Smart Path Planning in Flooded Regions Based on Route APIs

Ya-Hui Chang\textsuperscript{a}, Wei-Chi Hung\textsuperscript{a}, Shu-Han He\textsuperscript{a} and Chuan-Ming Liu\textsuperscript{b}

\textsuperscript{a}Department of Computer Science and Engineering, National Taiwan Ocean University, Keelung, Taiwan; \textsuperscript{b}Department of Computer Science and Information Engineering, National Taipei University of Technology, Taipei, Taiwan

ABSTRACT

The disaster brought by heavy rain has become more and more serious due to global warming. The intention of this paper is to automatically produce a route which does not pass through flooded areas to ensure the safety of travelers. We explore the techniques of using cloud services, which are very popular recently due to their efficiency and availability. The basic idea of the proposed approach is to first get the usual best path from the online route API provided by cloud services. If any portion of this route is flooded and unpassable, we identify their nearby unflooded intersections. From all the candidate sets, we then pick those which can form the shortest path between the source and the target locations, and invoke the route API again with the chosen intersections as waypoints to re-plan the route. Spatial indices and corresponding algorithms are designed to expedite the above process. We have implemented the proposed approach and conducted extensive experiments with real and synthetic data-sets. Experimental results indicate the feasibility and efficiency of the proposed method.

1. Introduction

The disaster brought by heavy rain has become more and more serious for the past few years due to global warming. It is important to build a flood forecasting system for early warning [1]. After knowing where will be flooded soon, the authorities should evacuate residents in time and the evacuation path should avoid those flooded areas. A possible way to get the appropriate path is to first collect all the road information, discard those flooded roads, and use the remaining roads to run the common shortest path algorithm, such as the Dijkstra [2] and A* [3] algorithms. However, it is not an easy task to build such a system, since it requires to maintain all the detailed spatial and road data for the whole region.

In contrast, the cloud service for route planning has been widely used recently. By giving the current position and the destination, the cloud service will recommend the best route, either the shortest or the fastest one. The popularity of such planning services is mainly gained from their efficiency, friendly interfaces, and availability for a variety of devices. They also provide the online route API (Application Programming Interface), e.g. Google Directions API.\textsuperscript{1} Many location-based services (LBSs) adopt such APIs to lessen the burden of maintaining huge amounts of spatial data and to obtain the most current information [4,5]. These route APIs usually allow one to give extra conditions, such as the mode of traveling, e.g. driving or walking, or the places one must pass by, referred as waypoints. However, seldom cloud services can let the
and cloud services, and propose a corresponding architecture.
- We design indices and corresponding algorithms to find alternatives for flooded roads and re-plan the route.
- We have implemented the proposed approach and evaluate its performance through extensive experiments. Empirical studies show that the proposed approach is effective and efficient in getting the alternative unflooded route.

The rest of this paper is organized as follows. In Section 2, we formally define the problem and explain the idea underlying the proposed approach. In Section 3, we introduce the whole cloud-based system and how to process the original route returned by the API to form a list of proper segments. In Section 4, we describe how to adjust the route to avoid flooded roads. Finally, experimental results are shown in Section 5, related works are compared in Section 6, and conclusions are given in Section 7.

2. Problem Definition

In this section, we define the terms used in this paper and formulate the problem to solve. For simplicity, we assume that the route API returns the shortest path between two given locations, and the flooded areas are formed by a set of blocks, as the grid shown in Figure 1.

Definition 1: A road network \( G = (V, E, W) \) is a directed connected graph with vertices \( V \) and weighted edges \( E \). Each vertex is assigned a unique identifier, and each weight in \( W \) represents the length of the corresponding edge.

In this paper, the terms vertex and intersection are used interchangeably, so are the terms edge and road. Based on the definition of the road network, the following definition describes the ideal path in a flooded region:

Definition 2: Given a road network \( G \), a set of flooded blocks \( \{F\} \), the source vertex \( s \), and the target vertex \( t \), the

Figure 1. Example paths returned by route APIs (a) the original flooded path (b) the adjusted unflooded path. (Map data: Google).
Unflooded Shortest Path, denoted as USP(s, t), is the shortest path between s and t where none of the component edges of USP(s, t) intersect F_i.

