# Adaptive Fastest Path Computation on a Road Network: A Traffic Mining Approach*

Hector Gonzalez, Jiawei Han, Xiaolei Li, Margaret Myslinska, John Paul Sondag
Department of Computer Science
University of Illinois at Urbana-Champaign

## ABSTRACT

Efficient fastest path computation in the presence of varying speed conditions on a large scale road network is an essential problem in modern navigation systems. Factors affecting road speed, such as weather, time of day, and vehicle type, need to be considered in order to select fast routes that match current driving conditions. Most existing systems compute fastest paths based on road Euclidean distance and a small set of predefined road speeds. However, "*History is often the best teacher*". Historical traffic data or driving patterns are often more useful than the simple Euclidean distance-based computation because people must have good reasons to choose these routes, *e.g.*, they may want to avoid those that pass through high crime areas at night or that likely encounter accidents, road construction, or traffic jams.

In this paper, we present an adaptive fastest path algorithm capable of efficiently accounting for important driving and speed patterns mined from a large set of traffic data. The algorithm is based on the following observations: (1) *The hierarchy of roads can be used to partition the road network into areas, and different path pre-computation strategies can be used at the area level*, (2) *we can limit our route search strategy to edges and path segments that are actually frequently traveled in the data*, and (3) *drivers usually traverse the road network through the largest roads available given the distance of the trip, except if there are small roads with a significant speed advantage over the large ones*. Through an extensive experimental evaluation on real road networks we show that our algorithm provides desirable (*short* and *well-supported*) routes, and that it is significantly faster than competing methods.

## 1. INTRODUCTION

Route planning systems such as MapQuest, MapPoint, or Google Maps have become essential tools for obtaining driving directions. In 2006 MapQuest alone reported that it had computed more than 10 billion routes since the online service launched in 1996. If we combine routes served by other websites and routes computed by car navigation systems, the number is much larger. It is only expected to grow as GPS and GIS systems become commonplace on ubiquitous devices such as cell phones. However, it is surprising to find that most route planning services have a very simple model for road speeds: for the most part roads are assumed to have constant speeds determined by their road class (*e.g.*, highway, interstate, city road, *etc.*); more recently these systems have started to track average road speeds for certain areas of the country, and when available this information is used to provide faster routes. But even with the use of current speed conditions at some edges, the model does not consider a multitude of other factors that are very important in the computation of desirable routes. Let us examine these factors with some illustrative examples.

**Example 1 (Importance of driving patterns)**. Suppose you are new to an area and need to drive to a nearby town to catch a flight at the airport. You would like to get to the airport safely and as quickly as possible. If you have the opportunity to ask a local driver on how to drive there, s/he will likely give you a nice and quick route, although it may not be necessarily the quickest. The suggested route, for example, will not send you through a high crime area, or if there is a snow storm, it will avoid the roads that are more likely to become icy and dangerous. Local experts will consider a multitude of important factors that are difficult to explicitly incorporate into a path finding algorithm. We propose that instead of trying to model all such factors explicitly, we mine historic traffic data and learn from the past driving behavior. In our algorithm we give preference to fast routes that have high support, *i.e.*, that are frequently traveled, over those, though fast, rarely taken by drivers. ■

**Example 2 (Importance of speed patterns)**. Suppose as a new comer, you would like to go downtown to work, your experienced locals will suggest to you the best routes by taking into the consideration many factors that influence your driving speed, *e.g.*, the time of departure, weather conditions, whether you are qualified to drive on a car pool lane, *etc.*. These important factors are also neglected by the current route planning software. Clearly, it is essential to have a new system that can learn from historic traffic data, construct a multi-condition-based road speed model, and plan the fastest route adaptively and dynamically. ■

*The work was supported in part by the U.S. National Science Foundation (NSF) IIS-05-13678/06-42771 and BDI-05-15813. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies.


*VLDB '07,* September 23-28, 2007, Vienna, Austria.

In this work, we develop a traffic-mining-based path-finding method that first mines speed and driving models from historic traffic data, and when a query is posed to the system (which contains the start point, the end point, and departure time, or some other information, *e.g.*, car pool), it computes the fastest route, based on additional conditions, such as weather forecast, or road construction/closure information. In contrast with the current route planning systems, our method is based on traffic data mining and has the following unique features:

- The suggested route is composed of as many frequent path segments as possible, *i.e.*, it matches historic driving behavior, and thus should account for subtle and difficult to model conditions.
- The suggested route takes into consideration the speed conditions expected to be encountered during the trip, given multiple factors that may influence the speed, such as relevant locations, time, type of vehicles, *etc.*.

The key technical contributions of the paper can be summarized as follows:

- **Road hierarchy-based partitioning**. We use the natural hierarchy present in road networks to partition the network into semantically meaningful areas. We construct high level areas by dividing the graph using the largest possible roads: Each area at this level is enclosed by large highways and will probably contain a large number of nodes and edges. We recursively subdivide areas by progressively decreasing the road-scale. An efficient algorithm is developed that automatically partitions an arbitrary road network and constructs a natural hierarchy of areas. These areas are essential to the algorithm as they will be used to guide the driving pattern mining and adaptive fastest path pre-computation.
- **Speed rule mining**. We take a traffic database that records observed road speeds under a variety of conditions (*e.g.*, weather, time of day, and accidents) and induce a set of concise rules of the form "if conditions $c$ for edge $e$ then speed factor = $f$", where speed factor is the speedup or slowdown for an edge with respect to the edge's base-speed. Speed factors are clustered to increase rule support. The concise set of rules is mined through a decision tree induction algorithm.
- **Driving pattern mining**. We mine frequently traveled edges or edge-sequences in order to obtain important driving hints that are hidden in the data and otherwise difficult to model. We propose a novel area-level mining that computes frequent path-segments at the area level with a support relative to the traffic in the area, *e.g.*, lower support thresholds for edges in rural areas than those in big cities. Such adaptive support will avoid generating too many or too few driving patterns.
- **Adaptive pre-computation**. Fastest path algorithms usually pre-compute a subset of fastest paths in order to speedup path computation at runtime. The problem is that pre-computation schemes assume unique fastest paths, and when we have variable edge speeds, fastest paths can be valid only for a given set of conditions. We develop an area-level pre-computation strategy that pre-computes high benefit paths at the area level, *i.e.*, the fastest paths do not change too much. This strategy allows us to speed up query processing without exploding the space requirements.
- **Road upgrading**. We observe that people usually drive between end points by going through the largest possible roads available unless there are smaller roads that are faster than the large ones. Based on this idea, we develop an algorithm that for each area computes the set of small roads that should be upgraded, *i.e.*, considered in routing because they have a significant advantage over large roads enclosing the area.
- **Adaptive fastest path algorithm**. Finally, we propose an efficient routing algorithm that uses the road hierarchy and pre-computed areas to limit the search space. This improves trip duration by using upgraded roads whenever beneficial, and finds routes that take into consideration both speed and driving patterns.

The rest of the paper is organized as follows. Section 2 presents a formal definition of the problem. Section 3 describes the structure of the database of traffic observations. Section 4 introduces the idea of road network partitioning and gives an efficient algorithm for automated road network segmentation. Section 5 proposes a method to mine speed and driving patterns. Section 6 presents the ideas of path pre-computation and road upgrades. Section 7 develops the fastest path algorithm. Section 8 reports on experimental results. We discuss related work in Section 9 and conclude our study in Section 10.

## 2. PROBLEM DEFINITION

In this section we first define a set of key concepts: *road network, driving patterns, speed patterns*, and *forecast functions*, and then present the problem statement.

