Smart traveling application

Carleton University
COMP 4905 Honors project
Supervisor Name: Michel Barbeau
Student Name: Coco Chen
Student Number: 100921653
Abstract

In this article, we attempt to provide a planning algorithm to help users find the best itineraries during their vacation time. We deploy an A* Search and heuristic algorithm to evaluate and define a good itinerary as one that takes the users between points of interests swiftly, visits the best locations first, and manages travel time in between. The smarting traveling application we show today is designed to help the user to make traveling plans in an unfamiliar destination by showing nearby the points of interests (POI) near the user’s entered a position. The application uses Google Places API to fetch a list of POIs to allow a user to generate a plan based on each destination of choice. The purpose of this report is to demonstrate an evaluation of each algorithmic approach and with the aim to describe the technologies used in our application to solve this problem by deploying the selected algorithm.
Table of Contents

Abstract 2

Table of Contents 3

1. Introduction 5
   1.1 Context 5
   1.2 Problem statement 5
   1.3 Result 6
   1.4 Outline 6

2. Background 7
   2.1 A Brief Introduction on Planning 7
   2.2 Potential Algorithms 9
      2.2.1 Travelling Salesperson Problem 9
      2.2.2 A* Search Algorithm 10
   2.3 Google Maps Service API 11

3. Result 13
   3.1 Algorithm Design and Implementation 13
      3.1.1 Static Planning 13
      3.1.2 A* Search Algorithm Planning 14
   3.2 Development Environment and Tools 19
   3.3 Frameworks 19
      3.3.1 Server Side Framework 19
   3.4 The Application 23
      3.4.1 Sequence diagram 23
      3.4.2 Use Case 27

4. Evaluation 31
   4.1 Evaluating the Algorithm 31
      4.1.1 Static Planning Algorithm 31
      4.1.2 A* Algorithm 31
   4.2 Evaluating the System 34
      4.2.1 Scalability 34
      4.2.2 Usability 34
   4.3 Performance 38

5. Conclusion 39
   5.1 Summary 39
5.2 Limitations 39
5.3 Future work 39

6. Appendix 40
   6.1 List of Figures 40
   6.2 List of Tables 40
   6.3 Usability Questionnaire 41

Reference 42
1. Introduction

1.1 Context

Year over year, an increasing number of people will choose to travel around the world during their vacation time to potentially experience different cultures, delightful environments, and the fascinating sceneries.

This application is targeted towards tourists on vacation, typically in an area in which they are unfamiliar with, to be used to plan recommended traveling paths for their days out. It is designed to propose points of interests for the users and enable them to select and make corresponding traveling plans based on chosen destinations.

1.2 Problem statement

The main part of the problem is planning which can be modeled by a graph. The considered problem can be represented as a destination network with a set of nodes and a set of links. Let \( G(V, E, W) \) be a graph, which \( V \) present the nodes and \( E \) present link sets, and \( W \) represent the weights correspondingly to the certain edges. Each vertex \( V \) represents a destination in the plan. The links \( E \) are the edges connect the vertices. The edges are often non-directed, meaning a person can travel back and forth between destinations, but the solution should direct the user to every node of the graph. When an edge is drawn from node \( A \) to node \( B \) with an arrow pointing from \( A \) to \( B \), it indicates that their route between points of interests are designed to visit the destination \( A \) before the destination \( B \). Each possible edge \( E \) will correspond to a weight \( W \), which illustrates the specific weight for the edge. In this problem, the graph we can treat our initial graph as a complete graph since the person is freely able to move from any points \( A \) and \( B \). However while we are constructing the graph, we do not necessarily construct all edges since some of the edges will fail the hard constraints, such as opening/closing time, and does not make sense to keep traversing or building every edge.

The objective of the optimal solution(s) is to find the lowest cost path on the graph which has the highest evaluating score or lowest cost. The algorithm described by this article will attempt to assign a score for certain edge referred to as the weight for the corresponding edge. The output of the algorithm will attempt to build a plan which is an ordered set of vertices of the graph.
1.3 Result

Initially, the application is designed to be a mobile application, but it ends up as a web application. Since a web page can display more information on a page, it can give the user a better experience when they are selecting destinations because all the destinations can be displayed on the same page. The goal of the application is trying to help the user generate a satisfying traveling plan. In the application, A* algorithm is used to find the best plan by searching for the set of possible plans states. A scoring mechanism is used to calculate the quality of the plan at each stage. The scoring mechanism measures the cost of each state by weighing in factors such as operations hours, the rating of the points of interests, and the distance between the pairs of destinations to consider in aims to return most pleasant traveling plan for the user.

1.4 Outline

The rest of this report is structured as follow. Section 2 presents background information relevant to this project and potential algorithms that can be used find a solution for the problem statement. Section 3 will describe the obtained result of the application in detail. Then we will evaluate to what degree does the result obtained from project solves the defined problem statement in section 4. Finally, section 5 concludes the report and the limitation of the application and the future work that may improve the result is explored.
2. Background

In this section, background knowledge of the application and relevant information is depicted. The first subsection, A Brief Introduction on Planning, provides a brief historical overview of the existing planning algorithms that are solved today. The second subsection, Potential Algorithms, proposes several algorithms that can be used to solve our planning problem. The last section gives a general idea about the Google Maps APIs the application used and how the application uses it.

2.1 A Brief Introduction on Planning

Planning is a term being used widely in the computer science field. In recent years, people are getting more and more familiar with robots and AI technology. For example, in automobile manufacturing industry, robots are used to help human assemble vehicles. How does the robot know when, where and what to move?

