Constrained Path Search with Submodular Function Maximization

Xuefeng Chen[1], Xin Cao[2], Yifeng Zeng[3], Yixiang Fang[4], Sibo Wang[5], Xuemin Lin[2], Liang Feng[1]
[1]College of Computer Science, Chongqing University, China
[2]UNSW, Australia [3]Northumbria University, UK [4]CUHK-Shenzhen, China [5]CUHK, China
xfchen@cqu.edu.cn, xin.cao@unsw.edu.au, yifeng.zeng@northumbria.ac.uk
fangyixiang@cuhk.edu.cn, swang@se.cuhk.edu.hk, lxue@cse.unsw.edu.au, liangf@cqu.edu.cn

***Abstract*—In this paper, we study the problem of constrained path search with submodular function maximization (CPS-SM). We aim to find the path with the best submodular function score under a given constraint (e.g., a length limit), where the submodular function score is computed over the set of nodes in this path. This problem can be used in many applications. For example, tourists may want to search the most diversified path (e.g., a path passing by the most diverse facilities such as parks and museums) given that the traveling time is less than 6 hours. We show that the CPS-SM problem is NP-hard. We first propose a concept called "submodular α-dominance" by utilizing the submodular function properties, and we develop an algorithm with a guaranteed error bound based on this concept. By relaxing the submodular α-dominance conditions, we design another more efficient algorithm that has the same error bound. We also utilize the way of bi-directional path search to further improve the efficiency of the algorithms. We finally propose a heuristic algorithm that is efficient yet effective in practice. The experiments conducted on several real datasets show that our proposed algorithms can achieve high accuracy and are faster than one state-of-the-art method by orders of magnitude.**

I. Introduction

The path search problem is an essential task in many applications such as navigation and tourist trip planning. Most path search systems (such as Google Maps) compute paths based on a single criterion which is usually either the total path length or the total travel time. However, in practice, the user often needs to consider multiple criteria during the path search. The studies on path search with constraints [1]–[5] aim to find the optimal path under a given constraint (such as a length limit).

Generally, in the path search with constraints, each path can compute two scores: an objective score (to be optimized) and a budget score (for constraint checking). The objective score of the path can be defined variously according to applications. In this paper, we assume a general submodular function computed over a set of nodes, and we aim to find the path with the maximized submodular function score (computed over the nodes in the path) from an origin s to a destination t satisfying a given budget limit Δ. We call this problem the constrained path search with submodular maximization (CPS-SM), which is first studied by Chekuri and Pal [6]. It can model various practical problems such as path-planning for robot-based environment sensing [7], travel route search for maximizing the satisfaction of a traveler's preference [8], door-to-door marketing [9], etc. We use a simple application called *the most diversified path query* to explain the CPS-SM problem intuitively. Assume a tourist who is spending holidays in a city, and she would like to plan a trip such that she will not travel too far away from her hotel. The tourist can express her preferences on points-of-interests (POIs) as some keywords (e.g., shopping malls, parks, and museums). Since not all preferences can be satisfied in a single trip due to the path length limit (the budget), the trip planner can return the most diversified path satisfying a given constraint that covers as many of her preferences as possible. This will enable the tourist to experience many different attractions and services interested to her. We can set the tourist's hotel as the origin and the destination, a distance threshold as the path length constraint, and the score of a path is the number of distinct types of POIs passed by it.

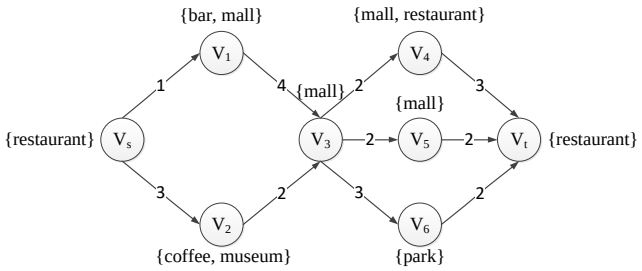


Fig. 1. A toy road network.

Example 1. *Fig. 1 shows a toy road network where each location is associated with some keywords representing its POIs' categories that are interested to a querying user, and each edge is associated with a cost (e.g., travel time). Assume that the user wants to find the most diversified path under a length limit $\Delta = 10$. If using a sum function to compute the number of keywords, $P_1 = \langle v_s, v_1, v_3, v_4, v_t \rangle$ is found because it has 7 keywords (not distinct!), but it only passes 3 types of POIs. In contrast, CPS-SM aims to return $P_2 = \langle v_s, v_2, v_3, v_6, v_t \rangle$ which passes by the most types of locations.*

In most constrained path search problems, the objective function is a simple sum function. For example, the objective

score is computed as the shortest path length [3]. Hence, given two partial paths P_1 and P_2 from the origin s to the same node v_j, if both two scores of P_1 are worse than those of P_2, P_1 can be pruned safely, because by further extending the two paths with the same node v_j, the two scores of path $P_1 \rightarrow v_j$ are also worse than those of path $P_2 \rightarrow v_j$. Unfortunately, this does not hold in the CPS-SM problem, since the objective score is computed by a submodular function over the set of nodes in a path. Given that P_1 is worse than P_2 on v_j, we cannot guarantee that $P_1 \rightarrow v_j$ is worse than $P_2 \rightarrow v_j$ due to the submodular properties. This renders existing methods on constrained path search inapplicable to CPS-SM. The submodular function poses more challenges on developing algorithms for CPS-SM, which can be viewed as a submodular optimization problem [10].

Most submodular optimization problems are hard to solve [8], [11]–[14], and we show that solving CPS-SM is NP-hard. There exist the recursive greedy (RG) algorithm [6] and the GreedyDP algorithm [7] to answer the CPS-SM query. However, the two algorithms have very high complexity and thus they are not scalable and are not practical to solve real-world problems. It is shown that RG costs more than 10^4 seconds on a small graph with only 22 nodes [15] and GreedyDP takes 4.0×10^3 seconds on a dataset with only 91 nodes [7]. The study [9] proposes the Generalized Cost-Benefit (GCB) algorithm for submodular optimization with routing constraints. The algorithm can be adapted to answer the CPS-SM query, but one problem is that it can only return paths with a guaranteed error bound under a budget limit much smaller than the original limit Δ. Its returned result has poor quality as shown in our experiments. The work [8] studies a special case of CPS-SM, and the proposed weighted A* (WA*) algorithm can get approximate solutions, but it has high costs as it stores a large number of partial paths during the search.

Due to the submodularity, it is meaningless to compare two partial paths using their current scores as analyzed above. Thus, we aim to find out their relationship when they are extended to the target node with the same path. We first propose a concept called "submodular α-dominance" by using the properties of the submodular function. This concept enables us to compare the quality of two partial paths from the origin to the same node. Specifically, for two such paths satisfying the submodular α-dominance condition, we can keep only one, and we prove that the final objective score obtained by extending the kept path is no smaller than $\frac{1}{\alpha}$ ($\alpha > 1$) times of that obtained using the discarded path. Thus, we can prune a large number of partial paths during path search with a guaranteed error bound.

Based on this concept, we develop the first algorithm SDD (Submodular α-Dominance based Dijkstra) which finally yields an approximation ratio of $(\frac{1}{\alpha})^c$, where α is a parameter larger than 1 and c is the maximum number of nodes in a path (excluding the source and target nodes) under limit Δ, which can be bounded by $\lfloor \frac{\Delta}{b_{min}} \rfloor - 1$ where b_{min} is the minimal budget value on edges. Next, we further improve the efficiency of the algorithm by relaxing the submodular α-dominance conditions and then propose another algorithm OSDD. OSDD algorithm has the same approximation ratio but it can discard more partial paths during the search and thus is much faster. To further improve the efficiency of SDD and OSDD, we utilize the bi-directional search to reduce the number of generated labels for both methods,and we obtain another two algorithms Bi-SDD and Bi-OSDD. We show that these two algorithms have an approximation ratio of $(\frac{1}{\alpha})^{c+1}$, and are more efficient than SDD and OSDD, respectively. Although the theoretical approximation ratios of our algorithms are small when c is large, in practice our algorithms can always achieve high-quality results as shown in the experimental study, as these ratios are the worst-case errors. We finally devise an efficient heuristic algorithm CBFS with a polynomial time complexity for solving the CPS-SM query. Although it does not have any guaranteed error bounds, it can return results with good quality in practice. The experimental study conducted on several real-world data sets shows that our proposed algorithms can achieve high accuracy and are faster than the state-of-the-art algorithm WA* [8] by orders of magnitude.

Contributions. In summary, we make the following contributions: i) We propose a concept called "submodular α-dominance" by using the properties of the submodular function, and design two algorithms with error bounds based on this concept to solve CPS-SM; ii) We utilize the bi-directional search method to speed up the two proposed algorithms; iii) We devise an efficient yet effective heuristic algorithm that has a polynomial-time complexity; iv) We conduct experiments on real-world datasets at the scale of millions of nodes, and the results demonstrate the excellent performance of our proposed algorithms in terms of both efficiency and accuracy.

II. Related Work

Path search is an important problem that has received much attention due to its wide applications. One of the most classic problems is the shortest path search [16], [17], and there exist a lot of variations of the problem such as finding near-shortest and k-shortest simple paths [18], [19]. However, in many applications, users often need to consider multiple criteria during the path search. One way to address this issue is through skyline path queries [20]–[22], which obtain a set of paths that cannot dominate each other in all criteria. Another way to search the multi-criteria paths is to find the best path based on one criterion under some constraints on other criteria. The orienteering problem [23] searches paths through a set of given nodes, such that the total collected score is maximized and a given time budget is not exceeded. The selected traveling sales man problem [24] determines a simple circuit of maximal total profit whose length does not exceed a preset value. The work [25] proposes new formulations for finding the s-t shortest path by visiting a given set of nodes. The trip planning query [26] aims to find the shortest path from a source location to a target location passing by all user-specified types of locations. The studies [27]–[29] consider the problem which searches the shortest path from a specified starting point passing by a sequence of user-specified types of locations. The

constrained shortest path query [1]–[5], [30] searches the s-t shortest path under a constraint on another criterion. The keyword-aware optimal route query [31], [32] considers the user's preferences which are expressed by keywords. It aims to return a route covering all specified keywords with the best objective value satisfying a budget limit.

All the aforementioned studies use an accumulative function to formulate the objective function in the path search, but such a function cannot measure the objective score in many applications such as the coverage of users' preferences [8], the entropy of selected locations [9], the likelihood of information [33], and the influence of regions [34], [35]. Our problem uses a submodular function to compute the objective score of a path, which makes the problem more challenging.

The CPS-SM problem is first studied by Chekuri and Pal [6] and they proposed the recursive greedy (RG) algorithm. RG obtains the solutions by recursively guessing the middle node and the budget cost reaching the middle node, and it is a quasi-polynomial time algorithm with high complexity. There also exists a GreedyDP algorithm [7] for CPS-SM, which constructs a zero-suppressed binary decision diagram (ZDD) to represent all feasible solutions for a query, and then uses the ZDD to find an approximate result. Since the size of the ZDD grows exponentially with the number of nodes in the graph, GreedyDP also has high time complexity. RG and GreedyDP cannot be used to solve real-world problems.

The studies [8], [9] of submodular optimization under routing constraints are relevant to our problem. The ORS-KC problem [8] defines a special submodular function for path search. The problem of submodular optimization with routing constraints [9] aims to find a route with the best submodular score under a length constraint, but it does not specify the source and target nodes. Their proposed algorithms can be adapted to answer our CPS-SM query, and we denote them by WA* and GCB respectively. As shown in the experimental study, WA* is not scalable to large-scale real-world datasets, and the result of GCB has poor quality.

