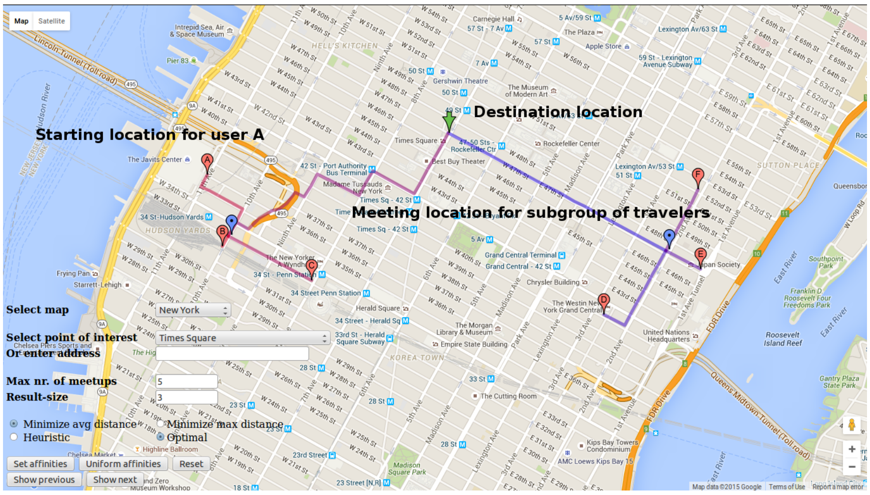


Figure 4: A route plan for six travelers split into two separate smaller groups that are joined at the destination.

3.7 Running Example

We will now present a simple problem instance for six travelers starting from locations $\alpha, \beta, \gamma, x, y$ and z, respectively, and discuss how our Clustered Processing (CP) algorithm runs. First, Alg. 3 is executed and will produce the initial grouping with just one group associated with each traveler right before entering the while-loop and expanding the best grouping retrieved at the time in each iteration with a call to Alg. 4. More specifically, each element of the group-list is combined with the rest of the elements, and we check whether their combination (by either joining or merging) produces a more complex group that benefits all its members, and therefore, it is more preferable than their previous separate form. For the purposes of this example, we assume that traveler t_α from starting location α can be combined only with travelers t_β and t_γ from locations β and γ, respectively, while a combination with travelers t_x, t_y or t_z would not meet the necessary constraints. Likewise, the rest of the travelers starting from locations denoted with greek letters, namely t_β and t_γ, should not be combined with travelers from locations denoted with latin, namely t_x, t_y and t_z, and vice versa. This could be either because those subsets of travelers like each other much more than the travelers from the opposite group, or due to their distances in the underlying network, or both. We illustrate such a scenario for six travelers in Fig. 4. We have two groups of friends, users A, B, C on one side belong to one group, while users D, E, F on the other side belong to the other. Each of those subgroups was formed by merging one traveler with another, say user A with B, and then merging user C with the group of two travelers that was formed earlier. Hence, two merging operations were required to form each of the smaller groups, and with a join operation we found where those two groups should be met so as to form a single solid group to reach Times Square. Conversely, a merge operation over the two subgroups would break them apart and their containment, six travelers in total, would constitute a new solid group that meets before or right at the destination. Hence, we would have just one meeting location for all six travelers instead.

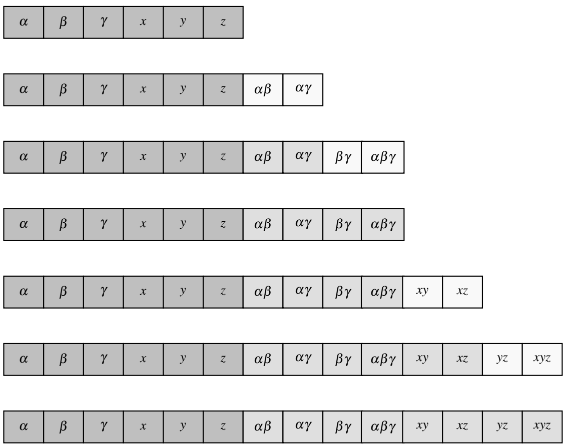


Figure 5: Derived group-list at first processing level for the six travelers scenario.