DEFINITION 2.1. *A road network is a directed graph $G(V, E)$, where $V$ is a set of vertices representing road intersections and terminal points, and $E$ is a set of edges representing road segments each connecting two vertices.*

Figure 1 presents the road network for the town of San Joaquin in California. Larger roads are shown in bold. The graph contains 24,123 edges and 18,496 nodes, and was obtained from TIGER line files provided by the US census [1].

DEFINITION 2.2. *A speed pattern is a tuple of the form $\langle edege\_id, t\_start, t\_end, (d_1, d_2, \ldots, d_k) : m\rangle$, where edge_id is an edge, $(t\_start, t\_end)$ is a time interval, each $d_i$ is a value for speed factor $D_i$, and $m$ is an aggregate function computed on edge speed.*

Speed patterns describe road speeds under a variety of conditions. Table 2 presents an example of speed patterns for a particular edge (road segment) in a road network. In the example we list edge speeds for three conditions: *time-of-day, $D_1$ = weather, and $D_2$ = vehicle-type.* In reality, it is possible to consider many more factors, such as road construction and accidents at nearby edges. The speed table can be constructed by integrating information from multiple sources, *e.g.*, we can obtain information on road speed, based on time, location, weather data, and road construction information. Currently, it is possible to obtain speed conditions for roads at certain important metropolitan areas like San Francisco and Chicago, and there is a trend for an increased availability of such information, evidenced by the

[1] `http://www.census.gov/geo/www/tiger/tgrcd108/tgr108cd.html`

| Time | Weather | Vehicle | Speed |
|---|---|---|---|
| 8am-10am | Good | Car | 45 mph |
| 8am-10am | Bad | Car | 35 mph |
| 8am-10am | Good | Truck | 40 mph |
| 8am-10am | Bad | Truck | 25 mph |
| 10am-5pm | Good | Car | 65 mph |
| ... | ... | ... | ... |

**Table 1: Speed Pattern for a Particular Edge**

recent incorporation of speed conditions into major route planning software from Google and Microsoft. Information on speed-affecting factors such as weather and road construction are readily available for most areas of the United States; and other factors such as accident data are expected to become available in the near future.

DEFINITION 2.3. *A driving pattern is a sequence $s$ of edges $e_{(1)}$, $e_{(2)}$, ..., $e_{(l)}$ that appears more than min_sup times in the path database, and that is a valid path in the road network graph $G(V, E)$. We define support$(s)$ as the number of paths that contain the sequence $s$. We define the length of the sequence, length$(s)$, as the number of edges that it contains.*

Driving patterns are edge sequences that are frequently traversed by drivers. A path database is a set of trajectories, one per driving session. Currently, the availability of path databases is quite limited, but there is a trend for the usage of sophisticated tracking mechanisms, such as RFID enabled tags in cars that can be read by toll ways' tag readers, road sensors, GPS devices, and cameras capable of identifying cars by their license plates. Currently, we do have edge-level traffic information available, and such data can be used to mine frequent driving patterns of length one, which can be quite useful in finding fast roads (at the edge level) that are consistently taken by drivers.

DEFINITION 2.4. *An edge forecast model $F(edge\_id, t)$, returns a tuple $(d_1, d_2, \ldots, d_k)$ with the expected driving conditions for edge edge_id at time $t$.*

The forecast model is a way for the route planner to estimate driving conditions at different edges in the graph. This is analogous, for example, to looking at the wether prediction for the day, before taking an extended trip to plan the route accordingly. An example of the forecast function may be: "*At 5 pm* [*time*]*, for highway 74 between Champaign and Normal* [*edge*]*, Weather = rain, and Construction = no* [*conditions*]".

With the above definitions available, we are ready to state our problem:

**Problem Statement.** *Given a road network $G(V, E)$, a set of speed patterns $S$, an edge forecast model $F$, and a query $q \leftarrow (s, e, start\_time)$, compute a fast route $q_r$ between nodes $s$ and $e$ starting from $s$ at time start_time, such that $q_r$ contains a large number of frequent driving patterns.*

We can see from the problem definition that we are interested in finding fast paths, that are aware of the expected driving conditions for the trip, and that give preference to routes that are historically preferred by drivers. This problem definition encompasses factors beyond what traditional research on fastest path computation has considered. And we believe these factors can be essential in selecting desirable driving routes in addition to being fast.

**Figure 1: San Joaquin Road Network**

# 3. TRAFFIC DATABASE

A traffic tracking system can generate information on the speed conditions for different times of day for each road in the network, such information can be represented as a set of traffic observations of the form $(edge\_id, time, speed)$, where *edge_id* is an edge, *time* is the time when the observation took place, and *speed* is the observed speed. A more sophisticated system, such as the one used to monitor the San Francisco Bay Area traffic conditions [2], can use radio-frequency tags placed in each car to track the paths traversed by individual vehicles. These tags can be the same ones used for automated toll collection, the city would just install readers at many non-toll roads. In this case, each traffic observation will be of the form (*car_id*, *edge_id*, *time*, *speed*), where *car_id* is a vehicle identifier, and other values are defined as before. Vehicle-level observations can be sorted on *car_id* and $t$ to generate a path database, where each entry is the sequence of edges traversed by a car during a driving session. We can use either form of data to determine frequent driving patterns. If only edge-level data is available, we can use the number of edge observations as support, but in this case only frequent length-1 patterns can be mined. If we have vehicle-level data, we can mine longer frequent driving patterns.

In addition to speed information at each edge, we can augment the traffic database with the set of driving conditions present during each edge observation. Given a set of available driving factors $D_1, \ldots, D_n$, we can augment each traffic observation with the tuple $(d_1, \ldots, d_n)$ where each $d_i$ is a value for driving factor $D_i$. Table 3 presents an example traffic database in path format, where each path stage is of

[2] `http://511.org`

| car_id | path $\langle (edge\_id, start\_time, end\_time), \ldots \rangle$ |
|---|---|
| 1 | $(e_1, 10, 15)(e_2, 15, 23)(e_7, 23, 29)$ |
| 2 | $(e_3, 20, 29)(e_1, 29, 33)$ |
| 3 | $(e_1, 9, 16)(e_2, 16, 22)$ |
| 4 | $(e_9, 10, 11)(e_2, 11, 17)(e_8, 17, 20)$ |
| ... | ... |

**Table 2: Traffic Database**

the form $(edge\_id, start\_time, end\_time)$, where $start\_time$ is the time when the car entered the edge, and $end\_time$ is the time when the car exited the edge. For lack of space we do not show observed driving conditions at path stages.

In the above example we have that $support(e_1) = 3$, $support(e_2) = 3$, $support(e_7) = 1$, etc.

# 4. ROAD NETWORK PARTITIONING

## 4.1 Road Hierarchy

Road networks are organized around a well-defined hierarchy of roads. An example of a typical road hierarchy is the road network of the United States, where highways connect multiple large regions, interstate roads connect locations within a region, multi-lane roads connect city areas, and small roads reach into individual houses. Information on road categories is available for both the Unites States where roads are classified into 4 levels, and for the European Union where roads are classified at a higher level of detail into 13 levels. For our San Joaquin example in Figure 1, we draw roads at the two highest levels in bold, and all other roads in gray.

Most existing work on hierarchical shortest path algorithms assume that a partition of the road network is provided, or that the partition can be generated by imposing a fixed grid over the network [18, 19]. Other approaches such as [26] use the idea of Highways to divide the graph, but their definition of Highway is graph theoretic, designed to preserve optimality of routes, and does not necessarily match the road size classification. We believe that the natural partition induced by the road hierarchy itself can be used to divide the network into semantically meaningful areas, with well defined driving and speed patterns. This idea can provide a significant speedup for fastest path queries when compared to arbitrary partitioning methods such as the grid-based one.