III. Problem Formulation

Our problem is defined on a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ which consists of a set of nodes $\mathcal{V}$ and a set of edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. $\mathcal{G}$ could be used to model many graph data. For example, if $\mathcal{G}$ is a road network, each node $v \in \mathcal{V}$ represents a location, and each edge $\langle v_i, v_j \rangle \in \mathcal{E}$ represents a directed path from v_i to v_j, and the edges can be associated with some weights such as travel time or distance. Although we only consider directed graphs in this paper, it is straightforward to extend our solutions to undirected graphs.

Definition 1. Path. *Given a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $P = \langle v_1, v_2 \cdots, v_m \rangle$ is called a path from v_1 to v_m, where each pair of consecutive nodes in P has an edge in $\mathcal{E}$.*

We consider general paths in our problem which can have cycles, and we use two scores to measure the paths, namely the *budget score* and the *objective score*. Each edge $< v_i, v_j > \in \mathcal{E}$ is attached with a budget value (e.g., the travel time or distance etc.), which is denoted by $b(v_i, v_j)$. For any set of nodes $V_x \subseteq \mathcal{V}$, we compute its objective score by a submodular function $f()$ over V_x, i.e., $f(V_x)$ (e.g., the degree of POI types coverage, social influence etc.). Given a path, its two scores are defined formally as below.

Definition 2. Path Budget Score. *Given a path $P = \langle v_1, v_2, \cdots, v_m \rangle$, its budget score is computed as the sum of the budget values on all the edges along the path, i.e.,*

$$BS(P) = \sum_{i=1}^{m-1} b(v_i, v_{i+1}) \tag{1}$$

Definition 3. Path Objective Score. *Given a path $P = \langle v_1, v_2, \cdots, v_m \rangle$, its objective score is is computed by $f()$ over the nodes in P, i.e.,*

$$OS(P) = f(\cup_{v_i \in P} v_i) \tag{2}$$

For example, in Fig. 1, assume that the objective value is the number of distinct POI types in a path, its objective score is computed as $OS(P) = |\cup_{v_i \in P} \mathcal{K}(v_i)|$, where $\mathcal{K}(v_i)$ is the set of keywords on v_i. Given $P_1 = \langle v_s, v_1, v_3, v_4, v_t \rangle$, we can get $OS(P) = |f\{restaurant, bar, mall\}| = 3$ and $BS(P) = 10$. To be specific, $OS(P)$ is submodular means that $OS(P_1 \rightarrow v_i) - OS(P_1) \geq OS(P_2 \rightarrow v_i) - OS(P_2)$, given $P_1 \subseteq P_2$ and $v_i \notin P_1$, where P_1(resp. P_2) $\rightarrow v_i$ composes a new path. It is obvious that computing the number of distinct POI types in a path is submodular.

Definition 4. Constraint Path Search with Submodular Maximization (CPS-SM) *Given a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and a search query $\mathcal{Q} = \langle v_s, v_t, \Delta \rangle$, where v_s is the source node, v_t is the target node, and Δ specifies a budget limit, we aim to find a path P starting from v_s and ending at v_t such that*

$$\begin{aligned} &P = argmax_P \;\; OS(P) \\ &subject\ to \;\; BS(P) \leq \Delta \end{aligned} \tag{3}$$

Example 2. *We use an example to explain the CPS-SM query. Fig. 1 shows a toy road network with locations and their POI categories represented by keywords. The most diversified path search is a special problem of the CPS-SM query. $OS(P)$ is computed as the number of distinct keywords in the path, while $BS(P)$ is the sum of the values on all the edges along the path. Given a query $\mathcal{Q} = \langle v_s, v_t, \Delta = 10 \rangle$, the optimal solution is $P_2 = \langle v_s, v_2, v_3, v_6, v_t \rangle$, as it contains the most distinct keywords and satisfies the budget constraint.*

The optimal route search for keyword coverage (ORS-KC) problem [8] is a special problem of the CPS-SM query, which is proved to be NP-hard. It is easy to prove that the problem of solving the CPS-SM query is also NP-hard by reducing the problem from the ORS-KC problem.

Theorem 1. *Answering the CPS-SM query is NP-hard.*

IV. Efficient Algorithm SDD

A simple idea to solve CPS-SM is to perform an exhaustive search from the source node v_s to the target node v_t. We

call a path from v_s to a node v_i ($v_i \neq v_t$) a "partial path". First, for each neighbor of v_s we can obtain a single-edge partial path. Next, from all the partial paths, we select the one with the smallest budget score and expand it to obtain more partial paths. If the newly generated path exceeds the budget limit, we discard it. We repeat this process until no new feasible partial paths can be obtained, and we return the best path on v_t as the result. It is obvious that such a method is computationally prohibitive, because all feasible partial paths on each node have to be kept during the search. As discussed in Section I, the A*-style algorithms [8] can solve CPS-SM, but it is also time-consuming since it still needs to maintain too many partial paths on each node.

In order to reduce the number of partial paths enumerated during the search, we develop an efficient algorithm that can prune a lot of partial paths and return results with a guaranteed error bound. Before presenting the algorithm, we first introduce the following important definitions.

Definition 5. Node Label. *Each node label L_i^k represents a path P_i^k from the source node v_s to a node v_i, which is in format of $\langle BS, OS, P_i^k \rangle$, where BS and OS represent the budget score and the objective score of P_i^k, respectively.*

The node label is used to store the information of the partial paths from the origin, and we store the labels on each node.

As analyzed in the introduction, existing studies on path search usually perform pruning based on the path dominance relationship, but this is not applicable in our problem due to the submodular objective score computation. It is meaningless to compare two partial paths from the origin to the same node using their current scores. Instead, we aim to compare them when they are extended to the target node with the same path. To achieve this, we define the relationship between the pruned partial path and the preserved partial path as "submodular α-dominance", as shown in Definition 6.

Definition 6. Submodular α-Dominance. *Let L_i^k and L_i^l be two labels representing two different paths from v_s to node v_i. We say L_i^k α-dominates L_i^l in submodularity iff $L_i^k.BS \leq L_i^l.BS$ and $\frac{OS(P_i^k)}{OS(P_i^l)-OS(P_i^k \cap P_i^l)+\varepsilon} \geq \frac{1}{\alpha-1}$, where $\alpha > 1$, ε is an arbitrarily small positive real number.*

This definition aims to guarantee that after they are extended with the same path to the target node, the objective score obtained from the preserved path is no smaller than $\frac{1}{\alpha}$ times of that obtained using the pruned path, which is shown in the following lemma. Note that the budget scores of the two paths are also checked. It is because, for the same extension from the current node to the target, it is possible that the path obtained by the shorter one is feasible but the path obtained by the longer one is infeasible (length exceeds Δ). Thus, if we use the longer path to dominate the short one, we may miss feasible solutions extended from the current node.

Lemma 1. *Let P_{ij} be any path from v_i to v_j, and L_i^k submodular α-dominates L_i^l, it is true that: $\frac{OS(P_i^k \cup P_{ij})}{OS(P_i^l \cup P_{ij})} \geq \frac{1}{\alpha}$.*

Proof. *As L_i^k submodular α-dominates L_i^l, we have $\frac{OS(P_i^k)}{OS(P_i^l)-OS(P_i^k \cap P_i^l)+\varepsilon} \geq \frac{1}{\alpha-1}$, we can get $(\alpha-1)OS(P_i^k) \geq OS(P_i^l) - OS(P_i^k \cap P_i^l) + \varepsilon$. According to the properties of submodular function, we have $OS((P_i^k \cap P_i^l) \cup P_{ij}) - OS(P_i^k \cap P_i^l) \geq OS(P_i^l \cup P_{ij}) - OS(P_i^l)$. Thus, $\frac{OS(P_i^k \cup P_{ij})}{OS(P_i^l \cup P_{ij})} \geq \frac{OS(P_i^k \cup P_{ij})}{OS((P_i^k \cap P_i^l) \cup P_{ij}) - OS(P_i^k \cap P_i^l) + OS(P_i^l) + \varepsilon} \geq \frac{OS(P_i^k \cup P_{ij})}{OS((P_i^k \cup P_i^l) \cup P_{ij}) + (\alpha-1)OS(P_i^k)} \geq \frac{1}{\alpha}$. The final step holds true because both $OS((P_i^k \cap P_i^l) \cup P_{ij})$ and $OS(P_i^k)$ are not larger than $OS(P_i^k \cup P_{ij})$.* ■

Algorithm 1: SDD

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $\mathcal{Q} = \langle v_s, v_t, \Delta \rangle$
Output: A path P

1 Initialize a min-priority queue $Q \leftarrow \emptyset$;
2 Initialize a label list $L(v_i)$ for each node $v_i \in V$;
3 Create a label $\langle 0, OS(v_s), (v_s) \rangle$ and insert it into Q;
4 **while** Q *is not empty* **do**
5 $\quad L_i^k \leftarrow Q$.dequeue();
6 $\quad$ **if** v_i *is* v_t **then** Insert L_i^k into $L(v_i)$;
7 $\quad$ **if** $DomCheck(L_i^k, L(v_i))$ == *true* **then continue**;
8 $\quad$ Insert L_i^k into $L(v_i)$, and obtain P_i^k from L_i^k;
9 $\quad$ **for** *each edge* (v_i, v_j) **do**
10 $\quad\quad$ Create a path $P_j^l \leftarrow P_i^k \cup v_j$;
11 $\quad\quad$ **if** $BS(P_j^l) > \Delta$ **then continue**;
12 $\quad\quad$ Create a new label $L_j^l \leftarrow \langle BS(P_j^l), OS(P_j^l), (P_j^l) \rangle$;
13 $\quad\quad$ Q.enqueue(L_j^l);
14 **If** $L(v_t)$ is $\emptyset$ **return** "No feasible path exists";
15 **Else return** the path P with the largest OS value in $L(v_t)$;

Next, we are ready to present our algorithm SDD (Submodular α-Dominance based Dijkstra). The basic process of SDD is similar to that of CP-Dijk [36] for constrained shortest path search. We traverse the graph in a shortest-first manner (in terms of the budget score), because based on Definition 6 only shorter paths can dominate longer paths. In addition, with this order, we can terminate the algorithm once we reach a label whose budget score exceeds Δ. We prune all the labels submodular α-dominated by other labels, and extend the non-dominant partial labels in ascending order of their budget scores.

The pseudocode is presented in Algorithm 1. The algorithm starts by initializing a min-priority queue Q which stores partial labels in ascending order of their budget scores, and a label list for each node, and inserting the label containing v_s into Q (lines 1-3). Next, it dequeues the candidate labels from Q one by one until Q becomes empty (lines 4-13). In each while-loop, the algorithm inserts the label whose path connects v_s and v_t into $L(v_i)$ (line 6). For the label L_i^k, the algorithm uses a function $DomCheck(L_i^k, L(v_i))$ to check whether L_i^k is dominated by any label in $L(v_i)$ according to Definition 6, and then only extends the non-dominant labels (lines 7-13). For a non-dominant label L_i^k, the algorithm obtains P_i^k and creates a new path P_j^l for each outgoing neighbor v_j of v_i (lines 8-10). If $BS(P_j^l) \leq \Delta$, the algorithm creates a new label and enqueues it into Q (lines 11-13). Finally, we return the path P with the largest OS value in $L(v_t)$ (lines 14-15).

We show the approximation ratio of SDD in the following theorem.

