Path Computation Algorithms for Advanced Traveller Information System (ATIS).

Shashi Shekhar, Ashim Kohli, and Mark Coyle

email: [shekhar | akohli | coyle]@cs.umn.edu
Phone: (612) 624-8307, 625-4002
Fax: (612) 625-0572
Address: Computer Science Department, University of Minnesota, Minneapolis, MN 55455.

ABSTRACT

Path planning has been approached by graph-theoretic algorithms for all pair (transitive closure) and single-source (partial transitive) path computation in databases. These algorithms compute many more paths beyond the single pair path that is of interest to ATIS, and hence may not be satisfactory for ATIS due to the dynamic nature of edge costs (travel-time). We explore specialized algorithms for single-pair path computation which are designed to reduce irrelevant computation and to quickly discover the shortest paths. Our cost models and performance studies show that single pair algorithms can outperform traditional algorithms in many situations.

Key Words: Advanced applications, performance studies, databases.

1. Introduction

1.1. Route Computations in Advanced Traveller Information Systems

Advanced Traveller Information Systems (ATIS) assist travellers with planning, perception, analysis and decision making to improve the convenience, safety and efficiency of travel [1]. ATIS is one facet of the Intelligent Highway Vehicle System (IVHS), which is currently being developed to improve the safety and efficiency of automobile travel. Route planning is an essential component of ATIS. It aids travellers in choosing the optimal path to their destinations in terms of travel distance, travel time and other criteria. Estimates made by government agencies show that approximately 6% of all driving time in the U.S. is due to an incorrect choice of routes [2]. An effective navigation system with static route selection, coupled with real-time traffic information, is crucial to eliminating unnecessary travel time. Reducing vehicles' exposure to congestion also reduces their exposure to accidents, reduces pollution, and allows efficient calculation of routes to new destinations. Furthermore, the commercial vehicle sector will derive additional advantages, including lower transportation costs and less delay at checkpoints.

Route planning services need to provide three facilities: route computation, route evaluation and route display. The goal of route computation is to locate a connected sequence of road segments from current location to destination. Route computation may be based on criteria such as shortest travel distance or smallest travel time. Route computation is also useful for travel during rush hour, travel in an unfamiliar areas, and/or travel to an unfamiliar destination. The goal of route evaluation is to find the attributes of a given route between two points. These attributes may include travel time and traffic congestion information, and thus route evaluation is useful for selecting travel time by a familiar path. The goal of route display is to effectively communicate the optimal route to the traveller for navigation. In this paper we focus on route computation and examine algorithms for real-time route computation on maps stored in a database.

1.2. Related Work and Our Contributions

The single-pair path computation problem is a special case of single-source path computation and all-pair path computation. Traditional research in database query languages[3-5], transitive closure [6-17], and recursive query processing[18-21] has approached single-pair path computation as a special case of more general problems. For example, partial transitive closure computation[22] and transitive closures [8] have been used for single-pair path computations. Previous evaluation of the transitive closure algorithms examined the iterative, logarithmic, Warren's, Depth first search (DFS), hybrid, and spanning-tree-based algorithms. [6, 10, 12, 13]. This study was based on implementing algorithms in Fortran, Quel* and Ingres. The database-based implementation outperformed main memory implementation for large graphs with more than 100 nodes. The study did not examine A* and other estimator-based algorithms, and did not examine the effect of path length and edge costs on the relative performance of search algorithms. In applications such as ATIS, the desired route will vary in length and rarely will traversal of the full map be necessary. Iterative algorithms need to examine all nodes in the graph to find the shortest path between any pair of nodes. They must carry out the same amount of work irrespective of the path length. There is a need for more thorough study of algorithms for single-pair path computations.