However, the above definition needs to be modified to cope with the characteristics of the cloud environment. First, the term point is used to represent a particular location on the map as follows:

Definition 3: Given a road network G = (V, E, W) on a map, the term point refers to a location on a road, which may be or may not be a vertex. Each point p_i is identified by a pair of (longitude, latitude) values, or called (x, y) values for simplicity.

It is obvious that the vertex set V is a subset of the set of points. We then give the following definitions for the cloud environment:

Definition 4: Given the source point s and the target point t, the path returned by a route API, called a Cloud Route for simplicity and denoted as CR(s, t), is formed by a point list \{p_1, …, p_n\}, where p_1 is s and p_n is t. We also call the line connecting p_i and p_{i+1}, denoted as (p_i, p_{i+1}), a segment of the path.

Definition 5: Given a set of flooded blocks \{F_i\}, the source point s and the target point t, the cloud route formed by \{p_1, …, p_n\} is an Unflooded Cloud Route for s and t, denoted as UCR(s, t), if p_1 is s, p_n is t, and none of the segments \{p_i, p_{i+1}\} intersect F_i.

To avoid heavy computation, the intention of this paper is not to get the real shortest one among all UCR(s, t), but to look for the best adjusted UCR. Specifically, if a segment \{p_i, p_{i+1}\} within the original CR(s, t) is flooded, we obtain their nearby intersections, respectively. We then choose one point from each set of candidate intersections as the basis of the alternative route. The example depicted in Figure 2 illustrates such an idea, where the road network consists of 9 vertices, 11 edges, and a flooded block.

Suppose \(v_i\) is the source vertex, \(v_j\) is the target vertex, and the original shortest path is \([v_i, v_2, v_3, v_6, v_9]\), with the path length as 11. Since the edge \((v_3, v_4)\) is flooded, we look for the intersections near \(v_2\) and \(v_3\), respectively, which are enclosed by dashed lines. From the two candidate sets, we pick \(v_3\) and \(v_6\), respectively, as alternatives, and the adjusted path becomes \([v_i, v_3, v_6, v_9]\), with the length as twelve. Note that the adjusted one is a little bit longer than the original one, but can safely avoid the flooded area. An obvious question will be how to choose the point from each candidate set, and we will explain our method in Section 4.

3. The Cloud-based System

In this section, we show the whole architecture of the path planning system which is constructed using the online route API. We will also discuss how to process the original cloud route returned by the API for performance improvement.

3.1. The System Architecture

Figure 3 shows the proposed path planning system based on route APIs. When the user inputs the source point s and the target point t, the Cloud Module will be first invoked to get the path recommended by the cloud service. If none of the segments of this path are flooded, it will be directly displayed to the user. Otherwise, the Adjustment Module will be invoked to find the nearby unflooded intersections for those blocked roads, get alternative ones as waypoints, and deliver them to the cloud service for planning a new path. The functionality of waypoints is originally designed to let the cloud route definitely pass through certain locations. In our system, they are used to force the re-planned route to pass through those chosen unflooded intersections and avoid the original flooded roads.

Besides, two indices are constructed in this system to increase efficiency. The first one is an R-tree index built on all the flooded blocks, which is referred as Flood Index. When a cloud route is produced, we will perform an intersection operation for each segment against this index to determine if it is flooded. The other index is used to quickly find the nearby intersections and will be explained in Section 4.