Theorem 2. *Algorithm SDD offers a $(\frac{1}{\alpha})^c$-approximate answer to the optimal solution, where c is the maximum number of nodes in a path (excluding the source and target nodes) under the budget limit Δ, bounded by $\lfloor \frac{\Delta}{b_{min}} \rfloor - 1$.*

Proof. Let the optimal path be $P^{opt} = \langle v_s, v_{o_1}, \cdots, v_{o_{c-1}}, v_{o_c}, v_t \rangle$, and the best path returned by SDD be P^{sd}.

Assume that P^{opt} is submodular α-dominated by another partial path $P^k_{o_1}$ on node v_{o_1} during the search. According to Lem. 1, we know that $\frac{OS(P^k_{o_1} \cup P_{o_1 t})}{OS(P^{opt})} \geq \frac{1}{\alpha}$, where $P_{o_1 t} = \langle v_{o_1}, \cdots, v_{o_{c-1}}, v_{o_c}, v_t \rangle$. Next, the path $P^k_{o_1} \cup P_{o_1 t}$ may be submodular α-dominated by a partial path $P^k_{o_2}$ on node v_{o_2}. Hence, we have $\frac{OS(P^k_{o_2} \cup P_{o_2 t})}{OS(P^k_{o_1} \cup P_{o_1 t})} \geq \frac{1}{\alpha}$. In the worst case, each partial path that dominates a path on a node v_x is dominated by another path on the subsequent node v_{x+1} in P^{opt}, which means that for any $0 < x \leq c$, $\frac{OS(P^k_{o_{x+1}} \cup P_{o_{x+1} t})}{OS(P^k_{o_x} \cup P_{o_x t})} \geq \frac{1}{\alpha}$.

Recall that SDD returns the path with the largest OS score on destination v_t, i.e., P^{sd}. Thus, we know that $OS(P^{sd}) \geq \frac{1}{\alpha} OS(P^k_{o_c} \cup P_{o_c t}) \geq (\frac{1}{\alpha})^2 OS(P^k_{o_{c-1}} \cup P_{o_{c-1} t}) \geq \ldots \geq (\frac{1}{\alpha})^c OS(P^k_{o_1} \cup P_{o_1 t}) \geq (\frac{1}{\alpha})^c OS(P^{opt})$, where c can be bounded by $\lfloor \frac{\Delta}{b_{min}} \rfloor - 1$. Finally, we can conclude that $\frac{OS(P^{sd})}{OS(P^{opt})} \geq (\frac{1}{\alpha})^{\lfloor \frac{\Delta}{b_{min}} \rfloor - 1}$. ■

Complexity Analysis. We denote the largest number of labels on a node as $|L_{max}|$. In the worst case, Algorithm SDD would generate $|L_{max}||\mathcal{V}|$ labels, and thus it requires at most $|L_{max}||\mathcal{V}|$ loops. To enqueue and dequeue all of those generated labels with the min-priority queue, Algorithm SDD takes $O(|L_{max}||\mathcal{V}|(\log(|L_{max}||\mathcal{V}|)))$ time in the worst case. In each loop, it takes at most $O(|L_{max}|)$ time to do submodular α-dominance checks. Therefore, in total the worst time complexity of SDD is $O(|L_{max}||\mathcal{V}|(\log(|L_{max}||\mathcal{V}|) + |L_{max}|))$. $|L_{max}|$ depends on α and Δ. In the worse case $|L_{max}|$ could be exponential to $|\mathcal{V}|$, but in practice it is much smaller. A larger α would reduce $|L_{max}|$ since more labels can be dominated and pruned during the search, and a smaller Δ would reduce $|L_{max}|$ as well since fewer partial paths need to be enumerated.

V. Efficient Algorithm OSDD

Utilizing the submodular α-dominance, SDD is able to prune partial paths while guaranteeing an error bound. Can we further improve its efficiency while preserving its accuracy?

From Definition 6, it can be observed that discovered partial paths with small objective scores can only dominate very few other partial paths. If we can improve the pruning power of discovered partial paths, we can obtain faster algorithms. Let's revisit Lem. 1, and we set v_j as the destination v_t. In the proof, if we have $L^k_i.BS \leq L^l_i.BS$ and $\frac{OS(P^k_i \cup P_{it})}{OS(P^l_i) - OS(P^k_i \cap P^l_i)} \geq \frac{1}{\alpha - 1}$ (where P_{it} is a path from v_i to the target node v_t), we still can guarantee that $\frac{OS(P^k_i \cup P_{it})}{OS(P^l_i \cup P_{it})} \geq \frac{1}{\alpha}$. This gives us an idea of relaxing the condition of submodular α-dominance to obtain larger pruning power, as $OS(P^k_i \cup P_{it})$ is usually much larger than $OS(P^k_i)$. However, to guarantee the error bound, we need to show that the above condition holds for all possible P_{it}, which is intractable to check this.

To this end, we propose a lazy extension optimization, and show that it is not necessary to check the relaxed condition for all possible paths from the current node to the target node. The optimization first quickly obtains some feasible paths from source to target (for getting $P^k_i \cup P_{it}$), and then extends some labels in a later stage to avoid unnecessary computations. We first introduce the following definitions which are used in this optimization.

Definition 7. Extended Node Label. *For a partial path P^k_i on node v_i, we denote its extended node label as $\mathcal{L}^k_i$ which is in format of $\langle BS, OS, OS_{fea}, \hat{OS}_{fea}, P^k_i \rangle$, where $\mathcal{L}^k_i.OS_{fea}$ and $\mathcal{L}^k_i.\hat{OS}_{fea}$ represent the OS score of a feasible result path from P^k_i to v_t (i.e., $OS(P^k_i \cup P_{it})$) and its estimated value (i.e., $\hat{OS}(P^k_i \cup P_{it})$), respectively.*

Given that $P^k_i \cup P_{it}$ is a feasible path, we can relax the condition of submodular α-dominance as below.

Definition 8. Forward-looking Submodular α-Dominance (FS-α-Dominance). *$\mathcal{L}^k_i$ FS-α-dominates $\mathcal{L}^l_i$ iff $\mathcal{L}^k_i.BS \leq \mathcal{L}^l_i.BS$ and $\frac{\mathcal{L}^k_i.OS_{fea}}{OS(P^l_i) - OS(P^k_i \cap P^l_i) + \varepsilon} \geq \frac{1}{\alpha - 1}$.*

This dominance cannot work when P_{it} is unknown. To address this, we define another type of dominance using $\mathcal{L}^k_i.\hat{OS}_{fea}$.

Definition 9. Potential Submodular α-Dominance (PS-α-Dominance). *$\mathcal{L}^k_i$ PS-α-dominates $\mathcal{L}^l_i$ iff $\mathcal{L}^k_i.BS \leq \mathcal{L}^l_i.BS$ and $\frac{\mathcal{L}^k_i.\hat{OS}_{fea}}{OS(P^l_i) - OS(P^k_i \cap P^l_i) + \varepsilon} \geq \frac{1}{\alpha - 1}$.*

PS-α-dominance is used to find some labels which are potentially FS-α-dominated at the beginning of the optimization. Hence, we need to do FS-α-dominance check for the PS-α-dominated labels in the next step.

We proceed to present SDD with the lazy extension optimization, which is denoted by OSDD (Optimized SDD). The algorithm has two phases: the first phase aims to quickly find some feasible solutions (i.e., paths from v_s to v_t satisfying the budget constraint) by extending some partial labels based on Definition 9, and the second phase searches the final solutions based on Definition 8. The details are shown in Algorithm 2.

OSDD uses two min-priority queues Q_1 and Q_2 to store partial labels in ascending order of their budget scores in the first and second phase respectively (line 1). In the first phase, the algorithm first creates label lists for all nodes, generates a new extended label on v_s, and then inserts it into Q_1 (lines 2-3). Next, it dequeues and deals with the candidate labels from Q_1 one by one until Q_1 becomes empty (lines 4-17). Different from SDD, the algorithm also uses $PSDomCheck()$ to check whether a non-α-dominated label $\mathcal{L}^k_i$ is PS-α-dominated by any label in $\mathcal{L}(v_i)$ before extending it (line 8). If it is true, the algorithm enqueues it into Q_2 for further processing in the second phase. Otherwise, the algorithm inserts the label into $\mathcal{L}^k_i$ and extends it by generating a new extended label for each outgoing neighbor v_j of v_i (lines 8-17). We estimate $\hat{OS}_{fea}$

Algorithm 2: OSDD

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $\mathcal{Q} = \langle v_s, v_t, \Delta \rangle$
Output: A path P

1 Initialize two min-priority queues $Q_1 \leftarrow \emptyset$ and $Q_2 \leftarrow \emptyset$;
2 Initialize a label list $\mathcal{L}(v_i)$ for each node $v_i \in V$;
3 Create a new extended label $\langle 0, OS(v_s), OS(v_s), OS(v_s), (v_s)) \rangle$ and insert it into Q_1;
4 **while** Q_1 *is not empty* **do**
5 $\quad \mathcal{L}_i^k \leftarrow Q_1$.dequeue();
6 $\quad$ **if** v_i *is* v_t **then** Insert $\mathcal{L}_i^k$ into $\mathcal{L}(v_i)$;
7 $\quad$ **if** $DomCheck(\mathcal{L}_i^k, \mathcal{L}(v_i))$ == *true* **then continue**;
8 $\quad$ **if** $PSDomCheck(\mathcal{L}_i^k, \mathcal{L}(v_i))$ == *true* **then**
9 $\quad\quad Q_2$.enqueue($\mathcal{L}_j^l$);
10 $\quad$ **else**
11 $\quad\quad$ Insert $\mathcal{L}_i^k$ into $\mathcal{L}(v_i)$, and obtain P_i^k from $\mathcal{L}_i^k$;
12 $\quad\quad$ **for** *each edge* (v_i, v_j) **do**
13 $\quad\quad\quad$ Create a path $P_j^l \leftarrow P_i^k \cup v_j$;
14 $\quad\quad\quad$ **if** $BS(P_j^l) > \Delta$ **then continue**;
15 $\quad\quad\quad$ $\hat{OS}_{fea} = \hat{OS}(P_j^l \cup P_{jt}) = \frac{OS(P_j^l) * \Delta}{BS(P_j^l)}$;
16 $\quad\quad\quad$ Create a new extended label $\mathcal{L}_j^l \leftarrow \langle BS(P_j^l), OS(P_j^l), null, \hat{OS}_{fea}, (P_j^l) \rangle$;
17 $\quad\quad\quad Q_1$.enqueue($\mathcal{L}_j^l$);
18 Update OS_{fea} for existing labels according to labels in $\mathcal{L}(v_t)$;
19 **while** Q_2 *is not empty* **do**
20 $\quad \mathcal{L}_i^k \leftarrow Q_2$.dequeue();
21 $\quad$ **if** v_i *is* v_t **then** Insert $\mathcal{L}_i^k$ into $\mathcal{L}(v_i)$;
22 $\quad$ **if** $FSDomCheck(\mathcal{L}_i^k, \mathcal{L}(v_i))$ == *true* **then continue**;
23 $\quad$ Insert $\mathcal{L}_i^k$ into $\mathcal{L}(v_i)$, and obtain P_i^k from $\mathcal{L}_i^k$;
24 $\quad$ **for** *each edge* (v_i, v_j) **do**
25 $\quad\quad$ Create a path $P_j^l \leftarrow P_i^k \cup v_j$;
26 $\quad\quad$ **if** $BS(P_j^l) > \Delta$ **then continue**;
27 $\quad\quad$ Create a new extended label $\mathcal{L}_j^l \leftarrow \langle BS(P_j^l), OS(P_j^l), OS(P_j^l), null, (P_j^l) \rangle$;
28 $\quad\quad Q_2$.enqueue($\mathcal{L}_j^l$);
29 **if** $\mathcal{L}(v_t)$ *is* $\emptyset$ **then return** "No feasible route exists";
30 **else return** P with the largest OS value in $\mathcal{L}(v_t)$;

for each new label in a simple way, as computing exact OS_{fea} is complex and has high costs. And OS_{fea} is not used in this phase and thus is set to $null$. (lines 15-16).

