Automatic Itinerary Planning for Traveling Services

Gang Chen, Sai Wu, Jingbo Zhou, and Anthony K.H. Tung

Abstract—Creating an efficient and economic trip plan is the most annoying job for a backpack traveler. Although travel agency can provide some predefined itineraries, they are not tailored for each specific customer. Previous efforts address the problem by providing an automatic itinerary planning service, which organizes the points-of-interests (POIs) into a customized itinerary. Because the search space of all possible itineraries is too costly to fully explore, to simplify the complexity, most work assume that user's trip is limited to some important POIs and will complete within one day. To address the above limitation, in this paper, we design a more general itinerary planning service, which generates multiday itineraries for the users. In our service, all POIs are considered and ranked based on the users' preference. The problem of searching the optimal itinerary is a team orienteering problem (TOP), a well-known NP-complete problem. To reduce the processing cost, a two-stage planning scheme is proposed. In its preprocessing stage, single-day itineraries are precomputed via the MapReduce jobs. In its online stage, an approximate search algorithm is used to combine the single day itineraries. In this way, we transfer the TOP problem with no polynomial approximation into another NP-complete problem (set-packing problem) with good approximate algorithms. Experiments on real data sets show that our approach can generate high-quality itineraries efficiently.

Index Terms—Map reduce, trajectory, team orienteering problem, itinerary planning, location-based service

1 Introduction

Traveling market is divided into two parts. For casual customers, they will pick a package from local travel agents. The package, in fact, represents a pregenerated itinerary. The agency will help the customer book the hotels, arrange the transportations, and preorder the tickets of museums/parks. It prevents the customers from constructing their personalized itineraries, which is very time-consuming and inefficient. For instance, Fig. 1 lists a four-day package to Hong Kong, provided by a Singapore agency. It covers the most popular POIs for a first-time traveler and the customers just need to follow the itinerary to schedule their trips.

Although the travel agencies provide efficient and convenient services, for experienced travelers, the itineraries provided by the travel agents lack customization and cannot satisfy individual requirements. Some interested POIs are missing in the itineraries and the packages are too expensive for a backpack traveler. Therefore, they have to plan their trips in every detail, such as selecting the hotels, picking POIs for visiting, and contacting the car rental service.

Therefore, to attract more customers, travel agency should allow the users to customize their itineraries and still enjoy the same services as the predefined itineraries. However, it is impossible to list all possible itineraries for users. A practical solution is to provide an automatic itinerary planning service. The user lists a set of interested POIs and specifies the time and money budget. The itinerary planning service returns top-K trip plans satisfying the requirements. In the ideal case, the user selects one of the returned itineraries as his plan and notifies the agent.

However, none of the current itinerary planning algorithms (e.g., [1] and [2]) can generate a ready-to-use trip plan, as they are based on various assumptions.

First, current planning algorithms only consider a single day's trip, while in real cases, most users will schedule an n-day itinerary (e.g., the one shown in Fig. 1). Generating an n-day itinerary is more complex than generating a single day one. It is not equal to constructing n single-day itineraries and combining them together, as a POI can only appear once in the itinerary. It is tricky to group POIs into different days. One possible solution is to exploit the geolocations, for example, nearby POIs are put in the same day's itinerary. Alternatively, we can also rank POIs by their importance and use a priority queue to schedule the trip.

Second, the travel agents tend to favor the popular POIs. Even for a city with a large number of POIs, the travel agents always provide the same set of trip plans, composed with top POIs. However, those popular POIs may not be attractive for the users, who have visited the city for several times or have limited time budget. It is impossible for a user to get his personal trip plan. The travel agent's service cannot cover the whole POI set, leading to few choices for the users. In our algorithm, we adopt a different approach by giving high priorities to the selected POIs and generating a customized trip plan on the fly.

Third, suppose we have N available POIs and there are m POIs in each single day's itinerary averagely. We will end

- *G. Chen and S. Wu are with the College of Computer Science, Zhejiang University, Yuquan Campus, Zheda Road, Hangzhou, Zhejiang 310027, P.R. China. E-mail: {cg, wusai}@zju.edu.cn.*
- *J. Zhou and A.K.H. Tung are with the School of Computing, National University of Singapore, Computing 1, Computing Drive, Singapore 117417, Singapore. E-mail: {jzhou, atung}@comp.nus.edu.sg.*

Manuscript received 16 Nov. 2012; revised 5 Feb. 2013; accepted 6 Feb. 2013; published online 12 Mar. 2013.
Recommended for acceptance by G. Yu.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2012-11-0783.
Digital Object Identifier no. 10.1109/TKDE.2013.46.

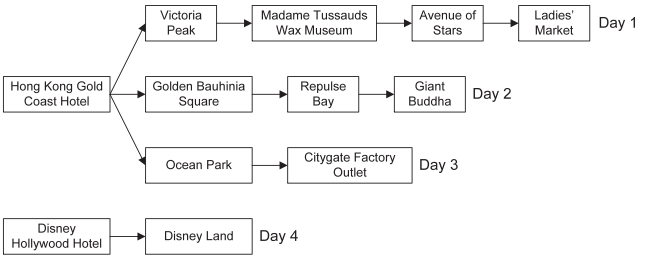


Fig. 1. A four-day trip to Hong Kong.

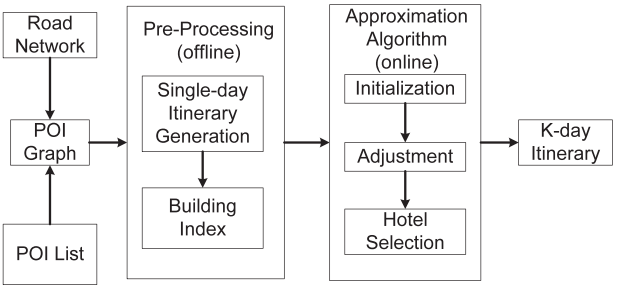


Fig. 2. Architecture of a trip planning system.

up with $\frac{N!}{(N-m)!m!}$ candidate itineraries. It is costly to evaluate the benefit of every itinerary and select the optimal one. Therefore, in [1] and [2], some heuristic approaches are adopted to simplify the computation. However, the heuristic approaches are based on some assumptions (e.g., popular POIs are selected with a higher probability). They only provide limited number of itineraries and are not optimized for the backpack traveler, who plans to have a unique journey with his own customized itinerary.

Last but not the least, handling new emerging POIs were tricky in previous approaches. The model needs to be rebuilt to evaluate the benefit of including the new POIs into the itinerary. For systems based on the users' feedback [2], we need to collect the comments for the new POIs from the users, which is very time-consuming.

To address the above problems, in this paper, a novel itinerary planning approach is proposed. The design philosophy of our approach is to generate itineraries that narrow the gap between the agents and travelers. We reduce the overhead of constructing a personalized itinerary for the traveler; and we provide a tool for the agents to customize their services. Fig. 2 shows an overall architecture of our trip-planning system. Specifically, our approach can be summarized as follows.

In the preprocessing, POIs are organized into an undirected graph, $\mathcal{G}$. The distance of two POIs is evaluated by Google Map's APIs.[1] Given a request, the system provides interfaces for the user to select preferred POIs explicitly, while the rest POIs are assumed to be the optional POIs. Different ranking functions are applied to different types of POIs. The automatic itinerary planning service needs to return an itinerary with the highest ranking. Searching the optimal itinerary can be transformed into the team orienteering problem (TOP), which is an NP-complete problem without polynomial approximations [3]. Therefore, a two stage scheme is applied.

In the preprocessing stage, we iterate all candidate single-day itineraries using a parallel processing framework, MapReduce [4]. The results are maintained in the distributed file system (DFS) and an inverted index is built for efficient itinerary retrieval. To construct a multiday itinerary, we need to selectively combine the single itineraries. The preprocessing stage, in fact, transforms the TOP into a set-packing problem [5], which has well-known approximated algorithms. In the online stage, we design an approximate algorithm to generate the optimal itineraries. The approximate algorithm adopts the *initialization-adjustment* model and a theoretic bound is given for the quality of the approximate result.

To evaluate the proposed approach, we use the real data from Yahoo Travel.[2] The experiments show that our approach can efficiently return high-quality customized itineraries. The remainder of this paper is organized as follows: In Section 2, we formalize the problem and give an overview of our approach. Then, Section 3 and Section 4 present the preprocessing stage and online stage of our approach, respectively. We evaluate our approach in Section 5 and review previous work in Section 6. Finally, the paper is concluded in Section 6.

2 Overview

2.1 Problem Statement

In the itinerary planning system, the user selects a set of interested POIs, S_p, and asks the system to generate a k-day itinerary. We use (S_p, k) to denote a user's request. To model the planning problem, we organize the POIs into a complete graph, the *POI graph*.