3.2. Producing Segment Lists

Given a source point and a target point, the cloud route returned by the route API consists of a list of points
specified by the longitude and latitude values. In addition to the obvious intersections, the list might include some points mainly for depicting purposes. For instance, suppose the source point is (121.477434, 25.077355) and the target point is (121.483442, 25.077423). Their distance is 700 m. The cloud route returned by the Google Maps route API actually consists of up to fifteen points, which are depicted in Figure 4. We can observe that some points are very near, especially around the corner or when the actual road is not very straight, e.g. v_4–v_7. If we let all the adjacent points in a cloud route form segments and utilize them to identify flooded locations, the resultant candidate sets of nearby intersections might overlap seriously, since it is very likely that the two endpoints of a short segment identify the same nearby intersection. Such redundancy is unnecessary and affects the performance of later operations.

To solve this problem, we process the cloud route to form a list of segments with proper lengths. The idea is to only keep the first and the last points for a list of adjacent points specified by the longitude and latitude values. In addition to the obvious intersections, the list might include some points mainly for depicting purposes. For instance, suppose the source point is (121.477434, 25.077355) and the target point is (121.483442, 25.077423). Their distance is 700 m. The cloud route returned by the Google Maps route API actually consists of up to fifteen points, which are depicted in Figure 4. We can observe that some points are very near, especially around the corner or when the actual road is not very straight, e.g. v_4–v_7. If we let all the adjacent points in a cloud route form segments and utilize them to identify flooded locations, the resultant candidate sets of nearby intersections might overlap seriously, since it is very likely that the two endpoints of a short segment identify the same nearby intersection. Such redundancy is unnecessary and affects the performance of later operations.

To solve this problem, we process the cloud route to form a list of segments with proper lengths. The idea is to only keep the first and the last points for a list of adjacent

**Algorithm FormSegment**

**Input:** a cloud route PointList, the merging threshold mt

**Output:** a segment list SegList

```
1: i ← 0; SegList ← null
2: start ← PointList[i]; NearFlag ← FALSE
3: while i < |PointList| - 1 do
4:     i ← i + 1
5:     if dist(PointList[i-1], PointList[i]) < mt then
6:         NearFlag ← TRUE
7:     else
8:         if NearFlag = TRUE then
9:             SegList.add(start, PointList[i-1])
10:        end if
11:     SegList.add(PointList[i-1], PointList[i])
12:     start ← PointList[i]
13:     NearFlag ← FALSE
14: end if
15: end while
16: if NearFlag = TRUE then
17:     SegList.add(start, PointList[i])
18: end if
```

**Figure 4.** A sample cloud route. (Map data: Google).

**Figure 5.** Algorithm FormSegment.
points which are all very near. As the above example continues, only one segment \((v_i, v_j)\) is produced, instead of three segments formed by all adjacent points. The corresponding algorithm is listed in Figure 5. It sequentially scans the input point list, and forms a segment if the distance of two adjacent points is larger than the merging threshold \(nt\) (L11). Note that the function \(\text{dist}\) returns the distance between two input parameters (L5) and the variable \(\text{NearFlag}\) is used to signal that there exists a sequence of near points (L6). The first point of the corresponding segment is represented by the variable \(\text{start}\), which is either the first point of the input list (L2) or the last point of the previous segment (L12). The last point of a segment will be identified either when the following point is distant enough to form a new segment (L7–9), or when the end of the input list is reached (L17).

4. The Adjustment Module

After producing a list of segments from the original cloud route, we identify unpassable ones based on Flood Index as described in Section 3. In this section, we explain how to further process those flooded segments for route adjustment. The main idea is to first find their nearby intersections and then re-plan the path based on the chosen ones.

4.1. Identifying Nearby Intersections

The nearby intersections (denoted as \(\text{NI}\)) for a particular point \(p\) on the map is defined as follows:

**Definition 6**: Given a road network \(G = (V, E, W)\) on a map, a particular point \(p\), and a near threshold \(nt\), \(\text{NI}(p) = \{v_1, \ldots, v_n\}\), where each \(v_i \in V\) and \(\text{dist}(p, v_i) \leq nt, 1 \leq i \leq n\).