After finishing the first phase, the algorithm can get some feasible solutions in $\mathcal{L}(v_t)$ which is used to update OS_{fea} values for stored labels (line 18). With the labels having accurate OS_{fea} values, the second phase can use FS-α-Dominance to prune the remaining labels in Q_2. This process is similar to the process of SDD, while it uses $FSDomCheck()$ to check the FS-α-Dominance relationship between labels, rather than α-Dominance (line 22). As the new extended labels created in the second phase do not have accurate OS_{fea} value, the algorithm sets $OS_{fea} = OS(P_j^l)$ and $\hat{OS}_{fea} = null$ for new labels (line 27). At the last step, the existing optimal path P will be returned (lines 29-30).

OSDD can prune more partial paths during the search, as in the first phase we quickly obtain some feasible paths which enable us to apply the FS-α-Dominance check which has larger pruning power.

Example 3. *Consider a CPS-SM query for the most diversified path search with a budget limit* $\Delta = 10$ *in* $\mathcal{G}$ *in Fig. 1. We run Algorithm OSDD with* $\alpha = 1.2$ *to solve the query. After some steps, OSDD now gets a label* $\mathcal{L}_3^1 = \langle BS = 5, OS = 4, OS_{fea} = 0, \hat{OS}_{fea} = 8, P_3^1 = \langle v_s, v_2, v_3 \rangle \rangle$ *stored on node* v_3*. Next, it dequeues* $\mathcal{L}_3^2 = \langle BS = 5, OS = 3, OS_{fea} = 0, \hat{OS}_{fea} = 6, P_3^2 = \langle v_s, v_1, v_3 \rangle \rangle$ *from the min-priority queue* Q_1*. In the submodular* α*-dominance check, as* $\frac{\mathcal{L}_3^1.OS}{OS(P_3^2) - OS(P_3^2 \cap P_3^l)} = 4 < 5 = \frac{1}{\alpha - 1}$*,* $\mathcal{L}_3^2$ *is not submodular* α*-dominated by* $\mathcal{L}_3^1$*. While in the PS-*α*-dominance check, as* $\frac{\mathcal{L}_3^2.\hat{OS}_{fea}}{OS(P_3^2) - OS(P_3^2 \cap P_3^l)} = 8 > 5 = \frac{1}{\alpha - 1}$*,* $\mathcal{L}_3^2$ *is PS-*α*-dominated by* L_3^1*. Thus, the algorithm holds on* $\mathcal{L}_3^2$ *by inserting it into another min-priority queue* Q_2*. After phase 1, OSDD can get a feasible path* $P_t^1 = \langle v_s, v_2, v_3, v_6, v_t \rangle$*. In phase 2, it dequeues* $\mathcal{L}_3^2$ *from* Q_2*, we can use* P_t^1 *and* $\mathcal{L}_3^1$ *to do FS-*α*-dominance check for* $\mathcal{L}_3^2$*. Since* $\frac{\mathcal{L}_3^2.OS_{fea}}{OS(P_3^2) - OS(P_3^2 \cap P_3^l)} = 5 \geq 5 = \frac{1}{\alpha - 1}$*,* $\mathcal{L}_3^2$ *is FS-*α*-dominated by* $\mathcal{L}_3^1$*. Therefore, OSDD can prune* $\mathcal{L}_3^2$ *(no need to extend it), and get a solution with an error bound at the end of the phase 2.*

We next show that OSDD has the same approximation ratio as SDD in the following theorem.

Theorem 3. *Algorithm OSDD also offers an approximation ratio of* $(\frac{1}{\alpha})^c$.

Proof. We first prove the following lemma.

Let P_{ij} *be a known path from* v_i *to* v_j*,* P_n *be any unknown path from* v_i *to* v_j*. If* $\mathcal{L}_i^k$ *FS-*α*-dominates* $\mathcal{L}_i^l$*, it is true that: either* $\frac{OS(P_i^k \cup P_n)}{OS(P_i^l \cup P_n)} \geq \frac{1}{\alpha}$ *or* $\frac{OS(P_i^k \cup P_{ij})}{OS(P_i^l \cup P_n)} \geq \frac{1}{\alpha}$.

As $\mathcal{L}_i^k$ FS-α-dominates $\mathcal{L}_i^l$, we have $\frac{OS(P_i^k \cup P_{ij})}{OS(P_i^l) - OS(P_i^k \cap P_i^l)} \geq \frac{1}{\alpha - 1}$, we can get $(\alpha - 1)OS(P_i^k \cup P_{ij}) \geq OS(P_i^l) - OS(P_i^k \cap P_i^l)$. According to the properties of submodular function, we have $OS((P_i^k \cap P_i^l) \cup P_n) - OS(P_i^k \cap P_i^l) \geq OS(P_i^l \cup P_n) - OS(P_i^l)$.

If $OS(P_i^k \cup P_{ij}) \geq OS(P_i^k \cup P_n)$, we have:

$$\begin{aligned}\frac{OS(P_i^k \cup P_{ij})}{OS(P_i^l \cup P_n)} &\geq \frac{OS(P_i^k \cup P_{ij})}{OS((P_i^k \cap P_i^l) \cup P_n) - OS(P_i^k \cap P_i^l)) + OS(P_i^l)} \\ &\geq \frac{OS(P_i^k \cup P_{ij})}{OS((P_i^k \cap P_i^l) \cup P_n) + (\alpha - 1)OS(P_i^k \cup P_{ij})} \\ &\geq \frac{OS(P_i^k \cup P_{ij})}{OS(P_i^k \cup P_n) + (\alpha - 1)OS(P_i^k \cup P_{ij})} \geq \frac{1}{\alpha}\end{aligned}$$

If $OS(P_i^k \cup P_{ij}) \leq OS(P_i^k \cup P_n)$, we have:

$$\begin{aligned}\frac{OS(P_i^k \cup P_n)}{OS(P_i^l \cup P_n)} &\geq \frac{OS(P_i^k \cup P_n)}{OS((P_i^k \cap P_i^l) \cup P_n) - OS(P_i^k \cap P_i^l)) + OS(P_i^l)} \\ &\geq \frac{OS(P_i^k \cup P_n)}{OS((P_i^k \cap P_i^l) \cup P_n) + (\alpha - 1)OS(P_i^k \cup P_{ij})} \\ &\geq \frac{OS(P_i^k \cup P_n)}{OS(P_i^k \cup P_n) + (\alpha - 1)OS(P_i^k \cup P_n)} \geq \frac{1}{\alpha}\end{aligned}$$

We thus complete the proof for this lemma. It shows that, if $\mathcal{L}_i^k$ FS-α-dominates L_i^l, either the two partial paths are extended to v_t in the same way, or $\mathcal{L}_i^k$ goes through a known path to v_t and $\mathcal{L}_i^l$ goes through an unknown path to v_t, discarding $\mathcal{L}_i^l$ can always guarantee an error bound. As this lemma also guarantees that each dominance only sacrifices $\frac{1}{\alpha}$ accuracy, it is straightforward to prove OSDD has the same approximation ratio as SDD following the proof of Theorem 2. ■

Speeding Up Dominance Check. In Alg. 2, the submodular α-dominance check is terminated until all existing labels are visited or we find an existing label that dominates the new label (in line 7 of Alg. 2). In order to speed up the process, we maintain the existing labels in the descending order of their objective scores. It is expected that when a label's objective score is larger, it is more likely to submodular α-dominate other labels. Note that the newly generated labels always have larger budget scores, and thus we only need to check the objective scores.

Dominance Pre-check for New Labels. In addition, we observe that inserting a new label into a min-priority queue Q has large costs (in lines 9, 17 and 28 of Alg. 2). To reduce the insertion operations for the new labels with low OS values, for a new label $\mathcal{L}_j^l$, we use the label with the maximum objective score in $\mathcal{L}(v_j)$ to check if $\mathcal{L}_j^l$ can be dominated before enqueuing it into Q.

For example, assume that $\alpha = 1.2$ and there exist 3 labels on node v_1, $\mathcal{L}_1^1$ with $\mathcal{L}_1^1.BS = 1$ and $\mathcal{L}_1^1.OS = 1$, $\mathcal{L}_1^2$ with $\mathcal{L}_1^2.BS = 3$ and $\mathcal{L}_1^2.OS = 2$ and $\mathcal{L}_1^3$ with $\mathcal{L}_1^3.BS = 5$ and $\mathcal{L}_1^3.OS = 5$. When we generate a new label $\mathcal{L}_1^4$ with $\mathcal{L}_1^4.BS = 7$ and $\mathcal{L}_1^4.OS = 1$, rather than inserting it into Q_1, we first use $\mathcal{L}_1^3$ to check whether it submodular α-dominates $\mathcal{L}_1^4$. If it is true, we can prune $\mathcal{L}_1^4$ directly to save time.

Complexity Analysis. Let $|\mathcal{L}_{max}^1|$ and $|\mathcal{L}_{max}^2|$ denote the largest numbers of labels on a node in the first and second phases respectively. The second phase is similar to SDD, and the complexity is $O(|\mathcal{L}_{max}^2||\mathcal{V}|(\log(|\mathcal{L}_{max}^2||\mathcal{V}|) + |\mathcal{L}_{max}^2|))$. In the first phase, we also need to enqueue and dequeue all of generated labels, and do submodular α-dominance check and PS-α-dominance check for those labels in the worst case. Then, the complexity of the first phase is $O(|\mathcal{L}_{max}^1||\mathcal{V}|(\log(|\mathcal{L}_{max}^1||\mathcal{V}|)+2|\mathcal{L}_{max}^1|))$. Thus, the complexity of OSDD is $O(|\mathcal{L}_{max}^1||\mathcal{V}|(\log(|\mathcal{L}_{max}^1||\mathcal{V}|) + 2|\mathcal{L}_{max}^1|) + |\mathcal{L}_{max}^2||\mathcal{V}|(\log(|\mathcal{L}_{max}^2||\mathcal{V}|) + |\mathcal{L}_{max}^2|))$. Both $|\mathcal{L}_{max}^1|$ and $|\mathcal{L}_{max}^2|$ are much smaller than $|\mathcal{L}_{max}|$ of SDD in practice, as the FS-α-dominance helps prune more partial paths, and thus OSDD is more efficient.

VI. Bidirectional Search Method

From the complexity analysis of Algorithm SDD and OSDD, we can observe that the time complexity mainly depends on the number of generated labels (partial paths) in both algorithms. More labels are generated during the process, more loops are required and more time is spent on dequeuing and enqueuing labels. The number of partial paths is affected by Δ. With a larger Δ, more possible partial paths need to be enumerated and maintained. Inspired by the bi-directional search of the Dijkstra algorithm, we also employ this method in our algorithms to reduce the number of generated labels.

The basic idea of this method is to first divide the budget limit Δ into two parts Δ_s and Δ_t. Next, we perform two types of search: a forward search from v_s under budget limit Δ_s, and a backward search from v_t under budget limit Δ_t. Hence, in this algorithm, we have two types of labels based on Definition5.