Definition 1 (POI Graph). *In the POI graph $G = (V, E)$, we generate a vertex for each POI and every pair of vertices are connected via an undirected edge in E. In G, the vertex and edge have the following properties:*

1. *$\forall v_i \in V$, $w(v_i)$ denotes the weight (importance) of the POI and $t(v_i)$ is the average time that tourists will spend on the POI.*
2. *$\forall (e_x = v_i \rightsquigarrow v_j) \in E$, $t(e_x)$ is the cost of the edge, computed as the average traveling time from v_i to v_j.*

Fig. 3 shows a POI graph with five nodes. Each node denotes a POI and has two properties: the weight and travel time (shown in the red blocks). The nodes are connected via weighted edges. The edge's weight is set to the average traveling time for the shortest path between the corresponding POIs in the map. In fact, there are two types of edges. The first type represents that the two nodes are directly connected in the map (no other POI exists in their shortest path, e.g., $0 \rightsquigarrow 1$). The second type contains multiple shortest paths in the map (e.g., $0 \rightsquigarrow 3 = (0 \rightsquigarrow 1) \oplus (1 \rightsquigarrow 3)$). Transforming the POI graph into a complete graph reduces the processing cost of our itinerary algorithm.

The definition of POI graph assumes that the costs of edges are symmetric. Namely, the traveling time from v_i to v_j is equal to the time from v_j to v_i. In fact, as our approach

1. https://developers.google.com/maps/.

2. http://travel.yahoo.com/.

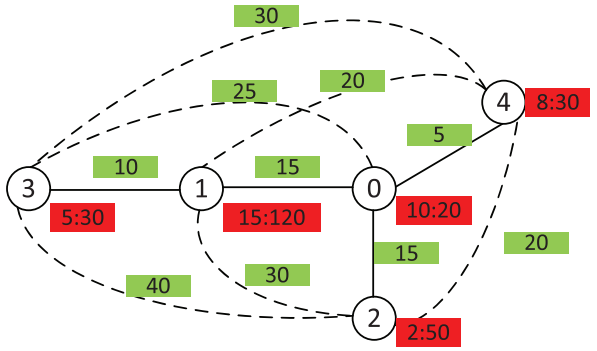


Fig. 3. POI graph.

does not rely on the assumption, it can be directly applied to the case of nonsymmetric cost (e.g., traffics are different for $v_i \rightsquigarrow v_j$ and $v_j \rightsquigarrow v_i$).

Let $w(v_i)$ denote the weight (importance) of POI v_i. The initial weight of v_i is generated from the users' reviews (e.g., in Yahoo Travel, users can specify score ranging from 0 to 5 for each POI. We accumulate the scores and use the average values as the initial weight).

Users can also select a set of preferred POIs, denoted as S_p. Given a request (S_p, k), if v_i is selected by the request ($v_i \in S_p$), we intentionally increase its weight to $\alpha(w(v_i)+1)$, where α can be set to an arbitrary integer. The intuition is that user-selected POIs are far more important than any other POIs.

For a request (S_p, k), if $k = 1$, we just need to generate a single-day itinerary. A single-day itinerary is represented as $L = v_0 \rightsquigarrow \cdots \rightsquigarrow v_n \rightsquigarrow h_j$, where h_j is a hotel POI. The elapsed time is estimated as

$$t(L) = \sum_{i=0}^{n} t(v_i) + \sum_{i=0}^{n-1} t(v_i \rightsquigarrow v_{i+1}) + t(v_n \rightsquigarrow h_j).$$

In the rest of the discussion, we remove the hotel part and focus on how to merge the POIs into itineraries. After all other POIs are fixed, we will solve the hotel-selection problem.

Assume there are H available hours per day for traveling. The itinerary L must satisfy that $t(L) \leq H$. For a common traveling request, it always includes a k-day ($k \geq 1$) trip, which is defined as

Definition 2 (k-Day Itinerary). *Given a POI graph G and time budget k, a valid k-day itinerary consists of k single-day itineraries, $\mathcal{L} = \{L_1, L_2, \ldots, L_k\}$, which satisfies that*

1. *$\forall i \forall j$, L_i and L_j do not share a POI.*
2. *$t(L_i) \leq H$ for all $1 \leq i \leq k$.*

Based on the POIs included in the itinerary, the score of a k-day itinerary can be computed as

$$w(\mathcal{T}) = \sum_{i=1}^{k} \sum_{v_j \in L_i} w(v_j). \quad (1)$$

The goal of our itinerary planning algorithm is to find the k-day itinerary with the highest score. However, we will show that finding the optimal itinerary is an NP-complete problem, which is equivalent to the team orienteering problem [3]. Even approximate algorithm within constant factor does not exist. The existing work [6] solves the problem by employing heuristic algorithms, which may generate arbitrarily bad results.

2.2 System Architecture

In our system, instead of trying to propose new algorithms for the TOP, we transform the optimal itinerary planning problem into a set-packing problem by an offline MapReduce process and an approximate algorithm is applied to solve the set-packing problem. If the maximal number of POIs in the single-day itinerary is bounded by m, the optimal result can be approximated within factor of $\frac{2(m+1)}{3}$ (m is the maximal number of POIs in each single-day itinerary).

Fig. 2 shows the architecture of our trip-planning system. In the first step, POI graph is constructed via the road network and POI coordinates. The Google Map's APIs are used to evaluate the distance between POIs. The average elapsed time of a POI is estimated from users' blogs and travel agency's schedules.

After the POI graph is constructed, a set of MapReduce jobs are submitted to iterate all possible single-day itineraries in the preprocessing. The number of itineraries is exponential to the number of POIs. However, using parallel processing engine, such as MapReduce, we can efficiently generate all itineraries in an offline manner. To speed up the single-day itinerary retrieval, an inverted index is built. Given a POI, all single-day itineraries involving the POI can be efficiently retrieved.

For a user request (S_p, k), POIs' weights are updated based on S_p and we compute the scores for each single-day itinerary. The problem of finding optimal k-day itinerary is transformed to select k single-day itineraries that maximize the total score. We show that the new problem can be reduced to the weighted set-packing problem, which has polynomial approximate algorithms. Therefore, we simulate the approximate algorithm for set-packing problem to generate the k-day itinerary. The algorithm uses a greedy strategy to create an initial solution, which is continuously refined in the adjustment phase. The adjustment phase scans the index to find a potentially better solution.

In the next two sections, we first present how we apply the MapReduce framework to generate and index the single-day itineraries. The parallel processing engine enables us to search the optimal solution in a brute-force manner. Next, we show after the preprocessing, the complexity of TOP is reduced and approximate algorithms are available.

3 Preprocessing

The preprocessing includes two steps. In the first step, a set of MapReduce jobs are submitted to produce all possible single-day itineraries. In the second step, the single-day itineraries are reorganized as an itinerary index, which supports efficient itinerary search.

3.1 Intractability of Optimal Itinerary Algorithm

Given a user request (S_p, k), the goal of an itinerary planning algorithm is to provide an itinerary, which ranks

highest among all possible itineraries. The score of the itinerary is computed based on the POI weights. However, as shown in the following theorem, this is an NP-complete problem and no polynomial time algorithm exists.

Theorem 1. *Finding optimal k-day itinerary in a POI graph $G = (V, E)$ is an NP-complete problem.*

Proof (Sketch). The optimal k-day itinerary can be reduced to the TOP [3], which is a well-known NP-complete problem. Consider a simple scenario where,

1. k vehicles are created, which start from the same position.
2. Each vehicle has a time limit (1 day) for traveling the POIs.
3. Each vehicle collects the profit by visiting the POIs.
4. The POI accessed by a vehicle will not be considered by other vehicles.
5. The POI's profit is equal to its weight.

The TOP is to find the traveling plan that generates the most profits. The results of the TOP are also the best k-day itinerary. □

Due to the complexity of TOP, it is impossible to find the exact solution. Instead, previous work focuses on proposing heuristic algorithms. The basic idea is to generate an initial plan and then adjust it based on some heuristic rules. Those algorithms have three drawbacks. First, the heuristic algorithms need many iterations to get a good enough result, which incur high computation cost [7]. Second, the adjusting rules are too complicated and the potential gains are unknown. Finally, there is no bound of the approximate result, which may be arbitrarily bad in some cases.

In this paper, we reduce the complexity of the TOP by transforming it into a set-packing [8] problem. As the transformation is done in an offline manner, the performance of online query processing is not affected.

3.2 Single-Day Itinerary

The basic idea of transformation is to iterate all possible single-day itineraries. This is done by a set of MapReduce jobs. In the first job, we generate $|\mathcal{P}|$ initial itineraries for the POI set $\mathcal{P}$. Each initial itinerary only consists of one POI. Iteratively, the subsequent MapReduce job tries to add one more POI to the itineraries. If no more single-day itineraries can be generated, the process terminates. In current implementation, we allow maximally m MapReduce jobs in the transformation process to reduce the overheads. Therefore, a single-day itinerary contains at most m POIs. This strategy is based on the assumption that users cannot visit too many POIs in one day. In our crawled data set from Yahoo travel, setting m to 10 is enough for Singapore data, which include more than 400 POIs. Only a few single-day itineraries can contain more than 10 POIs.