Specifically, \(\text{NI}(p)\) represents a set of vertices whose distances to \(p\) are all within the threshold \(nt\). To determine if a vertex \(v\) is in \(\text{NI}(p)\), we calculate the distance between \(v\) and \(p\) and see if it is smaller than the near threshold \(nt\). As illustrated in Figure 6, the dashed circle is centered at a particular point \(p\), with the threshold \(nt\) as the radius, where \(i = 1\) or \(2\). We can see that vertex \(v\) is located within the dashed circle centered at \(p_1\) in Figure 6(a), but outside the dashed circle centered at \(p_2\) in Figure 6(b). Therefore, \(v\) is in \(\text{NI}(p_1)\) but not in \(\text{NI}(p_2)\).

A straightforward way to obtain the set \(\text{NI}(p)\) is to compute the distances between all vertices in \(V\) and \(p\), and determine which vertex has the distance to \(p\) smaller than the near threshold \(nt\). To avoid such a time-consuming task, we propose to build an R-tree index on all the intersections, i.e. all the vertices of the road network, to streamline the search space. For each node of the index, in addition to the minimum bounding rectangle (MBR) covering all the component objects as the traditional approach, we also consider the circumcircle of that rectangle such as the dotted circles depicted in Figure 6. Suppose the MBR of a node \(n\) is centered at \(c\) and has the diagonal \(2r\). For the vertex \(v\) covered by node \(n\) and subsequently some of \(n\)'s descendants, \(v\) will NOT be in \(\text{NI}(p)\), if the following inequation holds:

\[
\text{dist}(p, c) > nt + r. \tag{1}
\]

As depicted in Figure 6(b), if the dotted circle and the dashed circle do not overlap, we can directly skip this index node and all its descendants. In contrast, if the two circles are near enough as depicted in Figure 6(a), there might exist a vertex like \(v\) which satisfies the threshold constraint, and we need to explore this node further. This index, called Near Index, is implemented based on the R-tree structure, and each node of the index has additional attributes \(\text{center}\) and \(\text{radius}\) to represent the circumcircle of the associated MBR. If the node is a leaf, it will have the attribute \(\text{vertices}\) to represent the vertices covered by this node. Otherwise, it will have the attribute \(\text{childnode}\) pointing to all its children. The algorithm utilizing Near Index to find nearby intersections for flooded roads is listed in Figure 7. It recursively traverses the index from top to bottom. In L2 of Algorithm FindNI and L1 of Algorithm RQ (RangeQuery), Inequation (1) is applied to determine if we should explore the descendants of this node further. If a leaf node of the index is reached (L4 of Algorithm RQ),
Therefore, with this index, we only need to examine the vertices represented by the left two leaf nodes, instead of all vertices.

### 4.2. Determining Alternative Intersections

Note that we do not directly identify the nearest intersection in the previous subsection to re-plan the path, since locally optimal points do not necessarily contribute to the globally optimal path. An example is illustrated in Figure 9. Suppose the source point is $s$, the target point is $t$, and the shortest path between them is $[s, p, q, t]$. If

**Algorithm FindNI**

**Input:** a point $p$, Near Index $R$, the near threshold $nt$

**Output:** NI($p$)

1: NI($p$) := {};
2: if dist($p$, R.root.center) > $nt$ + R.root.radius then
3:     return NI($p$)
4: end if
5: for each $c$ $\in$ R.chilnode do
6:     NI($p$) = NI($p$) $\cup$ RQ($p$, c, $nt$, NI($p$))
7: end for

**Algorithm RQ**

**Input:** a point $p$, a node $N$, the near threshold $nt$, NI($p$)

**Output:** the updated NI($p$)

1: if dist($p$, N.center) > $nt$ + N.radius then
2:     return NI($p$)
3: end if
4: if $N$ is a leaf node then
5:     for each $v$ $\in$ N.verts do
6:         if unflooded($v$) and dist($p$, $v$) < $nt$ then
7:             NI($p$).add($v$)
8:         end if
9:     end for
10: else
11:     for each $c$ $\in$ N.chilnode do
12:         NI($p$) := NI($p$) $\cup$ RQ($p$, c, $nt$, NI($p$))
13:     end for
14: end if

**Figure 7.** Algorithm FindNI.

we will directly examine each vertex covered by this node and pick out those unflooded and near ones.