In this paper we evaluate three path-planning algorithms for single-pair path computation. These algorithms are the iterative breath-first search, Dijkstra's single-source path-planning algorithm, and the A* single-path planning algorithm. The algorithms represent three classes of algorithms, namely transitive closure, partial transitive closure, and single-pair path planning. The algorithms have been chosen for their simplicity and for their representation of essential features of the algorithms in their class. The iterative algorithm does not utilize path-length information to reduce the work required for single-pair path computation, as is typical of most transitive-closure algorithms. Dijkstra's algorithm has limited lookahead of one edge during searching, therefore like most partial-transitive closure algorithms it is not able

to focus the search in the direction of the destination. A* represents the single-pair path-planning algorithms which use heuristic lookaheads to focus the search.

We also evaluate the performance of the algorithms on graphs representing the roadmap of Minneapolis. In order to get an insight into their relative performance, we also use synthetic grid maps as a benchmark computation. We examine the effect of two parameters, namely path length and edge-cost distribution, on the performance of the algorithms. Finally, we examine the effect of implementation decisions on the performance of the A* algorithm. The main hypothesis is that estimator functions can improve the average-case performance of single-pair path computation when the length of the path is small compared to the diameter of the graph. We examine this hypothesis via experimental studies and analytical cost modeling (studies include a grid graph as well).

1.3. Outline of the Paper

Section 2 defines the problems and introduces the terminology and notation used in the paper. Section 3 lists the candidate algorithms for single-pair path computation. Section 4 provides analytical cost modeling of the algorithms. Section 5 describes the experiments and observations to evaluate the effectiveness of estimator functions and implementation decisions. Section 6 presents the conclusions and future work.

2. Problem Definition and Scope of Work

A (directed) graph G = (N, E, C) consists of a node set N, a cost set C and an edge set E. The edge set E is a subset of the cross product N * N. Each element (u, v) in E is an edge joining node u to node v. Each edge (u,v) is associated with a cost C(u, v). Cost C(u, v) takes values from the set of real numbers. A node v is a neighbor of node u if edge (u, v) is in E. The degree of a node is the number of neighboring nodes. A path in a graph from a source node s to a destination node d is a sequence of nodes $(v_0, v_1, v_2, ..., v_k)$ where $s = v_0$, $d = v_k$, and the edges $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, d)$ are present in E. The cost of the path is the sum of the cost of the edges, i.e. $\sum_{i=1}^{k} C(v_{i-1}, v_i)$. An optimal path from node u to node v is the path with smallest cost.

3. Single Pair Algorithms

3.1. Iterative Algorithm

This is one of the oldest algorithms for transitive closure, all-pair path computation, and graph traversal[23], and is also known as the breadth-first search algorithm. The iterative algorithm can be described by the psuedo-code shown in Figure 1.

Lemma 1: Procedure Iterative(N,E,s,d) finds the shortest path between nodes s and d, if the edge costs C(u,v) are non-negative.

The optimality property, coupled with the termination condition of the procedure, guarantees discovery of the shortest path between s and d. The procedure terminates when the frontierSet becomes empty, i.e. the ExploredSet contains all nodes which have a path from s. This completes the proof.

```
Procedure Iterative( N, E, s, d);
begin
foreach u in N do { C(s,u) = ∞; C(u,u) = 0; path(u,v) := null;}
frontierSet := [ s ];
while not_empty(frontierSet) do
{ foreach u in frontierSet do
   { frontierSet := frontierSet - [u];
    fetch( u.adjacencyList);
    foreach <v, C(u,v)> in u.adjacencyList
        { if C(s,v) > C(s,u) + C(u,v) then
          { C(s,v) := C(s,u) + C(u,v); path(s,v) := path(s,u) + edge(u,v);
            if not_in(v, frontierSet) then frontierSet := frontierSet + [v];
          }
        }
   }
}
end;
```

Figure 1. Iterative Algorithm

We note that the iterative algorithm cannot be terminated before exploring the entire graph to find the shortest path between two nodes. It cannot reduce its work for paths with few edges, because its execution time is not sensitive to path length.