The partitioning process can first use the highest-level roads to divide the road network into large regions enclosed by such roads. Each large region can in turn be further subdivided by using the next lower-level roads. This process can be recursively applied until an area contains only roads at the lowest level of the hierarchy, or until a threshold on the minimum number of nodes in an area is reached.

DEFINITION 4.1. *Given a road network $G(V, E)$, with predefined edges classes $class(e)$ for each edge $e$, the class of a node $n$ denoted $class(n)$, is defined as the biggest (lowest class number) of any incoming or outgoing edge to/from $n$.*

The definition indicates that for an intersection of two highway segments with a small road (*i.e.*, the entry point into the highway), the intersection will be at the level of the highway, not at the level of the small incoming road.

DEFINITION 4.2. *Given edges of class $k$, a partition $P(k)$ of a road network $G(V, E)$ divides nodes into areas $V_1^k$, ..., $V_n^k$, with $V = \bigcup_i V_i^k$. Areas are defined as all sets of strongly connected components after the removal of nodes with $class(n) < k$ from $G$. A node $n$, with $class(n) > k$ in strongly connected component $i$, belongs to area $V_i^k$, and it is said to be interior to the area. A node $n$, with $class(n) \leq k$ belongs to all areas $V_i^k$ such that there is an edge $e$, with $class(e) > k$, connecting $n$ to $n'$ and $n' \in V_i^k$, such nodes are said to be border nodes of all the areas they connect to.*

Given a road hierarchy with $l$ levels, we can construct a hierarchy of areas as a tree of depth $l - 1$: The root node represents the entire road network, children of the root node represent the areas formed by partitioning the root using level-1 edges, the nodes in each area form a graph themselves, and all the edges from $E$ connecting two nodes in an area are said to belong to the area. A node at level-$k$ results from the partition of its parent node using the edges of class $k - 1$. Notice that according to this convention, road class 1 is the largest, and road class $l$ the smallest.

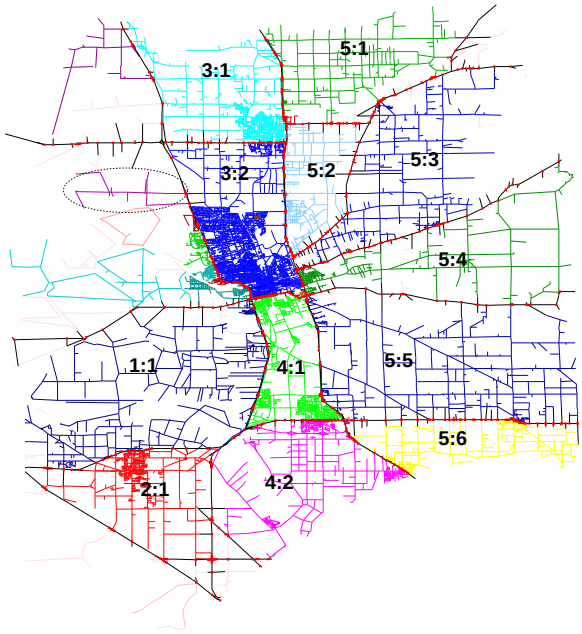


**Figure 2: San Joaquin Partitioned Map**

Figure 2 presents the partition of the road network for San Joaquin, CA. We have numbered each large area with two numbers $a : b$, $a$ is the area number when roads of level 1 are used, and $b$ is the subarea of $a$ when roads of level 2 are used to subdivide $a$. We can see in the upper left corner that we have not marked individual areas, the reason is that there are quite a few strongly connected components in this region, and each has its own area, *i.e.*, nodes inside each component cannot reach nodes in other components except by going through border nodes. We have marked one such component in the figure by encircling it with a dotted line to illustrate the point. As we can see the partition is quite natural. We also experimented with partitioning the entire road network for the San Francisco Bay Area, and for the complete state of Illinois among others (Figures not

included for lack of space), and in every case the hierarchical partitioning algorithm found balanced partitions, with may areas in dense regions of the graph, and less but larger areas at more sparse regions of the graph.

## 4.2 Area partitioning algorithm

In this section we develop an efficient algorithm that can automatically generate a semantically meaningful partition of the road network by using road hierarchy information. The algorithm uses a flood filling technique to identify strongly connected components delimited by high level edges. We take as input the road network $G$ and the edge class $k$ used for partitioning. Fist we assign to each node an empty set of areas. We then choose a node $n$ with $class(n) > k$ (*i.e.*, connected to edges less important than the ones used for partitioning), and add area $a$ to this node's area set; at this point we move to all neighbors of the node that are reachable through the edges of the classes greater than $k$, and add $a$ to their area sets. This process repeats until no further nodes can be reached. We are basically walking in every possible direction from the node until we reach large roads used for partitioning and we stop. At this point we increase our area number, move to the next node with an empty area set, and repeat the process until all nodes are assigned to at least one area. One nice feature of the algorithm is that it automatically identifies *interior nodes* (those with a single area in their area set), and *border nodes* (those with multiple areas in their area set).

**Analysis.** Algorithm 1 examines each interior node $O(1)$ times, and border nodes $O(|a|)$, where $|a|$ is the number of areas. So the order of the overall algorithm is $O(n \times |a|)$. In general, $|a| \ll n$. So the algorithm can be considered linear in the number of nodes. In our experiments we partitioned real road network graphs with more than a million nodes in just a few seconds.

**Algorithm 1** Area partitioning

**Input**: $G(V, E)$, $k$ edge class used to partition $G$.
**Output**: Partition $P(k) = V_1^k, V_2^k, \ldots, V_n^k$
**Method**:

```
1: area = 0;
2: Areas[n_i] = φ for all node n_i;
3: for Each node n_i do
4:    if Area[n_i] = φ and class(n_i) > k then
5:       q.push(n_i);
6:       while not q.empty() do
7:          n ← q.pop(); n.mark = true;
8:          Area[n] = Area[n] ⋃ {area};
9:          push into q all neighbors of n reachable through
            edges of class greater than k, s.t., n.mark = false;
10:      end while
11:      area = area + 1;
12:      q.clear();
13:   end if
14: end for
15: for each area i construct V_i^k as the set of nodes n such
    that i ∈ Area[n]
```

# 5. TRAFFIC MINING

An important contribution of our work is to take into consideration the factors that affect driving speed as well as driving behavior in the adaptive fastest path computation algorithm. We think that route planning software has to account for all such factors in order to provide routes that are not only fast and relevant given the conditions encountered by the driver, but also well supported, in the sense that many drivers in the past have opted for such a route under similar circumstances.

## 5.1 Speed pattern mining

In any road network multiple factors influence the speed at which we can travel through different roads. Weather, time of day, vehicle class, and road construction are just a few among the many dimensions that can influence road speed. In this section we will develop a method that mines, from a large database, a set of concise rules that identify the most relevant factors influencing road speed.

As we mentioned before, the traffic database augmented with extra factors, contains a collection of traffic tuples of the form $\langle edge\_id, time, (d_1, \ldots, d_k) : speed \rangle$ (if we have vehicle level traffic data, we compute the average speed for the edge under every condition for all cars). We can see the problem of speed pattern mining, as a classification problem where we would like to predict edge speed based on time and feature values $d_1, \ldots, d_k$. We can derive rules such as "if area $= a_1$ and weather $= icy$ and time $= rush\ hour$ then speed $= 1/4 \times base\ speed$". Looking at this rule we notice a few things. First, feature values reside at different levels of abstraction, *i.e.*, each factor has an associated concept hierarchy and the rules can use values at any level. Second, speed has been expressed as a relative value with respect to a base speed, such transformation allows us to build more general rules; in the above example "$1/4 \times base\ speed$" indicates that regardless of the initial speed of the road, it slows down to a quarter of its initial value. Third, in this case the class label which is speed, or more concretely, speed factor, is a continuous attribute that requires some form of discretization in order to perform rule induction.

