Interactive Itinerary Planning

Senjuti Basu Roy[‡], Gautam Das[‡], Sihem Amer-Yahia[†], Cong Yu[††]
[‡]Univ. of Texas Arlington, senjuti.basuroy@mavs.uta.edu, gdas@uta.edu,
[†]Yahoo! Labs, sihem@yahoo-inc.com, [††]Google Research, congyu@google.com

***Abstract*—[1] Planning an itinerary when traveling to a city involves substantial effort in choosing Points-of-Interest (POIs), deciding in which order to visit them, and accounting for the time it takes to visit each POI and transit between them. Several online services address different aspects of itinerary planning but none of them provides an interactive interface where users give feedbacks and iteratively construct their itineraries based on personal interests and time budget. In this paper, we formalize *interactive itinerary planning* as an iterative process where, at each step: (1) the user provides feedback on POIs selected by the system, (2) the system recommends the best itineraries based on all feedback so far, and (3) the system further selects a new set of POIs, with optimal utility, to solicit feedback for, at the next step. This iterative process stops when the user is satisfied with the recommended itinerary. We show that computing an itinerary is NP-complete even for simple itinerary scoring functions, and that POI selection is NP-complete. We develop heuristics and optimizations for a specfic case where the score of an itinerary is proportional to the number of desired POIs it contains. Our extensive experiments show that our algorithms are efficient and return high quality itineraries.**

I. Introduction

Planning an itinerary is one of the most time-consuming travel preparation activities. For a popular touristic city, it involves painstakingly examining the hundreds of Points-of-Interest (POIs) to select the POIs that one likes, figuring out the order in which they are to be visited, and ensuring the time it takes to visit them, and to transit from one POI to the next, satisfies the user's time budget. Many online services such as Lonely Planet provide packaged itineraries to their users. However, those itineraries suffer from two main drawbacks. First, they are often not tailored to one's own interests. For example, a first-time NYC tourist is likely to be interested in a trip to the Statue of Liberty, while a NYC regular may prefer to check out the latest MoMA exhibit. Second, suggested itineraries may not fit one's particular time budget. Someone who visits a place for a very short time frame, e.g, in the case of a layover in a city, or a very long time frame, e.g., in the case of a month-long backpacking trip, is unlikely to find an itinerary suggested by those services, satisfactory.

Constructing a personalized itinerary for a user is a big challenge because, even with a relatively small number of POIs, the number of possible itineraries can be combinatorially large. In this paper, we adopt an interactive process where the user provides feedback on POIs suggested by our itinerary planning system and the system leverages those feedback to suggest the next batch of POIs, as well as to recommend the best itineraries so far. The process repeats until the user is satisfied. In other words, instead of asking the user to examine all the POIs before deciding on the itinerary, our goal is to ask the user to examine only a subset of those POIs in multiple steps, each with a small number of increasingly relevant POIs, thereby reducing the overall efforts required on the user to construct the itinerary. To the best of our knowledge, this work is the first to address the question of formalizing interactive itinerary planning and explore efficient solutions to this problem.

More specifically, the itinerary planning process involves the following interactions.

1) It starts with a user providing a time budget and a starting point of the itinerary (usually corresponding to the hotel where the user is staying);
2) At each step, the system presents the user with a small fixed number of POIs that are *most probably liked* by the user, based on feedback provided by the user so far;
3) The system also recommends *highly ranked itineraries* to the user based on the feedback;
4) The user provides her *feedback* on suggested POIs to indicate whether or not she is interested in them, and the process continues;
5) The user can also choose to pick one of the recommended itineraries, at which point, the process stops.

Designing such an interactive system is a non-trivial task and raises both semantics and efficiency challenges. We provide a brief overview of those challenges here.

First, we need to define the **POI Feedback Model**, which dictates how the user can specify her preference for the individual POIs. The most generic model is the *star model* where the user provides 5-star ratings for POIs she really wants to visit and 1-star ratings for POIs she does not want to see. Two simpler models are also common: the *ternary model*, where the user specifies 'yes' (i.e., positive), 'do_not_care', and 'no' (i.e., negative) for the POIs, and the *binary model*, where the user is provided with only two feedback options 'yes' and 'do_not_care'. We note that the star model can often be converted into the ternary model. We will discuss the impact of different feedback models on the complexity of itinerary planning, and focus on the binary model within this work.

Second, we need to define the **Itinerary Scoring Semantics**, which dictates how an itinerary should be scored based on the user feedback. Similarly, it can also be defined using

[1]The work of Senjuti Basu Roy and Gautam Das was supported in part by the US National Science Foundation under grants 0916277, 0845644 and 0812601, a grant from the Department of Education, and unrestricted gifts from Microsoft Research and Nokia Research

multiple semantics. In the *set semantics*, the score of an itinerary positively correlates with the number of POIs with a 'yes' feedback and negatively correlates with the number of POIs with a 'no' feedback. In the strictest interpretation, a single POI with a 'no' feedback can render the entire itinerary ineligible. In the *chain* semantics, the score of an itinerary will further depend on how the positive and negative POIs are arranged in the itinerary. One such semantics could be to rank itineraries containing consecutive POIs marked with a 'yes' higher than ones containing more POIs marked with a 'yes' none of which being consecutive. Finally, an itinerary is only *valid* if it satisfies the budget constraint specified by the user. We focus on time budget in this paper and defer other kinds of budget for future work. We argue that during the interactive itinerary building process, previous user feedback has a direct impact on the score of a new itinerary. For example, when *Times Sq.* has been marked 'yes' by the user in previous steps, the score of an itinerary containing *Times Sq.* and *Madame Tussauds Wax Museum*, should increase, because those two POIs are frequently co-visited. In this work, we use a probabilistic model to compute the *expected score* of valid itineraries given user feedback using the set semantics. We leave the chain semantics to future work.

Third, we need to efficiently solve the **Optimal Itinerary Construction Problem**, i.e., how to construct the best scoring itinerary based on a given set of POIs, along with their feedback, and the user provided time budget. We argue that materialization of all itineraries is not practically feasible and design efficient algorithms for computing itineraries with the best expected scores *on the fly*.

Finally, we need to efficiently solve the **Optimal POI Batch Selection Problem**, i.e., how to select a fixed number of POIs to solicit future user feedback based on the feedback received so far. We argue that the best candidate POIs (to be suggested to the user next) are those which maximize the *expected scores* of the best itineraries. Any user feedback for those POIs is likely to lead to itineraries with high expected scores, and therefore satisfy user's needs sooner. We provide a formal definition of this problem and propose a probabilistic model to compute the expected score of a batch of POIs. There are two main efficiency challenges. First, selecting the optimal batch of k POIs according to the expected itinerary scores requires the system to go through all $m_{\mathcal{C}_k}$ sets of POIs, where m is the number of remaining POIs in the system, which can be large. We design a heuristic algorithm that selects POIs one by one to form partial batches, therefore significantly reducing the candidate POI sets to be examined. Second, the number of remaining POIs to be checked for each partial batch can still be large. In order to reduce that number, we design an efficient pruning strategy which accounts for the distance of the remaining POIs from the starting point and from POIs already in the batch.

Table I summarizes an example of a 2-step interactive itinerary planning for a user, whose starting location is *Ground Zero, NYC* and has a budget of 6 hours. At each step, the suggested batch of 5-POIs (column-2) is shown, the POIs for which user feedback is 'yes' (column-3), and the resulting top-1 itinerary based on her feedback (column-4) are also displayed. Note that, top-1 itinerary of step-2 considers both step-1 and step-2 feedback.

Step	POI batch	'yes' feedback	Top-1 Itinerary
1	*Trinity Church.*; *Brooklyn Bridge*; *NYC Stock Ex*; *Battery Park* ; *Statue of Liberty*	*Trinity Church.*; *NYC Stock Ex*; *Battery Park*	**1.** *Ground Zero - Trinity Church - NYC Stock Ex - Battery Park*
2	*Times Square* ; *Grand Central Terminal* ; *Chrysler Building* ; *UN Head Quarter* ; *Rockefeller Center*	*Times Square* ; *Grand Central Terminal*	**1.** *Ground Zero - Trinity Church - NYC Stock Ex - Battery Park - Times Square - Grand Central Terminal*

TABLE I
3-STEP ITERATIVE ITINERARY PLANNING

In summary, we make the following contributions.

(1) We introduce and formalize the novel approach of interactive itinerary planning based on user feedback and itinerary expected scores.

(2) We formally define the *optimal itinerary construction problem*, which is one of the two core problems in interactive itinerary planning. We prove NP-completeness of this problem and design an efficient real-time heuristic algorithm for computing itineraries based on user feedback and time budget.