More precisely, we demand from the new groups to outrank the aggregation of the scores of the groups they were produced from, namely, we want, $\sigma_{\alpha x} \leq \frac{|\alpha|\sigma_\alpha + |x|\sigma_x}{|\alpha|+|x|}$, $\sigma_{\alpha y} \leq \frac{|\alpha|\sigma_\alpha + |y|\sigma_y}{|\alpha+y|}$, and $\sigma_{\alpha z} \leq \frac{|\alpha|\sigma_\alpha + |z|\sigma_z}{|\alpha|+|z|}$ for *min-avg ranking*, or analogously, $\sigma_{\alpha x} \leq \sigma_\alpha, \sigma_{\alpha x} \leq \sigma_x$, $\sigma_{\alpha y} \leq \sigma_\alpha, \sigma_{\alpha y} \leq \sigma_y$, and $\sigma_{\alpha z} \leq \sigma_\alpha, \sigma_{\alpha z} \leq \sigma_z$ for *min-max ranking*. Otherwise, the overall ranking function that considers all travelers would deteriorate instead of ameliorate. This is actually how our first phase of pruning is applied so as to prevent from processing further the route plans that mix travelers from the one group with travelers from the other. Then, as shown in Fig. 5, the newly derived groups are appended at the end of the group-list, and will be then combined with the elements we will examine next. Now, simple group β manages to produce new larger groups $\beta\gamma$ and $\alpha\beta\gamma$, when combined with groups γ and $\alpha\gamma$, respectively. On the other hand, the group containing γ fails to produce new groups that meet the requirements when combined with the succeeding elements. Following exactly the same procedure, we generate groups xy and xz when examining group x, and groups yz and xyz from processing group y. Moreover, each derived group is associated with a unique location where *all* its members meet, regardless of their own recursive structure since only the part of the route to the next meeting location and beyond remains to be decided. Therewith, location $\mu_{\alpha\beta}$ serves as the meeting place for groups

α and β, so that from then on group $\alpha\beta$ is formed to represent the respective travelers' journey towards their common destination ω. Likewise, at $\mu_{\alpha\gamma}$ is where groups α and γ meet to form group $\alpha\gamma$. Similarly, β unites with γ at $\mu_{\beta\gamma}$. At $\mu_{\alpha\beta\gamma}$, three parts are met into a larger group. Additionally, locations μ_{xy}, μ_{xz}, μ_{yz} and μ_{xyz} serve as meeting places for the elements of groups xy, xz, yz and xyz, respectively.

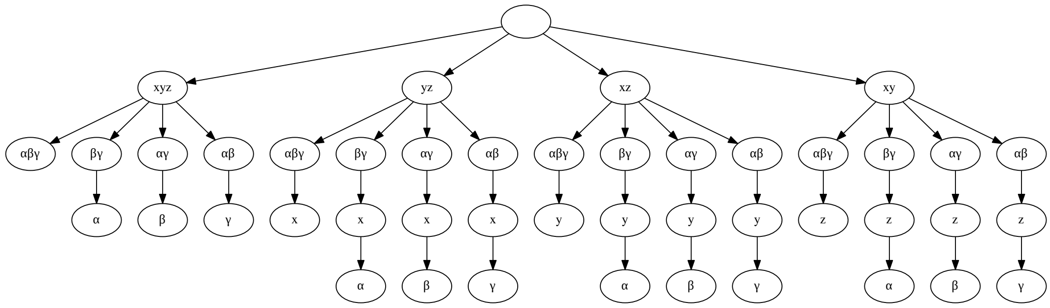


Figure 6: Grouping-tree instance for six travelers starting from locations $x, y, z, \alpha, \beta, \gamma$, respectively.