Definition 10. Forward Node Label. *Each forward node label $\overrightarrow{L}_i^k$ represents a path $\overrightarrow{P}_i^k$ from the source node v_s to a node v_i, which is in format of $\langle BS, OS, \overrightarrow{P}_i^k\rangle$, where BS and OS represent the budget score and the objective score of $\overrightarrow{P}_i^k$, respectively.*

Definition 11. Backward Node Label. *Each backward node label $\overleftarrow{L}_i^l$ represents a path $\overleftarrow{P}_i^l$ from a node v_i to the target node v_t, which is in format of $\langle BS, OS, \overleftarrow{P}_i^l\rangle$, where BS and OS represent the budget score and the objective score of $\overleftarrow{P}_i^l$, respectively.*

We still apply the submodular α-dominance to prune labels in both forward and backward searches. Finally, we can find a path from v_s to v_t having a high objective score by merging the forward and backward node labels. The improvement is due to that in both types of search the budget limit is reduced, and thus the number of both types of maintained labels is reduced significantly. Another advantage of this method is that it allows us to estimate the upper bounds of the objective scores for some paths, then we can prune such paths whose OS upper bounds are smaller than the current largest OS value.

We first apply the bi-directional search to the SDD algorithm, and we denote this algorithm as Bi-SDD (Bi-directional SDD), as shown in Alg. 3. The method first calls the function $computeBackwardLable()$ to compute the non-dominated backward node labels for each node (line 2). Next, we begin to compute the non-dominated forward node labels utilizing the backward labels and find the final result (lines 3-26). The process is similar to SDD. The difference is that, for a non-dominant label $\overrightarrow{L}_i^k$ whose budget score is not less than Δ_s, the method extends $\overrightarrow{L}_i^k$ directly utilizing the function $joinLabels()$, which gets a feasible path with the maximum OS value by joining $\overrightarrow{L}_i^k$ with backward node labels in $\overleftarrow{L}(v_i)$ (lines 12-18). In this process, before extending $\overrightarrow{L}_i^k$, we first compute an upper bound of the OS value for $\overrightarrow{P}_i^k$ as $OS_{ub}(\overrightarrow{P}_i^k) = \overrightarrow{L}_i^k.OS + \overleftarrow{L}_i^k.OS$, where $\overleftarrow{L}_i^k$ is the label with the maximum OS value in $\overleftarrow{L}(v_i)$ satisfying $\overleftarrow{L}_i^k.BS \leq \Delta_{rem}$. If $OS_{ub}(\overrightarrow{P}_i^k) \leq OS_{max}$, we can discard it safely. If the budget score of $\overleftarrow{L}_i^k$ is smaller than Δ_s, we further extend the label to obtain new labels (lines 19-24).

How to compute the backward node labels is shown in Alg. 4. The process is similar to SDD. We use the transpose of the original graph, and use v_t as the source node to find non-dominated partial paths satisfying the budget limit Δ_t for each node.

Next, as shown in Alg. 5, we present the details of function $joinLabels()$. It first initializes L_t^l as "null", and then checks labels $\overleftarrow{L}_i^m \in \overleftarrow{L}(v_i)$ one by one in descending order of their OS values, where $\overleftarrow{L}_i^m$ denotes the backward node labels on v_i. For a label $\overleftarrow{L}_i^m$, if the total budget of $\overrightarrow{P}_i^k \cup \overleftarrow{P}_i^m$ is not larger than Δ, the function creates a new path $\overrightarrow{P}_t^l$ by joining $\overleftarrow{P}_i^k$ and $\overleftarrow{P}_i^m$ (lines 3-4). If the upper bound of $\overrightarrow{P}_t^l$'s OS value is not larger than OS_{max}, the process can be terminated (lines 5-6). Otherwise, the function computes $OS(\overrightarrow{P}_t^l)$ and $BS(\overrightarrow{P}_t^l)$, and

Algorithm 3: Bi-SDD

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $\mathcal{Q} = \langle v_s, v_t, \Delta \rangle$, γ
Output: A path P

1 $\Delta_t = \gamma\Delta$, $\Delta_s = \Delta - \Delta_t$, $\mathcal{G}^T \leftarrow$the transpose of $\mathcal{G}$;
2 Compute backward labels on each node by using $computeBackwardLable(\mathcal{G}^T, v_t, v_s, \Delta, \Delta_t)$;
3 Initialize a min-priority queue $Q \leftarrow \emptyset$;
4 Initialize a label list $\overrightarrow{L}(v_i)$ for each node $v_i \in V$;
5 Create a new label $\langle 0, OS(v_s), (v_s)) \rangle$ and insert it into Q;
6 Initialize OS_{max} to store the current best path OS score;
7 **while** Q *is not empty* **do**
8 $\quad \overrightarrow{L}_i^k \leftarrow Q$.dequeue();
9 $\quad$ **if** v_i *is* v_t **then** Insert $\overrightarrow{L}_i^k$ into $\overrightarrow{L}(v_i)$, update OS_{max} and **continue**;
10 $\quad$ **if** $DomCheck(\overrightarrow{L}_i^k, \overrightarrow{L}(v_i))$ == *true* **then continue**;
11 $\quad$ Insert $\overrightarrow{L}_i^k$ into $\overrightarrow{L}(v_i)$;
12 $\quad$ **if** $\overrightarrow{L}_i^k.BS \geq \Delta_s$ **then**
13 $\quad\quad \Delta_{rem} = \Delta - \overrightarrow{L}_i^k.BS$;
14 $\quad\quad$ From $\overleftarrow{L}(v_i)$ find $\overleftarrow{L}_i^k$ with the maximum OS value satisfying $\overleftarrow{L}_i^k.BS \leq \Delta_{rem}$;
15 $\quad\quad OS_{ub}(\overrightarrow{P}_i^k) = \overrightarrow{L}_i^k.OS + \overleftarrow{L}_i^k.OS$;
16 $\quad\quad$ **if** $OS_{ub}(\overrightarrow{P}_i^k) \leq OS_{max}$ **continue**;
17 $\quad\quad \overrightarrow{L}_t^l \leftarrow joinLabels(\overrightarrow{L}_i^k, \overleftarrow{L}(v_i), \Delta_{rem}, OS_{max})$;
18 $\quad\quad$ **if** $\overrightarrow{L}_t^l != null$ **then** Insert $\overrightarrow{L}_t^l$ into $\overrightarrow{L}(v_t)$;
19 $\quad$ **else**
20 $\quad\quad$ obtain $\overrightarrow{P}_i^k$ from $\overrightarrow{L}_i^k$;
21 $\quad\quad$ **for** *each edge* (v_i, v_j) **do**
22 $\quad\quad\quad$ Create a path $\overrightarrow{P}_j^l \leftarrow \overrightarrow{P}_i^k \cup v_j$;
23 $\quad\quad\quad$ **if** $BS(\overrightarrow{P}_j^l) > \Delta$ **then continue**;
24 $\quad\quad\quad$ Q.enqueue($\langle BS(\overrightarrow{P}_j^l), OS(\overrightarrow{P}_j^l), (\overrightarrow{P}_j^l) \rangle$);
25 **if** $L(v_t)$ *is* $\emptyset$ **then return** "No feasible path exists";
26 **else return** P with the largest OS value in $\overrightarrow{L}(v_t)$;

Algorithm 4: $computeBackwardLable(\mathcal{G}^T, v_t, v_s, \Delta, \Delta_t)$

1 Initialize a min-priority queue $Q \leftarrow \emptyset$;
2 Initialize a label list $\overleftarrow{L}(v_i)$ for each node $v_i \in V$;
3 Create a label $\langle 0, OS(v_t), (v_t)) \rangle$ and insert it into Q;
4 **while** Q *is not empty* **do**
5 $\quad \overleftarrow{L}_i^k \leftarrow Q$.dequeue();
6 $\quad$ **if** $DomCheck(\overleftarrow{L}_i^k, \overleftarrow{L}(v_i))$ == *true* **then continue**;
7 $\quad$ Insert $\overleftarrow{L}_i^k$ into $\overleftarrow{L}(v_i)$, and obtain $\overleftarrow{P}_i^k$ from $\overleftarrow{L}_i^k$;
8 $\quad$ **for** *each edge* (v_i, v_j) **do**
9 $\quad\quad$ Create a path $\overleftarrow{P}_j^l \leftarrow \overleftarrow{P}_i^k \cup v_j$;
10 $\quad\quad$ **if** $BS(\overleftarrow{P}_j^l) > \Delta_t$ ***or*** $BS(\overleftarrow{P}_j^l) > \Delta$ **then continue;**
11 $\quad\quad$ Q**.enqueue(**$\langle BS(\overleftarrow{P}_j^l), OS(\overleftarrow{P}_j^l), (\overleftarrow{P}_j^l) \rangle$**);**

if a better path is found, it updates OS_{max} and creates a new label to update $\overrightarrow{L}_t^l$ using $\overrightarrow{P}_t^l$ (lines 7-10). Finally, it returns $\overrightarrow{L}_t^l$ (line 11).

Example 4. *Consider a CPS-SM query for the most diversified path search again. Assume that a query with budget limit is on $\mathcal{G}$ as shown in Fig. 1. When we run Bi-SDD with $\alpha = 1.2$ and $\gamma = 0.5$ to answer the query, we first obtain a backward label list $\overleftarrow{L}(v_3)$ on node v_3 using $computeBackwardLable(\mathcal{G}^T, v_t, v_s, \Delta, \Delta_t)$. $\overleftarrow{L}(v_3)$ contains 3 labels: $\overleftarrow{L}_3^1 = \langle BS = 5, OS = 3, \overleftarrow{P}_3^1 = \langle v_3, v_6, v_t \rangle \rangle$, $\overleftarrow{L}_3^2 = \langle BS = 4, OS = 2, \overleftarrow{P}_3^2 = \langle v_3, v_5, v_t \rangle \rangle$, and $\overleftarrow{L}_3^3 = \langle BS = 5, OS = 2, \overleftarrow{P}_3^3 = \langle v_3, v_4, v_t \rangle \rangle$. For the forward label $\overrightarrow{L}_3^1 = \langle BS = 5, OS = 3, \overrightarrow{P}_3^1 = \langle v_s, v_1, v_3 \rangle \rangle$, as $\overrightarrow{L}_3^1.BS \geq \Delta_s = 5$, Bi-SDD extends $\overrightarrow{L}_3^1$ by joining it with the labels in $\overleftarrow{L}(v_3)$. Firstly, $\overrightarrow{L}_t^1 = \langle BS = 10, OS = 5, P_t^1 = \langle v_s, v_1, v_3, v_6, v_t \rangle \rangle$ is found by joining $\overrightarrow{L}_3^1$ with $\overleftarrow{L}_3^1$, and OS_{max} is updated as 5. After that, the algorithm checks $\overleftarrow{L}_3^2$. Since $OS_{up}(\overrightarrow{P}_3^2) = 5 \leq OS_{max}$, which means the upper bound of OS value is not larger than OS_{max}, the algorithm stops and return $\overrightarrow{L}_t^1$ as the result of extending $\overrightarrow{L}_3^1$ to v_t. Similarly, Bi-SDD can extend $\overrightarrow{L}_3^2 = \langle BS = 5, OS = 4, \overrightarrow{P}_3^2 = \langle v_s, v_2, v_3 \rangle \rangle$ to v_t using $\overleftarrow{L}(v_3)$ directly. In this example, to extend $\overrightarrow{L}_3^1$ and $\overrightarrow{L}_3^2$ to v_t, Bi-SDD generates 6 labels in the initialization (line 2 of Alg. 3), where SDD needs to generate 12 labels which all are longer than the generated labels in Bi-SDD. This example illustrates that Bi-SDD is more efficient than SDD in practice.*

Algorithm 5: $joinLabels(\overrightarrow{L}_i^k, \overleftarrow{L}(v_i), \Delta_{rem}, OS_{max})$