For the example road network in Figure 2, we assume that each MBR covers three vertices. The road network along with the three MBRs are shown in Figure 8(a), where $M_1$ is the MBR of the vertices $v_1$, $v_2$, and $v_3$. Figure 8(b) shows the corresponding Near Index, where the order of each node is set as three, and its circumcircle is represented by the center $c$ and the radius $r$. Assume that the road $(v_2, v_5)$ is flooded and we need to find alternative intersections for $v_5$. Observe that the dashed circle centered at $v_2$ intersects the circumcircles of $M_1$ and $M_2$.

**Figure 8.** A sample Near Index (a) the road network with MBRs (b) the index.
the segment \((p, q)\) is blocked and we wish to identify the next shortest path without passing through it, we obtain all the nearby vertices for the two endpoints and get \(NI(p)\) as \([p_1, p_2]\) and \(NI(q)\) as \([q_1, q_2]\). Although \(p_1\) and \(q_1\) are the vertices nearest to \(p\) and \(q\), respectively, they are pretty far away and the length of the path \([s, p_1, q_1, t]\) is 14. Instead, the path \([s, p_2, q_2, t]\) only has the length 13 and is the best alternative. Therefore, to choose the vertices contributing to the globally optimal solution, we adopt the formula from [7], which is based on the idea of dynamic programming and is specified as follows:

\[
X(i, j) = \begin{cases} 
0, & \text{if } i = 0 \\
\min_{1 \leq L \leq |C_i|} \{ X(i-1, L) + \text{dist}(c_{i-1, L}, c_{i, j}) \}, & \text{if } i > 0
\end{cases}
\]

(2)

In our setting, the footnote \(i\) corresponds to a flooded vertex \(v_i\), \(C_i\) represents \(NI(v_i)\), and the footnote \(j\) identifies an element \(c_{i,j}\) in \(C_i\). In the algorithm FindALTP (Find ALTERNative Points) listed in Figure 10, we first construct the matrix \(C\) based on all \(c_{i,j}\)’s. Assuming that there exist \(k\) flooded points, \(k\) rows will be created to represent the nearby intersections of these flooded points, respectively (L3-L10). Besides, the source point is represented as the first entry of the first row (L1), and the target point is represented as the first entry of the last row (L11). Based on the matrix \(C\), Formula (2) is applied to compute the matrix \(X\) (L12). As proved in [10], \(X[k + 1][0]\) will represent the length of the shortest path, and we can get those points which form the shortest path by backtracking from there.

5. Experimental Study

We have designed several experiments to evaluate the performance of our cloud-based system, which is called CloudV. Note that although the waypoints are unflooded themselves, the alternative route planned by the route API might pass through other flooded areas. Therefore, we allow the system to repeat executions based on different nearby intersections at most three times. As a comparison, we also implement another system, called CloudV−, which can only perform path planning once.

5.1. Experimental Set-up

The programs of our system are implemented in Visual C++ and C#, and the experiments are performed on a personal computer with Intel i5-3570 3.4 GHz CPU and 8 GB memory, with the Microsoft Windows 7 operating system and IIS Web Server. The experiments are conducted on synthetic and real flooded blocks, where the real ones are got from the official Web site.4 There are also three road networks used in experiments as listed in Table 1. We randomly create several pairs of source points and target points for path planning, and average their execution time.