For the remainder of this section, we will study how Alg. 5 builds the grouping-tree given a collection of groups from the group-list produced in Alg. 4. At the end of the execution of Alg. 4, we start examining the elements of the list in reverse order, and each time we invoke Alg. 5. Hence, the first group we encounter, namely xyz, becomes the first child-node of the root of the grouping-tree, as depicted in Fig. 6. Next, group yz, will also be placed right under the root, since both its elements exist in the previously inserted group and it cannot be subsumed by it. Likewise, we insert groups xz and xy for they have at least one common element with each of the aforementioned groups. Now, upon the insertion of group $\alpha\beta\gamma$, we ascertain whether it is disjoint with any of the groups under the root. Since this is the case, it will become the child of each of these nodes, and therefore, will not be placed under the root. Similarly, we insert at depth 2 groups $\beta\gamma, \alpha\gamma, \alpha\beta$ under all nodes. Consequently, all nodes at depth 1 have exactly the same children. However, this in not the case at depth 3. When examining simple group z, we place it under xy only, and since all its child-nodes, $\alpha\beta\gamma, \beta\gamma, \alpha\gamma, \alpha\beta$, are disjoint with z, they will all become responsible for a copy of it. Likewise, group y is inserted under all children of group xz, whereas trivial group x finds its place under the children nodes of group yz. We find at depth 4 the last groups we shall examine, namely trivial groups α, β and γ, which are placed under trivial groups x, y and z.

Furthermore, it is hard to miss that the path from any leaf to the root corresponds to a valid grouping that congregates groups that were created at this processing phase mostly. More importantly, only when necessary, at the bottom levels of the tree, we will find groups of the previous processing phases that were needed so as to complement any valid grouping. In effect, whatever groups of the input failed to find a suitable spot in the grouping-tree, they are naturally pruned and prevented from being processed during the next phases. In plain words, these groups are safely discarded because they are better represented within larger groups that now improve the experience for a greater number of travelers. The reader may find in the appendix a series of lemmas on the soundness of the methods operating over the grouping-tree.

Arguably, the most influential groupings are the ones that comprise a small number of large groups. This is the case with grouping $\langle xyz, \alpha\beta\gamma \rangle$, the first grouping returned by an in-order traversal of the hierarchy. This is exactly the route plan that is shown in Fig. 4, where we have two groups of users that are joined at the destination. Each of those groups is produced through merging, the way we described earlier. Notably, which will be the groupings that will eventually prevail and be passed on to the next phase for further processing depends exclusively on their scores and the bounds thereof. All qualified groupings must have a lower bound that is less than the upper bounds of all groupings that were generated from. Now, let $\tau_{xyz,\alpha\beta\gamma}$ the lowest upper bound of all generated groupings. Hence, all groupings with a lower bound that surpasses $\tau_{xyz,\alpha\beta\gamma}$ will be rejected due to the fact that they cannot lead to a better solution than any of the eligible groupings can.

For simplicity and ease of presentation, we assume that only the groupings that congregate less than three groups are qualified, and the travelers starting from locations α, β and γ cannot be combined with the travelers starting from locations x, y and z, by convention. Consequently, Alg. 4 will eventually return a list containing the following groupings: $\langle xyz, \alpha\beta\gamma\rangle$, $\langle xyz, \beta\gamma, \alpha\rangle$, $\langle xyz, \alpha\gamma, \beta\rangle$, $\langle xyz, \alpha\beta, \gamma\rangle$, $\langle yz, \alpha\beta\gamma, x\rangle$, $\langle xz, \alpha\beta\gamma, y\rangle$ and $\langle xy, \alpha\beta\gamma, z\rangle$, which in turn will be inserted into the heap of Alg. 3, where they will be processed according to their rank. For example, when examining the next grouping $\langle xy, \alpha\beta\gamma, z\rangle$, a series of additional candidate groupings with more complex structure will be derived, namely groupings $\langle\langle xy, \alpha\beta\gamma\rangle, z\rangle$, $\langle xy, \langle\alpha\beta\gamma, z\rangle\rangle$ and $\langle\langle xy, z\rangle, \alpha\beta\gamma\rangle$, associated with locations $\mu_{\langle xy,\alpha\beta\gamma\rangle,z}$, $\mu_{xy,\langle\alpha\beta\gamma,z\rangle}$ and $\mu_{\langle xy,z\rangle,\alpha\beta\gamma}$, respectively. Same as before, a new instance of a grouping-tree will be constructed, and then, only the best of the new groupings will be returned for further processing, until we are left with only one grouping that outranks all the rest, either by comparison for those groupings that were eventually fully processed, or in terms of their bounds (for those groupings that were effectively pruned).