1 $\overrightarrow{L}_t^l \leftarrow null$;
2 **foreach** $\overleftarrow{L}_i^m \in \overleftarrow{L}(v_i)$ **do**
3 $\quad$ **if** $\overleftarrow{L}_i^m.BS + \overrightarrow{L}_i^k.BS \leq \Delta$ **then**
4 $\quad\quad \overrightarrow{P}_t^l \leftarrow \overrightarrow{P}_i^k \cup \overleftarrow{P}_j^m$;
5 $\quad\quad OS_{ub}(\overrightarrow{P}_t^l) = \overrightarrow{L}_i^k.OS + \overleftarrow{L}_i^m.OS$;
6 $\quad\quad$ **if** $OS_{ub}(\overrightarrow{P}_t^l) \leq OS_{max}$ **then break**;
7 $\quad\quad$ Compute $OS(\overrightarrow{P}_t^l)$;
8 $\quad\quad$ **if** $OS(\overrightarrow{P}_t^l) > OS_{max}$ **then**
9 $\quad\quad\quad OS_{max} = OS(\overrightarrow{P}_t^l)$;
10 $\quad\quad\quad L_t^l \leftarrow \langle \overrightarrow{L}_i^k.BS + \overleftarrow{L}_i^m.BS, OS(\overrightarrow{P}_t^l), (\overrightarrow{P}_t^l) \rangle$;
11 **return** $\overrightarrow{L}_t^l$;

Theorem 4. *The approximation ratio of Algorithm Bi-SDD is $(\frac{1}{\alpha})^{c+1}$ for the CPS-SM query, .*

Proof. We denote the optimal path as $P^{opt} = \langle v_s, \cdots, v_i, \cdots, v_t \rangle$. We use P_s^{opt} to represent the path $\langle v_s, \cdots, v_i \rangle$, and P_t^{opt} to represent the path $\langle v_i, \cdots, v_t \rangle$.

We prove this theorem based on Lem. 1, which shows that after each α-dominance, we lose at most $\frac{1}{\alpha}$ accuracy. Hence, similar to the proof of Theorem 1, if P_s^{opt} is dominated by other paths on v_i, even in the worst case, we must have a forward partial path $\overrightarrow{P}_i^k$ such that for any path P_{it} from v_i to v_t, and we have $OS(\overrightarrow{P}_i^k \cup P_{it}) \geq (\frac{1}{\alpha})^{\lfloor \frac{\Delta_t}{b_{min}} \rfloor} OS(P_s^{opt} \cup P_{it})$.

Next, let us consider the backward node labels on v_i. If P_t^{opt} is dominated on v_i, even in the worst case, we must have a backward partial path $\overleftarrow{P}_i^m$ such that $OS(\overleftarrow{P}_i^m \cup P_{si}) \geq (\frac{1}{\alpha})^{\lfloor \frac{\Delta_t}{b_{min}} \rfloor} OS(P_t^{opt} \cup P_{si})$, where P_{si} is any path from v_s to v_i. As $\overrightarrow{P}_i^k$ could be P_{si}, we can get that $OS(\overleftarrow{P}_i^m \cup \overrightarrow{P}_i^k) \geq (\frac{1}{\alpha})^{\lfloor \frac{\Delta_t}{b_{min}} \rfloor} OS(P_t^{opt} \cup \overrightarrow{P}_i^k)$. Next, as P_t^{opt} could be P_{it}, we can further get $OS(\overleftarrow{P}_i^m \cup \overrightarrow{P}_i^k) \geq (\frac{1}{\alpha})^{\lfloor \frac{\Delta_t}{b_{min}} \rfloor} (\frac{1}{\alpha})^{\lfloor \frac{\Delta_s}{b_{min}} \rfloor} OS(P_s^{opt} \cup P_t^{opt}) = (\frac{1}{\alpha})^{\lfloor \frac{\Delta}{b_{min}} \rfloor} OS(P^{opt})$.

$\overleftarrow{P}_i^m \cup \overrightarrow{P}_i^k$ is just one possible combination in our algorithm Bi-SDD, and thus the final result of Bi-SDD must be no worse than $(\frac{1}{\alpha})^{\lfloor \frac{\Delta}{b_{min}} \rfloor} OS(P^{opt})$, and we complete the proof. ■

Complexity Analysis. We use $|L^f_{max}|$ and $|L^b_{max}|$ to denote the largest number of labels on a node in the forward node label list and backward node label list respectively. $computeBackwardLable$ is similar to SDD, and the complexity is $O(|L^b_{max}||\mathcal{V}|(\log(|L^b_{max}||\mathcal{V}|) + |L^b_{max}|))$. For the forward search, in each loop we also need to do $joinLabels$, which takes $O(|L^b_{max}|)$ in the worst case. Thus, the complexity of the forward search is $O(|L^f_{max}||\mathcal{V}|(\log(|L^f_{max}||\mathcal{V}|) + |L^f_{max}| + |L^b_{max}|)$. $|L^f_{max}|$ and $|L^b_{max}|$ depend on Δ_s and Δ_t respectively, and thus are much smaller than $|L_{max}|$ in SDD.

Algorithm Bi-OSDD. We can also apply the bi-directional search to the OSDD algorithm, and we call this method Bi-OSDD (Bi-directional OSDD). Most steps are the same as that of Bi-SDD. The only difference is in the first phase. As the first phase in OSDD aims to find some feasible solutions quickly, the bi-directional search method cannot use OS_{max} to prune some paths in function $joinLabels()$. Thus, it always returns the label with the maximum OS value by joining $\overrightarrow{L}^k_i$ with the labels in $\overleftarrow{L}(v_i)$.

It is straightforward to prove that Bi-OSDD can also achieve an approximation ratio of $(\frac{1}{\alpha})^{c+1}$ for answering the CPS-SM query based on the proof of Theorem 4. This algorithm benefits from both the FS-α-dominance and the bi-directional search, and thus it has better performance than SDD, OSDD, and Bi-SDD, and this is also verified in our experiments as shown in Section VIII.

VII. A Polynomial Algorithm CBFS

Although Alg. 1 and Alg. 2 can answer the CPS-SM query with a guaranteed error bound, they are not polynomial-time algorithms. Here we propose a polynomial heuristic algorithm based on submodular α-dominance.

This algorithm utilizes a cost-effective first search which is a common method for solving the search problems with budget constraints [37], to find the path with a large objective score quickly, and we denote this algorithm as CBFS. In CBFS, the Cost-Effective value (CE) is computed as $CE(P^l_j) = OS(P^l_j)/BS(P^l_j)$. The most steps of CBFS are similar to those of Alg. 1, but it initializes a min-priority queue $\mathcal{Q}$ which stores partial labels in ascending order of their CE values in line 1 of Alg. 1. In addition, rather than storing all non-dominant labels on each node, CBFS only stores at most β labels on each node, thus it prunes the label L^k_i when the number of labels in $L(v_i)$ is equal to β (between line 6 and line 7 in Alg. 1). As the quality of pruned labels is uncontrolled, CBFS cannot maintain the error bound as does SDD. Note that storing at most β labels on each node can be used to speed up SDD and OSDD, but the performance is poor, as the two algorithms always keep the top-β shortest paths. Due to space limitations, we omit these experimental results.

Next, we analyze the complexity of CBFS. Since there exists at most β labels on each node, CBFS would generate at most $\beta|\mathcal{V}|$ labels. To dequeue and enqueue $\beta|\mathcal{V}|$ labels, it requires $O(\beta|\mathcal{V}|log(\beta|\mathcal{V}|))$ time. CBFS needs $\beta|\mathcal{V}|$ loops in the worst case, thus it takes $O(\beta^2|\mathcal{V}|)$ time to do submodular α-dominance checks. Therefore, the worst time complexity of CBFS is $O(\beta|\mathcal{V}|(log(\beta|\mathcal{V}|) + \beta))$. Note that β renders this algorithm to have polynomial running time.

VIII. Experimental Study

A. Experiments Setting

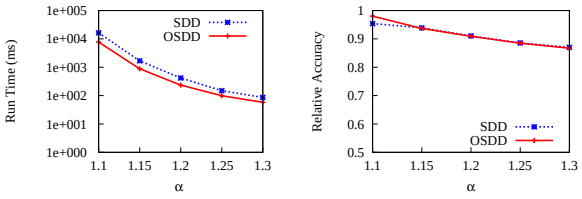
We compare the proposed algorithms SDD, Bi-SDD, OSDD, Bi-OSDD and CBFS with two state-of-the-art algorithms: Weighted A*(WA*) [8] and Generalized Cost-Benefit Algorithm(GCB) [9] using four real datasets on two applications. Note that the recursive greedy algorithm [6] and the GreedyDP algorithm [7] are too time-consuming and thus are not compared. The studies using an accumulative function (such as [23]) to formulate the objective function are also not compared since they cannot be applied to our problem. We set $\epsilon = 0.2$ for WA* according to the experimental results in [8], and use an approximation algorithm proposed by Frieze et al. [38] for the asymmetric TSP problem to compute the cost function for GCB, as computing the cost function for GCB (i.e., finding the shortest path visiting a given set of nodes) is NP-hard [25]. We adopt the H2H index [39] to compute the shortest distance between two nodes efficiently, as all of the proposed algorithms needs it to do some optimizations (e.g., Pruning Optimization [8] can be used in WA* and our algorithms, as it can prune the path satisfying $BS(P^l_j) + dist(v_j, v_t) > \Delta$ safely, where $dist(v_j, v_t)$ computes the shortest distance between v_j and v_t). We implement all the algorithms in JAVA on Windows 10, and run on a desktop with Intel i7-$9800X$ 3.8GHz CPU and 64 GB memory.

B. Application 1: Most Influential Path Query

We first investigate the Most Influential Path (MIP) Query whose target is to find an optimal path under a budget limit such that the number of users appearing on the path is maximal in a network. This problem is useful in many scenarios, e.g., a salesperson wants to find a path to market her products, a taxi driver intends to find a path for picking up new passengers, etc. Following the previous work [8], [34], we model the network as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and compute the probability that a user u_i appears on a P as $P^P_{u_i} = 1 - \prod_{v_j \in P, \mathcal{V}}(1 - P^{v_j}_{u_i})$, where $P^{v_j}_{u_i}$ is the probability that u_i visits the location v_j, and it is computed as $P^{v_j}_{u_i} = \frac{\#\ of\ check{-}ins\ in\ v_j\ of\ u_i}{\#\ of\ check{-}ins\ of\ u_i}$. Thus, the objective score of P is the number of users expected to appear on P, and it is computed as a submodular function: $OS(P) = \sum_{u_i \in \mathcal{U}} P^P_{u_i}$, where $\mathcal{U}$ represents all users.

In this problem, we use two real location-based social networks datasets BR and GO from SNAP (http://memetracker.org/data/index.html). BR has 4,491,143 check-ins made by 58,228 users at 772,966 locations on Brightkite's website, and GO contains 6,442,890 check-ins made by 196,591 users at 1,280,969 locations on Gowalla's website. We first remove the check-ins made by users who checked in a location less than 2 times and merge the locations whose distance is less than 300 meters. After that,

we generate the networks following the data distribution of the real road networks used in Application 2 by making an edge between two locations that were visited continuously in one day by the same user, and using Google Maps APIs to get the road network distance as the budget score for each edge. After the preprocessing, we obtain 33,671 nodes and 96,252 edges for BR, 70,480 nodes, and 198,430 edges for GO. For each experiment, we generate 100 queries randomly and report the average results.