3.2. Dijkstra's Algorithm

Dijkstra's algorithm has been influential in path computation research[22], but it has not been used a great deal because of its relatively poor CPU-cost. However, it provides competent worst-case I/O cost for partial transitive closure and single-pair path-computation problem[22]. The psuedo-code in figure 2 describes the algorithm.

```
Procedure Dijkstra( N, E, s, d);
begin
foreach u in N do { C(s,u) = ∞; C(u,u) = 0; path(u,v) := null;}
frontierSet := [ s ]; exploredSet := emptySet;
while not_empty(frontierSet) do
{ select u from frontierSet with minimum C(s, u);
  frontierSet := frontierSet - [u]; exploredSet := exploredSet + [u];
  if (u = d) then terminate
  else { fetch( u.adjacencyList);
         foreach <v, C(u,v)> in u.adjacencyList
         if C(s,v) > C(s,u) + C(u,v) then
         { C(s,v) := C(s,u) + C(u,v); path(s,v) := path(s,u) + (u,v);
           if not_in(v, frontierSet U exploredSet) then
             frontierSet := frontierSet + [v];
         }
    }
}
end;
```

Figure 2: Dijkstra's algorithm

Lemma 2: Assuming that the graph does not contain negative edge costs, then as soon as the destination node d in the frontierSet is selected and removed, its path(s,d) contains the shortest path from source node s to d.

The proof for this lemma is provided in [22]. We note that the procedure terminates after the iteration which selects destination node d as the best node in the frontierSet. The procedure can terminate quickly if the shortest path from s to d has fewer edges, since in many cases it does not have to examine all nodes to discover the shortest path.

3.3. Best First A* Algorithms

Best-first search has been a framework for heuristics which speed up algorithms by using semantic information about a domain. It has been explored in database contexts for single-pair path computation[24]. A* is a special case of best-first search algorithms. It uses an estimator function f(u,d) to estimate the cost of the shortest path between node u and d. A* has been quite influential due to its optimality properties[25]. The basic steps of these algorithms are described in figure 3. Best-first search without estimator functions is not very different from Dijkstra's algorithm.

```
Procedure A*( N, E, s, d, f);
begin
foreach u in N do { C(s,u) = ∞; C(u,u) = 0; path(u,v) := null;}
frontierSet := [ s ]; exploredSet := emptySet;
while not_empty(frontierSet) do
{ select u from frontierSet with minimum ( C(s,u) + f(u,d) );  **
  frontierSet := frontierSet - [u]; exploredSet := exploredSet + [u];
  if (u = d) then terminate
  else { fetch (u.adjacencyList);
         foreach <v, C(u,v)> in u.adjacencyList
         { if C(s,v) > C(s,u) + C(u,v) then
           { C(s,v) := C(s,u) + C(u,v); path(s,v) := path(s,u) + (u,v);
             if not_in(v, frontierSet) then frontierSet := frontierSet + [v];
           }
         }
    }
}
end;
```

Figure 3: A* Algorithm

Lemma 3: Assuming that the graph does not contain negative edge costs and that the estimator function never overestimates the cost of the shortest path(u,d), then as soon as the destination node d (f(d,d) =0) in the frontierSet is selected and removed by A, its path(s,d) contains the shortest path from source node s to d.*

The proof for Lemma 3 is provided in[26].

We note that the procedure terminates after the iteration which selects destination node d as the best node in the frontierSet. The procedures can terminate quickly if the shortest path from s to d has fewer edges. It does not have to examine all nodes to discover the shortest path, in many cases. Furthermore, the estimator can provide extra information to focus the search on the shortest path to the destination, reducing the number of nodes to be examined.