Algorithms 1 and 2 show the pseudocodes of the MapReduce job. The *mappers* load the partial paths from the DFS, which are generated in the previous MapReduce jobs. We try to append new POI to the existing itineraries. For each new path, we test whether it can be completed within one day. If not, we will discard the new path. If the old path cannot result in any new path, we will output the old path. For the last MapReduce job (the mth job), all the candidate itineraries are used as the results. The output key-value pair is using the sorted POIs in the itinerary as the key.

Algorithm 1. `map(Object` key`, Text` $value$`, Context` $context$`).`

// we allow maximally $m-$ round MapReduce jobs, i.e., the maximally length of path is m

//$value$: existing path, each MapReduce job tries to add one more POI to the path

1: Path $P =$ parsePath($value$)
2: **for** $i = 0$ to $POIGraph$.POINumber **do**
3: **if** isConnected(P, i) and !P.contains(i) **then**
4: Path $newPath = P$.append(i)
5: cost = P.cost + $POIGraph$.getCost(P.endPOI, i) + $POIGraph$.getCost(i)
6: weight = P.weight + $POIGraph$.getWeight(i)
7: $newPath$.cost = cost
8: $newPath$.weight = weight
9: **if** $newPath$.cost $\leq H$ **then**
10: Key $newKey$ = parsePath($newPath$).sort();
11: $context$.collect($newKey$, $newPath$)
12: **else**
13: DFS.write($resultFile$, P)

Algorithm 2. `reduce(Key` key`, Iterable` $values$`, Context` $context$`).`

1: $bestCost = \infty$
2: $bestPath = NIL$
3: **for** Path P: $values$ **do**
4: **if** $P.cost < bestCost$ **then**
5: $bestPath = P$
6: $bestCost = P.cost$
7: $context$.collect(key, $bestPath$)

In the *mappers*, to compute the weight and cost of new itinerary, we load the POI graph table from the DFS. As the graph table is small, each *reducer* maintains a copy in its memory. The table's schema is as follows:

$$(S\_POI, E\_POI, S\_weight, E\_weight, S\_cost, E\_cost, cost),$$

where S_POI and E_POI denote the two POIs linked by a specific edge, *cost* is the traveling cost from S_POI to E_POI, and S_POI is the primary key of the table.

In the *reducers* (Algorithm 2), we select the path with smallest cost of paths with the same POIs. In each reducer, all the paths have the same POIs. We only keep the path with smallest cost and output such path for the next round. Note that since all the paths have the same POIs, these paths have the same weight.

After all itineraries have been generated, a clean process is invoked to remove the duplication. For two itineraries ($L_0 = v_0 \rightsquigarrow \cdots \rightsquigarrow v_n$ and $L_1 = v'_0 \rightsquigarrow \cdots \rightsquigarrow v'_n$), L_0 contains L_1, iff

$$\forall v'_j \in L_1 \rightarrow \exists v_i \in L_0 (v_i = v'_j).$$

Namely, all POIs in L_1 are also included by L_0. If L_0 contains L_1, we will only keep L_0, as it provides more POIs for the users.

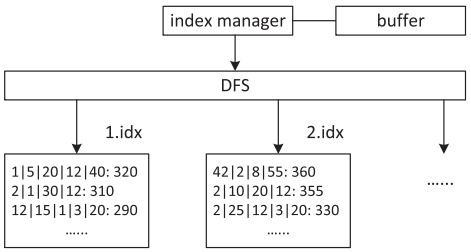


Fig. 4. Itinerary index.

3.3 Itinerary Index

To efficiently locate the single-day itineraries, an inverted index is built. The key is the POI and the values are all itineraries involving the POI. By scanning the index, we can retrieve all the itineraries. Fig. 4 illustrates the index structure. We create an index file for each POI in the DFS. The file includes all single itineraries involving the POI, which are sorted based on their weights. For example, in Fig. 4, "1.idx" contains all itineraries for the first POI. The itinerary "1|5|20|12|40" is the most important itinerary in the index file with weight 320.

The inverted index is constructed via a MapReduce job. Algorithms 3 and 4 show the process. The *mappers* load the single-day itinerary and generate key-value pairs for each involved POI. The *reducers* collect all itineraries for a specific POI and sort them based on the weights before creating the index file. In our system, the size of the index file may vary a lot. Some POI may have an extremely large index file, due to its popularity and short visit time. In *reducers*, those POIs may result in the exception of memory overflow in the sorting process. To address this problem, in the *map* phase, instead of using the POI as the key, we generate the composite key by combining the POI and the itinerary weight.

Algorithm 3. `map(Object` key`, Text` $value$`, Context` $context$`).`
//$value$: single-day itinerary
1: Itinerary $it =$ parse($value$)
2: **for** $i = 0$ to it.POISize() **do**
3: int $nextPOI = it$.getNext(i)
4: Key $key =$ new CompositeKey($nextPOI$, it.weight/bucketSize)
5: $context$.collect(key, it)

Algorithm 4. `reduce(Key` key`, Iterable` $values$`, Context` $context$`).`
1: CompositeKey $ck = key$, Set $s = \emptyset$
2: **for** Itinerary it: $values$ **do**
3: s.add(it)
4: sort(s)
5: DFSFile $f =$ new DFSFile(ck.first + "_"+ck.second)
6: f.write(s)

In particular, we partition the itineraries into n buckets. The bucket ID is used as a part of the composite key. In this way, we split the itineraries of a POI into n groups and each group can be efficiently sorted in the memory. Each group will result in an index file. However, it is not necessary to merge the files, as the files are partitioned based on the weights. By scanning all files from the nth bucket to the 1th bucket, we can get a sorted list for all itineraries involving a POI.

To simplify the index manipulation, an index manager is built in our query engine. The index manager only provides one interface *scan(POI)*, where POI denotes the owner of the index. The interface returns an iterator, which can be used to retrieve all itineraries of the POI. A memory buffer is established to cache the used itineraries and the LRU strategy is applied to maintain the buffer.

3.4 Discussion: Why MapReduce

Although the input data set (POI graph) is small in size, the partial results of the possible itineraries are extremely large (more than 100G or even 1T). The computation is also intensive, which cannot be completed by a single machine. MapReduce is the solution to partition the partial results and generate the itineraries in parallel. Its advantages are twofold:

1. Parallel computing effectively reduces the running time of preprocessing. The search space explodes, when the number of POIs and traveling days increases. It is impractical to generate all possible itineraries. But by exploiting the power of MapReduce, we can share and balance the workload between multiple machines. The scalability is achieved by adding more nodes into the cluster. In our experiment, the running time of preprocessing is significantly reduced with the number of nodes (see Fig. 12)
2. MapReduce algorithms can remove the duplicated itineraries in a simple way. In Algorithm 2, by leveraging the framework of MapReduce, we map all the itineraries with the same POIs into the same reducer and only keep one itinerary with the lowest cost. This approach can prune the low-benefit partial itineraries as early as possible and lead to less input for the next round of computation.

4 Greedy-Based Approximation Algorithm

After the itinerary indexes are constructed, the user request (S_p, k) can be processed by selecting k best itineraries from the indexes. Namely, the problem of generating optimal k-day itinerary is transformed into a weighted set-packing problem as shown in the following theorem.

Definition 3 (Weighted Set-Packing Problem). *In a universe $\mathcal{U}$, we assume that each element in $\mathcal{U}$ has a weight and the weight of any subset of $\mathcal{U}$ equals to the sum of the element weights in the subset. Given a family $\mathcal{S}$ of $\mathcal{U}$'s subsets, the set-packing problem is to select a subfamily $\mathcal{S}'$ from $\mathcal{S}$, where all subsets in $\mathcal{S}'$ are disjoint and the weight of $\mathcal{S}'$ is maximal among all possible selections.*

Theorem 2. *Finding optimal k-day itinerary can be reduced to the weighted set-packing problem.*

Proof (Sketch). By solving the set-packing problem, we can also get the optimal k-day itinerary, as

1. Each single-day itinerary can be considered as a subset of the POI set $\mathcal{P}$.
2. The subsets selected by the set-packing problem are disjoint, and hence in the k-day itinerary, we will not visit a POI twice.

1.idx	2.idx	3.idx	4.idx
$1\|2\|4\|X_1$	$5\|2\|4\|X_1$	$3\|7\|4\|X_1$	$4\|2\|1\|X_1$
$1\|2\|4\|X_2$	$5\|2\|4\|X_2$	$3\|7\|4\|X_2$	$4\|2\|1\|X_2$
$5\|1\|6\|X_1$	$2\|8\|9\|X_1$	$7\|5\|3\|X_1$	$3\|4\|5\|X_1$
$5\|1\|6\|X_2$	$2\|8\|9\|X_2$	$7\|5\|3\|X_2$	$3\|4\|5\|X_2$

Fig. 5. Example of Set-Packing

3. Each subset is replicated $k-1$ times and thus, we have k identical itineraries. For the ith itinerary, a virtual POI x_i is appended, denoting that the itinerary is designed for the ith day.
4. Apply the algorithm of set-packing to get the optimal solution. Let S_r be the result set. If $|S_r| > k$, there must be two itineraries for the same day and they are not disjoint. If $|S_r| < k$, we still have available days for traveling and new itineraries can be added. Therefore, $|S_r| = k$ and S_r can be considered as a k-day itinerary.