4 Experimental Evaluation

In this section we assess the performance of all methods we propose in this paper. We adopt two important metrics: (i) the total *execution time* required until each method returns its result, (ii) the *quality* of the result returned by each method. The scores of the respective cost functions are used for this purpose.

Parameter	**Range**	**Default**
number of travelers	$4,5,7$	4
coverage area	$40\%, 50\%, 70\%$	40%
distance to destination	$40\%, 50\%, 70\%$	40%
self-affinity parameter	$40\%, 50\%, 70\%$	40%

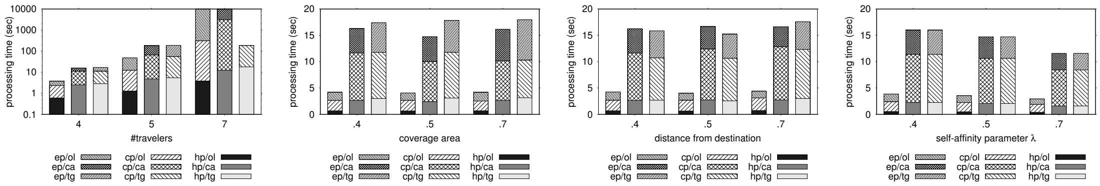
Table 2: System parameters.

A variety of different parameters has been employed so as to expose the strengths and the weaknesses of each method: (1) The number of travelers n. (2) The size of coverage area from which the starting posts are selected It is defined as a percentage over the nodes of the road network that are closer to a node selected at random. (3) The distance to the final destination that the travelers need to cover. This is expressed as a percentage over the remaining nodes of the road network that are immediately beyond the coverage area. Last but not least, (4) the affinity factor λ_{ii} has an indirect impact on performance by affecting the cardinality of the search space. Note that for simplicity we only vary the self-affinity factors, the remaining amount $(1-\lambda_{ii})$ is equally distributed among all fellow travelers. In Table 2, we present all parameters along with the range of values that they take and their default values.

Furthermore, we use real datasets that were obtained from [1] that correspond to the cities of Oldenburg (OL), San Joaquin County (TG) and to the road network of California (CA). Their sizes vary from over 7,000 edges to approximately 24,000, and are associated with a number of nodes that ranges between 6,000 and a little over 21,000 nodes. Upon each of these datasets we created query sets by varying the aforementioned parameters, one at a time, while the rest take their default values. All queries were run in an AMD Opteron 6134 processor running at 2.3GHz.

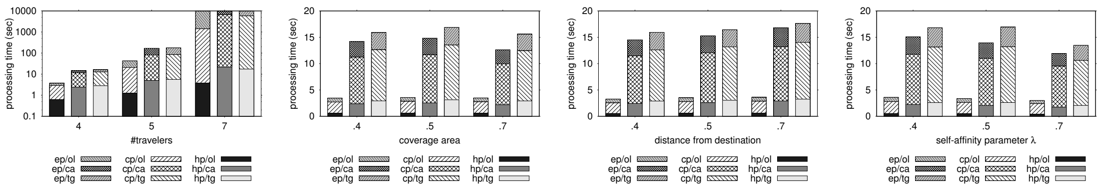
A general observation throughout our evaluation, whose results are depicted in Figures 7, 8, 9 and 10, is that albeit the fastest method, HP always returns a suboptimal solution due to the greedy heuristics employed. CP is faster than EP because of the better way it derives new groupings and adopts more effective pruning techniques at the cost of being a more complex method that has to manage and coordinate a number of heaps. Notably, we observe similar performance patterns for both types of ranking, namely *min-max* and *min-avg*. For each configuration we depict one bar per evaluated road network, starting from left to right (with regard to their sizes), we have OL for Oldenburg, CA for California, and TG for San Joaquin County. The road network of California has approximately the same size as TG but is qualitatively different in a sense that it describes a road network consisted of interstates and highways, unlike the rest that model urban areas.