There are several methods to perform rule induction, in this paper we chose decision tree induction [25] as it provides rule predicates of the desired form, which in general have good accuracy and generality, and the method can be applied to very large datasets efficiently [12]. Before running a decision tree algorithm we run a preprocessing step to discretize speed factors, which will be treated as our class label. Speed factors can be discretized through clustering [17], with each traffic tuple assigned to a cluster centroid. This is beneficial for route planning applications, as it allows us to derive speed rules that are supported by a large number of observations and are thus statistically significant. For the time dimension, we can use its concept hierarchy to register values at a higher level of abstraction than that of the raw data, *e.g.*, from seconds to minutes. We can then treat time at the minute level as a continuous attribute which can be handled by decision tree algorithms through binary splits [24], or multi-interval discretization methods [8].

## 5.2 Driving pattern mining

One of the most common ways that drivers use to determine good driving routes in an unfamiliar area is to ask local people for tips, we may find for example that route $R_1$ is very good in the summer but that in winter the road become unsafe, or that route $R_2$ although fast, goes through a high crime area and should be avoided at night. This is valuable information that has been largely ignored by route

planning algorithms, and that cannot be derived by looking at just distance and edge speeds.

Driving patterns can be derived from the traffic database by using frequent pattern mining [1, 15, 23]. We can define a minimum support level, and go through the traffic database identifying frequent edges, and when we have access to individual vehicle data, longer frequent path segments can be mined. The problem with this approach is that a uniform minimum support level is difficult to define, and it may filter many important local roads, or may keep infrequently traveled high-level roads.

We propose a frequent pattern mining method guided by the area and road hierarchies: Frequent edges are mined at the area level, using a minimum support relative to the traffic volume of each edge class in the area. This will allow us, for example, to distinguish support for edges at different levels of the road hierarchy.

With driving pattern mining, each edge (or path segment) in the road network can be marked as frequent or infrequent given a time interval and a set of driving conditions $(d_1, ..., d_k)$. The path-finding algorithm will use this information to guide the search mostly through frequent edges, and only expand infrequent ones when absolutely necessary (*e.g.*, usually at the start or end of a road when we may need to go into nearby neighborhoods).

# 6. PRE-COMPUTATION AND UPGRADES

In this section we will present two techniques aimed at improving the performance both in terms of run time and path accuracy of fastest path algorithms by the utilization of two techniques. The first is area level pre-computation of stable paths in order to improve efficiency. The second is to upgrade certain small but very fast roads to a higher level in the road hierarchy in order to improve the accuracy (path duration vs. best possible path duration) of the algorithm.

## 6.1 Area level pre-computation

Many fastest path algorithms rely on pre-computing a small set of fastest paths in order to improve performance [19, 5, 26]. At one end of the spectrum we can use algorithms such as Floyd Warshall [9] that pre-compute the shortest path between every pair of nodes. In this case fastest path queries can be answered in $O(1)$ lookups, but we need $O(n^2)$ space for storing those pairs, and $O(n^3)$ time for the initial computation. At the other end of the spectrum we can perform no pre-computation and dynamically find the shortest path using an algorithm such as $A*$ [16]. In between these two extremes we have several algorithms that do hierarchical decomposition of the original graph, and pre-compute a subset of paths that are helpful in connecting different areas in the graph [19, 5].

Most methods that do fastest path pre-computation assume that there is a unique path (or several but equally attractive, and thus undistinguishable) between any two nodes. When edge speed is a function of factors such as time, weather, or road conditions, the fastest path between two nodes may be different for different times and conditions, *e.g.*, it may differ when we leave at 8:00 am than at 10:00 am, or if we have to drive through icy roads or dry roads. In the presence of variable edge speeds, pre-computed fastest paths need to be annotated with the set of conditions under which they are valid.

DEFINITION 6.1. *We say that fastest path $p$ between nodes $(s, e)$ is conditionally stable for starting time interval $\langle t_1, t_2 \rangle$ given condition $c = (d_1, \ldots, d_n)$, where each $d_i$ is a value for factor $D_i$ or the special value * to indicate any value for $D_i$ is valid, if duration$(p)$ is minimal among all possible paths between $(s, e)$, when the starting time is between $t_1$ and $t_2$, and when condition $(d_1, \ldots, d_n)$ is forecasted for all edges along paths connecting $s$ and $e$.*

The benefit of pre-computing a path between two nodes is proportional to the number of path queries for which the path can be used to speedup the fastest path algorithm. We can check two conditions to determine benefit. First, how many fastest path queries will go through nodes of the pre-computed path. For example, pre-computing the fastest path between two houses in an area has little value, as it will likely help a single query, while pre-computing a path between two important intersections may benefit many queries. Second, how stable is the path, *i.e.*, for how long, and for how many speed patterns is the path a fastest path. For example, pre-computing a path that is valid only between 10:07am and 10:11am has less value than pre-computing a path that applies for the entire rush hour interval, 8:00am - 10:00am. A sensible strategy is thus to pre-compute high benefit value paths.

A naive method to determine what paths to pre-compute would be to list every fastest path in the road network, under every possible condition, and rank them according to benefit value. This strategy would yield optimal results but it is clearly unfeasible as the number of paths and conditions to consider is enormous. We propose an area level pre-computation strategy where we compute certain fastest paths only within the nodes inside the area. We first define a minimum level $l_m$ in the area hierarchy at which path pre-computation will be conducted. Within each area at level $l_m$ we choose the set of nodes $S_p$ of class $l_m + 1$ (one level smaller than the borders of the area), and class $l_m$, and pre-compute fastest paths between nodes in $S_p$. For this step, we guide our pre-computation by the set of speed rules mined for the area, and limit the analysis paths involving edges with few speed rules. We can handle time intervals by using the algorithm presented in [20], which efficiently computes the set of fastest paths between two nodes for different time intervals.

## 6.2 Small road upgrades

The main assumption of hierarchical path finding algorithms is that drivers take the largest road available in order to reach their destination, and thus the search space for route finding can be significantly reduced. Our observation is that although this strategy is generally true, there is an important exception, if there is a small road that is faster than a large road, people will take it. For example, people driving to or from cities usually do it through highways, but during rush hour highways can become so congested that taking smaller roads yields shorter travel times. If we ignore such cases we may incur significant error.

Our strategy in dealing with this problem will be to *upgrade* certain edges inside an area if under some driving conditions they have a significantly higher speed than the edges at the area borders under the same driving conditions. For the above example, we would upgrade the internal edges in the area where the city resides, to the level of highways but only for the driving condition of rush hour. This way we can still compute most routes considering only highways,

and incur the extra cost of looking at the upgraded edges only when absolutely necessary.

More formally, an edge $e$, residing in area $a$ with border edges at level $l$ of the road hierarchy, will be upgraded to level $l$ if three conditions are met (i) the edge speed under driving conditions $(d_1, \ldots, d_n)$ for some time $t$ is faster than the average edge speed of border edges in the area under the same conditions and time, (ii) edge $e$ is at level $l+1$, and (iii) the edge is frequent. For all such edges we will register a conditional road class tuple $\langle edge\_id, t, (d_1, \ldots, d_n) : upgraded\_class \rangle$.