(3) We formally define the *optimal POI batch selection problem*, which is the other core problem, and propose a probabilistic model based on the notion of expected itinerary score given user feedback on a POI batch. We prove NP-completeness of this problem and design efficient heuristics for selecting a good batch of POIs.

Finally, we run extensive experiments validating our approach on real datasets. Quality experiments confirm the effectiveness of our algorithms for interactive itinerary planning and performance experiments demonstrate their efficiency.

The paper is organized as follows. Section II contains a formalization of the interactive itinerary planning approach. Section III describes the algorithms. Our experiments are reported in Section V. The related work is summarized in Section VI. We conclude with future directions in Section VII.

II. FORMALISM AND PROBLEM STATEMENT

In this section, we discuss the formal data model of interactive itinerary planning. We begin by describing different notations and their corresponding interpretations to be used throughout the paper. A summary of those notations is listed in Table II for easy reference.

Data Model: The underlying data model is a directed *complete graph* $G = (\mathcal{M}, E)$. Each node $m \in \mathcal{M}$ represents a POI and each edge (m_i, m_j) in E represents a transit between the two nodes and is annotated with an edge cost $\texttt{transit}(m_i, m_j)$. The edge cost is not always symmetric. For example, traveling time between two POIs can be different because it is downhill in one direction and uphill in another. Each POI m_i is also annotated with $\texttt{visit}(m_i)$, which represents the cost

associated with visiting the POI. For example, it takes about 3 hours to visit the *Statue of Liberty.*

Itinerary: An itinerary is a path in the input graph starting from the start POI. Each itinerary τ has a total visit time $\texttt{totalVisit}(\tau) = \Sigma_{m_i \in \tau} \texttt{visit}(m_i)$, and a total transit time, $\texttt{totalTransit}(\tau) = \Sigma_{(m_i,m_j) \in \tau} \texttt{transit}(m_i, m_j)$. A *valid* itinerary is one such that $\texttt{totalVisit}(\tau) + \texttt{totalTransit}(\tau) \leq \mathcal{B}$, where $\mathcal{B}$ is a user provided budget constraint.

Notation	Interpretation
$\mathcal{M}$	set of all POIs in a city
$\mathcal{M}_{seen}$	set of POIs for which feedback has been received
$\mathcal{M}_{remain}$	$= \mathcal{M} - \mathcal{M}_{seen}$
$\texttt{transit}(m_i, m_j)$	transit time from POI m_i to m_j
$\texttt{visit}(m_i)$	time to visit POI m_i
$FeedbackOptions$	set of different feedback values a user can assign a POI (e.g., $\{`yes', `no', `do\_not\_care'\}$)
n	number of feedback options
$\langle id, feedback \rangle$	a POI as an ordered pair of id and feedback option
$\mathcal{I}$	a POI batch
k	number of POIs in a batch
$\mathcal{B}$	total budget
$AllFeedbacks(\mathcal{I})$	$= \{\mathcal{I}_1, \ldots, \mathcal{I}_{n^k}\}$, i.e., set of all possible feedback combinations of $\mathcal{I}$
$\mathcal{I}_j$	$= \{< id_1, feedback_1^j >, \ldots, < id_k, feedback_k^j >\}$, i.e., j-th feedback combination for the POI batch $\mathcal{I}$
τ	an itinerary, expressed as a sequence of POIs
$\tau_{\mathcal{I}_j}$	best itinerary corresponding to j-th feedback combination for the POI batch $\mathcal{I}$
$S_{\mathcal{I}_j}$	score of the best itinerary, given $\mathcal{I}_j$, $\mathcal{B}$ and $\mathcal{M}_{seen}$
$ExpScore(\tau \vert \mathcal{M}_{seen})$	expected score of itinerary, given feedback $\mathcal{M}_{seen}$
$ExpBatchScore(\mathcal{I} \vert \mathcal{M}_{seen})$	expected value (over all $\mathcal{I}_j$) of $S_{\mathcal{I}_j}$

TABLE II
NOTATIONS AND THEIR CORRESPONDING INTERPRETATIONS

A. System Overview

The input to the system is the graph G that obeys metric properties, a budget $\mathcal{B}$ (e.g., the user has 8 hours to spend in the city), and a starting POI (e.g., an airport or a hotel). The task of the system is to interact with the user and gather her preferences, and build the best possible itinerary for her via this iterative feedback process. In each iteration, the system suggests a batch of k POIs to the user, and the user provides feedback on these POIs, i.e., her preference for including them in her itinerary. Based on the feedbacks, the valid itineraries are re-ranked according to the *scoring semantics* and the top itineraries are suggested to the user. Since G is complete, therefore the POIs that the user has preferred to have included in the itinerary can always be connected with each other with direct edges based on their shortest transit time paths subject to the budget constraints, and does not need to involve any POI that she has not chosen. At each step, the user is shown the next batch of POI suggestions from the system. This interactive process ends when the user is satisfied with the top itineraries suggested by the system and decides not to proceed with the next batch.

Two computational problems form the heart of the system. The first is the *Optimal POI Batch Selection Problem*, where the system has to determine at every iteration the next batch of k POIs to be shown to the user. Once these POIs have been presented and user feedback collected and updated, the system then has to solve the *Optimal Itinerary Construction Problem*, which re-ranks all itineraries and presents the top-ranked ones to the user. In fact, the *Optimal POI Batch Selection Problem* also requires solutions to multiple instances of the *Optimal Itinerary Construction Problem*, as each candidate set of k POIs have to be considered and top itineraries have to be computed for each possible combination of user feedback. In the rest of this section we develop formal notations and definitions of both problems. We begin by describing the feedback models that we consider.

POI Feedback Model: When one (or more) POIs are shown to the user, the user expresses her preference for them according to a specific feedback model. Let *FeedbackOptions* be the set of different ways in which a user can show her preference to a POI. As an example, for the *ternary feedback model*, $FeedbackOptions = \{`yes', `no', `do\_not\_care'\}$. A simpler model *binary feedback model* has the options $\{`yes', `do\_not\_care'\}$. An alternate binary feedback model may have the options $\{`yes', `no'\}$.

Interestingly, since in this paper we consider recommending itineraries only for a *single user*, the specific feedback model is irrelevant. We only need to be concerned with the POIs marked as $`yes'$ by the user, as the POIs marked as $`no'$ or $`do\_not\_care'$ are never considered by the recommendation algorithm. This is because the underlying graph is a complete graph, and the recommended itinerary should try to visit as many $`yes'$ POIs as the budget allows, and will never need to visit any a POI marked as 'no' or 'do_not_care'. The different feedback models only differ in their "user friendliness", and do not impact the underlying solution. [2]

In our system, a POI may be regarded as an ordered pair $\langle id, feedback \rangle$, where id identifies the POI (e.g., '*Statue of Liberty*). Initially each POI's feedback is set to the value $`unseen'$, and, after the POI is seen by the user, is set to a value from $FeedbackOptions$. At any stage during the interactive process, let $\mathcal{M}_{seen}$ (respectively, $\mathcal{M}_{remain}$) be the set of POIs that have currently been seen (respectively, remain to be seen) by the user; thus initially $\mathcal{M} = \mathcal{M}_{remain}$. At every step of the iteration, the system selects a batch $\mathcal{I}$ of k POIs from $\mathcal{M}_{remain}$ and shows them to the user. The user provides feedback for each POI in $\mathcal{I}$ indicating her preference for including the POIs in the output itinerary. Let $n = |FeedbackOptions|$. We note that there can be n^k

[2]However, if an itinerary has to be shared by a group of users (e.g., a set of people sharing a tour bus), then a POI marked as 'no' by some users may be marked as 'yes' by other users, and the recommendation algorithm will have to carefully consider the impact of visiting a POI with conflicting preferences by the user group. Recommending itineraries for user groups is left for future work.

feedback combinations, each of which represents a possible user feedback for POIs in $\mathcal{I}$. The following notation will be convenient: $AllFeedbacks(\mathcal{I}) = \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_{n^k}\}$, where each $\mathcal{I}_j$ represents a specific combination of feedback by the user for each POI. Thus for the ternary model there are 3^k feedback combinations, whereas the simpler binary model leads to 2^k feedback combinations.

B. Probability Model

For any candidate set $\mathcal{I}$ of k POIs considered during an iteration, it is crucial that the system be able to derive the probability distribution of these n^k feedback combinations. Such a probability distribution will be useful in steering the system towards choosing the subset $\mathcal{I}$ that maximizes the chances of getting highly ranked itineraries. We adopt probabilistic models that are intended to combine users' general preferences (e.g., statistics derived from past query logs may reveal that most users who wish to visit the *Status of Liberty* would also like to visit the *Empire State Building*) with personalization (e.g., the specific feedback obtained from the user on previous batches of POIs may reveal that this particular user prefers art related places). We describe our models in more details below.