4. Cost Analysis

We discuss the cost model and derive the cost formulas for the algorithms in this section. The analysis is based on the following representation of the data-structures used by various algorithms. Directed graphs are represented as pairs of relations: edge (S) and node (R). The edge relation S is a read-only relation and it stores the edges of the graph along with their costs. Its fields include: Begin-node, End-node, and Edge-cost. The node relation R stores the internal data-structures of various routing algorithms. Its fields include: node-id, x-coordinate, y-coordinate, status, a path to the source node and path-cost. The fields Begin-node and End-node of S contain values of node-id's from the node relation R of the graph to represent an edge in the graph, and the edge-cost contains the cost of traversing this edge. The neighbors of a node v, i.e. its adjacency list, can be found by retrieving the edges with S.Begin-node = v. An undirected graph can be represented by storing two directed-edge entries in S for each undirected edge in the graph. The relation S has a primary index (random hash) on the field S.Begin-node. Each tuple in R represents a node with status and path attributes. The status field is used to represent node lists such as frontierSet and exploredSet. Nodes with status = open represent the frontierSet. Nodes with status = closed represent the exploredSet. Node(s) with status = current represent the current node(s) being explored. Nodes with status = null are ones that are not open, "closed" or current. The path field in R points to a neighboring node on the best path to the source node. The complete path to the source node can be constructed by traversing this pointer starting at the destination node. The path-cost field for a node n represents the total cost of the path from n to the source node. The relation R has a primary index (ISAM) on node-id. Furthermore, we choose between four join algorithms for the joins computed by the algorithms.

Duplicate management in the frontierSet is an important design decision. It can be done in three ways: avoiding duplicates, removing duplicates, or allowing duplicates. Allowing duplicates leads to redundant iterations of the algorithm. Duplicates can be avoided by checking the status of the node to be null before adding it to the frontierSet. Duplicates can also be eliminated after insertion in frontierSet by duplication-elimination algorithms, but we prefer duplicate avoidance for its cost effectiveness.

Although the algebraic cost formulas do not account for system overheads, they still provide us with an insight into the relative performance of the algorithms. In order to account for other detailed costs, we implemented a query optimizer simulation in C, which was able to choose between several Select and Join strategies and produce very accurate approximations of the I/O cost of the algorithms, based on varying parameters. Using the simulation, we were able to verify the validity of the experimental results obtained by measuring the execution-times of the implementations of various algorithms on the INGRES DBMS. We use the notation in table 1 to derive the cost models.

4.1. Cost Model of the Iterative Algorithm

The Iterative algorithm can be broken into eight steps for cost modeling. The steps and their associated costs are shown in table 2.

Each iteration of the iterative algorithm is composed of steps 5,6,7 and 8. Step 6. performs a join which helps in fetching the adjacency lists of all current nodes. The cost of this join is described by a function $F(B_1, B_2, B_3)$. The input parameters for F are the sizes (in blocks) of the relations to be joined and the size of the resultant relation. The value of this function depends on the join strategy that is chosen to carry out the join. The function uses the input parameters to choose the cheapest join strategy from among four viable choices: (1) Hash Join, (2) Nested-Loop Join, (3) SortMerge Join, and (4) Primary Key Join. In the case of the Iterative algorithm the size of the resultant relation is estimated as $B_{join} = (JS * |C| * |S|)/Bf_{rs}$. Step 8 scans the relation R to determine the size of the

Symbol	Meaning
S	Edge relation = (Begin-node, End-node, Edge-cost)
R	Temporary Node relation = (Node-id, x-coordinate, y-coordinate, status, path, path-cost)
JOIN	Paths to neighbors of current node(s)
L	Path Length, Number of edges in path
I	I/O cost of creating a temporary relation
I_l	ISAM index level
S_r	Selection cardinality of nodes in R
|A|	Average Number of nodes in the adjacency list (neighbors)
n	Number of nodes in the Graph
|S|	Number of tuples in S
|R|	Number of tuples in R
|C|	Number of nodes in R that are marked current
JS	Join selectivity = $\frac{\|S \text{ join } R\|}{\|S\|*\|R\|} = \frac{\|JOIN\|}{\|S\|*\|R\|}$
D_t	Cost for deleting all tuples in a relation t
B	Disk block size (in bytes)
T_s	Tuple size of the source relation (edge-relation) S
T_r	Tuple size of the resultant relation (node-relation) R
Bf_s	Blocking factor for S = B / T_s
Bf_r	Blocking factor for R = B / T_r
Bf_{rs}	Blocking factor for RxS = $B / (T_r + T_s)$
B_s	$\|S\| / Bf_s$ Number of blocks of source relation S
B_r	$\|R\| / Bf_r$ Number of blocks of resultant relation R
B_{join}	Number of blocks of JOIN
B_c	$\|C\| / Bf_r$ Number of blocks of current-nodes
t_{read}	Time for reading one block from disk
t_{write}	Time for writing one block to disk
t_{update}	Time for updating one tuple ($t_{read} + t_{write}$)
C_j	Cost of step j