The terms motion planning and trajectory planning are used to describe the algorithms that convert high-level specifications of tasks from humans into low-level descriptions of how to move [8]. Robot motion planning usually ignores the dynamics and other potential constraints. It tends to focus more on the translations and rotations themselves, while trajectory planning takes into account how to adjust the movement in a way that it respects the mechanical limitations of the robot [8].

Control Theory is another type of planning, which manages the control of continuously operating dynamical systems. The main focus of this theory is to instantiate stability, which ensures that the dynamics of the environment do not cause an unexpected situation [8].

In Artificial Intelligence, the planning considers a problem in a more discrete flavor [8]. For example, instead of planning to move a piano through a continuous space, the problem involves more on problem-solving such as solving puzzles which the environment brings about. We have talked about several kinds of planning, but why do we want to solve these problems? Let’s see some motivational examples and applications.

A classical example of motion planning is the Piano Mover’s Problem. Imagine we have a piano sitting in the middle of the room and we want to move it to another room. Besides the piano, there are other pieces of furniture. The algorithms applied in the robot needs to consider avoiding collisions between robots and other pieces of furniture in the room. As you can see, the problem becomes more complex as the space and variability of the environment grow. The AI planning algorithms can be used to solve discrete puzzles, operations, and scheduling [8]. For example, a video game named F.E.A.R uses a STRIPS-style planner for the enemy AI. STRIS is a planning algorithm that searches through the entire state space by applying actions [8].
Terminologies used in the following article
In planning, there are several basic ingredients we need to understand.

**State:** A state in planning refers to the snapshot of the situation at a certain time [8]. For example, in a motion planning, a state can be used as to describe what gesture is the robot in right now. A state space captures all possible situations that could arise.

**Actions:** Actions manipulate the states [8]. The term action is used to describe the job has been transitioned between two states.

**Initial and goal states:** The initial state refers to the state hasn’t been applied any actions yet. The goal state is the specified state we would like to achieve. There might be a set of goal states. The actions are selected in a way that tries to accomplish arriving at the goal state [8].

**Feasibility:** The ability to find a plan that causes arrival at a goal state, regardless of its efficiency [8].

**Optimality:** Find a feasible plan that optimizes performance in some carefully specified manner, in addition to arriving in a goal state [8].

**A plan:** Some sequence of actions you need to apply to achieve the goal state.

**Evaluation function:** A function calculate the estimated cost from the current state to the goal state.
2.2 Potential Algorithms

2.2.1 Travelling Salesperson Problem

The traveling salesperson problem (TSP) is stated as: A salesperson starts his travel from his home city, with a set of cities he needs to visit and the distances between each pair of cities given, he needs to visit each one of them exactly once and return to the original city [9]. What is the best order in which to visit them such that the distance or the cost of his travels are minimized in total [9]?

![Traveling Salesperson Graph](image)

**Figure 1.** An example of travelling salesperson graph with the solution on the right

The traveling salesperson (salesman) problem was treated in the 1800s by the Irish mathematician Sir William Rowan Hamilton and by the British mathematician Thomas Penyngton Kirkman [10]. The book *Graph Theory 1736-1936* first discussed the early work of Hamilton and Kirkman did about this problem [11]. Hamilton’s Icosian Game was a puzzle with an objective to finding a Hamiltonian cycle in a graph [10]. A Hamilton’s cycle is defined as a cycle in an undirected or directed graph that visits each vertex exactly once and comes back to the original point. The general form of the TSP was first discussed by mathematicians Karl Menger at Harvard in 1930s [10]. Karl Menger defined the problem state and considered an obvious brute-force algorithm for it. In the paper *On the history of combinatorial optimization*, Hassler Whitney introduced the name ‘Travelling salesman problem’ [12].

Currently, there are many variations of algorithms for TSP. The most direct one is to generate all possible permutations of each destination and evaluate which one of them is the cheapest to visit all of them. This is called the exact algorithm and has a running time of $O(n!)$, where $n$ is the number of cities or destinations it needs to visit. Nowadays TSP can be optimized by solving the program using dynamic programming techniques.

*Dynamic programming* is an algorithm paradigm where a problem is solved by tackling the collection of subproblems one by one [9]. By starting from the smallest or easiest one first, it
uses the answers of small problems to figure out the solutions for bigger ones until the whole problem is solved [9]. Let’s first define the sub-problem of TSP:

For a subset of cities \( S \subseteq \{1, 2, \ldots, n\} \) that includes 1, and \( j \in S \), let \( C(S, j) \) be the length of the shortest path visiting each node in \( S \) exactly once, start at 1 and ending at \( j \).

When \( |S| > 1 \), we define \( C(S, 1) = \infty \) since it does not make sense to travel between the same node. The subproblem is we need to start at 1 and end at some other node \( j \). Then second-to-last city has to be some \( i \in S \), so the overall path length is the distance from 1 to \( i \). We can denote this as: \( C(S - \{j\}, i) + d_{ij} \), where \( d_{ij} \) is the length of the final edge. We need to pick the best one such \( i : C(S, j) = \min_{i \in S \setminus \{j\}} C(S - \{j\}, i) + d_{ij} \).