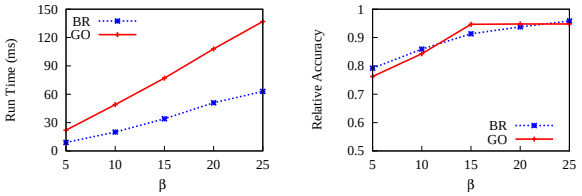
(a) BR (b) BR

(c) GO (d) GO

Fig. 2. Effect of parameter α with $\Delta = 50$ $kilometer$ on MIP

1) Effect of Parameter α: In all proposed algorithms, the parameter α balances the runtime and the solution quality. To study the effect of α with $\Delta = 50$ $kilometer$ on processing the CPS-SM query, we run SDD and OSDD on the BR and GO datasets and get the result as shown in Fig. 2.

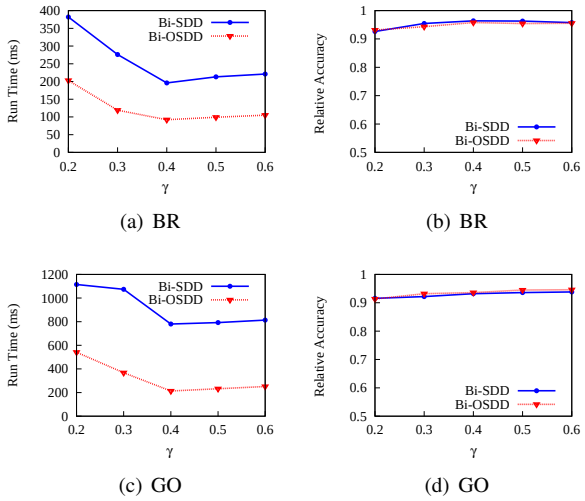
As the problem is NP-hard and we do not have an exact algorithm, we get an approximate result by running OSDD with $\alpha = 1.05$, and we compute the "relative" accuracy of the results using different parameter values by comparing with this. We can see that with the increase of α, the efficiency improves (shorter runtime) while the accuracy drops. We set $\alpha = 1.2$ for both algorithms since the query processing time is short and about 90% accuracy can be achieved. We use $\alpha = 1.2$ as a default setting for other proposed algorithms in this application.



(a) run time (b) precision

Fig. 3. Effect of parameter β with $\Delta = 50$ $kilometers$ for CBFS on MIP

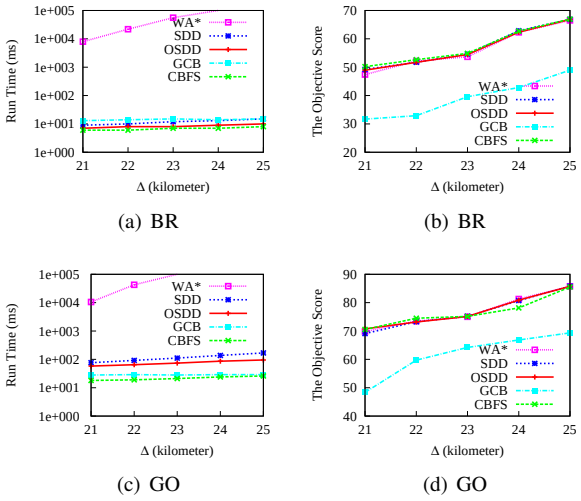
2) Effect of Parameter β: The value of β controls the maximal number of labels stored on each node in CBFS. When β is larger, more labels are generated and maintained on nodes, thus the solution quality is better, while the runtime is longer. To tune β, we run CBFS to process the query with $\Delta = 50$ $kilometers$ on both datasets. Fig. 3 validates the analysis, and we set $\beta = 15$ for CBFS for the experiments in the application. Note that the effect of β on CBFS is similar to the effect of the number of labels on SDD and OSDD. Due to space limitations, we omit these experiments here.



(a) BR (b) BR

(c) GO (d) GO

Fig. 4. Effect of parameter γ with $\alpha = 1.2$, $\Delta = 50$ $kilometer$ on MIP



(a) BR (b) BR

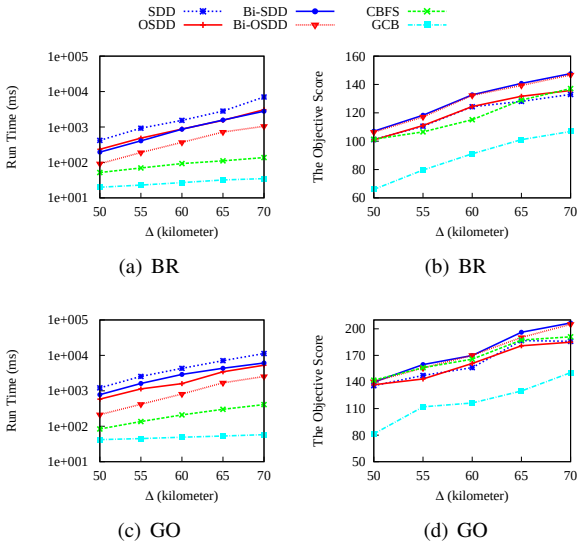
(c) GO (d) GO

Fig. 5. Comparison with baselines varying Δ on MIP

3) Effect of Parameter γ: The value of parameter γ adjusts the search ranges of two directions search in Bi-SDD and Bi-OSDD. We vary γ from 0.2 to 0.6 and run the two algorithms on the FL dataset. Fig. 4 shows that the runtime of both algorithms first drops, and then increases slightly. The reason is that when γ is small, the forward search still has to generate and maintain a lot of labels. On the other hand, when γ is large, the initialization for getting the backward node labels has very high costs. $\gamma = 0.4$ is used for Bi-SDD and Bi-OSDD in the subsequent experiments in this application.

4) Performance comparison: We compare 3 proposed algorithms SDD, OSDD and CBFS with WA* and GCB on

both efficiency and accuracy. As WA* is extremely slow, we conduct the experiments by varying Δ from $21km$ to $25km$ using BR and GO. Fig. 5 shows that WA* is very time-consuming when Δ is large, as it stores too many partial paths. The quality of the solution of GCB is the worst, as GCB relaxed the budget limit and always returned a local optimal value. Meanwhile, we observe that, the maximum number of nodes (c) in the returned paths is 26 in our algorithms. Although the worst-case error bound is poor, our proposed algorithms are faster than WA* by orders of magnitude, and can find high-quality paths like WA*.



(a) BR (b) BR

(c) GO (d) GO

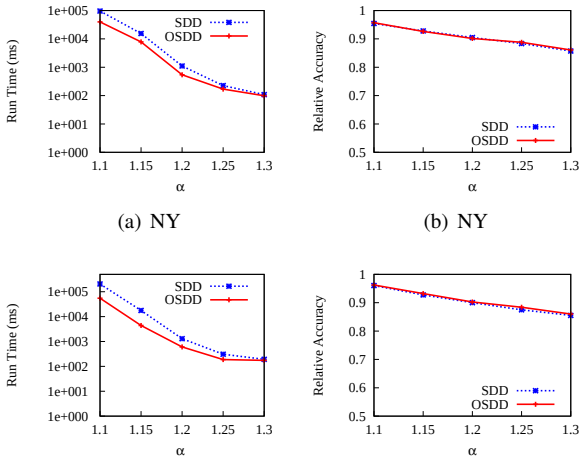
Fig. 6. Comparison of algorithms varying Δ on MIP

It is hard to show the performance difference of our algorithms on small Δ. We conduct another experiment and run the five proposed algorithms on larger Δ using BR and GO to answer MIP queries. As shown in Fig. 6, OSDD is at least 2 times faster than SDD, and the two have similar accuracy. For SDD and OSDD, the bi-directional search method can improve efficiency. Surprisingly, the solution quality is also improved slightly, although the theoretical error bound is worse. The reason might be that the bi-directional search reduces the number of generated labels and the length of stored partial paths in both algorithms. Note that the pruning power of submodular α-dominance depends on the OS value of the label. Thus, if the partial paths are shorter, their OS values are smaller, and the pruning power is weaker, which leads to better solution quality. With the increase of Δ, the runtime of our algorithms grows faster than that of GCB. This is because GCB is affected by the number of nodes scanned, and our algorithms are affected by the number of paths enumerated. Moreover, Bi-OSDD and CBFS scale well when varying Δ.

C. Application 2: Most Diversified Path Query

The second application is to find the most diversified path (MDP) given a source node, a target node, and a budget constraint. MDP considers a set of locations in road networks that is associated with a set of keywords, e.g., "mall" and "bar". The diversity of a path is measured by the number of different keywords covered by this path, and thus the objective function is $OS(P) = |\cup_{v_i \in P} \mathcal{K}(v_i)|$, where $\mathcal{K}(v_i)$ is the set of keywords on v_i.

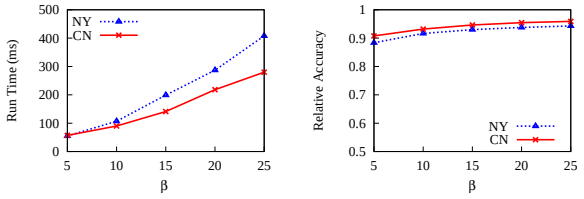
We use two real road networks New York City (NY) and California & Nevada (CN) from DIMACS (http://users.diag.uniroma1.it/challenge9/download.shtml), where the NY dataset contains 264,346 nodes and 733,846 arcs and the CN dataset contains 1,890,815 nodes and 4,657,742 arcs. We adopted the Foursquare APIs to obtain the categories of locations (keywords) for NY, and randomly generate keywords for CN. We also generate 100 queries randomly and report the average results for each experiment.



(a) NY (b) NY

(c) CN (d) CN

Fig. 7. Effect of parameter α with $\Delta = 50$ $kilometer$ on MDP



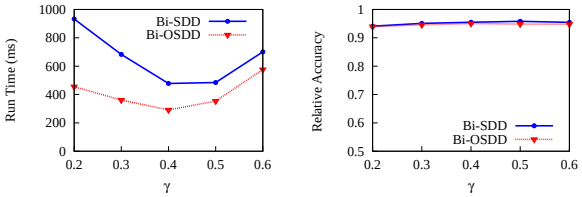
(a) run time (b) precision

Fig. 8. Effect of parameter β with $\Delta = 50$ $kilometer$ for CBFS on MDP

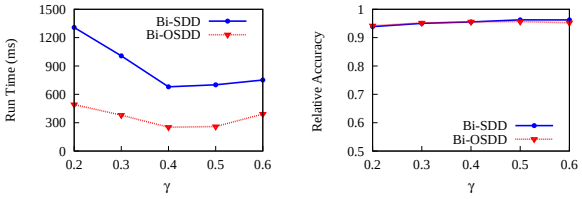
1) Effect of Parameter α: We vary α from 0.1 to 0.3 to run SDD and OSDD with $\Delta = 50$ $kilometer$ on both datasets for answering the MDP query. This application is more complicated and the datasets are larger than in the previous one, we get an approximate result by running OSDD with $\alpha = 1.05$, and then use it to compute the relative accuracy of the results. The results are shown in Fig. 7. When $\alpha = 1.2$ both algorithms can obtain solutions with 90% relative accuracy in a short time, and thus we use $\alpha = 1.2$ as a default setting for all our algorithms in this application.

2) Effect of Parameter β: We tune β by running CBFS to answer the query with $\Delta = 50$ $kilometers$ on both datasets.

The results are shown in Fig. 8, and we set $\beta = 10$ for CBFS for the experiments in this application.