Generic Probability Model: A generic probability model can be used to compute the probability of j-th feedback combination: $Pr(\mathcal{I}_j|\mathcal{M}_{seen})$. This probability model can be learned from two training sources: the past activities (e.g., past itineraries accepted by other users of the system), and current ongoing activities (i.e., the POIs that have been seen and marked by the current user). Several classical machine learning solutions can be used for this purpose, e.g., graphical models such as Bayesian Networks or Markov Random fields [1].

Specific Probability Model: In this paper, however, instead of relying on complex solutions involving a generic probability model, we adopt a much simpler probability model using the assumption of a limited form of *conditional independence*[3]. We assume the POIs in $\mathcal{I}_j$ are not *totally independent* but rather are *conditionally independent*.

Under the conditional independence assumption, we have:

$$Pr(\mathcal{I}_j|\mathcal{M}_{seen}) = \prod_{m_i \in \mathcal{I}_j} Pr(m_i|\mathcal{M}_{seen})$$

Using Bayes' Theorem [2], this can be rewritten as:

$$Pr(\mathcal{I}_j|\mathcal{M}_{seen}) = \prod_{m_i \in \mathcal{I}_j} \frac{Pr(\mathcal{M}_{seen}|m_i) \times Pr(m_i)}{Pr(\mathcal{M}_{seen})}$$

Since $Pr(\mathcal{M}_{seen})$ is a constant for that particular iteration, we therefore have:

$$Pr(\mathcal{I}_j|\mathcal{M}_{seen}) \propto \prod_{m_i \in \mathcal{I}_j} Pr(\mathcal{M}_{seen}|m_i) \times Pr(m_i)$$

Applying conditional independence again:

$$Pr(\mathcal{I}_j|\mathcal{M}_{seen}) \propto \prod_{m_i \in \mathcal{I}_j} \prod_{m_l \in \mathcal{M}_{seen}} Pr(m_l|m_i) \times Pr(m_i)$$

Even though the probability formula is a proportionality formula, it suffices for our purpose as it is used in the scoring function for ranking itineraries, since all we need to know is whether one itinerary has a higher score than the other—the exact score is irrelevant. Computing the probability formula requires us to know the value of quantities such as $Pr(m_l|m_i)$ and $Pr(m_i)$ where m_i and m_l are POIs. However, singleton and pairwise probabilities can be computed in a preprocessing step from itineraries chosen by previous users. For example, $Pr(m_l|m_i)$ can be estimated as the fraction of previous itineraries containing m_i that also contain m_l, and $Pr(m_i)$ can be estimated as the fraction of itineraries that contain m_i.

[3]Conditional independence assumption is used in building Naive Bayes classifiers [2]

C. Itinerary Scoring Semantics

An itinerary consists of two sets of POIs: the seen POIs for which user feedback has already been collected, and the remaining POIs for which we can only estimate the user feedback. Thus the score of an itinerary should be a combination of the score of the seen part, as well as the expected score of the remaining part, where the expectation is over the probability distribution of all possible user feedback. The probability model proposed earlier can be used to model the expected score of the unseen part.

Generic Scoring Function: Consider an itinerary τ as $\tau_{seen} \cup \tau_{remain}$. A generic scoring function has the form:

$$ExpScore(\tau|\mathcal{M}_{seen}) = Combine(Score(\tau_{seen}), ExpScore(\tau_{remain}|\mathcal{M}_{seen}))$$

where the two parts may be combined using any meaningful operation (such as addition, weighted or un-weighted). There can be numerous ways of defining reasonable forms of the function $Score(\tau_{seen})$. For example, a reasonable function is positively correlated with the number of 'yes' POIs, or a sophisticated scoring function may even consider the sequence of the 'yes' POIs in the overall itinerary score.

Specific Scoring Function: While we do not advocate for a specific scoring function in this paper, we illustrate several optimization opportunities in conjunction with a specific scoring function in Section IV. This scoring function is related to the binary feedback model, and has a simple but compelling form—the score of an itinerary is the expected number of POIs that will be marked as 'yes' by the user.

D. Problem Definitions

We are now ready to describe the two fundamental problems that our system needs to solve.

Optimal Itinerary Construction Problem: Given $\mathcal{B}$, $\mathcal{M}_{seen}$, and $\mathcal{I}_j$ (i.e., a specific batch of k POIs with their feedbacks from the user), compute the valid itinerary τ such that $ExpScore(\tau|\mathcal{M}_{seen} \cup \mathcal{I}_j)$ is maximized.

We next introduce some useful notation. Let $\tau_{\mathcal{I}_j}$ be the output of the *Optimal Itinerary Construction Problem*, i.e., the valid itinerary with the maximum expected score, and let its expected score be $S_{\mathcal{I}_j}$. Next, given $\mathcal{B}$, $\mathcal{M}_{seen}$, and a batch of k unseen POIs $\mathcal{I}$ (i.e., without any specific user feedback combination), let $ExpBatchScore(\mathcal{I}|\mathcal{M}_{seen})$ be the expected value (over all possible user feedback combinations $\mathcal{I}_j$) of the random variable $S_{\mathcal{I}_j}$.

Optimal POI Batch Selection Problem: Given $\mathcal{B}$ and $\mathcal{M}_{seen}$, compute the batch of k unseen POIs that maximizes $ExpBatchScore(\mathcal{I}|\mathcal{M}_{seen})$.

Intuitively, we wish to select a batch of k unseen POIs such that, no matter how the user responds with her preferences to these POIs, the expected score of the top ranked itinerary over all possible user feedback is maximized.

As will be discussed in the next sections, the choice of the itinerary scoring function as well as the probability model affects the efficiency of our solutions to these problems. We discuss a general solution framework for these problems in Section III, and more efficient solutions tailored to a specific scoring function and the simpler probability model in Section IV. Our solutions are designed to solve one iteration step in the interactive itinerary planning problem.

III. General Algorithms for Itinerary Planning

In this section we shall develop the framework of a generic algorithm for solving the *Optimal POI Batch Selection Problem*. We refer to this as a "generic" algorithm because it is essentially a framework that assumes any arbitrary scoring function for itineraries, as well as any arbitrary probabilistic model for predicting user preferences for the remaining unseen POIs, given the current user feedback. We also develop a generic subroutine to solve the *Optimal Itinerary Construction Problem*. We analyze the computational complexity of the problems as well as the proposed algorithms. In the next section, we show how a specific probabilistic model (based on conditional independence), as well as a specific scoring function (based on user feedback restricted to only 'yes' and 'do_not_care' for POIs), can be leveraged, along with several algorithmic optimizations, to achieve extremely efficient approximate solutions to these problems.

A. A Generic Optimal POI Batch Selection Algorithm

Our generic algorithm for the *Optimal POI Batch Selection Problem* is shown in Algorithm 1. As can be seen, the main body consists of generating all possible k-sized batches of potential POIs from the remaining unseen POIs, and for each potential batch, computing the expected score of the optimal itinerary—where the expectation is over the probability distribution of all possible user feedback to those k POIs. This calculation is performed by the `ExpBatchScore` subroutine (which will be discussed next). The set of k POIs selected are those that maximize this expected optimal score.

Algorithm 1 Algorithm OptPOIBatchSelection

Require: $\mathcal{M}_{seen}$, $\mathcal{M}_{remain}$, batch size k, budget $\mathcal{B}$;
1: $RS = \{\mathcal{I} \mid \mathcal{I} \subseteq \mathcal{M}_{remain}, |\mathcal{I}| = k\}$;
2: $\mathcal{I}_{max} = argmax_{\forall \mathcal{I} \in RS} ExpBatchScore(\mathcal{I}|\mathcal{M}_{seen}, \mathcal{B})$;
3: **return** $\mathcal{I}_{max}$;

We next discuss the `ExpBatchScore` subroutine as described in Algorithm 2, which computes the expected score of the top itinerary given the POI batch ($\mathcal{I}$), conditioned upon the feedback of the seen POIs ($\mathcal{M}_{seen}$). For each of the n^k possible user feedback combinations $\mathcal{I}_j$, we need to recompute the scores of all valid itineraries, and determine the one with the highest score. This is achieved by repeated calls to the `OptItn` subroutine (which will be discussed next). Finally, the expected value of the score of the optimal itinerary is returned (where the expectation is computed over the probability distribution of the user feedback $\mathcal{I}_j$).