**Algorithm 2** Edge upgrading

**Input**: $G(V, E)$: road network, $T$: area hierarchy, $s$: speed threshold
**Output**: List of upgraded edges
**Method**:

```
1: Precompute the average border speed for every area under every valid driving condition and time
2: q ← push all leaf areas in T;
3: while q not empty do
4:    A ← q.pop();
5:    for each edge e in A do
6:       if e.level = A.level + 1 and e is frequent then
7:          for each driving condition c and time t for which
            e.speed > s× average border speed for c in A,
            make e.level = a.level and output ⟨e, t, c, e.level⟩;
8:       end if
9:    end for
10:   q.push(A.parent);
11: end while
```

**Analysis**. Algorithm 2 presents the method used to upgrade internal area edges when they are faster than the border. The algorithm starts by computing the average speed of border edges in an area for all valid conditions. This can be done efficiently in a single scan of the list of edges. We then traverse the area tree in a bottom up order, upgrading edges at the lowest areas, before upgrading edges at larger areas. Notice that an edge can be upgraded multiple times if it is consistently faster than the borders of several successively larger areas. During the edge upgrade process each edge is touched $O(l)$ times where $l$ is the number of levels in the tree, so in total we touch at most $O(|E| \times l)$ edges.

# 7. FASTEST PATH COMPUTATION

In this section we will introduce an approximate fastest path algorithm for road networks, that computes fast paths between a source and destination node, such that the computed route has the following properties:

- Fast routes should be well supported by the historical driver behavior, *i.e.*, they should contain as many frequent driving patterns as possible.
- Fast routes between a source and a destination will go through the largest possible roads connecting the two locations as long as there are no smaller roads along the way that have a significant advantage over the large ones.
- Fast routes will account for all relevant factors affecting driving speed expected to occur during the trip such as weather, time of day, and road construction status.

Before running the algorithm we assume that the following components have been computed:

- The road network $G$ has already been partitioned using algorithm 1, and we have an area hierarchy tree $T$ that encodes the parent/child relationship between areas.
- Speed patterns have been mined and we can use the function *get_edge_speed* (*edge_id*,$t$,$(d_1, \ldots, d_k)$) to get the speed of *edge_id* when it is taken at time $t$, and for driving conditions $(d_1, \ldots, d_k)$. This information is retrieved from our mined set of rules for driving conditions, by selecting most specific rule(s) applicable given the conditions.
- Driving patterns have been mined and we can use the function $is\_frequent(edge\_seq, t, (d_1, \ldots, d_n))$ to determine if the edge sequence *edge_seq* is frequent under conditions $(d_1, \ldots, d_n)$ at time $t$.
- We have pre-computed a set of area-level fastest paths with high benefit value.
- We have used algorithm 2 to upgrade internal roads to an area when they are faster than roads along the area border for a given set of time and driving conditions. We can retrieve upgraded edges with the function *get_edge_class* (*edge_id*, $t$, $(d_1, \ldots, d_k)$) that returns the class of *edge_id* for driving conditions $(d_1, \ldots, d_k)$ at time $t$.

## 7.1 Algorithm

At this point we are ready to state our fastest path algorithm, it is a variation of $A*$, where we dynamically compute edge costs, take advantage of pre-computed paths, follow edges in ascending/descending order of their level in the road hierarchy, and give priority to frequent edges (or edge sequences).

The key technical contributions of the algorithm are three. First, it incorporates previously neglected factors such as speed and driving patterns into route finding. Second, we improve performance by utilizing a novel area level precomputation scheme. And third, although hierarchical path finding has been used in the past, to the best of our knowledge this is the first study that uses the idea of small road upgrading to improve path quality with minor impact to efficiency.

The key concepts of the algorithm are summarized below:

1. We maintain a priority queue of expanded paths (represented by the last node of the path), for each path we keep $g(n)$ *the current cost* (*travel time*) *spent to get from start to* $n$, and $h(n)$ *the expected cost to reach the end node from* $n$.
2. At each step of the search process we pick the node with lowest $g(n) + h(n)$ value that is frequent[3], if no frequent node is present in the queue we pick the best infrequent one. We prefer to travel through frequent roads, but sometimes it is necessary to pick a small number of infrequent paths in order to reach the destination. This is especially true around the starting and ending nodes.
3. At the beginning of the search we determine, using the area hierarchy tree $T$, what is the lowest common ancestors of both start and ending nodes. We use the lowest common ancestor to set the phase of the algorithm, which can be *Ascending* or *Descending*. We are in an *Ascending* phase when the currently examined node is in an

[3]A more sophisticated strategy is possible, we can give preference to paths that have longer frequent driving patterns, than shorter ones.

area that is below or at same level of the lowest common ancestor, and we are in a *Descending* phase otherwise.

4. At each iteration we look at the neighbors of the node currently being examined. If we are ascending, we only consider neighbors that are connected through edges that have an edge class lower or equal to the previous edge (bigger road) in the path. If we are descending, we only take edges with class greater or equal to the previous edge (smaller road) in the path. The class of each edge is dynamically computed by calling the function *get_edge_class* (*edge_id*, $t$, $F$(*edge_id*, $t$)), where $F$ is the predictor function that returns the tuple $(d_1, \ldots, d_k)$ of expected driving conditions (*e.g.*, weather, road construction, etc) for the edge at time $t$, $t$ is the projected time at which the edge will be taken. The set of neighbors of a node are all those nodes directly reachable through a single edge, or indirectly reachable through a pre-computed path.
5. Whenever we insert a new path into the priority queue, we update its $g(n)$ value by adding to the path's total travel time the time to traverse the new edge, which is computed by dividing the edge distance by the expected edge speed retrieved with the function *get_edge_speed* (*edge_id*, $t$, $F(edge\_id, t)$). $h(n)$ is also updated for the path by using a conservative estimate of the total travel time from the current node to the goal node. Several different estimation policies are possible, more accurate ones can significantly speed up the search [6]. In our implementation we used the simple heuristic $h(n) = distance(n, end)/max\_speed$, but any other heuristic could be plugged into the algorithm.

LEMMA 7.1. *The adaptive fastest path algorithm, when computing a path between* $(start, end)$ *nodes, in areas* $a_i$, $a_j$ *respectively will consider at most* $O(|a_i| + |a_j| + |bn| + |un|)$ *distinct nodes, where* $|a_i|$ *is the number of nodes in area* $a_i$, $|a_j|$ *is the number of nodes in area* $a_j$, $|bn|$ *is the total number of border nodes in all areas, and* $|un|$ *is the number of nodes connected to upgraded edges in all areas.*

**Proof Sketch**. The worst case for path finding is when no pre-computed path is available. In this case we need to examine all nodes in the starting area until border nodes are reached, and all nodes in the ending area until we reach the destination. Once we have reached the border nodes of the first area, the algorithm only goes through border nodes, or upgraded nodes until the destination area is reached. ∎

The implication of lemma 7.1 is that our search algorithm will need to consider significantly fewer nodes than traditional $A*$ even in the case when no area pre-computation is possible. We verify this result running the search algorithm on real road networks for the United States where we observed an order of magnitude savings in the number of nodes examined by the algorithm.