In summary, the algorithm:

\[
C(\{1\}, 1) = 0 \\
\text{for } s=2 \text{ to } n: \\
\quad \text{for all subsets } S \subseteq \{1, 2, \ldots, n\} \text{ of size } s \text{ and containing 1:} \\
\quad \quad C(S, 1) = \infty \\
\quad \quad \text{for all } j \in S, j \neq 1: \\
\quad \quad \quad C(S, j) = \min \{C(S - \{j\}, i) + d_{ij} : i \in S, i \neq j\} \\
\text{return } \min_j C(\{1, \ldots, n\}, j) + d_{1j}
\]

There are at most \( 2^n \cdot n \) subproblems. For every single one of them, it takes \( O(n) \) time to solve. As a result, the total running time is \( O(n^2 2^n) \) [9].

The TSP has several applications such as planning, logistic, and the manufacture of microchips, which is closely related to the problem for the system STA we describe in this paper to solve. For the application of TSP on our planning problem, the destinations users entered are the cities corresponding to the TSP. The evaluation function can be used to generate the ‘distance’ between destinations. The ‘distance’ is not the actual distance between cities, but the evaluation of the planning scores. So the TSP finds the plan will travel each destination once with the highest score or lowest score.

### 2.2.2 A* Search Algorithm

Another algorithm can be potentially used to solve the problem is using A* Algorithm to make the process relatively efficient. For a planning algorithm, we need to define an initial state and a goal state. In general, plans are created by searching through the space of all possible actions, a graph in our case, until the final goal state is achieved [7]. The subset of a state space consists the possible states when a certain action is applied. The search terminates when the goal state is achieved [7]. The goal state where we reach the end of our search when all destinations have been reached. Solutions are a sequence of actions...
leading from the initial to a goal state. The following picture shows us an example. The paths marked in red is the solution. A* algorithm uses a heuristic to speed up the searching process and avoid expending unnecessary nodes.

![Space of Possible actions](image)

**Figure 2.** An overview of planning

Sometimes, there might be multiple paths can be taken to get to the goal state. In order to select the better one, each action corresponds to a cost. The cost can be calculated by using the evaluation function, and the optimal solution is the path which has a lower cost.

### 2.3 Google Maps Service API

In the application, a user gets prompted to enter a location to the page. This will trigger a service call to Google’s places API to retrieve a list of candidate points of interests and returned back to the browser. We have chosen to use the free Google Places API to retrieve free information on each candidate points of interest. The API we used in this application can search for local businesses by category, retrieve place details such as ratings and opening hour, and perform auto broadening search based on any location in the world. This service serves as the foundation of our scoring algorithm which the data is used to evaluate the cost each candidate plan.

The service is call is initiated by a user request, every time a user enters in their location on the planning application, the proceeding requests will trigger a Google API service call. Google service response is highly detailed and includes many of the facets you would normally find if you search for something similar to Google’s very own search engine. In order to process the response, we have to extract and transform the data into the planning algorithm’s format. Some of the very important POI details to extract for planning are the destination ratings, the geographic information and operation hours of the place, and geographic positionings in longitude and latitude (to be used for measuring the distance between two points of interest). We load an embedded Google Maps iframe into the
application which simulates the functionality of this common to that many of us are fond of. The embedded maps are where the application cycle begins, where a user will start their travel planning. One downside of using the embedded map to gather POI information is that it can become a burden for large and dense geographic areas like Tokyo. As a result, this can stress the running time since the application algorithm can only begin to process the information when all our requests have responded.
3. Result

With much of the background information and potential algorithms defined, this section of the report will describe the implementation result for the application and the challenges that we have encountered while we implement the algorithms. This section is broken up into several subsections to introduce the implemented algorithms, development environment, and the framework used in the application.

3.1 Algorithm Design and Implementation

We have a couple algorithms that have contributed to the final result of the application. The first algorithm, Static Planning, calculates the scoring of each plan by brute force. The second algorithm is A* based, which is designed focuses on improving the weakness of static planning algorithm.

3.1.1 Static Planning

The algorithm

The Static Planning (SP) algorithm is one of the algorithms implemented in the application to help users plan the best routes for their travels. The Static Planning algorithm can guarantee to find the optimal solution among all possible permutations of the destinations. The first step of this algorithm is to generate all the possible combination of the points of interests that the user wishes to tour. In this algorithm, we make the assumption that the number of POIs are not high since a traveling tourist has limited time to see all the destinations they wish. This bounds the running time to O(n!) where n is never really too big. After all the possible plans generated, the algorithm uses an evaluation function to evaluate each plan. The evaluation function considers multiple factors and calculates corresponding scores for each solution. There are many factors being considered but we can categorize them into two main types. The first type, the hard-constraints, and the second type, soft-constraint. For the hard-constraint type, it refers to a constraint or deadline the algorithm has to follow. If the current plan disagrees with the hard-constraint, an extremely negative number will be added to the plan evaluation score. Comparing with the soft-constraint evaluation score, the weight of the hard-constraint will be much higher because we want to make sure the plan fails if the hard-constraints fail so the solution will be added at the bottom of the planning list and will be least likely to be selected. The algorithm will read in permutation after permutation and calculate the cost of each plan by evaluating the hard and soft constraints of the order of destinations to travel. A final score is concluded after considering each destination’s score per plan, the equations below show how we calculate this score. Once each plan’s score is fully evaluated, with is a composition of the cost from each destination of the plan to other destinations, the plan, along with the score, is
added into a priority queue. The plan with the highest evaluation score will have the highest priority and will be the first element in the queue and is presented to the user.

\[
C(P) = R(i) \times 10 + t + d
\]

- \( C(P) \) - Cost of plan \( P \)
- \( R(i) \) - Rating of POI \( i \)
- \( t \) - Time score for the time of the day the user is expected to get there. Will have a negative effect if the POI is closed by this time
- \( d \) - Distance score, cost of users traveling between the destinations in the plan

**Special cases**

When all possible plans do not meet the hard constraints, all the scores in the result list will be negative because of the big weight set for hard constraint failure. In this case, no plans will be selected.