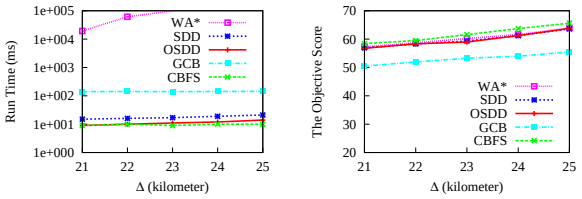
(a) NY (b) NY



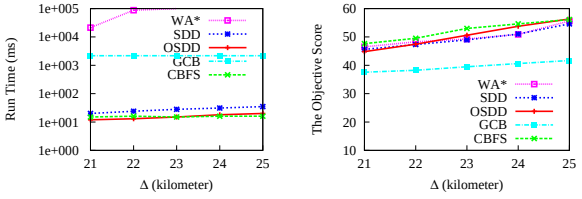
(c) CN (d) CN

Fig. 9. Effect of parameter γ with $\alpha = 1.2$, $\Delta = 50$ $kilometer$ on MDP

3) Effect of Parameter γ: We vary γ from 0.2 to 0.6 to run Bi-SDD and Bi-OSDD with $\Delta = 50$ $kilometer$ on NY and CN to process the MDP query. Fig. 9 shows the results, and the trend is similar to that in the first application. We set $\gamma = 0.4$ for Bi-SDD and Bi-OSDD in the subsequent experiments.



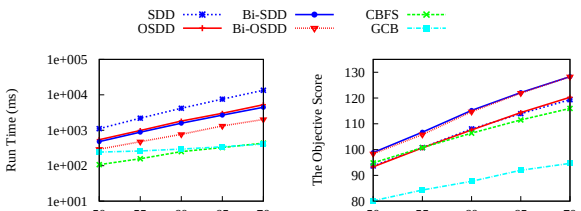
(a) NY (b) NY



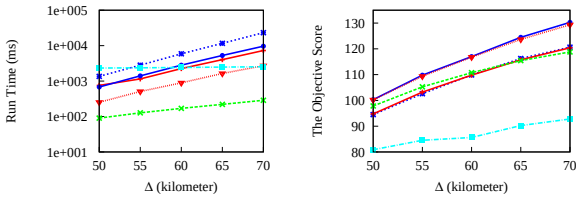
(c) CN (d) CN

Fig. 10. Comparison with baselines varying Δ on MDP

4) Performance comparison: Considering WA* is very time-consuming, we also compare our algorithms SDD, OSDD, and CBFS with WA* and GCB by varying Δ from $21km$ to $25km$ using NY and CN. Fig. 10 also illustrates that WA* is too slow to solve the queries with large Δ. GCB cannot achieve high-quality solutions, and its runtime is long when the dataset is large, since GCB needs to scan all nodes for searching paths. Moreover, the results show the maximum number of nodes (c) in the returned paths is 40 in our algorithms, thus the worst-case approximation ratio is small. But our proposed algorithms can always get paths of high quality in a short time.



(a) NY (b) NY



(c) CN (d) CN

Fig. 11. Comparison of algorithms varying Δ on MDP

In order to compare the performances of our algorithms and GCB in this application, we run them on NY and CN using larger Δ. Fig. 11 demonstrates that the solution quality of GCB is poor, while our algorithms can achieve much higher accuracy. The bi-directional search speeds up SDD and OSDD significantly in this application, as it reduces a lot of labels in the two algorithms. Meanwhile, CBFS can return high-quality results in practice (90% of the Bi-OSDD result) in a short time (less than 500ms, even faster than GCB). Thus, on larger datasets and larger Δ, CBFS is a good choice.

IX. Conclusion

In this paper, we study the CPS-SM problem which is to find the optimal path from a source node to a target node such that its submodular function score is maximized under a given budget limit. Solving the CPSSM problem is NP-hard. To answer the problem efficiently, we propose an algorithm with a guaranteed error bound based on a concept called "sumodular α-dominance" utilizing the properties of the submodular function. We develop another algorithm with the same approximation ratio which is more efficient by relaxing the sumodular α-dominance conditions. We also present a bi-directional search method to speed up the proposed algorithms, and devise a heuristic algorithm with polynomial runtime. The experimental study on two applications and four real-world datasets demonstrates the excellent performance of the proposed algorithms in terms of both efficiency and accuracy. In future work, we would like to design parallel algorithms to solve the problem more efficiently.

Acknowledgment

This work was supported in part by NSFC Grants (No. 61876025, 62176225, 61836005, 61772442 and 62102341), ARC DE190100663, and CUHK-SZ Grant UDF01002139. Liang Feng is the corresponding author of this paper.

REFERENCES

[1] H. C. Joksch, "The shortest route problem with constraints," *Journal of Mathematical analysis and applications*, vol. 14, no. 2, pp. 191–197, 1966.

[2] R. Hassin, "Approximation schemes for the restricted shortest path problem," *Mathematics of Operations research*, vol. 17, no. 1, pp. 36–42, 1992.

[3] S. Wang, X. Xiao, Y. Yang, and W. Lin, "Effective indexing for approximate constrained shortest path queries on large road networks," *Proceedings of the VLDB Endowment*, vol. 10, no. 2, pp. 61–72, 2016.

[4] S. Lu, B. He, Y. Li, and H. Fu, "Accelerating exact constrained shortest paths on gpus," *Proceedings of the VLDB Endowment*, vol. 14, no. 4, pp. 547–559, 2020.

[5] Z. Liu, L. Li, M. Zhang, W. Hua, P. Chao, and X. Zhou, "Efficient constrained shortest path query answering with forest hop labeling," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1763–1774.

[6] C. Chekuri and M. Pal, "A recursive greedy algorithm for walks in directed graphs," in *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2005, pp. 245–253.

[7] S. Sakaue, M. Nishino, and N. Yasuda, "Submodular function maximization over graphs via zero-suppressed binary decision diagrams," in *32th AAAI Conference on Artificial Intelligence (AAAI)*, 2018, pp. 1422–1430.

[8] Y. Zeng, X. Chen, X. Cao, S. Qin, M. Cavazza, and Y. Xiang, "Optimal route search with the coverage of users' preferences," in *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, 2015, pp. 2118–2124.

[9] H. Zhang and Y. Vorobeychik, "Submodular optimization with routing constraints," in *Proceedings of 30th AAAI Conference on Artificial Intelligence (AAAI)*, 2016, pp. 819–825.

[10] A. Krause and D. Golovin, "Submodular function maximization." 2014.

[11] S. Stan, M. Zadimoghaddam, A. Krause, and A. Karbasi, "Probabilistic submodular maximization in sub-linear time," in *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 2017, pp. 3241–3250.

[12] N. Buchbinder, M. Feldman, and M. Garg, "Deterministic $(1/2+\varepsilon)$-approximation for submodular maximization over a matroid," in *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2019, pp. 241–254.

[13] R. Haba, E. Kazemi, M. Feldman, and A. Karbasi, "Streaming submodular maximization under a k-set system constraint," in *Proceedings of the 37th International Conference on Machine Learning (ICML)*, 2020.

[14] M. Feldman and A. Karbasi, "Continuous submodular maximization: Beyond dr-submodularity," in *Proceedings of the 33rd Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[15] A. Singh, A. Krause, C. Guestrin, W. J. Kaiser, and M. A. Batalin, "Efficient planning of informative paths for multiple robots," in *Proceedings of the Twenty International Joint Conference on Artificial Intelligence (IJCAI)*, vol. 7, 2007, pp. 2204–2211.

[16] E. W. Dijkstra *et al.*, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[17] R. K. Ahuja, K. Mehlhorn, J. Orlin, and R. E. Tarjan, "Faster algorithms for the shortest path problem," *Journal of the ACM*, vol. 37, no. 2, pp. 213–223, 1990.

[18] W. Matthew Carlyle and R. Kevin Wood, "Near-shortest and k-shortest simple paths," *Networks: An International Journal*, vol. 46, no. 2, pp. 98–109, 2005.

[19] J. Hershberger, M. Maxel, and S. Suri, "Finding the k shortest simple paths: A new algorithm and its implementation," *ACM Transactions on Algorithms (TALG)*, vol. 3, no. 4, pp. 45–es, 2007.

[20] H.-P. Kriegel, M. Renz, and M. Schubert, "Route skyline queries: A multi-preference path planning approach," in *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 2010, pp. 261–272.

[21] D. Ouyang, L. Yuan, F. Zhang, L. Qin, and X. Lin, "Towards efficient path skyline computation in bicriteria networks," in *International Conference on Database Systems for Advanced Applications*. Springer, 2018, pp. 239–254.

[22] Q. Gong, H. Cao, and P. Nagarkar, "Skyline queries constrained by multi-cost transportation networks," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 926–937.

[23] A. Gunawan, H. C. Lau, and P. Vansteenwegen, "Orienteering problem: A survey of recent variants, solution approaches and applications," *European Journal of Operational Research*, vol. 255, no. 2, pp. 315–332, 2016.

[24] G. Laporte and S. Martello, "The selective travelling salesman problem," *Discret. Appl. Math.*, vol. 26, no. 2-3, pp. 193–207, 1990.

[25] R. C. de Andrade, "New formulations for the elementary shortest-path problem visiting a given set of nodes," *European Journal of Operational Research*, vol. 254, no. 3, pp. 755–768, 2016.

[26] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng, "On trip planning queries in spatial databases," in *International symposium on spatial and temporal databases (SSTD)*. Springer, 2005, pp. 273–290.

[27] M. Sharifzadeh, M. R. Kolahdouzan, and C. Shahabi, "The optimal sequenced route query," *VLDB Journal*, vol. 17, no. 4, pp. 765–787, 2008.

[28] R. Levin, Y. Kanza, E. Safra, and Y. Sagiv, "Interactive route search in the presence of order constraints," *Proceedings of the VLDB Endowment*, vol. 3, no. 1, pp. 117–128, 2010.

[29] M. N. Rice and V. J. Tsotras, "Engineering generalized shortest path queries," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pp. 949–960.

[30] D. H. Lorenz and D. Raz, "A simple efficient approximation scheme for the restricted shortest path problem," *Operations Research Letters*, vol. 28, no. 5, pp. 213–219, 2001.

[31] X. Cao, L. Chen, G. Cong, and X. Xiao, "Keyword-aware optimal route search," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1136–1147, 2012.

[32] Z. Feng, T. Liu, H. Li, H. Lu, L. Shou, and J. Xu, "Indoor top-k keyword-aware routing query," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1213–1224.

[33] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, and N. Glance, "Cost-effective outbreak detection in networks," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, 2007, pp. 420–429.

[34] K. Feng, G. Cong, S. S. Bhowmick, W.-C. Peng, and C. Miao, "Towards best region search for data exploration," in *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, 2016, pp. 1055–1070.

[35] X. Chen, X. Cao, Y. Zeng, Y. Fang, and B. Yao, "Optimal region search with submodular maximization." in *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI)*, 2020, pp. 1216–1222.

[36] G. Tsaggouris and C. Zaroliagis, "Multiobjective optimization: Improved fptas for shortest paths and non-linear objectives with applications," *Theory of Computing Systems*, vol. 45, no. 1, pp. 162–186, 2009.

[37] S. Khuller, A. Moss, and J. S. Naor, "The budgeted maximum coverage problem," *Information Processing Letters*, vol. 70, no. 1, pp. 39–45, 1999.

[38] A. M. Frieze, G. Galbiati, and F. Maffioli, "On the worst-case performance of some algorithms for the asymmetric traveling salesman problem," *Networks*, vol. 12, no. 1, pp. 23–39, 1982.

[39] D. Ouyang, L. Qin, L. Chang, X. Lin, Y. Zhang, and Q. Zhu, "When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks," in *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, 2018, pp. 709–724.