□

In step 3 of our proof, we replicate the itinerary $k-1$ times. That is to guarantee that the solution of set-packing problem returns exactly k subsets. Fig. 5 illustrates the idea. Suppose we have four index files and want to generate a two-day itinerary. Without the replication, the set-packing algorithm may return a three-day itinerary, such as "$5|1|6$," "$2|8|9$," and "$3|7|4$." By replicating the itineraries and adding the virtual elements X_1 and X_2, the above selection cannot work, as two itineraries will share at least one virtual element. In this case, the set-packing algorithm will return another solution (e.g., "$1|2|4|X_1$" and "$7|5|3|X_2$"), which satisfies our time requirement.

Although set-packing is also an NP-complete problem, different from the TOP, in a special case, set-packing problem has approximate algorithms. As mentioned in the preprocessing, we set the maximal number of MapReduce jobs in generating the single-day itineraries to m. Therefore, each itinerary can have at most m POIs. It was shown that when the size of subsets is bounded by a constant, the weighted set-packing problem can be solved by polynomial approximations [8], [9]. By following the above ideas, in this paper, we design a variant of the approximate algorithm in [8], which provides a bound of $\frac{2(m+1)}{3}$ for the quality of the approximate answers. The algorithm includes an initialization phase and an adjustment phase.

4.1 Initialization

For the user request (S_p, k), we adjust the weights of POIs in S_p to emphasize the user's selection. If $v_i \in S_p$, v_is weight is increased to $\alpha(w(v_i)+1)$, where α is an integer larger than 0 and $w(v_i)$ is the original weight of POI v_i. Algorithm 5 shows how we generate the seed itineraries using the greedy strategy.

Algorithm 5. `Initialization(POIList` L`, Day` k`).`

```
1: sortByWeight(L)
2: int i = 0, Set seed = ∅, Set rev = ∅
3: while i < k and L.size() > 0 do
4:     int poi = L.nextPOI();
5:     Set group = new Set()
6:     group.add(poi)
7:     int lastpoi = poi
8:     while not L.isEmpty() do
9:         int newpoi = getNearest(lastpoi, L)
10:        int time = getTravelTime(group, newpoi)
11:        if time ≤ one day then
12:            group.add(newpoi)
13:            L.remove(newpoi)
14:            lastpoi = newpoi
15:        else
16:            break;
17:    i++, seed.add(group)
18: for i = 0 to seed.size() do
19:    Set group = seed.get(i)
20:    IndexIterator iter = indexManager.scan(group.
       get(0))
21:    while iter.hasMoreElements() do
22:        Itinerary I = iter.next()
23:        if I.contains(group) then
24:            removeReplicatedPOI(I, rev)
25:            rev.add(I)
26:            break
27: return rev
```

We first sort the selected POIs by their weights (line 1). Then, in each iteration, we try to form a group, which contains a subset of POIs that can be accessed within one day (line 3-17). We greedily select the POI with shortest distance and add it into our group (line 9-14). There are maximally k groups generated. All groups are used as our seeds for searching the index. We will use the first itinerary that contains all the POIs in the group as our candidate itinerary (line 18-26). Although after the weight adjustment, itineraries in the index file are no longer sorted by the weights. We can still retrieve the itinerary with maximal weight as shown in the following theorem.

Theorem 3. *Given a list of POIs $L = \{v_0, v_1, \ldots, v_n\}$ that can be accessed within one day, by scanning the index of v_i in L, we can get the itineraries that contain all POIs in L and the first candidate is the itinerary with maximal weight.*

Proof (Sketch). Because L can be finished within one day, there must be some itineraries containing all the POIs in L. Let I_0 and I_1 be first and second candidate itineraries, respectively. I_0s weight is larger than I_1s, as before weight adjustment, I_0 has a higher weight than L_1 and after weight adjustment, both of them receive the same weight boost. □

To improve the weights of the obtained itineraries in the greedy algorithm, we adopt the adjustment phase.

4.2 Adjustment

In the adjustment phase, new solutions are searched and used to replace the greedy itineraries. The process repeats until no improvement can be obtained. In the following discussion, we discard the virtual POIs to simplify our representations. Suppose $idx(v_j)$ returns the itineraries in the index of POI v_j. We define the neighborhood of an itinerary as

Definition 4 (Neighborhood). *Given an itinerary L_i, its neighborhood $ngb(L_i)$ is an itinerary set satisfying:*

$$ngb(L_i) = \bigcup_{v_j \in L_i} idx(v_j).$$

For example, in Fig. 5,

$$ngb(1|2|4) = \{5|1|6, 5|2|4, 2|8|9, 4|2|1, 3|4|5\}. \quad (2)$$

The neighborhood of L_i represents the candidate itineraries that can replace L_i. However, some itineraries share the common POIs, which cannot coexist in the result. Therefore, we define the independent set as

Definition 5 (Independent Set). *An independent set $IS(L_i)$ is a subset of $ngb(L_i)$. Any two itineraries in $IS(L_i)$ do not share a common POI. Namely, $\forall L_0, L_1 \in IS(L_i) \rightarrow (L_0$ and L_1 are disjoint).*

Neighborhood of each itinerary can have multiple independent sets and each set denotes a different adjustment strategy. Let $\mathcal{S}$ be the initial itinerary set returned by Algorithm 5. An alternative solution $\mathcal{S}'$ can be constructed from $\mathcal{S}$ by replacing the itineraries by their independent sets. More formally,

$$\mathcal{S}' = \mathcal{S} - f(\mathcal{S}, ngb(L_i)) + IS(L_i),$$

where $f(S_a, S_b)$ returns a subset of S_a, which shares at least one POI with itineraries in S_b.

For itinerary "1|2|4" in Fig. 5, its independent set is $\{2|8|9, 3|4|5\}$. If $\mathcal{S} = \{1|2|4, 7|5|3\}$, after the adjustment, we will get $\mathcal{S}' = \{2|8|9, 3|4|5\}$. All itineraries are replaced by new ones. To avoid the case of cascading replacement, the size of $IS(L_i)$ should be less than k, as only k single-day itineraries are required. In our implementation, we limit the size of $IS(L_i)$ to $\frac{k}{2}$. Namely, at most half of the itineraries are replaced.

The benefit of itinerary adjustment is computed as

$$B = weight(\mathcal{S}') - weight(\mathcal{S}).$$

If $B > 0$, we assume that the adjustment improves the quality of the results. Hence, a better itinerary can be produced by replacing the old itineraries with corresponding independent sets.

Algorithm 6 summarizes the idea of adjustment process. We set a threshold for the maximal number of adjustments. In each iteration, we find the independent sets for the existing itineraries. If one itinerary has multiple independent sets, we will select the one with maximal weight (line 6). The new results are then computed by performing the replacement (line 7) and we record the benefit (line 8). After all possible replacement strategies have been checked, we will select the one with maximal benefit. If the benefit is larger than 0, the result itineraries are updated as the new ones (line 13). Otherwise, we will perform the updates, only with a small probability (line 15-16). The idea is to simulate the hill-climbing algorithm to avoid the suboptimal solution. The algorithm guarantees the quality of the returned itinerary as shown in the below theorem.

Algorithm 6. `Adjustment(Set` $\mathcal{S}$`, double` P`, int` $step$`).`

```
1: int j = 0;
2: while j < step do
3:    Set cand = ∅, int max = −∞, int idx = −1
4:    for i = 0 to S.size() do
5:       Set ngb = S.get(i).getNeighborhood()
6:       Set ind =
         getIndependentSetWithMaximalWeight(ngb)
7:       Set S' = S − f(S, ngb) + ind
8:       double B = weight(S') − weight(S)
9:       cand.add(S')
10:      if B > max then
11:         max = B, idx = i
12:   if max > 0 then
13:      S = cand.get(idx)
14:   else
15:      if randProb() > P then
16:         S = cand.get(idx)
17:   j++
```

Theorem 4. *Algorithm 6 returns a k-day itinerary, which approximates the optimal solution with the bound $\rho = \frac{2(m+1)}{3}$.*