When two plans get the same evaluating score, the first one in the list will be selected.

### 3.1.2 A* Search Algorithm Planning

The A* Planning algorithm can help improve upon the running time SP by eliminating the need to calculate worse scoring plans than our best seen so far. Imagine this as pruning the search tree. To begin explaining the details of the algorithm, we must first touch base on the state representation. In our application, an array of numbers is used to represent the state of a current node. The length of the array is the number of destinations selected. Each number in the array indicates whether the destination is already visited and a fully fulfilled array indicates the order of the destination that is visited. For example, if we have three destinations - A, B, C to visit, an array of [0,0,0] represents none of the destinations have been visited. The array [3,1,2] indicates we visit the destination B first, and then the destination C and A. Each array is used to represent the state of a current node in the tree. The number of node in the tree is \( n! \) when \( n \) indicates the number of destinations we select.

On each iteration of our algorithm, we apply an action in our application representing a visit to a different destination. The action defines a change in the state. The goal state is defined as the summation of all numbers in the array equals to \( n^2(n-1)/2 \) where \( n \) is the number of destinations in the plan.

**Goal state:** \( \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \), where \( n \) is number of the destinations

The state space is a natural representation scheme and it consists of a set of states. Each state in the state space can be interpreted as a snapshot of the current situation. In our application, the state space is all the possible traveling plans we can generate.
The following pictures show us a precedent of the state space when the number of destinations is three. The letter ‘I’ stands for the initial state; the letter ‘S’ stands for intermediate states between the initial state and the goal state; the ‘G’ stands for the goal state.

![State Space Diagram](image)

**Figure 3.** The state space when number destinations equals three.

The solution of the problem involves a tree search through the state space. As the example shown, there are multiple ways to get to the goal state. We need to introduce an evaluation function to calculate the cost. A modification of the Best-first Search with an A* search heuristic is used. A Best-first Search uses an evaluation function to appraise the desirability of the current node. If a node has a desirable score, this node considers being more likely in the optimal path.

The idea of A* search is to avoid expanding paths that are expensive. The evaluation function $f(n)$ is broken into two parts. The function can be written as: $f(n) = g(n) + h(n)$, where $g(n)$ indicates the cost to get to the current node and $h(n)$ is the estimated cost from the current node to the goal state, so $f(n)$ is the estimated total cost of a path through $n$ to goal. The evaluation function $g(n)$ is calculated based on destination information such as rating, distance, operation hours etc. If the destination score is high, it means the cost goes to the node will be low. The heuristics, $h(n)$, is calculated by height minus depth of the tree. Then, the A-star search algorithm is applied to the tree.
The algorithm

- Initialize VISITED to initial state
- Until a Goal is found or no nodes left in VISITED do:
  - Pick the best node in VISITED
  - Generate its successors (recording the successors in a list);
  - Place in FINISHED
  - For each successor do:
    - If not previously generate (not found in VISITED or FINISHED)
      - Evaluate, add to VISITED, and record its parent
    - If previously generated (found in VISITED or FINISHED), and if the new path is better than the previous one
      - Change parent pointer that was recorded in the found node
    - If parent changed
      - Update the cost of getting to this node
      - Update the cost of getting to the children
        - Do this by recursively ‘regenerating’ the successors using the list of successors that had been recorded in the found node
      - Make sure the priority queue is reordered accordingly

In the following figure, an example is provided to demonstrate this algorithm in details. The starting node is ‘S’ and ‘Gx’ are the goals. The numbers in red represent the scores the heuristics functions returned for each node. The numbers in purple indicate the actual cost for the corresponding paths. A* algorithm will run like this:

Figure 4. An example with heuristic function cost and actual cost for all possible states.
Step 1: Traverse node ‘S’.

Step 2: Traverse node ‘A’ since it has the lowest cost so far.

Step 3: Traverse node ‘D’ since it has a lower cost so far.

Step 4: We have reached the first goal state. Keep searching for a better path. Traverse the node ‘B’ because it has the lowest cost.
We keep doing the same step until we hit this case because there is no node in the graph that has a lower cost than the cost of node ‘G4’ which is already a goal state.

This example established how the A* search can be used to solve the problem in our application. One very important properties of the heuristic function for A* search is that it needs to be admissible. A heuristic \( h(n) \) is admissible if for every node \( n \), \( h(n) \leq h^*(n) \), where \( h^*(n) \) is the actual cost to reach the goal state from \( n \) [7]. If a heuristic overestimates the cost to reach the goal, A* might not find the optimal path.
3.2 Development Environment and Tools

This application in originally developed under macOS with Node.js and Express installed. Since it is a web application, it can be run on any operating system with the required framework installed.

3.3 Frameworks

In this subsection, we introduce the frameworks and technologies used in the application.