Algorithm 2 Subroutine ExpBatchScore

Require: $\mathcal{M}_{seen}$, $\mathcal{I} \subseteq \mathcal{M}_{remain}$, budget $\mathcal{B}$;
1: $AllFeedbacks(\mathcal{I}) = \{\mathcal{I}_1, \mathcal{I}_2, \ldots \mathcal{I}_{n^k}\}$;
2: # each $\mathcal{I}_j$ is a possible feedback combination on $\mathcal{I}$
3: ExpBatchScore = $Pr(\mathcal{I}_j|\mathcal{M}_{seen}) \times$
$\Sigma_{1 \le j \le n^k} ExpScore(OptIt(\mathcal{M}_{seen}, \mathcal{I}_j, \mathcal{B})|\mathcal{M}_{seen})$;
4: **return** ExpBatchScore;

Algorithm 3 Subroutine OptItn

Require: $\mathcal{M}_{seen}$, $\mathcal{I}_j$, budget $\mathcal{B}$;
1: $\mathcal{T} = \{\tau \mid \texttt{totalVisit}(\tau) + \texttt{totalTransit}(\tau) \le \mathcal{B}\}$, where τ is an itinerary
2: $\tau_{max} = argmax_{\tau \in \mathcal{T}} ExpScore(\tau|\mathcal{M}_{seen} \cup \mathcal{I}_j)$;
3: **return** τ_{max};

The `OptItn` subroutine solves the *Optimal Itinerary Construction Problem*. It takes as input the user feedbacks from previous batches ($\mathcal{M}_{seen}$, along with a candidate user feedback combination $\mathcal{I}_j$), and computes the valid itinerary with the highest expected score. As can be seen from Algorithm 3, one straightforward (but inefficient) way of doing this is to first compute all valid itineraries, compute the expected scores of each of them (conditioned by the user feedback in previous batches and candidate user feedback combination), and return the one with the highest score.

In summary, the general algorithms discussed above do appear rather inefficient. However, in what follows, we show that the problems are NP-complete in general, and one may not be able to improve over such naive approaches in the generic case. To improve efficiency, one has to resort to specific scoring functions, approximation heuristics, and other optimizations—such approaches are discussed in Section IV.

B. Complexity Analysis

The generic *Optimal POI Batch Selection* algorithm described above is very inefficient. The inefficiency stems from three sources:

1) There are $\binom{|\mathcal{M}_{remain}|}{k} = O(|\mathcal{M}_{remain}|^k)$ possible batches of k POIs that need to be considered.
2) For a given batch $\mathcal{I}$, all possible n^k user feedback need to be considered.
3) For a given user feedback (i.e., a potential user feedback for a given batch, in conjunction with the user feedback for earlier batches), the itinerary with the highest expected score needs to be computed.

Thus, if we assume that the cost of a single optimal itinerary computation is T, then the total time taken by the `OptPOIBatchSelection` algorithm is $O(|\mathcal{M}_{remain}|^k \times n^k \times T)$. Unfortunately, as the following arguments show, it appears impossible to improve this in general, as even the third task in the list above, i.e., the problem of computing

the itinerary with the optimal expected score for a given user feedback (essentially the Optimal Itinerary Construction Problem), is NP-complete.

Theorem 1: The *Optimal Itinerary Construction Problem* is NP-complete.

Proof: (sketch) We can reduce the NP-complete *Rooted Orienteering Problem* [3] to this problem. The rooted orienteering problem is defined as follows: Given a complete weighted graph (in a metric sense, i.e., satisfying the triangle inequality), a start node, and a length budget, determine a path from the start node that visits as many nodes as possible without going over the length budget.

The reduction proceeds as follows. Consider a very simple scenario where:

- the original POIs are connected by a complete weighted graph where each edge weight represents the transit time to go from one vertex to the other along the edge,
- the visit times of all POIs are 0,
- there is no prior probability model: thus all possible user feedback for the next batch are equally likely,
- the user feedback is restricted to 'yes'/'do_not_care' for each POI that is shown to her,
- the score of a valid itinerary is simply the number of POIs that have been marked as 'yes' by the user in her feedback, and
- we are considering the very first batch, i.e. user feedback has not been collected for any POI yet.

Let $\mathcal{I}$ be any subset of k POIs. Let $\mathcal{I}'$ be any subset of $\mathcal{I}$, representing a specific subset of the batch that the user may potentially mark as 'yes'. Consider the induced complete subgraph graph over $\mathcal{I}'$. Let this induced graph be isomorphic to the input graph of the rooted orienteering problem. It is easy to see that computing the valid itinerary with the highest score is equivalent to solving the rooted orienteering problem whose length does not exceed the budget. ■

The above theorem shows that computing itinerary with the optimal expected score is NP-complete even for a simple scoring function. Moreover, since the *Optimal POI Batch Selection Problem* is more general than the *Optimal Itinerary Construction Problem*, the former is also easily seen to be NP-complete. Also, as can be seen, the `OptItn` subroutine is called inside the innermost loop of the overall `OptPOIBatchSelection` algorithm, and is therefore called numerous times, making the overall algorithm extremely inefficient. In the next section, we consider several ways to avoid these sources of intractability.

IV. Efficient Algorithms for Itinerary Planning

In this section we discuss more efficient solutions to the itinerary planning problems, by focusing on the simple but practice scoring function (discussed in Section II) based on the binary feedback model, and the simple probabilistic model for scoring itineraries based on the assumption of conditional independence. Our solutions are based on fast heuristics to compute optimal itineraries approximately, thus overcoming the intractability of `OptItn`. We also assume that the batch size k is reasonably small (which is true in practice as the value of k is limited by the screen size used to display the selected POIs to the end user), thus making the n^k factor in the running time of `ExpBatchScore` small. We also use a greedy approach to construct the k POIs, thus eliminating having to examine all $|\mathcal{M}_{remain}|^k$ subsets of POIs. Finally we develop several other algorithmic and data structure optimizations to achieve very efficient overall performance of `OptPOIBatchSelection` in practice. In the rest of this section we provide more details of our techniques.

A. Efficient Approximation Algorithm for POI Batch Selection

One of the main bottleneck in the `OptPOIBatchSelection` algorithm is that a large number of candidate POI batches need to be considered and the best one chosen from among them. Instead, we follow a greedy approach where we construct a POI batch one POI at a time, thus trading off batch quality (i.e., $ExpBatchScore(\mathcal{I}|\mathcal{M}_{seen})$) for efficiency, with the hope that small quality degradation can bring in huge performance improvements.

Consider the algorithm `GreedyPOIBatchSelection` shown in Algorithm 4. The first step is to prune from consideration those POIs in $\mathcal{M}_{remain}$ that are simply too far away from the start POI to be involved in valid itineraries. For tight budgets, this can be a very effective step in practice. Next, the batch of k POIs are constructed greedily in k iterations. In each iteration i, each of the remaining POIs in $\mathcal{M}_{pruned}$ are considered as candidate for adding to the batch, and the one that creates a batch with i POIs with the maximum batch score is selected for inclusion in the batch.

Thus, unlike the `OptPOIBatchSelection` algorithm in the previous section which makes $O(|\mathcal{M}_{remain}|^k)$ calls to subroutine `ExpBatchScore` (which evaluates each candidate batch), the new `GreedyPOIBatchSelection` only makes at most $O(|\mathcal{M}_{remain}| \times k)$ calls to subroutine `FastExpBatchScore` (which itself is a more efficient subroutine than the earlier `ExpBatchScore` subroutine for evaluating each candidate batch, to be discussed later). Since the value of k is small is practice, the number of calls to `FastExpBatchScore` is acceptably small.

B. Efficient Computation of a Batch Score

We next discuss the subroutine `FastExpBatchScore` which is shown in Algorithm 5. This subroutine takes as input a candidate batch, and evaluates its "expected score", i.e., for the distribution of all possible user feedback for the candidate batch, the expected score of the optimal itinerary according to the specific scoring function being used. The structure of this subroutine is very similar to that of the corresponding subroutine `ExpBatchScore` in Section III, because it also enumerates all possible user feedback combinations to the candidate batch, and makes a total of $n^{|\mathcal{I}|}$ calls to another subroutine to determine the optimal itineraries for each possible user feedback combination (this subroutine, called `ApproxItn`, will be discussed later). Since we assume

Algorithm 4 Algorithm GreedyPOIBatchSelection

Require: $\mathcal{M}_{seen}$ consisting of 'yes' and 'do_not_care' feedback, $\mathcal{M}_{remain}$, batch size k, budget $\mathcal{B}$;

1: $\mathcal{M}_{pruned} = \{m | m \in \mathcal{M}_{remain}, \texttt{transit}(StartPOI, m) + \texttt{visit}(m) \leq \mathcal{B}\}$; {prune $\mathcal{M}_{remain}$ by removing POIs that are very far away from the start POI}
2: $\mathcal{I}_{max} = \{\}$;
3: $i = 0$;
{construct POI batch greedily by adding POIs one by one to initially empty batch}
4: **while** $i \neq k$ **do**
5: $m = argmax_{m_j \in \mathcal{M}_{pruned}} FastExpBatchScore(\mathcal{I}_{max} \cup \{m_j\} | \mathcal{M}_{seen})$;
6: $\mathcal{I}_{max} = \mathcal{I}_{max} \cup \{m\}$;
7: $\mathcal{M}_{pruned} = \mathcal{M}_{pruned} - \{m\}$;
8: i++;
9: **end while**
10: **return** $\mathcal{I}_{max}$;

that k is small, and $|\mathcal{I}| \leq k$, the total number of user feedback combinations will be reasonably small.