Table 1: Notations Used in Cost Analysis

Cost	Steps of Iterative Algorithm
$C_1 = I$	
	-1. Creating the resultant relation R (node-relation)
$C_2 = B_s * t_{read} + B_r * t_{write}$	
	-2. Initializing R with all nodes in S
$C_3 = 2 * (B_r * log(B_r) + B_r) * t_{update}$	
	-3. Indexing and Sorting the node-relation by node-name
$C_4 = (I_L + S_r) * t_{update} + B_r * t_{read}$	
	-4. Mark start node in R as "current" and count current-nodes
For each iteration i:	
$C^i_5 = B_r * t_{read}$	
	-5. Fetch all current-nodes from R
$C^i_6 = F(B_c, B_s, B_{join})$	
	-6. Perform a Join to get the neighbors of all current-nodes
$C^i_7 = 2 * B_r * t_{update}$	
	-7. Update status and path of nodes in R (mark explored nodes as "closed")
$C^i_8 = B_r * t_{read}$	
	-8. Scan R to count the number of current-nodes
Cost of iteration i = $\Gamma_i = C^i_5 + C^i_6 + C^i_7 + C^i_8$	
Total Cost $T = C_1 + C_2 + C_3 + C_4 + \sum_{i=1}^{B(L)} \Gamma_i$	

Table 2: Cost of Iterative BFS Algorithm

current node-list. The algorithm terminates if the current node-list is empty. The number of iterations taken by the BFS algorithm can be expressed as a function B(L), which is dependent on several factors such as the start node and the graph diameter. It is difficult to predict the number of current nodes at any point in time because they are dependent on the amount of backtracking performed by this algorithm. This backtracking is a dynamic factor dependent on the edge costs. On the average, the number of current nodes (|C|) per iteration before Step 6 is estimated to be $|R| / B(L)$ if there is no backtracking at all. The average number of tuples (paths) after the Join in Step 6 is estimated to be $|S| / B(L)$. The join selectivity (JS) is estimated to be $\frac{|S| / B(L)}{|R| / B(L) * |S|} = \frac{1}{|R|}$. The total cost can be approximated as $T = C_1 + C_2 + C_3 + C_4 + B(L) * \Gamma_{average}$, where $\Gamma_{average}$ is the average cost per iteration.

4.2. Cost Models for Dijkstra's and A* Algorithms

Cost	Steps of Dijkstra's / A* Algorithms
$C_1 = I$	
	-1. Creating the resultant relation R
$C_2 = B_s * t_{read} + B_r * t_{write}$	
	-2. Initializing the node-relation with all nodes in S
$C_3 = 2 * (B_r * log(B_r) + B_r) * t_{update}$	
	-3. Indexing and Sorting the node-relation by node-id
$C_4 = (I_L + S_r) * t_{update} + B_r * t_{read}$	
	-4. Mark start node in R as "current" and count current-nodes
For each iteration i:	
$C^i_5 = B_r * t_{read}$	
	-5. Find the minimum cost of nodes in the frontierSet
$C^i_6 = B_r * t_{update}$	
	-6. Fetch the minimum cost node X from the frontierSet and mark it "current"
$C^i_7 = F(B_c, B_s, B_{join})$	
	-7. Perform a Join to get the neighbors of node X
$C^i_8 = (B_r * t_{read} * B_{join} + B_{join} * (t_{read} + t_{write}))$	
	-8. Scan R and the result of Step 7 and update the status and node-costs in R
$C^i_9 = B_r * t_{update}$	
	-9. Update status of nodes in R by marking explored nodes as "closed"
$C^i_{10} = B_r * t_{read}$	
	-10. Scan R to count the number of current-nodes
Cost of iteration i = $\Gamma_i = C^i_5 + C^i_6 + C^i_7 + C^i_8 + C^i_9 + C^i_{10}$	
Total Cost $T = C_1 + C_2 + C_3 + C_4 + \sum_{i=1}^{Z(n,L)} \Gamma_i$	