3.3.1 Server Side Framework

Express with Node.js

The application we describe in this article is built with Node.js and Express as the server side framework. Express is an unopinionated web framework, written in JavaScript and hosted within the Node.js runtime environment [1]. Node provides the user a super fast server with an event-driven, non-blocking I/O model [3]. By using Express, it provides an easy way to generate initial skeleton code with common features like routing, APIs, and HTTP verbs [3]. There are several reasons that we choose to use Node.js as the server side framework. First, it uses JavaScript as its main language and JavaScript is widely used among web developers. This is convenient for this project because it uses JavaScript for all layers of the stack. For example, ASP.NET is another framework, but it requires developers to know another language on server-side like VB.NET, Java, or C# [4]. Second, since Node.js is powered by V8, it runs at a remarkably fast since Google maintains it [4]. The great contributor to the speed of Node is from its event loop mechanism. It’s essentially a while loop which evaluates different functions and events on a single core. This method means the sequences of events are laid out in a one-dimensional vector, the evaluation window, and the event evaluator calls each function as it moves down the vector. Functions can attach more functions to this queue, which will eventually be called. What this event loop gives to the system is the ability to make asynchronous I/O operations. As we know, I/O operations are considered the most time-consuming tasks a CPU has to bear. Node handles with very well in the following way and we will use a disk read as an example. If the following representation is the evaluation vector:

\[[\text{do1()}, \text{do2()}, \text{diskRead()}, \text{do3()}, \text{do4()}]\]

The evaluator will run functions do1(), do2(), then diskRead(). In traditional systems, the process will be blocked until diskRead finishes before do3 and do4 are run. In node, we would generally attach a event callback to diskRead() making the evaluation vector look like this during the evaluation of diskRead. Node handles diskRead() by offshoring the disk job back to the operating system which usually runs on some other thread but attaching a call
back function to call when the disk read is done. Node is able to continue evaluating do3() and do4() before diskRead() finishes. So when the disk read is actually done, the operating system adds a new function to the back of event loop and passes back the data. The allows Node to handle important web requests without bogging down on slow I/O operations. Paypal converted parts of their backend to Node.js and has reported that the technology had reduced the response time by 35% with doubling the number of requests per second [4].

**Require.js**

One of the main framework used on the server side is Require.js. Require.js supports the use of Asynchronous Model Definition (AMD) for JavaScript modules [2]. AMD is a specification for Javascript which defines an application programming interface (API) for interactions between code modules and their dependencies. It loads module asynchronously when the module is requested or required which results in reduced page load time and initial script download size. The following figure demonstrates the loading order for an AMD model.

![Asynchronous Module Definition (AMD)](image)

**Figure 5.** The AMD model.

By using the Require.js framework, we are able to offload heavy dependencies that are unneeded when the page loads for the first time. Chrome inspect is a great tool to see how much time is spent on script evaluation, DOM reflows, and total page load times. Require.js will offload that stress of loading the page initially and spread this computation over time. The framework has the ability to automatically request new script dependencies when it’s needed. For example, if we have a submit button on the page that has some javascript it needs to run when the button is clicked. It’s likely that this script is not needed on page load because the user hasn’t clicked it yet, so therefore the event handler script for the corresponding button is not needed. When the button is finally clicked, the Require.js AMD model can fetch the JS file by making an extra HTTP request. This makes the page load faster since fewer scripts need to be interpreted at page load but can increase the amount of network traffic. Although network traffic is not cheap, the round trip time for fetching a script
from a CDN is not long and is usually unnoticeable by the user. This technique that we employed in our application makes the web page feel “fast and snappy”.

**CSS extension LESS**

LESS is a dynamic stylesheet language which extends the Cascading Style Sheets (CSS) syntax with the ability to build composable styles for websites. It requires a preprocessor to parse the LESS syntax, which borrows a lot of semantics from CSS to output styles which the browser understands. Using LESS during web development has brought a few advantages over traditional CSS. It allows programmers to compose styles sheets together by sharing mixins, which are essentially macros [5]. LESS includes variables in the language, which is super helpful when we want to change primary and secondary colors of our site. LESS also encapsulates CSS classes very well with nesting which prevents accidentally class namespaces bleeding into the global namespace. Below are some examples of the benefits of LESS.

**Variables**: In Less, developers can define variables for css and the variable is accessible throughout the program [5]. For example:

```
Less syntax

@button-color-red: #4D926F;

button {
  Color: @button-color-red;
}

Css syntax

button {
  Color: #4D926F;
}
```

**Figure 6.** An example demonstrating the usage of a variable syntax in Less.

By using the variable feature, it makes the program more extensible and maintainable. For instance, if the variable button-color-red is used by multiple classes and we would like to change the color from red to green, we only need to change the value or the variable instead of changing all the values in all the classes used this value.

**Mixins**: This allows developers embed all the properties in one class into another class. It is used by simply including the class name as one of its properties.

**Nested rules**: Less also support nest selectors inside other selectors.

The following example provide us an example how Less gets compiled toCss and how to use these two techniques in Less.
Figure 7. An example for Less using mixins and nested rules.

Namespaces for grouping variables: This functionality is very useful in a gigantic application when there are many classes to avoid naming conflicts.