Proof. (sketch) In Algorithm 6, we add a virtual POI to each itinerary to mark its traveling day. Therefore, the adjustment algorithm at most returns k disjoint itineraries. Otherwise, there are two itineraries sharing the same virtual POI. Namely, they are supposed to be traveled in the same day, which is not possible. If the algorithm returns less than k itineraries, we can still repeat the initialization and adjustment to fill in the left days. In this way, we guarantee that Algorithm 6 returns exactly a k-day itinerary. Based on Theorem 2, the problem of selecting the k-day itinerary can be reduced to the weighted set-packing problem. Therefore, in Algorithm 6, we simulate the heuristic set-packing algorithm. The heuristic algorithm has been analyzed in [8]. Suppose there are X iterations in the algorithm. Let I_i be the results of the $X - i - 1$ iteration. I_1 will be the final result. Let d_i be the payoff factor of each iteration. We have

$$(m+1)w(I_1) \geq \left(2 - \frac{1}{d_1} + \frac{1}{2d_1^2}\right) w(opt),$$

where $w(I_1)$ and $w(opt)$ represent the weights of the itinerary returned by the heuristic algorithm and the optimal itinerary, respectively. The right side of the equation is minimized when $d_1 = 1$. In that case, we have

$$(m+1)w(I_1) \geq \left(1 + \frac{1}{2}\right) w(opt).$$

Therefore, we have a bound $\rho = \frac{2(m+1)}{3}$ for the heuristic approach, where m is the maximal number POIs in the itinerary (m is the number of MapReduce jobs in our preprocessing). □

The most expensive operations in Algorithm 6 are retrieving the neighborhood sets. We need to scan the indices of involved POIs to find all itineraries. We find that as Algorithm 6 only selects one independent set for each itinerary, we can save I/O costs by scanning a small portion of the index file. Therefore, in our implementation, we read

the first n itineraries of an index file in batch and if independent sets are found, the process stops. Otherwise, we will continue to load the next n itineraries.

4.3 Hotel Selection

In fact, hotels can be considered as a special type of POIs. It must appear as the last POI in the itinerary. We need to calculate the traveling time from other POIs to the hotel POIs. Hotel POIs do not incur access cost and their weights are set as users' rankings for the hotels. Based on the user's preference, we have two processing strategies.

4.3.1 Multiple Hotels

If the user does not insist on staying in the same hotel (e.g., he can select k different hotels, one for each day), we can extend the preprocessing algorithm to handle the hotels. In the MapReduce jobs, when a new itinerary L_i is generated, we test every hotel POI and try to append it to the end of L_i. Given a hotel POI h_j, we use $L_i|h_j$ to represent the combined itinerary. $L_i|h_j$ is considered as a single-day itinerary, if

1. The total traveling time of $L_i|h_j$ is less than H. H is the average traveling time per day.
2. For any other nonhotel POI $\bar{v}$ which is not included by L_i, $L_i|\bar{v}|h_j$ cannot be completed within H time.

When we detect a new single-day itinerary, we output it to the DFS for indexing.

The itinerary generation algorithm is exactly the same, except that the hotel POI can appear in different itineraries. In Algorithm 6, we do not consider the hotel POIs, when performing the disjoint test for itineraries. The output itinerary may contain multiple hotels (h_i represents the hotel POI):

$$2|5|10|h_1, 3|7|8|h_1, 9|10|0|h_2.$$

4.3.2 Single Hotel

If the user prefers to stay in the same hotel, the itinerary generation problem cannot be easily reduced to the set-packing problem. Instead, we adopt a best-effort solution. In particular, we still apply Algorithm 6 to find the candidate k-day itinerary without hotel POIs. After that, we invoke Algorithm 7 to append the hotel POI.

Algorithm 7. `HotelSelection(Set` *hotels*`, Set` *itinerarySet*`)`.

```
1: double max = 0, Set result = ∅
2: for i = 0 to hotels.size() do
3:     Hotel h_i = hotels.get(i)
4:     Set copy = itinerarySet
5:     for j = 0 to copy.size() do
6:         Set L_j = copy.get(j)
7:         while getTravelTime(L_j, h_i) > H do
8:             L_j.removelast()
9:         L_j.append(h_i)
10:    double weight = getTotalWeight(copy)
11:    if max < weight then
12:        max = weight
13:        result = copy
14: return result
```

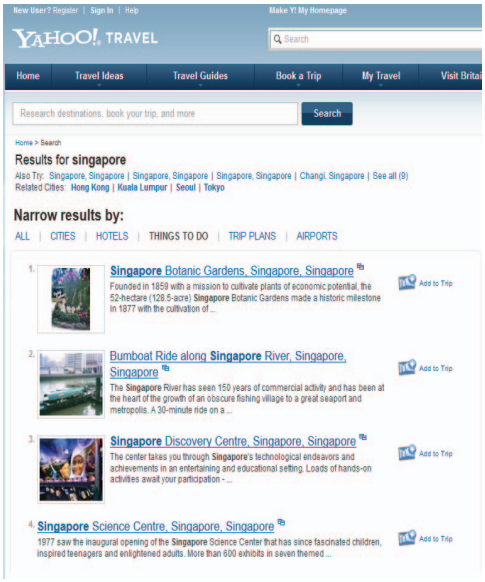


Fig. 6. Yahoo POIs.

The idea is to discard a few POIs from the end of each itinerary and try to append the hotel POIs to the shortened itinerary. In line 7-8, the itinerary progressively removes the last POI, until it can include the hotel POI to form a single-day itinerary. For example, the total traveling time is less than H. In line 10, we will get a new set of k itineraries, where all itineraries contain the same hotel POI. We will generate such a k-day itinerary for each hotel. After comparing weights of the itineraries, the one with maximal weight is returned as our final k-day itinerary.

5 Experiment Evaluations

5.1 Data Set Description

To evaluate the performance of our proposed approaches, we crawl the traveling information from Yahoo Travel (http://travel.yahoo.com). In particular, we focus on the Singapore POIs. Fig. 6 illustrates our crawling strategy. Yahoo classifies the POIs into *hotels*, *things to do*, and *cities*. We use the first two types in our experiments, as the last one is the geolocations for the city. *Things to do* contains 254 POIs of Singapore and *hotels* contain 276 hotels from unranked to five stars. After removing the duplicated and meaningless POIs, we keep 400 POIs for our experiments. As far as we know, this is the largest data set for the automatic itinerary generation. In [2], the largest data set only contains 163 POIs.

The POI's weight is also crawled from Yahoo Travel. As shown in Fig. 7, for each POI, Yahoo maintains a page for users' reviews. We accumulate the user scores for each POI as its weight. If a POI has not been reviewed, we assign it an initial weight (e.g., 1).

The average visiting time of a POI is estimated from the shared travel plans in Yahoo Travel. The edge cost between any two POIs are estimated using Google Map. Specifically, the public transit time for the shortest path between two POIs is used as the edge cost. We assume that each user will spend at most 8 hours for traveling per day.

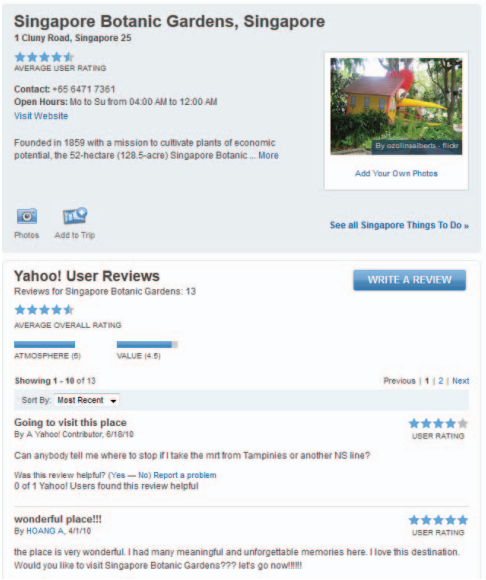


Fig. 7. User reviews.

TABLE 1
Experiment Settings

Settings	
Parameter	Range and Default Value
k	3 (1-5)
α	2
buffer size	5 million itineraries
size of S_p	10 (5-20)
total POIs	400 (100-400)
number of MapReduce nodes	32 (8-64)
number of Map/Reduce per node	2
data chunk size of HDFS	64M
number of MapReduce jobs (m)	10

In our experiments, the query is (S_p, k), where S_p is randomly selected from the nonhotel POIs. We allow the users to select the same hotel POIs for different days. The traveling time is set to three days by default. For comparison, we implement the original TOP algorithm proposed in [3].

Table 1 lists the parameters used in our experiments. The experiments are conducted on our in-house cluster, Awan (http://awan.ddns.comp.nus.edu.sg/ganglia/). We use 64 nodes exclusively. Each node has one Intel X3430 2.4-GHz processor, 8-GB memory, two 500-GB SATA hard disks, and gigabit ethernet. Hadoop [10] is used as our MapReduce engine.

5.2 Single-Day Itinerary Generation

In the preprocessing, m MapReduce jobs are submitted sequentially to iterate all possible single-day itineraries. The input are our crawled POIs and the output contain all single-day itineraries. This is, in fact, a brute-force search strategy, but we exploit the parallel processing engine to reduce its cost. After the single-day itineraries are generated, we start another MapReduce job to remove the duplicate itineraries. We call it the *Dup-Clean* job (the previous m jobs are named *MR-Scan*). *Dup-Clean* generates a special namespace for each itinerary by combining its POIs. The namespace is used as the key in the shuffling phase. All duplicated itineraries will be shuffled to the same *reducer*, where a local clean process is conducted.