Figures 7 and 8 illustrate the total processing time over all studied parameters. The combination XX/YY in the legend means that method XX was used with dataset YY, e.g., the bar for CP/CA depicts the results obtained by using the CP (clustered processing) method for the CA (California) road network, and so on.



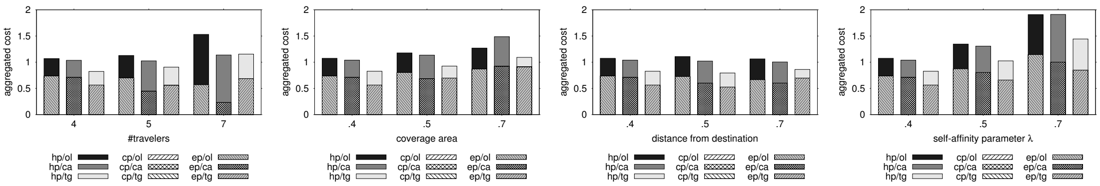
(a) (b) (c) (d)

Figure 7: Processing time for different configurations for *min-max* ranking.



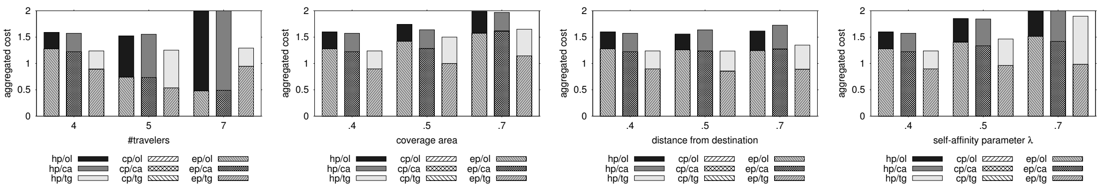
(a) (b) (c) (d)

Figure 8: Processing time for different configurations for *min-avg* ranking.



(a) (b) (c) (d)

Figure 9: Aggregated costs for different configurations for *min-max* ranking.



(a) (b) (c) (d)

Figure 10: Aggregated costs for different configurations for *min-avg* ranking.

Clearly, time grows exponentially in terms of the number of travelers in Figures 7(a) and 8(a) shown in log-scale. This manifests the importance of this parameter and the significant computation cost imposed when a large number of travelers is involved. This result manifests the exponential complexity of the problem arising from the increased number of route plans that can be derived with the different combinations of travelers. In Figures 7(b) and 8(b), we observe that the cover area, which corresponds to the percentage over the overall area from which the starting posts are selected, has a slightly irregular effect on execution time. On the other hand, the distance of the destination from the starting posts has also a negative effect on performance, as shown in Figures 7(c) and 8(c). Those two parameters affect only the time spent for graph search and do not increase the number of possible route plans to be examined. Consequently, their impact is not as strong as the number of travelers. Therefore, we see that relatively slight change in performance. Quite remarkably, we can easily observe in Figures 7(d) and 8(d) that by increasing the self-affinity parameter (or equivalently by increasing how much each traveler enjoys traveling with others other than alone), makes it easier and faster to retrieve the globally optimal route plan. The reason lies with the fact that when the relaxation factors maximize their effect (high λ_{ii}-values and low λ_{ij}-values with $i \neq j$), the cost function over the search space takes a steeper form. Hence, the prioritization policy we enforce with the usage of the heaps is more effective, and along with our pruning approach, processing becomes much more efficient.