![Figure 9. An example of globally optimal points.](image)

![Figure 10. Algorithm FindALTP.](image)

**Algorithm FindALTP**

**Input:** \(v_s, v_t\), the endpoints of flooded segments \(fpSet\), Near Index \(R\), the near threshold \(nt\)

**Output:** the matrix \(X\)

1: \(C[0][0] \leftarrow v_s\)
2: \(i \leftarrow 1\)
3: for each \(f_e \in fpSet\) do
4: \(j \leftarrow 0\)
5: \(NI(f_e) \leftarrow \text{FindNI}(f_e, R, nt)\)
6: for each \(e \in NI(f_e)\) do
7: \(C[i][j++] \leftarrow e\)
8: end for
9: \(i++\)
10: end for
11: \(C[i][0] \leftarrow v_t\)
12: \(X \leftarrow \text{DP}(C)\)
in Figure 11(a), Algorithm FindNI is not affected by the near threshold much since the proposed Near Index can help identify nearby intersections efficiently. However, we can see Algorithm FindALTP requires a lot more time when the near threshold increases. The main reason is that it applies Formula (2) to choose the globally optimal points and is very sensitive to the input size, which in turn makes the total execution time sensitive too.

We then examine how the near threshold affects the success rate, and contrast the performance of CloudV with that of CloudV-. As shown in Figure 11(b), both systems increase the success rate along with the increase of the near threshold, since the size of candidate sets will be enlarged along with the value and the route API has more choices for path planning. It is also evident that CloudV has a better success rate than CloudV since it may execute more times for sure. From these two figures, we can see that the near threshold 1.5 km with up to three execution times can almost get a successful path every time, but it is quite slow. If we wish to get the route faster, running the CloudV system with the near threshold 1 km may be the best choice, since it can output a path around 1.5 s and has the 80% success rate.

5.2. Effects of the Near Threshold and the Merging Threshold

In this section, we investigate how the two thresholds affect the performance of our system. Recall that the near threshold is applied in choosing nearby intersections, and the merging threshold helps determine if two adjacent points of a cloud route are far enough to form a segment by themselves. Performance comparison is based on two metrics, the execution time and the success rate, where the latter measures how often the system can successfully produce an unflooded cloud route.

The experiment is performed with the roads in Taipei. We first examine how the near threshold affects the execution time, particularly that of Algorithm FindNI (denoted as Cand) and that of Algorithm FindALTP (denoted as Alt). The execution time of the remaining operations is summed up and denoted as Other. As shown

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|c|}
\hline
City & # of Vertices & # of Roads & # of Flooded blocks \\
\hline
Keelung & 1464 & 1600 & 148 \\
Hsinchu & 10151 & 11526 & 330 \\
Taipei & 18872 & 21652 & 655 \\
\hline
\end{tabular}
\caption{Data-sets.}
\end{table}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure11}
\caption{Effects of the near threshold (a) execution time (b) success rate.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure12}
\caption{Effects of the merging threshold (a) execution time (b) success rate.}
\end{figure}
We then perform the similar set of experiments on the merging threshold. Recall that the larger this threshold is, the smaller the number of produced segments is. As shown in Figure 12(a), the increase of the merging threshold does not affect the time of producing segments (Cand), but will reduce the execution time of the later operation (Alt) and the total execution time. For another metric, as shown in Figure 12(b), the success rate of CloudV significantly drops when the merging threshold increases from 50 m to 100 m, but does not change afterwards. The reason is that the number of segments does not necessarily have direct relationships with the number of candidate intersections, and in turn the produced path. Since using a larger merging threshold will decrease the execution time to a large extent, the value 200 m may be the best choice when taking both factors into account. Based on the results of the experiments above, we will fix the near threshold as 1 km and the merging threshold as 200 m in the following experiments.

5.3. Scalability Study on the Number of Roads

In this section, we investigate how the size of the road network affects the execution time. We apply the three data-sets as listed in Table 1 and fix the number of flooded blocks as 148 for all cities. Note that the numbers of roads in the three data-sets are roughly in linear scale.

To compare the time required by the major modules of our system, in addition to the time of identifying nearby intersections for flooded roads (Cand) and choosing the globally best alternative ones (Alt) as before, we measure the time of determining if any segment of the cloud route is flooded (denoted as Flood). Besides, recall that Near Index is an R-tree built on vertices. As a comparison, we build another R-tree based on roads, which are line segments instead of points. The execution time of the original system and its component modules will be ended with the letter ‘V’, while those of the new one will be marked using ‘R’.