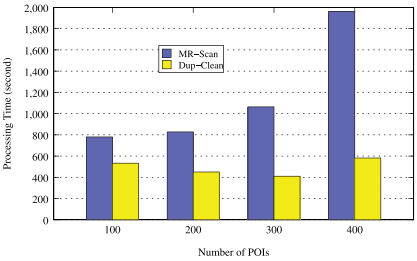


Fig. 8. Preprocessing cost.

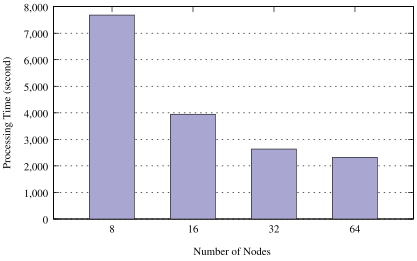


Fig. 9. Scalability of preprocessing.

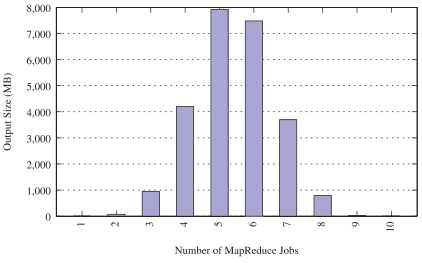


Fig. 10. Size of a single-day itinerary.

Fig. 8 shows the accumulate costs of all m jobs and the cost of the clean job. We vary the number of POIs in the preprocessing from 100 to 400. The cost of the *MR-Scan* increases for a larger POI set. However, even for 400 POIs, the preprocessing only takes less than 1 hour. This is an offline process and only needs to be invoked once. In fact, most travel agencies do not maintain such a large number (400) of POIs for a single city. Interestingly, the performance *Dup-Clean* is not correlated to the POI number. Its cost is neutralized by the parallel processing strategy. We observe that most nodes are not fully exploited in *Dup-Clean*. Fig. 9 shows the scalability of the MapReduce jobs (*MR-Scan+Dup-Clean*). We vary the number of nodes in our cluster from 8 to 64 and we observe a near-linear improvement over the performance. Therefore, to handle a larger POI-graph, we can simply add more processing nodes into our cluster.

In the preprocessing, the maximal number of MapReduce jobs (m) is set to 10. Namely, each single-day itinerary can contain at most 10 POIs. m is a configurable parameter. As shown in Fig. 10, in our data set, most itineraries consist of 4-7 POIs. Setting m to 10 can iterate most itineraries in our case.

5.3 Itinerary Indexing

The second step of preprocessing is to build the itinerary index. The index process only requires one MapReduce job and is much faster than the itinerary iteration process. In

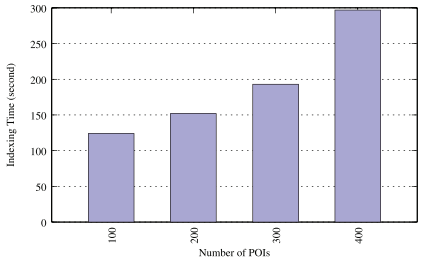


Fig. 11. Indexing cost.

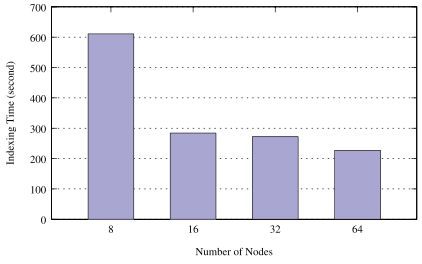


Fig. 12. Scalability of indexing.

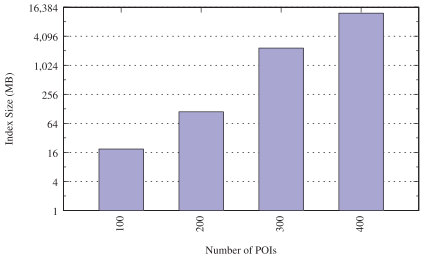


Fig. 13. Size of index.

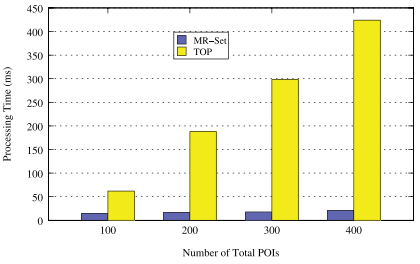


Fig. 14. Effect of graph size (processing time).

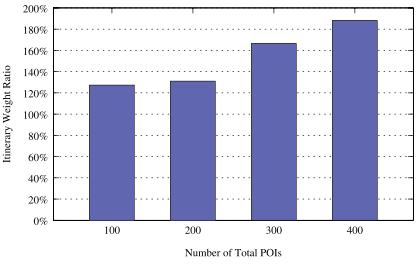


Fig. 15. Effect of graph size (quality).

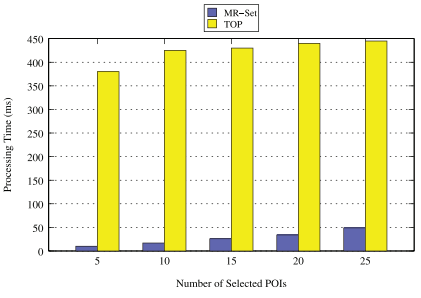


Fig. 16. Effect of selected POIs (processing time).

Fig. 11, we show the indexing cost for different sizes of POI graphs. We can efficiently recreate the index within a few minutes. Fig. 12 conducts the scalability test for the indexing process. The indexing process benefits from a larger cluster. Fig. 13 shows the total index size for different POI graphs. The size of index increases exponentially with the size of POI graph. But even for the graph with 400 POIs (a large enough POI graph for most cities), only 12 GB index data are generated. The index is maintained in the DFS and hence, the storage cost is not the system bottleneck.

5.4 Effect of POI Graph Size

In the experiments, we compare our approach (*MR-Set*) with the original *TOP* approach in [3]. To evaluate the query performance, two metrics, processing time, and weight ratio are adopted. The weight ratio is used to measure the quality of the generated itineraries. In particular, let W_i and W_j denote the total weights of *MR-Set* and *TOP*, respectively. The weight ratio is defined as $\frac{W_i}{W_j}$.[3]

We first vary the graph size to test the query performance. Figs. 14 and 15 show the processing time and weight ratio, respectively. Our new approach significantly reduces the processing cost, as we have already computed the single-day itineraries in the preprocessing. The previous *TOP* approach is not scalable. The query cost increases linearly with the number of POIs. If more POIs (e.g., restaurants) are added in the set, the *TOP* approach may fail to provide a satisfied performance. On the contrary, our technique enables the itinerary to be generated within 30 milliseconds. It is not affected by the POI graph size. Moreover, the traveling plan system is accessed by multiple users concurrently. In the case of 400 POIs, the *TOP* approach can serve up to two requests per second, while our approach can provide a throughput of 40 requests per second. Our approach is more scalable and feasible for the real-time processing.

In fact, our approach not only reduces the processing overhead, it also provides results with higher qualities. Fig. 15 shows the change of weight ratio. We have 20 to 80 percent improvement over the original TOP algorithm. The gap increases for a larger POI graph, as our approach can efficiently exploit the POI combinations. More POIs indicate a higher possibility of finding a good itinerary.

5.5 Effect of Selected POIs

In our query model, we allow the user to explicitly select some POIs as their preferences. The weights of the selected POIs are adjusted to reflect the selection. This strategy may increase the importance of some unpopular POIs and avoids generating the itinerary with the same set of top-popular POIs. This is how the users customizes their itineraries in our system. Figs. 16 and 17 show the effect of varied number of selected POIs (from 5 to 25). The default traveling time is set to three days. In fact, most people will not select too many POIs (e.g., 25) for a three-day itinerary.

3. In idea case, we should compare the approximate results with the optimal ones. However, it is impossible to generate the optimal results, given the size of POI graph and complexity of the problem.

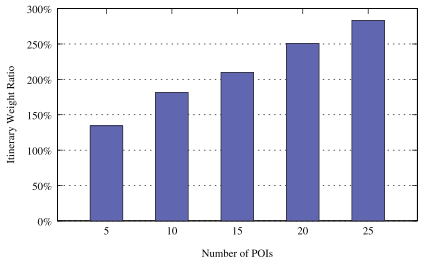


Fig. 17. Effect of selected POIs (quality).

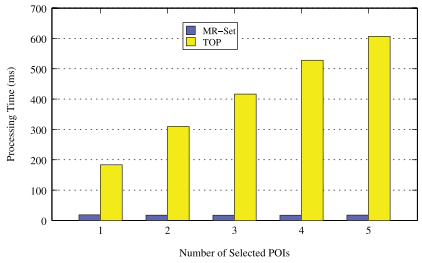


Fig. 18. Effect of traveling time (processing time).