**Example 3 (Search)**. Figure 3 presents a road network, there are 3 levels of roads, and we have partitioned the graph along the first 2 levels. The first level gives us the large grid, and the second level the finer grid (shown only for two areas). The size of nodes indicate the level at which they are, larger nodes are associated with edges at higher level. The graph also shows edges that have been upgraded by painting them with dotted lines. The name of an area indicates its position

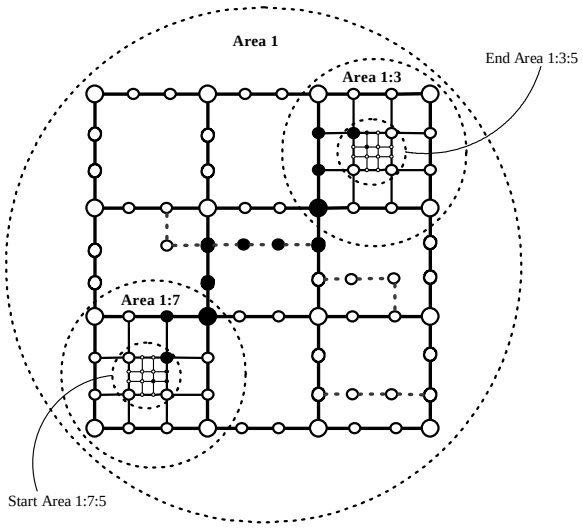


**Figure 3: Example Hierarchical Search**

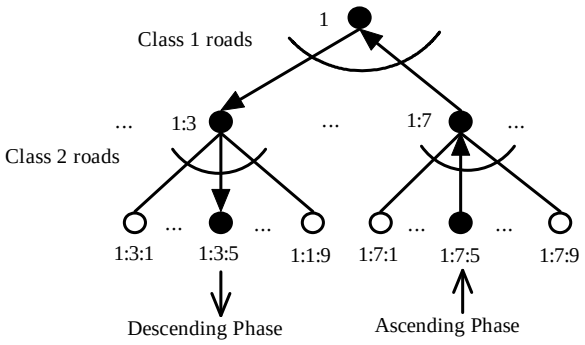


**Figure 4: Example Area Hierarchy**

in the area tree, *i.e.*, area $1:7:5$ is a child of area $1:7$ which in turn is a child of area 1 (more clearly seen in Figure 4). The total number of nodes in this graph is $28 \times 28 = 784$.

We would like to find the fastest path between a starting node in area $1:7:5$, and an ending node in area $1:3:5$. Figure 4 presents the area hierarchy, in this case see that the lowest common ancestor of the start and end nodes is area 1. The algorithm will proceed in two phases: First, it ascends from area $1:7:5$ to area 1, and then it descends to area $1:3:5$. We first expand nodes in area $1:7:5$ by following roads of level 3, until we reach edges in the border of the area which are at level 2, and are inside area $1:7$. At this point we are still ascending, as we have not reached the lowest common ancestor, and we will not look at roads of level 3 any more. We now move along edges of level 2 until we reach the borders of area $1:7$, which are roads of level 1. At this point we will consider only roads of level 1, and roads of level 2 upgraded to level 1 (the dotted ones) until we reach area $1:3$, at which point we enter the descending phase of the algorithm and the order of road classes to follow is reversed. We follow roads of level 2 to area $1:3:5$, and finally roads of level 3 to the destination node.

In this example, in order to simplify the explanation we

have ignored edge frequency. But if we consider that all edges of level 3 are infrequent, it should be clear that when we start the search, and when we end the search, the priority queue will contain only nodes that are reached through an infrequent edge and thus we would have to use it. We have also assumed that there are no pre-computed paths; otherwise, the search process is the same, but instead of following direct neighbors of the node we also follow indirect neighbors connected through pre-computed paths.

In this example the maximum number of distinct nodes that the algorithm would have to consider is 32 at the start and end areas, 32 at the level 2 areas, 64 at level 1, and 7 nodes along upgraded edges. This is much smaller than the potential nodes examined by A* which in the worst case is the entire network, 784 nodes.

## 7.2 Online path re-computation

When we compute a fastest route, the predictor function $F$ is used to estimate driving conditions throughout the entire trip, but it is possible that our initial estimate is wrong. For example, if we computed the route using a prediction of good weather for the duration of the trip, but halfway into the trip a snow storm happens, our fastest path may be far from optimal. Another example of changing conditions would be an unexpected road closure or an accident. If we plan a complete trip before the starting time using a static model, there is not much that the system can do about these unexpected cases (*e.g.*, when we ask a site such as MapQuest to give us a route before the trip starts). But if we are operating in an online navigation environment, we can do better, as it is possible to recompute the best route as driving conditions change. In this case we can just apply the algorithm presented in the previous section with a starting node changed to the current position, starting time to the current time, and an updated forecast model.

In an online navigation system, having access to an *efficient* route computation algorithm is even more critical than in the static route computation model, and the power of our proposed method would be more evident. Another possible change to our algorithm, when forecast is updated, would be to invalidate our speed model for edges near the vehicle's current location, as we may know the exact speed for those, and use mined speed patterns only for farther away edges.

# 8. EXPERIMENTAL EVALUATION

In this section we perform a thorough analysis of the adaptive fastest path finding algorithm proposed in the paper, and compare its performance against a basic A* implementation, and a version of the adaptive of our own algorithm that does not perform area pre-computation. All the experiments were conducted on a single core of an AMD Athlon 64 X2 Dual Core processor with 2GB of RAM. The system ran cygwin and gcc 3.4.4.

## 8.1 Data Synthesis

In all of our experiments we used real road maps from areas of the United States. We used three maps. San Francisco Bay area (90 by 125 miles) with 175,343 nodes, and 223,606 edges. A map for the entire state of Illinois with 831,524 nodes, and 1,048,080 edges. A smaller map for San Joaquin, CA with 18,496 nodes, and 24,123 edges. We simulated different traffic conditions using the Network-based Generator of Moving Objects by Thomas Brinkhoff [3], which is a well known traffic simulator. For each map we ran two simulations, one with 10,000 objects used to generate rush hour like speed conditions, and one with 1,000 objects used to simulate non-rush hour speed conditions. In each simulation we defined two object classes, cars with faster speeds, and trucks with slower speeds. We also incorporated a weather factor into the simulation by running the simulation with and without external objects, which in the data generator can be used to slow down certain areas of the road network as if bad weather were occurring. The output of the simulation was a list of edge observations of the form $\langle edge\_id, car\_id, time, weather, speed\rangle$. The output files were a few hundred megabytes long. Using this information we mine speed patterns for each edge, such patterns involve the dimensions of *time*, *weather*, and *vehicle_type*.

In most of the experiments we compare three methods. The first is an implementation of $A*$, we run this algorithm on the entire road network. We maintain a priority queue of paths to expand. At each iteration we select the path with minimal $g(n) + h(n)$, where $g(n)$ is the current cost of the path, and $h(n)$ is the expected cost to the goal. For each path we update $g(n)$ by retrieving the appropriate speed for the edge at the time when it will be taken, and for the type of vehicle for which the route is being planned, and for the forecasted weather. $A*$ always finds the fastest possible path, and is thus our baseline for correctness. The second algorithm $Hier$ is our adaptive fastest path algorithm implemented without area pre-computation, *i.e.*, no fastest paths have been pre-computed at all, and without considering road upgrades, *i.e.*, small roads through an area that are faster than roads in the area border. The third algorithm, *Adapt* is the fastest path algorithm proposed in this paper.

## 8.2 Query Length

In this set of experiments we vary the distance between the starting node and the ending node of the query. For this experiment we used the San Francisco road network. The road network was partitioned with edges of level 1, and level 2. We artificially upgraded a single random path in 20% of the lowest level areas, and set the speed for edges in upgraded paths higher than the average speed of the border edges of the area. We pre-compute fastest paths in 30% of the lowest level areas. Results are the average of 100 random queries. We varied the average distance between the starting and ending nodes. The longer the distance the larger the search space. Query Length measures the distance of the end points in the query as a percentage of the map diameter.