The experimental results are depicted in Figure 13(a). We can see that the Alt module still dominates all modules due to the heavy computation required by dynamic programming. In contrast, the other two modules are pretty efficient and insensitive to the size of the input, with the help of the underlying indices. Note that Near Index only affects the execution time of the Cand module, and we can see that using the original index (CandV) takes less time than using the new index (CandR). The reason is that roads usually span a lot of space, which makes the ranges

![Figure 13. Scalability study on the number of roads (a) time of each module (b) total execution time.](image1)

![Figure 14. Scalability study on the number and distribution type of roads (a) random distribution (b) clustered distribution.](image2)
covered by index nodes overlap a lot, and causes more traversal time when processing a query.

Figure 13(b) illustrates the total execution time, and we can see that it is roughly linear to the number of roads, in sync with the time required by the Alt module. However, even for a big city, like Taipei, which consists of many roads, the route can be computed and displayed to the user within one second. It is pretty acceptable.

5.4. Performance Study on the Number and Distribution Type of Flood Blocks

Recall that flooded blocks are used to build Flood Index, which is applied to determine if a road is flooded or not. In this section, we study how the number of flooded blocks affects the total execution time. To perform the scalability test more easily, we produce synthetic flooded blocks with the number ranging from 100 to 200. We also investigate the effect of three different types of distributions, which are random, gaussian, and clustered. The experimental result of random flooded blocks in the Taipei road network is depicted in Figure 14(a). The execution time does not differ a lot when the number of blocks increases from 100 to 150, since the heights of the two corresponding indices are similar. Although the execution time increases more when the number of flooded blocks reaches 200, our original system, i.e. building Near Index on vertices (CloudV), is more efficient than the contrasted system whose Near Index is constructed on roads (CloudR).

Finally, we perform the same set of experiments on the Gaussian and clustered data-sets, respectively. The experimental result of the clustered data-set is shown in Figure 14(b), where the execution time is also insensitive to the size of the input. The result of the Gaussian data-set is similar and omitted here.

6. Related Work

This paper investigates the techniques of using cloud services and route APIs. They are adopted by many LBSs for getting the up-to-date traffic or road information, but external API requests are needed. Therefore, some researchers proposed to reduce the number of such requests using the recently obtained route [4], and others used direction sharing and optimal waypoints [5]. Besides, the caching techniques are also explored to answer shortest paths efficiently, by determining the optimal way to cache certain paths [8], or efficiently caching and reusing historical queried-paths [9].

To effectively derive the flooded route segments and possible alternative intersections, two R-tree-based indices are proposed, i.e. Flood Index and Near Index, respectively. R-tree and its variations are widely studied for years to efficiently index multi-dimensional spatial data [6,10,11]. Particularly, range queries and NN (Nearest-Neighbor) queries are two of the most important query types supported in different computing environments. The first NN search algorithm on an R-tree [12] follows the branch-and-bound algorithmic design pattern and can be generalized to kNN search. In [13], both query types are discussed on broadcasted index tree structures and R-tree is one of the considered indices. The range nearest-neighbor (RNN) problem [14] is also discussed, where the nearest neighbor (NN) for every point in a given range is retrieved. Such problems are further investigated on road networks by constructing more complicated indices [15,16]. In the future, we will also study if adopting the real road distance instead of Euclidean distance for finding the alternative intersections near a flooded road will obtain better results.

7. Conclusion

It is common that a particular region may temporarily become flooded due to a sudden heavy rain, and certain roads are unpassable. If the route recommended by the cloud service consists of those flooded roads, the route will be useless and even endanger the user. In this paper, we explore the possibilities of utilizing route APIs and the indexing techniques to efficiently produce a safe path without needing to maintain the huge amount of detailed road data by ourselves. Experimental results demonstrate the effectiveness and the efficiency of the proposed approach. Particularly, by setting the near threshold 1 km and the merging threshold 200 m, we can often successfully get an unflooded route within around 1 s even for a big city like Taipei. Besides, the proposed indices let the performance of our system insensitive to the number of flooded blocks.