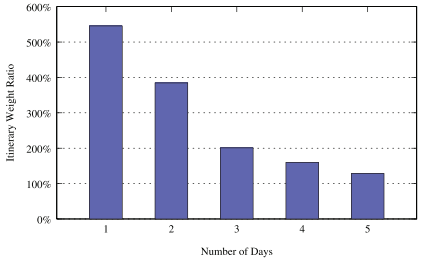


Fig. 19. Effect of traveling time (quality).

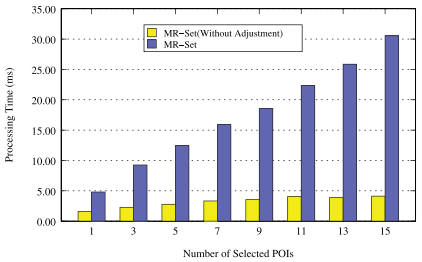


Fig. 20. Effect of adjustment (processing time).

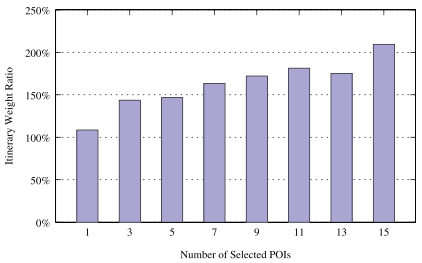


Fig. 21. Effect of adjustment (quality).

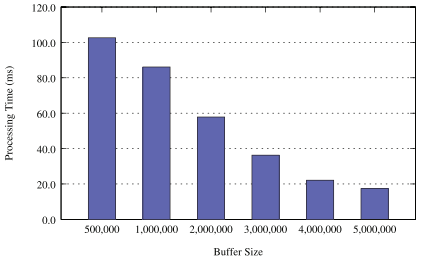


Fig. 22. Buffer size.

In Fig. 16, the cost of *MR-Set* increases for a larger number of selected POIs. This is because in the adjustment phase, *MR-Set* needs to look up the index of the corresponding POIs to search for the replacements. Index is maintained by the DFS and the I/O costs dominate the query cost. However, *MR-Set* is still much more efficient than the original TOP.

Fig. 17 reveals that the quality gap between *MR-Set* and the TOP approach enlarges, when the user selects more POIs as his preference. *MR-Set* can effectively find the itinerary that includes as many selected POIs as possible. It can optimize the way of how to combine the selected POIs and other POIs into the itinerary.

5.6 Effect of Traveling Budget

Besides the POIs, the user can change his expected traveling time as well. With more time budget, his itinerary can include more interested POIs. Figs. 18 and 19 show the effect of varied time budgets. The original TOP algorithm incurs a higher overhead for the increased time budget, because it needs to generate and refine each single-day itinerary progressively. *MR-Set* adopts a different strategy. When it tries to adjust the itinerary, it may replace multiple single-day itineraries with new ones. It considers the k-day itinerary as a whole solution, instead of treating each single-day itinerary independently. It is interesting to observe that Fig. 19 shows a different result from Fig. 17. The weight ratio decreases, when more traveling budget is given. In fact, the TOP algorithm benefits from a loose time budget, as it can arrange more high-weight POIs into different single-day itineraries.

5.7 Effect of Adjustment

The query processing of *MR-Set* splits into the initialization phase and adjustment phase. The initialization phase applies the greedy-based heuristic approach to generate a k-itinerary as the seed, which is further improved in the adjustment phase by replacing the itineraries with their independent sets. In this experiments, we show the effect of the adjustment phase. We vary the number of selected POI from 1 to 15.

Fig. 20 shows that the adjustment phase greatly increases the processing cost. Algorithm 6 may repeat for several iterations before converging to a high-quality result. As mentioned before, in the adjustment phase, the query engine loads the itinerary index from the DFS, which incurs high I/O cost. One way to reduce the cost is to increase the index buffer size. After an indexed itinerary is loaded from the DFS, we cache it in the buffer. If the buffer is full, we apply the LRU strategy to remove the less used entries.

In Fig. 22, we change the number of buffered single-day itineraries in the index buffer and test the query performance. Not surprisedly, we can get a huge performance boost by deploying a large enough index buffer. In fact, the single-day itinerary is less than 64 bytes and caching 5 million entries only takes about 300 M memory. Any modern server can effectively reduce the processing cost by employing a large buffer.

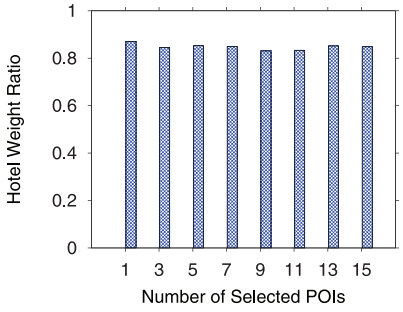


Fig. 23. Effectiveness of single hotel selection.

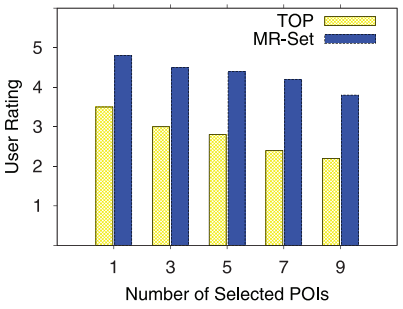


Fig. 24. User study.

Although the adjustment phase incurs high processing cost, it can significantly improve the result quality. As shown in Fig. 21, the adjustment phase can double the weight of generated itinerary if more than 15 POIs are selected.[4] With more POIs selected, the adjustment phase can generate more replacement itineraries and therefore, has a better chance of finding the high-quality result.

5.8 Effect of Single Hotel Selection

In this section, we justify the effectiveness of hotel selection algorithm. In Algorithm 7, we adopt a "best-effort" solution to append the hotel to the end of each itinerary. To evaluate the performance of such a solution, we define a new metric, the hotel weight ratio. In particular, let W_m and W_s denote the total weights of generated itineraries in the multiple hotel case and single hotel case, respectively. The hotel weight ratio is defined as $\frac{W_s}{W_m}$. Our "best-effort" solution still provides high-quality results. Fig. 23 shows the change of the hotel weight ratio. We can see that, in the single hotel case, the total weight of generated itineraries is penalized as each single-day itinerary should end in the same hotel POI. However, the "best-effort" solution can provide an approximate result with 85-90 percent of the total weight as in the multiple hotel case. This indicates that Algorithm 7 is still able to find good itineraries with the single hotel constraint.

5.9 User study

To evaluate the quality of the generated itineraries, we conduct a user study, which asks the users to manually rank the itineraries. Our study hires 20 undergraduate students as the users. Given a set of selected POIs, we use the *TOP* and *MR-Set* methods to generate 20 groups of itineraries (three-day itineraries in the experiment). Each participant assigns a score (ranging from 1 to 5) to each itinerary in his group. The average ranks are then computed for the itineraries generated by different approaches. Fig. 24 shows the results. Most users prefer the results generated by *MR-Set*. We also observe that the ratings of both the *TOP* and *MR-Set* are reduced, when more POIs are selected as the necessary POIs. It is because that some of the user selected POIs are missing in the itineraries due to the constraint of travel time.

6 Related Work

Most existing work on itinerary generation take a two-step scheme. They first adopt the data mining algorithms to discover the users' traveling patterns from their published images, geolocations and events [11], [12], [13]. Based on the relationships of those historical data, new itineraries are generated and recommended to the users [14], [15], [16]. This scheme leverages the user data to retrieve POIs and organize the POIs into itinerary, which is based on a different application scenario to ours. We help the traveling agency provide the customized itinerary service, where all details of POIs are known and each user prefers different itinerary instead of adopting the most popular ones. In our case, the itinerary generation problem is a search problem for the optimal POI combinations.

In fact, searching for the optimal single-day itinerary has been well studied. It can be transformed into the traveling salesman problem (TSP) [5], which is a well-known NP-complete problem. For example, in [17], given a set of POIs, the system will generate a shortest itinerary to access all the POIs. If the distance measure is a metric and symmetric, the TSP has the polynomial approximate solution [18], but the approximate solution incurs high overhead for a large POI graph [19]. Therefore, some heuristic approaches [1] are adopted to simplify the computation.

Some interactive search algorithms [2], [20] are proposed in recent years. These algorithms still focus on optimal single-day itinerary planning. To reduce the computation overhead and improve the quality of generated itineraries, users' feedbacks are integrated into the search algorithm. The search algorithm works iteratively. It proposes new itineraries for users based on their previous feedbacks and the users can adjust the weights of POIs in the itinerary or select new POIs into the itinerary. In the next iteration, the algorithm will refine its results based on the collected information. Those work can be considered as variants of optimal single-day itinerary planning problems, whereas our algorithms focus on generating multi-day itineraries. Moreover, interactive algorithms pose requirements for the users, who may be reluctant to provide the feedbacks.