In Figures 9 and 10, we illustrate the aggregated scores that the returned results achieve with regard to the studied parameters. Obviously, EP and CP return results of better quality when compared with HP. This is due to the fact that HP, as its name implies, relies on greedy heuristics, and hence, cannot consider all possible grouping outcomes. CP and EP return the same (optimal) result corresponding to equi-height bars. In Figures 9(a) and 10(a) we study the impact that the number of travelers has on the result quality. In particular, for *min-max* ranking criteria, there is an insignificant change in the normalized aggregated cost for the optimal methods as the number of travelers grows, and it slightly increases for the heuristic method. For *min-avg*, the aggregated cost apparently diminishes, especially for the optimal methods, because of the additional affinity parameters, the product of which in the ranking function causes the decrease. Moreover, the gap of HP from the optimal solution is further deepened, even though a diminishing trend is still evident. In other words, the heuristic solution impairs as the number of travelers increases. Arguably, this observation provides a clear evidence of the trade-off and when the time overhead is justified the usage of CP is the appropriate choice. On the other hand, the effect of the growing coverage area becomes evident in Figures 9(b) and 10(b), where the costs increase mainly due to the longer distances the travelers have to cover until they meet. In Figures 9(c) and 10(c), the distance of the destination has an insignificant impact on the scores because of the aggregated relaxation factor that diminishes the part of the cost associated with the final step of the route plan after all travelers meet. Last but not least, the aggregated costs surprisingly increase in Figures 9(d) and 10(d) due to the first parts of the route plans where the travelers travel alone, as the relaxation factors increase. We would expect those scores to drop as larger groups would be formed early on in an effort to minimize the overall travel cost. However, when the travelers are that spread apart (default value for the cover area is 40%), this is still a difficult task, and the overhead from those early steps makes its presence known here.

5 Related Work

To the best of our knowledge, this is the first time that the OMCP query is proposed. Additionally, no other similar method has been ever proposed to consider multiple meeting locations, mainly due to the complexity of the problem, which can be proved to be NP-hard. On the other hand, the simpler form of the query that searches for just one such point that optimizes the objective, is related to queries focusing on finding the *Optimal Meeting Point* (OMP). This line of work does not consider any notion of common destination shared by the travelers to succeed the meet up. Previous works on the OMP problems focus on finding a location (on a road network or in the Euclidean space) that minimize the aggregate distance (e.g., maximum or average) from a set of query locations, e.g., the starting points of users who want to meet. The OMP problem has been well studied in the Euclidean space in [3], where bounds on the MBRs of R-trees are used to guide search.

The first work to study and solve the OMP problem in road networks was [7]. The authors propose three methods: (i) the Incremental Euclidean Restriction (IER), that exploits spatial data structures by utilizing Euclidean lower bounds in combination with A* search and an incremental Euclidean Aggregate Nearest Neighbor method, (ii) the Threshold Algorithm (TA), which is motivated by aggregate top-k query processing techniques and (iii) the Concurrent Expansion (CE) algorithm, which concurrently expands the searched area in an effort to avoid the shortest path computations, since they can be expensive because they traverse the network nodes in a less systematic way. CE is a more general method which does not rely on bounds and can be applied for any cost function on the network's edges. More recently, the authors of [5] address the OMP problem relying on finding all the split-points of the road network that satisfy the query requirements to minimize the specified distance parameters. For any point of the road network its split-points are the points that halve the distance from that point over any cyclic path it participates over the graph. Among these points they select the point with the smallest sum of network distances from all starting posts. However, this paradigm imposes a significant computational cost. The authors in [6] aim at retrieving the OMP, by associating the nodes of the road networks with their spatial locations. However, this is somewhat limiting as the road network could have much different characteristics than the Euclidean space where the nodes lie (e.g., the weights in a road network can represent the traveling time between any two nodes, or the max speed of a vehicle along that path, etc.) Nonetheless, two distinct approaches are proposed in [6]. The first, limits the search among the nodes that are located within the convex hull formed by the query points. We note that, this is only a filtering step and there is no sense of prioritization during search. On the other hand, the second approach calculates the centroid of the query points and then enacts a nearest neighbor search. We note that the centroid is a fictitious point and its nearest node in the road networks does not always capture the desired properties as it does not necessarily minimize the aggregated cost. Moreover the solutions obtained cannot be claimed to be optimal. The continuous monitoring of the OMP in the Euclidean space was more recently studied in [2], where the query input points (i.e., user locations) are assumed to be constantly moving and the objective is to minimize the computation and communication cost for continuously updating their OMP.