Operators and functions: Less supports operation such as addition, subtraction, division and multiplication for property values and colors.

```
Less syntax

.rounded-corners (@radius: 5px 10px) {
  border-radius: @radius;
}
#header {
  h1 {
    font-size: 16px;
  }
  p {
    .rounded-corners;
    &:hover {
      color: red;
    }
  }
}

CSS syntax

#header h1 {
  font-size: 16px;
}
#header p {
  border-radius: 5px 10px;
}
#header p:hover {
  color: red;
}
```
3.4 The Application

In the application, the server communicates with the client through HTTP requests. When a user submits a location, the client sends a POST request to the server. The server will look up the details on the location that the user provided from Google Maps Service. With detailed information such as longitude and latitude, the server can deploy “nearby” search for popular POI candidates for which it will respond to the original request through HTTP. After that, the page is updated by using the partial-page rendering. Partial-page rendering does not refresh the whole page. It only updates the regions of the page that have changed and the server is smart enough to know what parts of the page to update. Therefore, users do not see a refresh for the whole page when they change their desired location every time. After users have made selections in the list, a list of locations are sent back to the planning service on the server. The server uses the A* algorithm or the static algorithm to generate a plan. When the plan is generated, the page is directed to URL ‘/planning’ to display the resulted plan.

3.4.1 Sequence diagram

In order to show the sequence better, the following picture demonstrates the events and the order of the events happening when the user is generating a plan.
Figure 9. The sequence diagram for a user generating a plan
The stack of function calls can be listed as below:
When a user finishes entering the location they would like to make a plan for and presses
the submit button, the input is sent to the server to get candidates POIs near the target
location.

1. `googleMapsClient.geocode(query).asPromise()`:
This function looks up the geometric code of the location of the location by using the name of
the location like ‘Ottawa’. It returns the code which is comprised of a longitude and latitude
through a promise so that the next function can execute. Promises are joins in asynchronous
calls. Promise.all([...]) takes an array of promises and works like a fork. Each promise of the
array is delegated to some I/O handler on the operating system. When all the promises
(forks) are done, the results are joined into the next task of the algorithm.

2. `googleMapsClient.placesNearby(request).asPromise()`:
The parameter of this function `request` is an object defined by some properties shown below:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>LatLng</td>
<td></td>
<td></td>
</tr>
<tr>
<td>language</td>
<td>String</td>
<td>&lt;optional&gt;</td>
<td></td>
</tr>
<tr>
<td>radius</td>
<td>number</td>
<td>&lt;optional&gt;</td>
<td></td>
</tr>
<tr>
<td>keyword</td>
<td>string</td>
<td>&lt;optional&gt;</td>
<td></td>
</tr>
<tr>
<td>minprice</td>
<td>number</td>
<td>&lt;optional&gt;</td>
<td></td>
</tr>
<tr>
<td>maxprice</td>
<td>number</td>
<td>&lt;optional&gt;</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>boolean</td>
<td>&lt;optional&gt;</td>
<td></td>
</tr>
<tr>
<td>opennow</td>
<td>boolean</td>
<td>&lt;optional&gt;</td>
<td></td>
</tr>
<tr>
<td>rankby</td>
<td>string</td>
<td>&lt;optional&gt;</td>
<td>Either ‘prominence’ or ‘distance’</td>
</tr>
<tr>
<td>type</td>
<td>string</td>
<td>&lt;optional&gt;</td>
<td></td>
</tr>
<tr>
<td>pagetoken</td>
<td>string</td>
<td>&lt;optional&gt;</td>
<td></td>
</tr>
</tbody>
</table>

**Table 1.** Properties of `placesNearby` function
The location of the `request` is set as the geocode returned. The default radius is 1 kilometer
and currently, the application does not support customized radius. The maximum number of
nearby places returned by this function is twenty. Normally, if the user searches within a
reasonably sized city, the list of recommended location always has a size of 20. This
function also returns as a Promise, so when this function returns, the result is sent to the client side for the user to select.

3. `googleMapsClient.place(query).asPromise()`:
The properties of parameter `query` are listed below:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>placeid</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>language</td>
<td>string</td>
<td>&lt;optional&gt;</td>
<td></td>
</tr>
</tbody>
</table>

*Table 2. Properties of `place` function*

For each location the user selects, the `planningService` looks up the place details information. There are \( n \) Promises added to a list. When all of the functions return, it catches by `Promise.all()`. After the series of promises return, the `planningService` starts to generate the plan through either static planning algorithm or A* search algorithm. When the `planningService` finishes calculating the results, the results are sent to the client and displayed to the user.
3.4.2 Use Case

The following section shows a use case of the application. This section provides screenshots about steps users need to take to generate a plan.

![Input boxes on the home page.]

When users launch the application, there are three input boxes on the homepage for them to enter inputs. Users need to enter a destination they are traveling to, a date the plan to generate for, and a time users want the plan to start. Figure 10 shows the user interface (UI) for users to enter the information.
Users can enter any points of interest or geographic locations since the full query string is sent through an asynchronous POST request to the server. The server first forwards the query string to the GoogleMapsAPI and tries its best to locate the latitude and longitude of the POI or geographic location. Once this call returns, a second service call to the GoogleMapsAPI call is sent to get ‘points of interest’ near the latitude and longitude, which is capped to a maximum of 20 locations near the point. The data for these locations are then packaged into the response and sent back to the browser. When this response returns, the browser will render the map pins to highlight each of the candidate locations on the embed Google Maps panel. Figure 11 is a screen for ‘points of interest’ in Ottawa region.
The same list of the locations which were returned in the nearby search is rendered into a separate panel on the screen. Users can check the boxes to include the corresponding location into the plan. Figure 12 selects three locations in the list to generate a plan. When users click the 'Submit' button, a plan generating request is made by the browser to the server. As we see above, the final plan will consist of the three locations selected.
Figure 13. Display recommended plan page

In the ‘Recommended Plan’ page will display two plans that are generated by different algorithms. The first table displays the plan which is generated by the Static Planning Algorithm, while the second table displays the plan generated by A star Planning algorithm. On the first column in the table, the suggested visiting order is shown. The third column of the table displays the calculated score of the plan generated by the corresponding algorithm. Figure 13 shows two sample plans generated. The associated scores are the result of the algorithms for which we have described in section 3.1.1 and 3.1.2.
4. Evaluation

This section is dedicated to evaluating the algorithms we employ, the system, and the performance perspectives.