Algorithm 5 Subroutine FastExpBatchScore

Require: $\mathcal{M}_{seen}$ consisting of 'yes' and 'do_not_care' feedbacks, a set $\mathcal{I}$ of $\leq k$ POIs from $\mathcal{M}_{remain}$;

1: $AllFeedbacks(\mathcal{I}) = \{\mathcal{I}_1, \mathcal{I}_2, \ldots \mathcal{I}_{2^{|\mathcal{I}|}}\}$
{above sequence should correspond to Hamiltonian path in $\mathcal{I}$-dim hypercube}
2: FastExpBatchScore = $\Sigma_{1 \leq j \leq 2^{|\mathcal{I}|}} (Pr(\mathcal{I}_j | \mathcal{M}_{seen}) \times ExpScore(ApproxItn(\mathcal{M}_{seen}, \mathcal{I}_j) | \mathcal{M}_{seen}))$; {above calculation should be run in Hamiltonian path sequence to enable incremental computation of ApproxItn and $Pr(\mathcal{I}_j | \mathcal{M}_{seen})$}
3: **return** FastExpBatchScore;

Hamiltonian Paths in Hypercubes: However, there is scope for optimizing `ExpBatchScore` even further. The crucial difference between `FastExpBatchScore` and the earlier generic `ExpBatchScore` is the *order in which* all the user feedback combinations are processed. `ExpBatchScore` processes the user feedback combinations in any arbitrary order, but we observe that certain specific orders can be leveraged to improve overall efficiency. Since we are considering the specific binary feedback model, for a given candidate batch $\mathcal{I}$, there are $2^{|\mathcal{I}|}$ different user feedback combinations. Consider any specific user feedback combination $\mathcal{I}_j$. If we consider $\mathcal{I}$ as an ordered set (in any order) of POIs, then $\mathcal{I}_j$ can be considered as a Boolean vector of length $|\mathcal{I}|$, in which a 1 implies that the corresponding POI has been potentially marked as 'yes', and a 0 implies that the corresponding POI has been marked as a 'do_not_care'. Thus the set of $2^{|\mathcal{I}|}$ user combinations can be viewed as the vertices (i.e., corners) of a $|\mathcal{I}|$-dimensional hypercube.

The subroutine `FastExpBatchScore`'s order for processing all user feedback combinations is as follows: it finds a *Hamiltonian path* in the hypercube, and then processes each user feedback combination in the order in which it appears in this path. For example, consider the 3-dimensional hypercube in Figure 1, where a Hamiltonian path is shown traversing the 8 vertices.

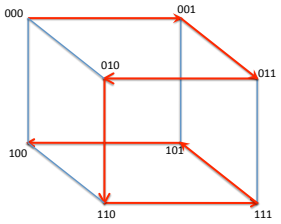

Fig. 1. Hamiltonian paths in hypercubes.

The reason for using a Hamiltonian path for ordering the user feedback combinations is because the Hamming distance between any pair of consecutive vertices on this path is exactly 1, i.e., the corresponding subsets of 'yes' POIs differ by exactly one POI. This has important efficiency implications. For every user feedback combination $\mathcal{I}_j$, the subroutine `FastExpBatchScore` has to perform two computations (see line 2 in Algorithm 5) (a) it has to call a subroutine `ApproxItn`, and (b) it has to compute $Pr(\mathcal{I}_j | \mathcal{M}_{seen})$, i.e., the probability that the user will give this specific feedback combination, given her earlier feedbacks. We defer the details of `ApproxItn` till later. However, $Pr(\mathcal{I}_{j+1} | \mathcal{M}_{seen})$ can be incrementally computed very efficiently from $Pr(\mathcal{I}_j | \mathcal{M}_{seen})$ if they differ by only one 'yes' POI—as can be seen from the specific probability model formula in Section II, $Pr(\mathcal{I}_{j+1} | \mathcal{M}_{seen})$ can be computed from $Pr(\mathcal{I}_j | \mathcal{M}_{seen})$ in $O(|\mathcal{M}_{seen}|)$ time rather than in $O(|\mathcal{M}_{seen}| \times |\mathcal{I}|)$ time, which would be required if $Pr(\mathcal{I}_{j+1} | \mathcal{M}_{seen})$ had to be computed from scratch.

The following lemma shows that all d-dimensional hypercubes have Hamiltonian paths, and moreover they are easy to construct.

Lemma 1: Each d-dimensional hypercube has a Hamiltonian path, and such a path can be computed in $O(2^d)$ time.

Proof: The proof is by induction. Assume that all hypercubes up to dimension d have Hamiltonian paths. Consider a Hamiltonian path $v_1, v_2, \ldots, v_{2^d-1}, v_{2^d}$. Now, consider a $(d+1)$-dimensional hypercube. It is easy to see that the path $0v_1, 0v_2, \ldots, 0v_{2^d-1}, 0v_{2^d}, 1v_{2^d}, 1v_{2^d-1}, \ldots, 1v_2, 1v_1$ is a Hamiltonian path in the $(d+1)$-dimensional hypercube. Figure 1 illustrates this construction for the case $d = 2$, and $d + 1 = 3$. Clearly, this also implies a simple linear time recursive construction of such Hamiltonian paths. ■

What if we did not use a Hamiltonian path ordering? If we use any arbitrary ordering, the changes between successive user feedback combinations may be quite large, thus making probability calculations expensive. For example, suppose we used a random ordering (i.e., a random permutation of all user feedback combinations). Then between successive user feedback combinations in such an ordering, it is easy to see that the expected Hamming distance may be $O(|\mathcal{I}|)$. Thus every time `ApproxItn` is called, the incremental probability computation may take $O(|\mathcal{M}_{seen}| \times |\mathcal{I}|)$ time rather than $O(|\mathcal{M}_{seen}|)$ time if the Hamiltonian path ordering was used.

The Hamiltonian path order is also crucial in the efficient execution of `ApproxItn`, which shall be discussed next.

C. Approximation Algorithm for Itinerary Construction

The `ApproxItn` subroutine solves the *Itinerary Construction Problem* using approximation heuristics. It takes as input a certain set of user inputs marked as 'yes' ($\mathcal{M}_{seen}$, enhanced with a candidate feedback combination $\mathcal{I}_j$), and computes the valid itinerary with the (approximate) highest expected score. Since this problem was shown to be NP-complete in Section III, we use a "Best-Benefit" approximation heuristic to solve this problem approximately.

The subroutine is shown in Algorithm 6, which adopts a greedy approach. Starting from the start POI, at every iteration, the algorithm adds the POI (chosen from the remaining POIs, i.e., those not yet in the partially constructed itinerary) that has the best *benefit*, as defined in line 5. Intuitively, the benefit correlates positively with the probability that the user will mark the POI as 'yes', and negatively with the time needed (transit plus visit) to reach this POI from the last POI added to the itinerary.

Algorithm 6 Subroutine ApproxItn

Require: $\mathcal{M}_{seen}$, and a candidate user feedback combination $\mathcal{I}_j$
1: $\mathcal{M}_{tempseen} = \mathcal{M}_{seen} \cup \mathcal{I}_j$
2: $\tau_{max} = StartPOI$
3: $RemainB = \mathcal{B} - \texttt{visit}(StartPOI)$
{Construct itinerary greedily using a "best benefit" heuristic}
4: **while** $RemainB > 0$ **do**
5: $NextPOI = argmax_{m_i \in \mathcal{M}_{prune} - \tau_{max}} \frac{Pr(m_i.feedback=yes|\mathcal{M}_{tempseen})}{\texttt{transit}(m_i, \tau_{max}.LastPOI) + \texttt{visit}(m_i)}$;
6: $RemainB = RemainB - \texttt{transit}(NextPOI, \tau_{max}.LastPOI) + \texttt{time}(NextPOI)$;
7: **if** $RemainB > 0$ **then**
8: $\tau_{max} = \tau_{max} \cup \{NextPOI\}$;
9: **end if**
10: **end while**
11: **return** τ_{max};

Heap Data Structures for Maintaining Benefits: For the `ApproxItn` subroutine to be efficient, at every iteration it needs to be able to quickly determine, from the remaining POIs in $\mathcal{M}$[4] that are not a part of the partially constructed itinerary, the POI with the best benefit with regard to the last POI added to the itinerary. A naive way of doing so is to pre-compute, before each execution of `ApproxItn`, for all pairs of POIs $m_x, m_y \in \mathcal{M}$ the benefit of reaching m_y from m_x. Then, while `ApproxItn` is executing, the benefit of reaching each POI in $\mathcal{M}$ from the last POI of the itinerary can be compared and the POI with the best benefit can be selected. Clearly this approach takes at least $O(|\mathcal{M}|^2)$ time, not accounting for the pre-computation time.