6 Conclusions

To recapitulate, in this paper we proposed a novel query type that retrieves the *Optimal Multiple Connecting Points (OMCPs)*, or the best locations where groups of travelers having a common destination should meet so that the aggregated (either maximum, or average) traveler cost is minimized. In this context, we proposed three distinct approaches for retrieving the OMCPs. Namely, (i) *Heuristic Processing (HP)* relies on a fast and greedy approach that finds a sub-optimal solution, (ii) *Extensive Processing (EP)* adopts a pruning approach based on score bounds that a refined route can achieve, so as to reduce the number of processed routes, and (iii) *Clustered Processing (CP)*, which employs a best-first search approach to process the most promising routes first, so as to enforce stricter bounds, and hence, impose even more effective pruning, in an effort to ameliorate overall response-times. Notably, EP and CP retrieve the globally optimal solution. Albeit the fastest method, HP returns a suboptimal solution due to the greedy heuristics employed. CP is faster than EP because of the smarter way it derives new candidate solutions and adopts more effective pruning techniques at the cost of being a more complicated method that has to manage and coordinate a number of heaps. Most importantly, as shown in our evaluation, our methods are efficient overall, in spite of addressing an intractable problem. Finally, we developed a framework in Java leveraging the methods proposed in this paper, and made it available at `http://connect.cs.ualberta.ca/omcp/`.

Appendix

Lemma 1 *The union of the groups associated with the nodes on any path of the grouping-tree produced at any processing phase congregates all travelers.*

Proof By induction, we show that it holds after processing phase 0, we accept it is true for processing phase i, and last, we prove that it also holds after processing phase $i+1$.
Phase 1: We start from the fact that the initial grouping used as input comprises all travelers. This is the input for the first processing phase where each traveler naturally corresponds to a disjoint group to each of his fellow travelers' groups. In effect, the grouping-tree of that rudimentary grouping takes the form of a plain list for that matter. Then, at the end of the first processing step, the output group-list consists of two parts, the first containing a number of the input elements, and the second, the qualified groups derived from mergings and joins that can occur.

Now, the produced grouping-tree, given this basic input, will have at its first level the groups that are pairwise non-disjoint, meaning that each group from the first level intersects somehow with every other group of the same level. The second layer of the hierarchy congregates again groups that are pairwise non-disjoint, but also they have the property of being disjoint to the group that is associated with their parent node. Therefore, it is impossible to find groups from the first layer at any subsequent layer. Now, even if none derived group fulfils that specification to find a spot at the second layer, we can be certain that the remaining elements of the input can complement the top groups, arranged in the form of a linear list, one (complementary) group per node. However, we have given priority to the larger newly derived groups that congregate those simpler elements in more preferable combinations.

Similarly, the k-th layer of each subtree is built, and whenever we run out of derived groups, we know that the remaining elements from the input will fill the tree-branch, one group per node. Therefore, all paths from the root comprise a mixture of new and old groups, arranged in such a way that a valid grouping is formed. And thus, only valid groupings are passed on for further processing.
Phase i: We accept that the hypothesis is true and the i-th processing phase constitutes a grouping-tree with paths that correspond to valid groupings. This also holds for its output, which is produced based on the paths of the grouping tree.
Phase i+1: We start from the fact that the input of this processing phase is necessarily a valid grouping, as the output of the i-th phase. In addition, its elements are the atomic parts that will constitute any derived group.