4.1 Evaluating the Algorithm

4.1.1 Static Planning Algorithm

The highlight of this algorithm
This algorithm can guarantee to find the optimal solution if there is one.

The downsides of this algorithm
The running time of this algorithm is not optimal since it will generate and evaluate all the possible plans. When there are more than ten locations, the running time for the algorithm will increase exponentially. Additionally, this algorithm is not flexible with edge cases like when there are two plans having the same score, it won't be able to determine which plan is a preferable one.

4.1.2 A* Algorithm

The highlight of this algorithm
This algorithm guarantee to find the optimal solution when the heuristic function is admissible. The A* Searching Algorithm speeds up searching in the state space as well as guaranteeing to find the optimal solution.

Comparing with Depth-first search and Breadth-first search
Other two possible search algorithms can be used are Breadth-first search (BFS) and Depth-first search (DFS).

A breadth-first search always expands the shallowest unexpanded node. BFS can guarantee to find the optimal solution. However, the disadvantage of the BFS is the space it uses. It uses $O(b^{k+1})$ space because it needs to keep every node in memory, where b stands for the branching factor. Besides, BFS is inefficient if branching factor is very high.
A Depth-first search always expands the deepest unexpanded node. Comparing with BFS, DFS uses linear space which is much smaller than the space usage of BFS. It is also quite efficient when solution path is known to be long. Additionally, DFS sometimes cannot guarantee to find the optimal solution. It may also fail in infinite-depth spaces since it keeps expanding to the deepest node.

Comparing with other possible algorithms

**Travelling salesperson problem**

In section 2.2, we have talked about how the TSP related to the problems of this application. However, the algorithm does not perfectly fit our situation. Firstly, the original TSP algorithm needs to go back to the origin city but in our plan, we only need to visit each city once. Secondly, the cost of each city is static. In our planning, we cost between each city dynamic changes because it changes when the visiting order changes. For example, the total distance traveled from points A-B-C is the same as the distance of city C-B-A, but the scores for them are different because if destination C is visited last, it may be closing time. Then the score for the plan with place order A-B-C has a higher score. Because of this reason, the accuracy of the score generated by the evaluation function may be influenced.
Dijkstra’s algorithm (single-source shortest path problem)

Dijkstra's algorithm is designed to solve the problem of a single-source shortest path on a non-negatively weighted graph [6]. The planning algorithm while using Dijkstra’s algorithm strategy is to use the evaluation function to calculate the distance between two nodes.

![Graph](image)

**Figure 16.** An example of a graph constructed for Dijkstra’s algorithm

Though, there are few problems with Dijkstra’s algorithm. The first problem is that for Dijkstra’s algorithm, we need to determine the start node. In our application, we can start from any destinations. To ensure a guarantee that it can find the best plan for all the possible plans, we need to have more than one possible starting points. This means we will need to apply this algorithm n times when n indicates the number of destinations we need to visit, and the source of each graph will be set to the first destination we would like to visit. Another problem with Dijkstra’s algorithm is that there cannot exist negative weight on the graph. In our application, the evaluation function does not guarantee to return a non-negative score due to hard constraints of each destination.

The downsides of A* algorithm

A* algorithm does not necessarily enforce the hard constraint depicted when a destination is closed. It always returns a plan nevertheless if the plan has a negative score in the case where the destination is currently closed for example. It is also not flexible enough on the number of destinations you can visit because it always includes all destinations within the plan.
4.2 Evaluating the System

4.2.1 Scalability

Every time a user enters a location, a series of requests needed to be made to the Google Maps APIs to get the information about the nearby locations, place detail information, or geocode of a location etc. The Google Maps API has a standard usage limit of 50 requests per second and 2,500 requests per day, calculated as the sum of client-side and server-side queries [13]. On average, each user needs to make 8 to 12 requests assuming each user selects 3 to 5 visiting destinations. The application is capable of serving 4-6 users making requests at the simultaneously within a second and 200 to 300 users per day. Anymore, we will need to pay Google for using their map services.

4.2.2 Usability

In order to study the usability of the application, we conducted a usability trail with twelve people. Each interviewer is given a brief introduction about what this application does and how to use the application. The following is the results and analysis of the results were recorded. The questionnaire used in this survey is included in the Appendix.

![Figure 17. Bar graph of Q1 results](image)

The first question in the questionnaire asks interviews how often they would like to use the application while traveling. Two of the interviewers say this application is a useful tool for them when they are not familiar with the place they are going to visit. Also, almost half of the interviewers among who select ‘Everytime’ and ‘Often’ claim that it is a good tool to use when you run out of ideas about where to go. For the interviewers who choose ‘Rarely’ and ‘Never’, the common feedback they provide is that they do not think they need to plan ahead of the trip. Also, they mention that this application is not mandatory to use because it does
not have the features like booking hotels, flights or renting cars, which are more critical for a trip.