In Figure 5, we see the number of expanded nodes for the three algorithms. We see that the number of nodes expanded by $A*$ grows very rapidly for large paths. This is expected as there are many more possible paths to consider between nodes that are separated by a large number of edges. At the same time we see that $Hier$ and *Adapt* expand a number of paths that is almost constant. The reason is that both algorithms limit their search to larger roads and only go into individual areas at the start or end of the search. We see that although *Adapt* has to consider all upgraded edges, it only expands slightly more nodes than $Hier$ which ignores those edges.

Figure 6 presents the travel time for the three methods. We see that $A*$ always gives us the fastest path. The path found by *Adapt* is almost as good as the $A*$ path, but at

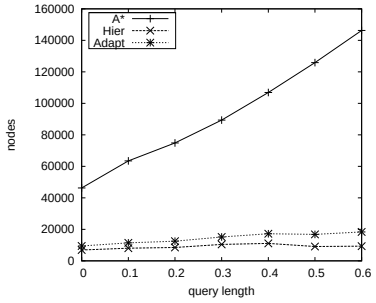


**Figure 5: Query Length vs. Expanded Nodes. Depth: 2**

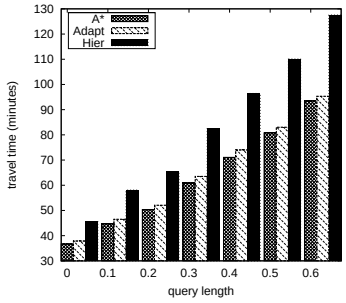


**Figure 6: Query Length vs. Travel time. Depth: 2**

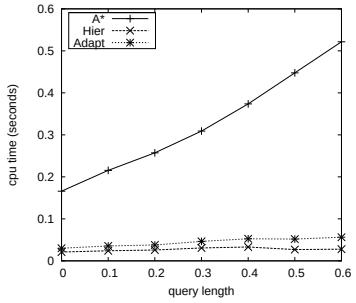


**Figure 7: Query Length vs. CPU time. Depth: 2**

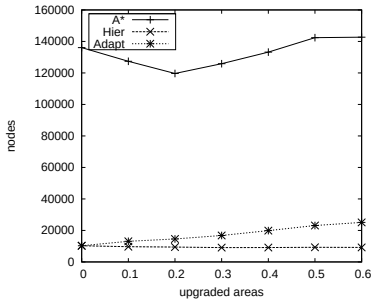


**Figure 8: Upgraded paths vs. Expanded Nodes. Depth: 2**

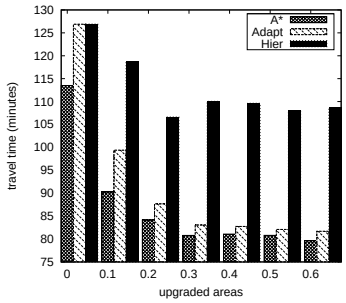


**Figure 9: Upgraded paths vs. Travel time. Depth: 2**

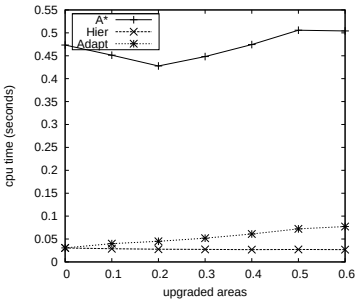


**Figure 10: Upgraded paths vs. CPU time. Depth: 2**

only a fraction of the cost. *Hier* suffers significantly in terms of path travel time, the reason is that it completely ignores roads internal to areas that are faster than border roads. This experiment shows the power of our algorithm, not only in terms of efficiency, but also accuracy, and highlights the relevance of considering small fast roads in addition to large roads. In Figure 7, which presents the average CPU time per query using the 3 algorithms, we observe the same pattern as in the expanded nodes figure.

## 8.3 Upgraded paths

In this set of experiments we vary the percentage of lowest level areas that contain a path that is faster than the border paths and thus needs to be upgraded (Given that the speed of the edges in the areas along paths that need upgrading is changed, $A*$ may need to expand slightly different edges). For this experiment we used the San Francisco road network. Average path length is 50% of the area diameter, and all the conditions are the same as the ones used for the query length experiments, except for upgraded paths which we vary.

We can see in Figures 8 and 10 that neither $A*$ or *Hier* are significantly affected in terms of the number of expanded nodes, or the CPU speed. This is because $A*$ always considers the entire network, and *Hier* disregards upgraded edges completely. The performance of *Adapt* suffers as we have more upgraded edges that need to be considered in the search process. But we can see that the degrade in performance is quite gradual we go from around an average of 18,000 expanded nodes when no edges are upgraded to just around 20,000 when 60% of the areas contain an upgraded path. Where we really see the importance of upgraded edges is in Figure 9, we see that when no edges are upgraded both *Hier* and *Adapt* perform equally, as we increase the number of upgraded edges *Adapt* starts closing the gap with $A*$. This is expected as we have ever more options to find a good path, while the quality of paths found by *Hier* continues to decrease. This experiment is important because we see that we can use a fairly aggressive edge updating strategy to improve path quality without incurring any significant performance penalty. We could consider, for example, interior edges as long as they are 80% as fast as border edges to improve path quality.

## 8.4 Area Pre-computation

In this experiment we examine the performance gain for different levels of pre-computation. We use the San Francisco road network, partitioned using roads of classes 1 and 2. The average path length used is 50% of the area diameter. The percentage of areas with an upgraded path is 20%. We compare two methods, *Adapt* which is our algorithm, and *Adapt_nopre* which is the same algorithm but without using pre-computed areas. For this experiment we select a percentage of the lowest level areas, and pre-compute every fastest path in it. We vary the number of pre-computed areas at the lowest level from 0% to 100%. We can see that the performance improvement is very significant as we can use more pre-computed areas, *i.e.*, the algorithm is basically reaching for the destination taking very long jumps. In this experiment we did not pre-compute any higher level area, if we had done it, the performance gains would have been even more noticeable.

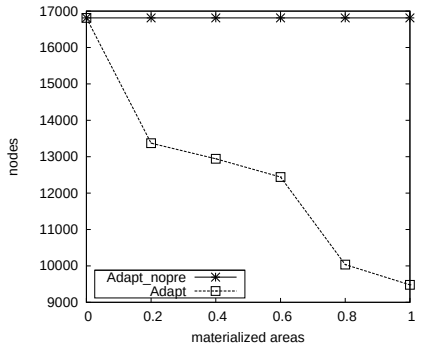


**Figure 11: Pre-computed areas vs Expanded Nodes. Depth: 2**

## 8.5 Road Network Size

In this experiment we compare query processing efficiency for 3 road networks in the United States of different sizes: San Joaquin (sj) with 18,496 nodes and 24,123 edges, San Francisco (sf) with 175,343 nodes and 223,606 edges, and Illinois (il) with 831,524 nodes and 1,048,080 edges. All the maps where partitioned using the top two levels of roads. Average path length was 50% of map diameter. 20% of areas had a single path upgraded.

We can see in Figure 12 that the adaptive algorithm has excellent scalability in terms of road network size. The reason is that the number of nodes usually grow much slower than the number of small roads, and thus our algorithm is able to significantly restrict the search space to a manageable size even in the presence of millions of nodes and edges. This result is encouraging, as it indicates that the algorithm can handle route planning in very large maps quickly, a factor that is essential when we consider that the number of queries that such system needs to handle is in the billions.

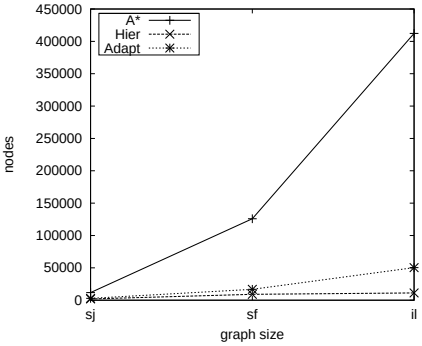


**Figure 12: Graph size vs Expanded Nodes. Depth: 2**

# 9. RELATED WORK

Shortest path computation is an area that has received extensive research attention for almost half a century. There are so many results in this area that we can only highlight a few selected publications to put our work into perspective. A good survey of shortest path methods is found in [22, 11].