Of course, if some are not used in a more complex combination, they can definitely be used to complement some groupings when needed. In essence, there is no chance to be needing just some part of these atomic elements so as to form a valid grouping, and therefore, all the necessary pieces of the puzzle are already in the group-list, and arranged appropriately inside the grouping-tree.

As far as the tree construction is concerned, we examine the input in reverse and we put at the top levels of the hierarchy the groups that were derived in this phase, and each time we encounter a disjoint node, we add a child node. Otherwise, if none such node is encountered, we place it as a sibling node under the same disjoint parent node. The elements of the group-list that were not derived during this phase can be placed in tandem as leaf-nodes. When no other disjoint group can be found within the group-list, it means that all travelers are already involved in that particular route plan. ■

Corollary 1 *All paths from the root of the grouping-tree from any processing phase correspond to valid groupings.*

Proof By contradiction. Let a path of the grouping tree, that it does not correspond to a valid grouping for either of the two following reasons:

there is at least one pair of non-disjoint groups does not hold by construction: a node is not subsumed by another at any level unless they are associated with fully disjoint groups.

does not involve all travelers contradicts Lemma 1.

Thus, the statement holds for the opposite cannot be true. ■

Lemma 2 *The grouping-tree produced at any phase includes all possible groupings that involve the newly generated groups.*

Proof By contradiction. Let grouping $\langle c_1, c_2, \cdots, c_k \rangle$, that was not represented in the grouping-tree. Since, it constitutes a valid grouping, all its elements are mutually disjoint and the union of all their elements corresponds to the set of all travelers. For this grouping to have been missed, it means that at least one of its elements was not encountered when building the grouping-tree.

Now, we assume that the elements of the missed grouping appear inside the group-list in the following order $\{c'_1, c'_2, \cdots, c'_k\}$, with other groups intervening between them in the original group-list. Hence, since we examine the elements in reverse order, the first element of the missed grouping we encounter is c'_k. The elements of c'_k are naturally represented in all other groupings by definition (otherwise they would not constitute valid groupings). All groups of the group-list preceding c'_k are non-disjoint, and therefore, c'_k will not be subsumed by any of them, but instead will be placed as their sibling. If c'_k was disjoint with any preceding group, then it would appear in another position subsumed by any of them, and therefore, no valid grouping with these exact elements would exist. From then on, when we will encounter each remaining element of the missed grouping, it will be put (among other places) exactly below the previously seen group of the sequence, since it is disjoint with all groups of the sequence that we have encountered.

And thus, there is no missing grouping in the grouping-tree since the initial hypothesis is false. Therefore, all groupings are well represented within the hierarchy. ■

References

[1] Real datasets for spatial databases: Road networks and points of interest. `http://www.cs.fsu.edu/~lifeifei/SpatialDataset.htm`.

[2] Jing Li, Man Lung Yiu, and Nikos Mamoulis. Efficient notification of meeting points for moving groups via independent safe regions. In *ICDE 2013*, pages 422–433, 2013.

[3] Dimitris Papadias, Yufei Tao, Kyriakos Mouratidis, and Chun Kit Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Trans. Database Syst.*, 30(2):529–576, 2005.

[4] George Tsatsanifos, Alexandr Petcovici, and Mario A. Nascimento. Meet-and-go: Finding optimal single connecting points considering companionship preferences (demo paper). In *SIGSPATIAL '15, Belleview, WA, USA, 2015*, 2015.

[5] Zhengdao Xu and Hans-Arno Jacobsen. Processing proximity relations in road networks. In *SIGMOD 2010*, pages 243–254, 2010.

[6] Da Yan, Zhou Zhao, and Wilfred Ng. Efficient algorithms for finding optimal meeting point on road networks. *PVLDB*, 4(11):968–979, 2011.

[7] Man Lung Yiu, Nikos Mamoulis, and Dimitris Papadias. Aggregate nearest neighbor queries in road networks. *IEEE Trans. Knowl. Data Eng.*, 17(6):820–833, 2005.