We can reduce the execution time of `ApproxItn` from $O(|\mathcal{M}|^2)$ to $O(|\mathcal{M}| \log(|\mathcal{M}|))$, using the data structuring techniques described below. Since `ApproxItn` is called in the innermost loop of our overall itinerary planning algorithms, this can be a substantial savings in practice.

Pre-Computation: Two data structures are prepared before each call to `ApproxItn`:

1) The first is $ProbOrder$, an ordered list of the POIs in $\mathcal{M}$, in decreasing order of $Pr(m_i.feedback = yes|\mathcal{M}_{tempseen})$ for each POI m_i. Note that these quantities are the numerators of the *benefit* of each POI (see line 5 in Algorithm 6). Instead of naively constructing $ProbOrder$ from scratch every time `ApproxItn` is called, we can leverage the fact that the calls are made in sequence along the Hamiltonian path ordering of the user feedback combinations. Thus for each POI m_i, we update $Pr(m_i.feedback = yes|\mathcal{M}_{tempseen})$ from its previous value in constant time, since $\mathcal{M}_{tempseen}$ has changed by only one POI since the last execution. Thus $ProbOrder$ can be updated and re-sorted in overall $O(|\mathcal{M}| \log(|\mathcal{M}|))$ time.
2) The second is a set of *priority queues/heaps* [4] $H_1, H_2, \ldots H_{\mathcal{M}}$, one for each POI in $\mathcal{M}$. For each POI, the corresponding heap contains the time (transit plus visit) needed to reach every other POI in $\mathcal{M}$. Note that these quantities correspond to the denominators of the benefit of each POI (see line 5 in Algorithm 6). These heaps allow the operation *find best time POI* to be performed in constant time, and the operations *delete best time POI* and *insert POI* to be performed in $O(\log(|\mathcal{M}|))$ time. Although it may appear that the total size of all the heaps is $O(|\mathcal{M}|^2)$, these heaps are constructed *only once* by the `FastExpBatchScore` subroutine. During each of the $2^{|\mathcal{I}|}$ executions of `ApproxItn`, these heaps change due to *delete best time POI* operations, but are restored to their original status before the next execution of `ApproxItn` by undoing the delete operations with corresponding *insert POI* operations, as shall be discussed next.

In-Computation: During the execution of `ApproxItn`, the main task at each iteration is to determine, for the last added POI, the POI from the remaining with the best benefit. However, as described above, we do not store the benefits of each POI directly in any data structure (since that will be expensive to maintain), but rather store the numerators and denominators in separate data structures. Thus to find the POI with the best benefit, we have to *simultaneously* scan both data structures in a round-robin manner, $ProbOrder$ as well as $H_{LastPOI}$ (the latter is done by repeated *delete best time POI* operations), until we determine the remaining POI with the best benefit. This is essentially an application of the popular *Threshold Algorithm* (TA) [5]. While in the worst case it can take $O(|\mathcal{M}|)$ if both data structures need to be completely scanned, in practice, it is expected to stop very early. Once the next best POI has been determined, then the heap $H_{LastPOI}$ can be restored by undoing the *delete best time POI* operations with corresponding *insert POI* operations. Thus the in-computation cost of each execution

[4]Actually, this should be $\mathcal{M}_{pruned}$, but in this discussion we assume that in the worst case there may not be any pruning, and $\mathcal{M}_{pruned} = \mathcal{M}$.

of ApproxItn takes $O(|\mathcal{M}|\log(|\mathcal{M}|))$ time, assuming that the TA algorithm only goes to a constant depth on each data structure on average.

In summary, in this section we presented efficient approximation heuristics for the *POI Batch Selection Problem* as well as the *Itinerary Construction Problem*. We leveraged a simple itinerary scoring function based on the binary feedback model, assumed that the batch size k is reasonably small, and applied a greedy strategy for constructing the batch of k POIs. This is facilitated by making calls to an approximation algorithm for itinerary construction that is based on the *best benefit* heuristic. Moreover, we employ interesting algorithmic and data structure optimizations, such as using the heap data structure for indexing the POI benefits, and maintaining the heaps as well as the probability quantities efficiently by following update strategies based on Hamiltonian path ordering in hypercubes.

V. Experiments

We conduct a set of comprehensive experiments for popular travel destinations using real world datasets extracted from *Lonely Planet*[5] and *Flickr*[6]. In this section, we describe our experimental set-up, data generation and explain our quality and performance results.

We implemented our prototype using JDK 5.0. All performance experiments were conducted on a 2.66GHz Intel Core i7 processor, 4GB Memory, and 500GB HDD, running OS X. The Java Virtual Memory size is set to 512MB. All numbers are obtained as the average of three runs.

A. Data Generation

City Names and POI Generation: We consider popular tourist destinations and their POIs for our itinerary planning problem. 12 geographically distributed cities are considered and the popular POIs of those cities are extracted using the *Lonely Planet* dataset. City names, corresponding number of POIs in each city and some example POIs are shown in table III. For each POI, we used Wikipedia[7] to extract latitude and longitude information associated with it.

Transit Time, Visit Time Generation: Given a city, we generate the transit time between every pair of POIs in that city. We use *Google Maps*[8] to calculate the transit time *by car* between a pair of POIs using the underlying road network. This process gives rise to a POI graph, one for each city and each of these generated graphs are complete and directed. Note that, in general, the pairwise transit times generated in this process are asymmetric, which is usually true for any road network. Visit time of each POI is generated using the Flickr log.

Atomic and Pair-wise Probability Generation using Flickr Log: We use the publicly available Flickr data[9] to generate atomic and pair-wise probabilities of POIs. Flickr data captures user itineraries in the form of photo streams, where the photos are tagged with corresponding POI names and the respective date/time associated with the photos define the set of possible itineraries (such as, a set of POIs visited on the same day). Given a Flickr log of a particular city, each row in that log corresponds to a user itinerary that is visited in a 12 hour window. We use this log to generate the atomic probabilities of the POIs, and the pair-wise probabilities of every POI pair for a particular city. Using three years worth of Flickr logs, the atomic probability of a POI is the fraction that a POI appears out of the total number of itineraries in the query log. The conditional pair-wise probabilities, $Pr(POI_i|POI_j)$ are calculated as the fraction that POI_i was also visited out of the total number of times POI_j was visited.

City Name	Number of POIs	Example POIs
Amsterdam	118	Diamond Museum, Museum Amstelkring, Oosterpark
Bangkok	48	Phayathai Palace, Siam Ocean World, Wat Traimit
Barcelona	73	Arc de Triomf, Museu Picasso, Plaza Reial
Chicago	91	Flat Iron Building, Lincoln Park, Soldier Field
London	163	Brick Lane, Buckingham Palace, Hyde Park
New Orleans	35	French Quarter, Pitot House, St Roch Cemetery
New York	119	Chelsea Art Museum, Lincoln Center, Russian & Turkish Baths
Paris	114	Bois de Vincennes, Jardin des Tuileries, Petit Palais
Rome	134	Arco di Costantino, Colosseum, Gianicolo
San Francisco	78	Alcatraz, Mission Dolores Park, Union Square
Sydney	96	Bondi Beach, Customs House, Taronga Zoo
Toronto	48	Cn Tower, Ontario Place, Spadina Museum

TABLE III
Example Cities and POIs

The Flickr log may be considered as a collection of itineraries selected by prior users. This enables us to perform quality experiments evaluating the effectiveness of interactive itinerary planning, without requiring a user study involving actual users. Our interactive approach chooses the next batch of POIs suggestions based on the probabilistic model learned from Flickr itineraries. User response is also simulated by the same probabilistic model.