Exact algorithms for static graphs have used the idea of edge hierarchies to reduce computation time. [19, 18] and more recently [26, 5] have looked at the techniques to decompose the graph using hierarchies and pre-computed selected paths. The hierarchy defined by these methods is similar in spirit to ours, but theirs is usually graph theoretic and not based on the physical size of roads. Approximate algorithms have also considered the idea of graph partitioning. [4] uses large roads to partition the graph, but they do it manually, and no driving or speed patterns are considered. Other graph-theoretic algorithms have focused on improving the heuristics used by $A*$. [6] is a recent example of work along this direction. It uses the concept of landmarks to improve route cost estimation. These algorithms are a complement to our method, we could directly use them to estimate $h(n)$ and improve performance.

Shortest path computation on dynamic graphs has two interpretations. The first is graphs where edges are updated, deleted, and added. Under this interpretation the idea of most methods is to apply a static shortest path algorithm such as Dijsktra's [7] only to the subset of the graph where fastest paths change after the upgrade. [10, 6] compare the performance of a few of the most popular single source dynamic shortest path methods. [21] presents an all-pair dynamic fastest path algorithm. The second interpretation is that edge speed is a function of time. [20] looks at this problem, they define the CapeCod pattern to match time of day to edge speed. Their algorithm is an adaptation of A*: Instead of sorting the priority queue on a scalar cost, they maintain a function of cost based on starting time. The focus of this work is to provide the complete set of fastest paths for a given time interval. This technique can be used by our algorithm in performing path pre-computation at the area level. [2] uses a statistical approach based on clustering to compute rules useful in predicting fastest paths. We are different to this work in that our approach to route finding is search, not prediction.

We make use of different data mining techniques in order to derive driving and speed patterns. Frequent driving patterns can be computed using frequent pattern algorithms such as [1, 15], or sequential pattern mining algorithms such as [23]. Speed patterns are computed using an efficient decision tree [25] induction algorithm such as [12]. Speed factors can be clustered using a number of algorithms [17]. And the treatment of time, and other continuous attributes in speed pattern induction can be handled through the methods presented in [24, 8].

Finally, our traffic database, especially when individual vehicle tracking information is available, is similar to path databases studied in the management of RFID data [13, 14]. But these models are based on the concept of bulky object movements and predictable flow patterns, which for the case of cars in a road network are not directly applicable.

# 10. CONCLUSIONS

We developed an adaptive fastest path algorithm, that bases routing decision on driving and speed patterns mined from historical data. This is a radical departure from traditional algorithms that have focused only on speed and Euclidean distance considerations. The routes computed by our algorithm are not only fast given a set of driving conditions but also reflect observed driving preferences. This is in sharp contrast to existing algorithms that may send a

driver through high crime areas of a city at night, or through unsafe roads in order to save a few minutes of travel time.

A road network partitioning algorithm was introduced. The algorithm uses the hierarchy of roads to segment the network into areas that are enclosed by large roads. This method yields very natural partitions, where large areas are observed at regions with low road densities, and much finer areas are observed at dense regions such as big cities. Areas are a central concept to route planning, as they provide the basis for hierarchical path finding, area level precomputation, and area sensitive support of driving patterns.

We showed that significant query processing gains can be obtained, by following the principle that drivers tend to travel through the largest roads available for the trip, unless small roads along the way have a speed advantage over the large ones. We also presented a method to identify such fast roads, and efficiently incorporate them into route planning. In our experimental study we demonstrated that incorporating fast small roads into the hierarchical path finding algorithm can significantly improve the quality of routes.

Through an empirical study, on real road networks, using a realistic traffic information, we verify the large performance gains of our algorithms vs. competing methods, while showing that computed routes are close to optimal.

# 11. REFERENCES


[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. Int. Conf. on Data Engineering (ICDE'95)*, 1995.

[2] A. Awasthi, Y. Lechevallier, M. Parent, and J.-M. Proth. Rule based prediction of fastest paths on urban networks. In *Proc. Conf. on Intelligent Transportation Systems*, 2005.

[3] T. Brinkhoff. Network-based generator of moving objects. Technical report, IAPG, `http://www.fh-oow.de/institute/iapg/personen/brinkhoff/generator/`.

[4] Y.-L. Chou, H. E. Romeijn, and R. L. Smith. Approximating shortest paths in large-scale networks with an application to intelligent transportation sytems. *INFORMS Journal on Computing*, 10:163–179, 1998.

[5] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Highway hierarchies star. In *Proc. 9th DIMACS Implementation Challenge*, 2006.

[6] C. Demetrescu, S. Emiliozzi, and G. F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. In *Proc. Symposium on Discrete Algorithms (SODA'04)*, 2004.

[7] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[8] T. Elomaa and J. Rousu. General and efficient multisplitting of numerical attributes. *Machine Learning*, 36(3):201–244, 1999.

[9] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[10] D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasquale. Experimental analysis of dynamic algorithms for the single-source shortest-path problem. *ACM Journal of Experimental Algorithms*, 3:5, 1998.

[11] L. Fu, D. Sun, and L. R. Rilett. Heuristic shortest path algorithms for transportation applications: state of the art. *Computers in Operations Research*, 33(11):3324–3343, 2006.

[12] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest: A framework for fast decision tree construction of large datasets. In *Proc. Int. Conf. on Very Large Data Bases (VLDB'98)*, 1998.

[13] H. Gonzalez, J. Han, and X. Li. Flowcube: Constructuing RFID flowcubes for multi-dimensional analysis of commodity flows. In *Proc. Int. Conf. on Very Large Data Bases (VLDB'06)*, 2006.

[14] H. Gonzalez, J. Han, X. Li, and D. Klabjan. Warehousing and analysis of massive RFID data sets. In *Proc. Int. Conf. on Data Engineering (ICDE'06)*, 2006.

[15] J. Han and J. Pei. Mining frequent patterns by pattern-growth: Methodology and implications. *SIGKDD Explorations (Special Issue on Scalable Data Mining Algorithms)*, 2:14–20, 2000.

[16] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on system science and cybernetics*, 4:100–107, 1968.

[17] J. A. Hartigan. *Clustering Algorithms*. John Wiley & Sons, 1975.

[18] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical optimization of optimal path finding for transportaton applications. In *Proc. Int. Conf. on Information and Knowledge Management (CIKM'96)*, 1996.

[19] S. Jung and S. Pramanik. HiTi graph model of topographical road maps in navigation systems. In *Proc. Int. Conf. on Data Engineering (ICDE'96)*, 1996.

[20] E. Kanoulas, Y. Du, T. Xia, and D. ZXhang. Finding fastest paths on a road network with speed patterns. In *Proc. Int. Conf. on Data Engineering (ICDE'06)*, 2006.

[21] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. Symposium on Foundations of Computer Science*, 1999.

[22] S. Pallottino and M. G. Scutellà. Shortest path algorithms in transportation models: classical and innovative aspects. Technical Report TR-97-06, 14, 1997.

[23] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. Int. Conf. Data Engineering (ICDE'01)*, 2001.

[24] J. Quinlan. Improved use of continuous attributes in c4.5. *Journal of Artificial Intelligence Research*, 4:77–90, 1996.

[25] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[26] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *Proc. 17th European Symposium on Algorithms (ESA)*, 2005.