**Figure 18.** Pie chart of Q2 results

Question 2 asks about if the users like the plan the application generated. Four out of twelve people say that they are not quite satisfied with the plan. Two out of four interviewers mention the reason as the plan is too intensive. Both of them select 4-5 destinations for generating the plan. One of the interviewers provided a negative feedback because most of the destinations proposed in his plan are closed according to the time when the plan was built. The last interviewer did not like the plan because there are very few destinations on the available list interested him.

**Figure 19.** Pie chart of Q3 results

For this question, seven out of twelve interviewers are satisfied with the locations recommended while five out of twelve interviewers are not. One interviewer provides the feedback that the application does not provide the functionality to create a new list of recommended location if the user does not like the current one. Other interviewers who
provided negative feedback explain that there too many hotels recommended in the list, but hotels are not considered as points of interests to them.

**Figure 20.** Pie chart of Q4 results
For this question, the application crashed while the Google Maps failed to respond. The application froze when it's waiting for the response. The application restores normally after refreshing the page.

**Figure 21.** Pie chart of Q5 results
For this question, most of the interviewers are satisfied for the response time of the application. The only negative feedback is given by the interviewer who crashed the application due to the failure of getting a response from the Google Maps API service. Deferred script loaded may have contributed the response time of the web page.
Every interviewer claims that this application is easy to use. Most of them can generate the plan without asking me more than two questions after the brief introduction. Clearly, the UI is easy to use.

Figure 23. Bar graph of Q8 result

Most of the interviewers spend less than 5 minutes to set up their plans. For the interviewers which spent more than five minutes, they claim that they spent most of the time to look at the options provided in the recommended list, and it is not because the application is difficult to use.

For question 9, interviewers respond that they would like more features added. For example, as we mentioned in question 3, interviewers state it would be better to add a filter to filter the choices that users want to visit. Also, they would like the planning functionality can be customized. For instance, they can add a break time between two destinations.
4.3 Performance

The average response time of each request is about 0.03 second. When users are trying to get a candidate list of locations, the response time is seamless. However, when the number of destinations is more than 5, the user is expected to experience a short period of waiting time around 0.5 seconds to 1 second.
5. Conclusion

5.1 Summary
The Smart Traveling Application uses the Static Planning algorithm to solve the problem and also adopt A* algorithm to increase the efficiency. The application uses Google Maps API to retrieve information to promote a plan with higher rating destinations and best ordering. The application can be used on any platform and location where the required frameworks and Google Maps API service support.

5.2 Limitations
As we mentioned in section 4.2.1, because of the fact that the Smart Traveling Application strongly relies on the information the Google Maps APIs provide, there is a limitation on the number of users can use the application at the same time. It also draws a limitation for the total number of users can use the application per day.

5.3 Future work
In this section, we discuss about the future work and potential optimizations can be done to improve the application.

Persistent Storage
In the current application, the evaluation for every new request or selection made by the user, the application needs to make an API call to get corresponding information in order to calculate the cost. Since the application is making the calls a public API, the waiting duration for the response is quite elongate. If the frequently used information can be cached or stored in the database for the application, it can decrease the response time by avoiding waiting for the response from API calls.

Besides, another potential optimization can be done is reducing the dependency on a single API service. The application only uses the Google Maps API to currently which is not optimal, because Google has an upper limit on the number of requests it handles each second. Also when Google service is not available, the application cannot function. If the application can have an alternative service it can use, there will be a backup option when Google Map is not available.
6. Appendix

6.1 List of Figures

Figure 1. An example of travelling salesperson graph with the solution on the right
Figure 2. An overview of planning
Figure 3. The state space when number destinations equals three.
Figure 4. An example with heuristic function cost and actual cost for all possible states.
Figure 5. The AMD model.
Figure 6. An example demonstrating the usage of a variable syntax in Less.
Figure 7. An example for Less using mixins and nested rules.
Figure 8. An example explaining how the operators and functions will be compiled from Less to CSS.
Figure 9. The sequence diagram for a user generating a plan
Figure 10. Input boxes on the home page.
Figure 11. Candidate locations on google map
Figure 12. Selecting page
Figure 13. Display recommended plan page
Figure 14. The order of BFS expands nodes.
Figure 15. The order of DFS expands node.
Figure 16. An example of a graph constructed for Dijkstra’s algorithm
Figure 17. Bar graph of Q1 results
Figure 18. Pie chart of Q2 results
Figure 19. Pie chart of Q3 results
Figure 20. Pie chart of Q4 results
Figure 21. Pie chart of Q5 results
Figure 22. Pie chart of Q6 results
Figure 23. Bar graph of Q8 result

6.2 List of Tables

Table 1. Properties of placesNearby function
Table 2. Properties of place function
6.3 Usability Questionnaire

Smart Traveling Application Questionnaire

1. How often do you think you will use STA when you go traveling?
   Every time          Often          Rarely          Never

2. Do you like the plan STA generated for you?
   Yes                  No

3. Do you like the locations recommended in the list?
   Yes                  No

4. Were you able to crash the app?
   Yes                  No

5. Is the waiting time reasonable?
   Yes                  No

6. Is the application easy to use?
   Yes                  No

7. If you choose ‘No’ for question 6, please explain________________________.

8. How long did it take you to generate a plan for your destination?
   < 1 min          1 min to 5 mins          5 mins to 10 mins          >10 min

9. Do you have any feedbacks you would like to provide?
   ____________________________________________________________.
Reference