In the future, we will further investigate how to apply the relevant research results as discussed in Section 6 to improve the efficiency and success rate of our system. We will also study how to formally define the concept of the best adjusted route and how to obtain one.

Notes

1. https://developers.google.com/maps/documentation/directions.
2. As far as we know, Google Maps does not provide such functionality.
3. http://code.google.com/apis/maps/terms.html.
4. http://satis.ncdr.nat.gov.tw.
5. The total execution time corresponds to the top of the bar.
Disclosure Statement
No potential conflict of interest was reported by the authors.

Funding
This work was partially supported by the Ministry of Science and Technology, Taiwan (TW) under [contract number MOST 105-2221-E-019-069].

ORCID
Chuan-Ming Liu http://orcid.org/0000-0001-9005-5715

References
[1] Chang YH, Wu PS, Liu YT, et al. An effective flood forecasting system based on web services. Adv Intell Syst Appl. 2013;2:681–690.
[2] Dijkstra EW. A note on two problems in connexion with graphs. Numer Math. 1959;1:269–271.
[3] Hart PE, Nilsson NJ, Raphael B. A formal basis for the heuristic determination of minimum cost paths. IEEE Trans Syst Sci Cybern. 1968;4:100–107.
[4] Li Y, Yiu ML. Route-saver: leveraging route apis for accurate and efficient query processing at location-based services. IEEE Trans Knowl Data Eng. 2015;27:235–249.
[5] Zhang D, Chow CY, Li Q, et al. Efficient evaluation of shortest travel-time path queries in road networks by optimizing waypoints in route requests through spatial mashups. Proceedings of APWeb Conference. Suzhou: Springer; 2016. p. 104–115.
[6] Guttman A. R-trees: a dynamic index structure for spatial searching. Proceedings of SIGMOD Conference. Boston (MA): ACM Press; 1984. p. 47–57.
[7] Rice MN, Tsotras VJ. Engineering generalized shortest path queries. Proceedings of ICDE Conference. Brisbane: IEEE Press; 2013. p. 949–960.
[8] Thomsen JR, Yiu ML, Jensen CS. Effective caching of shortest paths for location-based services. Proceedings of SIGMOD Conference. Scottsdale (AZ): ACM Press; 2012. p. 313–324.
[9] Zhang Y, Hsueh YL, Lee WC, Jhang YH. Efficient cache-supported path planning on roads. IEEE Trans Knowl Data Eng. 2016;28:951–964.
[10] Beckmann N, Kriegel H, Schneider R, et al. The R*-tree: An efficient and robust access method for points and rectangles. Proceedings of SIGMOD Conference. Atlantic City (NJ): ACM Press; 1990. p. 322–331.
[11] Gaede V, Gunther O. Multidimensional access methods. ACM Comput Surv. 1998;30:170–231.
[12] Roussopoulos N, Kelley S, Vincent F. Nearest neighbor queries. Proceedings of SIGMOD Conference. San Jose (CA): ACM Press; 1995. p. 71–79.
[13] Hambrusch S, Liu CM, Aref WG, et al. Efficient query execution on broadcasted index tree structures. Data Knowl Eng. 2007;60:511–529.
[14] Hu H, Lee DK. Range nearest-neighbor query. IEEE Trans Knowl Data Eng. 2006;18:78–91.
[15] Sun W, Chen C, Zheng B, et al. An air index for spatial query processing in road networks. IEEE Trans Knowl Data Eng. 2015;27:382–395.
[16] Zhong R, Li G, Tan KL, et al. G-tree: an efficient index for knn search on road networks. Proceedings of CIKM Conference. San Francisco (CA): ACM Press; 2013. p. 39–48.