B. Summary of Experimental Results

We conduct quality and performance experiments by varying the number of POIs, the budget and the size of the suggested POI batch. Each of these parameters impacts the running time and the score of the returned results. We consider a starting POI for each experiment that provides the starting point for the itinerary. All performance experiments are reported for a running time of a single batch. We argue that pre-computation of itineraries is not possible. We observe in our dataset, that, for a budget of 6 hours, any set of 5 POIs are permissible and can form a valid itinerary. Given a city that consists of about 150 POIs, roughly the number of valid

[5]http://www.lonelyplanet.com/
[6]http://www.flickr.com/services/api/
[7]http://en.wikipedia.org/
[8]http://maps.google.com/
[9]http://www.flickr.com/services/api/

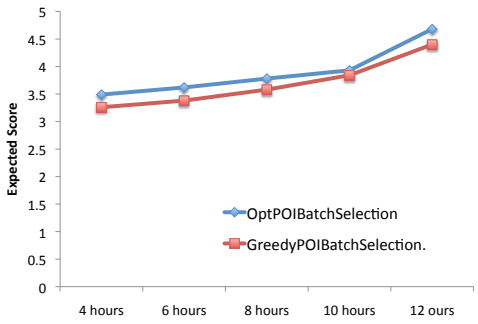

Fig. 2. Expected Score of `GreedyPOIBatchSelection` and `OptPOIBatchSelection`.

itineraries that consist of all 5 POIs could be in the range of 0.5 billion (the total number of itineraries would be much more), which certainly is not feasible to pre-compute.

In short, our experimental results substantiate our claim that the greedy algorithm for interactive itinerary planning is a feasible solution for interactive itinerary planning, both quality and performance wise. In addition, we propose several optimizations of the greedy algorithm and our results accordingly corroborate our theoretical analysis, by generating faster running times for the optimized variants.

C. Quality Experiments

In this subsection, we discuss and report the results of our quality experiments.

Greedy Interactive Itinerary Planning Algorithm: In this experiment, we vary the budget and observe the expected score of the optimal itinerary in one step of the interactive itinerary planning process. We compare the optimal itinerary scores produced by `OptPOIBatchSelection` and `GreedyPOIBatchSelection`. Both of these algorithms use the greedy *best benefit* heuristic to obtain the best itinerary. Input to these algorithms is a set of user feedbacks ('yes' to 3 different POIs) and a batch size (3). This experiment is run on New York City, which has 119 POIs.

Figure 2 shows the output of this experiment. We note that with increased budget, since more POIs can be added to the optimal itinerary, its expected score increases. The figure corroborates the fact that `GreedyPOIBatchSelection` is comparable in the quality of its optimal itinerary, to the more expensive `OptPOIBatchSelection`.

Effectiveness of Interactive Itinerary Planning: In this experiment, we select prior Flickr-based 25 static itineraries (we refer to this as `OfflineItinerary`) instead of actual users, where each itinerary consists of 10 POIs, and is visited in 12 hours. We consider a simpler scoring function to assign score in each of them - the score of an itinerary is the number of POIs in it. For each static itinerary, we apply our interactive itinerary planning algorithm (known as `InteractiveItnPlanning`), where the next batch of POIs suggestion is based on the probabilistic model learned from those Flickr itineraries. `InteractiveItnPlanning` calls `GreedyPOIBatchSelection` to select a POI batch at each iteration. In each batch, user response is akin to the

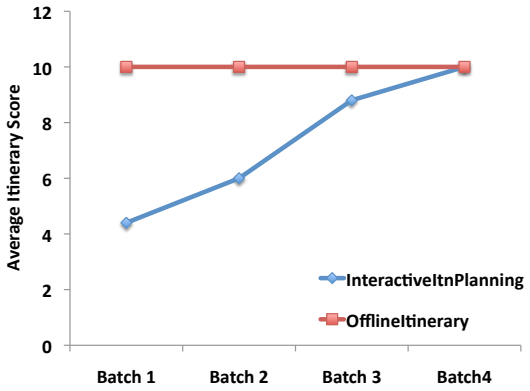

Fig. 3. Effectiveness of Itinerary Planning Algorithm

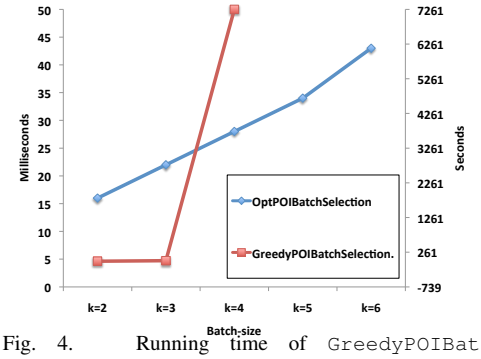

Fig. 4. Running time of `GreedyPOIBatchSelection` and `OptPOIBatchSelection`.

actual POIs present in that itinerary,i.e., response is 'yes' for those POIs which actually surface in that static itinerary.

Figure 3 records the average itinerary score in each batch. It shows that `InteractiveItnPlanning` reaches the same score of offline itineraries in 4 batches on an average. Thus this result demonstrates that our interactive approach effectively generates itineraries that are liked by prior Flickr users .

D. Performance Experiments

In this subsection, we discuss the efficiency aspects of the interactive itinerary planning algorithms, describe the running time attained by performing proposed optimizations and compare that with the optimal brute-force algorithm. Performance is recorded by mainly varying 3 parameters - budget, batch size and number of POIs.

Feasibility of the Optimal Algorithm: We record the running time of `OptPOIBatchSelection` and `GreedyPOIBatchSelection`, for varying batch sizes k in Figure 4. The number of POIs is set to 119 for this experiment, whereas the budget is fixed at 6 hours. `OptPOIBatchSelection` algorithm runs in seconds, whereas `GreedyPOIBatchSelection` runs in milliseconds. Also, beyond batch-size 4, `OptPOIBatchSelection` does not terminate within 10 hours, whereas `GreedyPOIBatchSelection` scales well with increasing batch size. This observation confirms that `GreedyPOIBatchSelection` is an efficient solution for interactive itinerary planning.

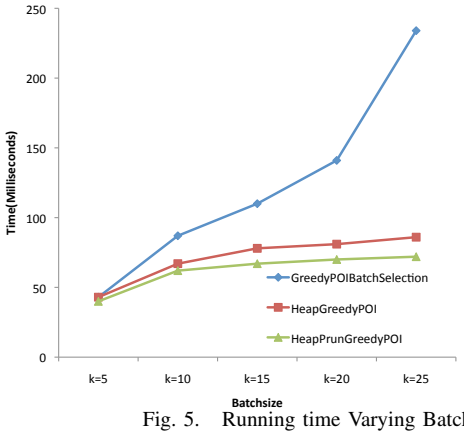

Fig. 5. Running time Varying Batch Size.

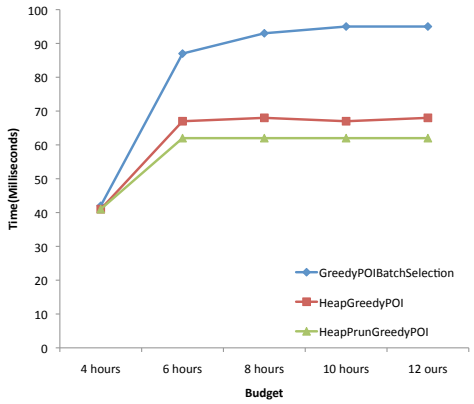

Fig. 6. Running Time Varying Budget.

Varying Batch Size: In this set of experiments, we vary the batch size k and profile the running time of the different optimizations performed in combination with `GreedyPOIBatchSelection`. This algorithm is compared with its optimized variants - greedy that uses a heap to calculate *best time POI* and processes user feedback combinations in the heap following the Hamiltonian path computation(`HeapGreedyPOI`), and the the most optimized variant, (`HeapPrunGreedyPOI`), which in addition to efficient heap processing, also prunes the set of remaining POIs, subject to the budget.

The number of POIs is set to 119 for this experiment, while the budget is fixed at 6 hours. Figure 5 records the running time of this experiment. We observe that with an increasing batch size, the most optimized variant `HeapPrunGreedyPOI` performs substantially better than `GreedyPOIBatchSelection`. This confirms that our proposed optimizations are important to improve the overall performance.

Varying Budget: We vary the budget constraints and keep the batch size and the number of POIs (10 and 119 respectively) fixed, and record the running time of different variants of the greedy algorithm in Figure 6. The figure shows that with the increasing budget, the running time increases in general for all variants. The most optimized variant `HeapPrunGreedyPOI` outperforms others in all cases. One interesting observation here is, the running time does not increase linearly with the budget. This phenomenon is due to the fact, that, for a large enough budget (while everything else is fixed), there cannot be any pruning based on budget and hence it does not impact performance anymore.