To the best of our knowledge, no previous work studied the problem of generating multiday itinerary. This problem is more challenging than the single-day itinerary, because simply combining multiple optimal single-day itineraries may result in a suboptimal solution. The multiday itinerary, as shown in this paper, can be reduced to the team orienting problem (TOP) [3], which is an NP-complete problem with no approximate solution. Therefore, many heuristic approaches are proposed [6], [21], [22]. The heuristic approaches cannot guarantee the quality of generated itineraries. To address the problem, in this paper, we apply

4. In this figure, the weight ratio is computed between the *MR-Set* with adjustment and *MR-Set* without adjustment.

the MapReduce framework to generate the single-day itineraries. The parallel engine of MapReduce allows us to solve some NP-complete problems more efficiently. Other work [23], [24] also try to leverage the power of MapReduce to reduce the processing cost of NP-complete problems. The beauty of our approach is that after the transformation, the itinerary planning problem is reduced to the weighted set-packing problem, which has approximate solutions under some contraints.

7 Conclusion

In this paper, we present an automatic itinerary generation service for the backpack travelers. The service creates a customized multiday itinerary based on the user's preference. This problem is a famous NP-complete problem, team orienting problem, which has no polynomial time approximate algorithm. To search for the optimal solution, a two-stage scheme is adopted. In the preprocessing stage, we iterate and index the candidate single-day itineraries using the MapReduce framework. The parallel processing engine allows us to scan the whole dataset and index as many itineraries as possible. After the preprocessing stage, the TOP is transformed into the weighted set-packing problem, which has efficient approximate algorithms. In the next stage, we simulate the approximate algorithm for the set-packing problem. The algorithm follows the initialization-adjustment model and can generate a result, which is at most $\frac{2(m+1)}{3}$ worse than the optimal result. Experiments on real data set from Yahoo's traveling website show that our proposed approach can efficiently generate high-quality customized itineraries.

Acknowledgments

The work of Sai Wu was supported by the National Science Foundation of China (NSFC Grant 60970124, 61170034). The work of Sai Wu, Jingbo Zhou, and Anthony K.H. Tung was carried out at the SeSaMe Centre. It is supported by the Singapore NRF under its IRC@SG Funding Initiative and administered by the IDMPO. Sai Wu was the corresponding author.

References

[1] S. Dunstall, M.E. Horn, P. Kilby, M. Krishnamoorthy, B. Owens, D. Sier, and S. Thiebaux, "An Automated Itinerary Planning System for Holiday Travel," *Information Technology and Tourism,* vol. 6, no. 3, pp. 195-210, 2004.

[2] S.B. Roy, G. Das, S. Amer-Yahia, and C. Yu, "Interactive Itinerary Planning," *Proc. IEEE 27th Int'l Conf. Data Eng. (ICDE),* pp. 15-26, 2011.

[3] I.-M. Chao, B.L. Golden, and E.A. Wasil, "The Team Orienteering Problem," *European J. Operational Research,* vol. 88, no. 3, pp. 464-474, Feb. 1996.

[4] J. Dean and S. Ghemawat, "MapReduce: A Flexible Data Processing Tool," *Comm. ACM,* vol. 53, pp. 72-77, Jan. 2010.

[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms,* second ed. The MIT Press and McGraw-Hill Book Company, 2001.

[6] C. Archetti, A. Hertz, and M.G. Speranza, "Metaheuristics for the Team Orienteering Problem," *J. Heuristics,* vol. 13, pp. 49-76, Feb. 2007.

[7] P. Vansteenwegen, W. Souffriau, and D.V. Oudheusden, "The Orienteering Problem: A Survey," *European J. Operational Research,* vol. 209, pp. 1-10, Feb. 2011.

[8] M.M. Halldórsson and B. Chandra, "Greedy Local Improvement and Weighted Set Packing Approximation," *J. Algorithms,* vol. 39, pp. 223-240, May 2001.

[9] E.M. Arkin and R. Hassin, "On Local Search for Weighted K-Set Packing," *Math. Operations Research,* vol. 23, pp. 640-648, Mar. 1998.

[10] http://hadoop.apache.org/, 2013.

[11] T. Rattenbury, N. Good, and M. Naaman, "Toward Automatic Extraction of Event and Place Semantics from Flickr Tags," *Proc. 30th Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval (SIGIR '07),* pp. 103-110, 2007.

[12] D.J. Crandall, L. Backstrom, D.P. Huttenlocher, and J.M. Kleinberg, "Mapping the World's Photos," *Proc. 18th Int'l Conf. World Wide Web (WWW),* pp. 761-770, 2009.

[13] M. Clements, P. Serdyukov, A.P. de Vries, and M.J. Reinders, "Using Flickr Geotags to Predict User Travel Behaviour," *Proc. 33rd Int'l ACM SIGIR Conf. Research and Development in Information Retrieval (SIGIR),* 2010.

[14] C.-H. Tai, D.-N. Yang, L.-T. Lin, and M.-S. Chen, "Recommending Personalized Scenic Itinerary with Geo-Tagged Photos," *Proc. IEEE Int'l Conf. Multimedia and Expo (ICME),* pp. 1209-1212, 2008.

[15] M.D. Choudhury, M. Feldman, S. Amer-Yahia, N. Golbandi, R. Lempel, and C. Yu, "Automatic Construction of Travel Itineraries Using Social Breadcrumbs," *Proc. 21st ACM Conf. Hypertext and Hypermedia (HT),* pp. 35-44, 2010.

[16] H. Yoon, Y. Zheng, X. Xie, and W. Woo, "Smart Itinerary Recommendation Based on User-Generated GPS Trajectories," *Proc. Seventh Int'l Conf. Ubiquitous Intelligence and Computing (UIC),* pp. 19-34, 2010.

[17] I. Hefez, Y. Kanza, and R. Levin, "TARSIUS: A System for Traffic-Aware Route Search under Conditions of Uncertainty," *Proc. 19th ACM SIGSPATIAL Int'l Conf. Advances in Geographic Information Systems (GIS),* pp. 517-520, 2011.

[18] N. Christofides, "Worst-Case Analysis of a New Heuristic for the Traveling Salesman Problem," Technical Report 388, Graduate School of Industrial Administration, Carnegie-Mellon Univ., 1976.

[19] G. Laporte, "The Traveling Salesman Problem: An Overview of Exact and Approximate Algorithms," *European J. Operational Research,* vol. 59, no. 2, pp. 231-247, June 1992.

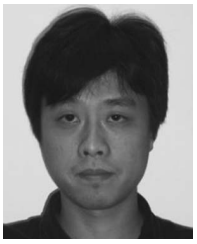
[20] R. Levin, Y. Kanza, E. Safra, and Y. Sagiv, "Interactive Route Search in the Presence of Order Constraints," *Proc. VLDB Endowment,* vol. 3, no. 1, pp. 117-128, 2010.

[21] W. Souffriau, P. Vansteenwegen, G.V. Berghe, and D.V. Oudheusden, "A Path Relinking Approach for the Team Orienteering Problem," *Computers and Operations Research,* vol. 37, pp. 1853-1859, 2010.

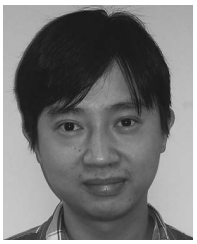
[22] M.V.S.P. de Aragao, H. Viana, and E. Uchoa, "The Team Orienteering Problem: Formulations and Branch-Cut and Price," *Proc. Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS),* vol. 14, pp. 142-155, 2010.

[23] F. Chierichetti, R. Kumar, and A. Tomkins, "Max-Cover in Map-Reduce," *Proc. 19th Int'l Conf. World Wide Web (WWW),* pp. 231-240, 2010.

[24] Z. Zhao, G. Wang, A.R. Butt, M. Khan, V.A. Kumar, and M.V. Marathe, "SAHAD: Subgraph Analysis in Massive Networks Using Hadoop," *IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS),* 2012.



Gang Chen received the BSc, MSc, and PhD degrees in computer science and engineering from Zhejiang University in 1993, 1995, and 1998, respectively. He is currently a professor at the College of Computer Science, Zhejiang University. He is also the executive director of Zhejiang University—Netease Joint Lab on Internet Technology. His research interests include database, information retrieval, information security, and computer supported cooperative work.



Sai Wu received the bachelor's and master's degrees from Peking University, and the PhD degree from the National University of Singapore in 2011. Now he is an assistant professor at the College of Computer Science, Zhejiang University. His research interests include P2P systems, distributed database, cloud systems, and indexing techniques. He has served as a program committee member for VLDB, ICDE, and CIKM.

Jingbo Zhou is currently working toward the PhD degree in the School of Computing, National University of Singapore. His research interests include indexing and query processing on the complex structure, such as trajectories, trees and graphs.

Anthony K.H. Tung received the BSc (second class honor) and MSc degrees in computer science from the National University of Singapore (NUS), in 1997 and 1998, respectively, and the PhD degree in computer sciences from Simon Fraser University in 2001. He is currently an associate professor in the Department of Computer Science (NUS). His research interests include various aspects of databases and data mining (KDD) including buffer management, frequent pattern discovery, spatial clustering, outlier detection, and classification analysis.