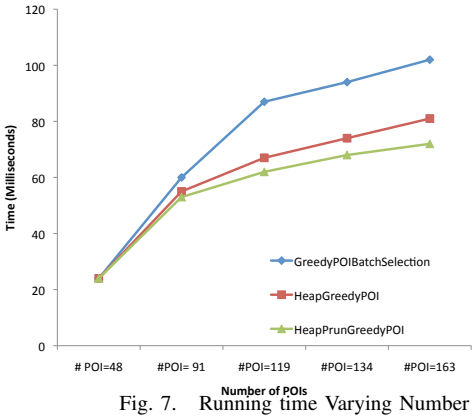

Fig. 7. Running time Varying Number of POIs.

Varying Number of POIs: We vary the number of POIs for a fixed budget (6 hours) and batch size (10). The running time of different variants are recorded in Figure 7. With the increase in POIs, the running time increases in general. This can be explained since the greedy algorithm has to probe more POIs for selecting the k best POIs in each batch. The most optimized variant `HeapPrunGreedyPOI`, outperforms the rest in all cases. One noteworthy observation here is, the running time of `HeapPrunGreedyPOI` increases the least with the increase in the number of POIs. The role of pruning becomes significant in this case, hence with the increase in the number of POIs, `HeapPrunGreedyPOI` effectively prunes the remaining POIs in a batch, and becomes the winner.

VI. Related Work

Our work of interactive itinerary planning is an effort towards returning complex objects (i.e., an itinerary constructed of several POIs) to the user based on user interaction, subject to the constraints. In a recent work, we first propose the notion of composite items [6] towards that goal. However, an itinerary is *not any arbitrary* ordering of a set of POIs, but it renders a strict ordering between the POIs, subject to the constraints. The ordering imposes a complex relationship between POIs and makes this problem significantly different from our earlier model [6].

The interactive itinerary planning facilitates effective navigation through the information space. Our interactive POI selection strategy is akin to exploratory browsing interfaces such as faceted search [7]. However, the interaction here is on the suggested set of POIs.

Existing work related to travel itineraries can be classified into touristic data analysis and touristic information synthesis. Regarding the former, there are a number of studies on analyzing POI visitation patterns from geo-spatial and temporal evidences left by travelers [8], [9], [10], [11], [12]. However, those works generally do not synthesize POIs into itineraries and instead focus solely on the analysis itself. In the context of touristic information synthesis, a number of works construct

and recommend tourist itineraries at various granularities [13], [14], [15], [16], [17] but none of them provides the ability to query constructed itineraries. Our work is tangentially related to other vast fields such as visualizing geo-spatial data, tracking movements based on sensor networks, and constraint optimization. The closest works to ours are [18] and [19] which merge touristic data analysis and synthesis to recommend itineraries based on user's input. However, none of them does so in an interactive manner.

A recent work proposes interactive route search in the presence of order constraints [20]. The proposed approach is different from our work in that it does not consider user budget, does not synthesize user's previous feedback to learn future probability of user preferences, and more importantly, tries to build an itinerary POI by POI, whereas we consider a navigational approach that starts with all possible valid itineraries, which is then iteratively narrowed by suggesting POIs in batches based on highest expected itinerary scores.

Our *optimal itinerary construction problem* is akin to the *vehicle routing problem* and *traveling salesman problem* [4], [21], widely studied in the field of Computer Science and Operation Research. These problems and several of their variants are known to be NP-complete. One variant of vehicle routing problem is the *Orienteering problem*, which and many of its variants are also known to be NP-complete [22], [23], [24]. In particular, we deal with the Rooted Orienteering problem in non-Euclidean asymmetric metric space. Efficient polynomial time approximation scheme is known for this problem problem in the plane [3]. Unfortunately, to the best of our knowledge, there are no known approximation algorithms with provable bounded factors for its non-Euclidean asymmetric variant.

Our greedy solution to the itinerary construction problem requires efficient searching for the next *best time* POI. We resort to a heap data structure [4] for that, which facilitates efficient look up operation for the next best time node. The *next benefit* POI is retrieved by performing a threshold style [5] computation on *ProbOrder* lists and heaps.

Our greedy algorithm for POI selection problem processes feedback combinations in a current batch such that the heap requires only one update between subsequent combinations. We leverage on computing a Hamiltonian Path on a hypercube graph [4] to accomplish that task.

VII. Conclusion

In this paper, we formalized interactive itinerary planning, showed that it is an NP-complete problem and developed intuitive optimizations for the case where the score of an itinerary is proportional to the number of Points Of Interest (POIs) desired by the user. In order to do so, we reduced our problem to the rooted orienteering problem. Our optimizations are based on computing a Hamiltonian path in a hypercube and on using an efficient heap-based data structure to efficiently prune POIs. In the future, we are planning to explore optimizations for more sophisticated itinerary scoring functions such as the chain semantics, and to consider more complex budget constraints which incorporate both time and price. Our algorithms would need to be revisited for that purpose.

References

[1] J. Whittaker, *Graphical Models in Applied Multivariate Statistics*. Wiley, 1990.

[2] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.

[3] K. Chen and S. Har-Peled, "The Euclidean orienteering problem revisited," *SICOMP*, vol. 38, no. 1, pp. 385–397, 2008.

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

[5] R. Fagin and et. al., "Optimal Aggregation Algorithms for Middleware," in *PODS*, 2001.

[6] S. B. Roy, S. Amer-Yahia, A. Chawla, G. Das, and C. Yu, "Constructing and exploring composite items," in *SIGMOD Conference*, A. K. Elmagarmid and D. Agrawal, Eds. ACM, 2010, pp. 843–854.

[7] K.-P. Yee, K. Swearingen, K. Li, and M. A. Hearst, "Faceted metadata for image search and browsing," in *CHI*, G. Cockton and P. Korhonen, Eds. ACM, 2003, pp. 401–408.

[8] S. Ahern, M. Naaman, R. Nair, and J. Yang, "World explorer: Visualizing aggregate data from unstructured text in geo-referenced collections," in *Proc. Joint Conference on Digital Libraries (JCDL'07)*, June 2007, pp. 1–10.

[9] D. Crandall, L. Backstrom, D. Huttenlocher, and J. Kleinberg, "Mapping the world's photos," in *Proc. 18th International World Wide Web Conference (WWW'2009)*, April 2009, pp. 761–770.

[10] F. Girardin, "Aspects of implicit and explicit human interactions with ubiquitous geographic information," Ph.D. dissertation, Universitat Pompeu Fabra, Barcelona, Spain, 2009.

[11] A. Popescu and G. Grefenstette, "Deducing trip related information from flickr," in *Proc. 18th International World Wide Web Conference (WWW'2009)*, April 2009, pp. 1183–1184.

[12] T. Rattenbury, N. Good, and M. Naaman, "Toward automatic extraction of event and place semantics from flickr tags," in *Proc. 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'07)*, July 2007, pp. 103–110.

[13] L. Ardissono, A. Goy, G. Petrone, M. Segnan, and P. Torasso, "Intrigue: Personalized recommendation of tourist attractions for desktop and handset devices," *Applied Artificial Intelligence*, vol. 17, no. 8-9, pp. 687–714, 2003.

[14] K. Cheverst, N. Davies, K. Mitchell, A. Friday, and C. Efstratiou, "Developing a context-aware electronic tourist guide: some issues and experiences," in *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM, 2000, pp. 17–24.

[15] S. Dunstall, M. E. T. 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, 2004.

[16] D. Leake and J. Powell, "Mining large-scale knowledge sources for case adaptation knowledge," in *Proc. ICCBR 2007*, 2007, pp. 209–223.

[17] ——, "Knowledge planning and learned personalization for web-based case adaptation," in *Proc. ECCBR 2008*, 2008, pp. 284–298.

[18] C. H. Tai, D. N. Yang, L. T. Lin, and M. S. Chen, "Recommending personalized scenic itinerary with geo-tagged photos," in *Proc. IEEE International Conference on Multimedia and Expo (ICME'2008)*, 2008, pp. 1209–1212.

[19] M. D. Choudhury, M. Feldman, S. Amer-Yahia, N. Golbandi, R. Lempel, and C. Yu, "Automatic construction of travel itineraries using social breadcrumbs," in *HT*, M. H. Chignell and E. Toms, Eds. ACM, 2010, pp. 35–44.

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

[21] G. Dantzig and J. Ramser, "The truck dispatching problem," in *Operations Research*, 1959, pp. 80–91.

[22] K. Chen and S. Har-Peled, "The euclidean orienteering problem revisited," *SIAM J. Comput.*, vol. 38, no. 1, pp. 385–397, 2008.

[23] A. Blum, S. Chawla, D. R. Karger, T. Lane, A. Meyerson, and M. Minkoff, "Approximation algorithms for orienteering and discounted-reward tsp," *SIAM J. Comput.*, vol. 37, no. 2, pp. 653–670, 2007.

[24] C. Chekuri, N. Korula, and M. Pál, "Improved algorithms for orienteering and related problems," in *SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2008, pp. 661–670.