Table 3: Cost Model for Dijkstra and A* (version 3)

Dijkstra's algorithm and A* (version 3) can be decomposed into 10 steps for cost analysis. The steps and their associated costs are shown in table 3. The number of blocks occupied by the paths in JOIN is expressed as $B_{join} = (JS * 1 * |S|) / Bf_{rs} = |A| / Bf_{rs}$

The number of iterations taken by any algorithm is denoted by a function Z(n,L). In general, Z(n,L) for a given algorithm depends on the number of nodes in the graph and the path length from the source to the destination. For long paths, the function Z(n,L) could equal the number of nodes in the graph (n). For short paths, Z(n,L) may be much smaller than the total number of nodes.

In each iteration, A* (version 3) and Dijkstra perform similar computations. The main difference appears in the selection of the minimum-cost node to expand at each iteration. Dijkstra minimizes only the actual cost for this selection, whereas A* (version 3) selects the node with the minimum actual + heuristic cost. This makes C_5 and C_6 depend on the algorithm. Since it is difficult to algebraically predict the number of iterations, we extract it from the trace of the actual execution of the algorithms. The average join selectivity in Step 7 for both these algorithms is JS = $|A| / |S|$ because there can only be one current node at each iteration and it can have |A| neighbors. The average number of neighbors for each node in a grid graph is four (4). The total cost for A* (version 3)

and Dijkstra can be approximately $T = C_1 + C_2 + C_3 + C_4 + B(L) * \Gamma_{average}$, where $\Gamma_{average}$ is the average cost per iteration.

4.3. Example

Using the algebraic cost formulas described in Tables 2 and 3, we can compute the algorithm costs for different path lengths. For illustration, we assume that all the algorithms choose the nested-join approach for Step 7 of the algebraic cost models. Under this assumption, the join function can be expressed as $F(B_1, B_2, B_3) = B_1 * t_{read} + (B_1 * B_2) * t_{read} + B_3 * t_{write}$. Let us also assume certain average values for the size of the JOIN relation, the number of current node(s) |C|, and the join selectivities for the different algorithms, and also for several other parameters for the illustration to be those shown in Table 4A.

Parameter	Value	Parameter	Value
I	0.5 units	T_s	32 bytes
I_l	3	T_r	16 bytes
S_r	1	Bf_s	128 records/block
|A|	4	Bf_r	256 records/block
|S|	3480 edges	Bf_{rs}	86 records/block
|R|	900 nodes	t_{read}	0.035 units
D_t	0.5 units	t_{write}	0.05 units
B	4096 bytes	t_{update}	0.085 units

Table 4A: Parameter Values

Let us assume the ratio *Number of nodes in the graph* |R| / *Number of iterations N of the algorithm* as the average number of current-nodes per iteration of the iterative algorithm. The join selectivity for the same algorithm is JS = $1/|R|$. Therefore, B_{join} for the iterative would be = $|S| / (B(L) * Bf_{rs})$. Since in each iteration of the A* (version 3) and Dijkstra's algorithms, there exists exactly one current node, therefore the average join selectivity would be JS = *Average number of neighbors of a node* |A| / *Total number of edges* |S|. The B_{join} for these algorithms would be = $|A| / Bf_{rs}$. Let us take the number of iterations (refer to table 5, section 5) from the execution trace of the EQUEL programs to predict the execution time.

Algorithm / Path	Horizontal	Semi-Diagonal	Diagonal
Dijkstra	1055.6	1656.8	1941.2
A* (version 3)	66.7	881.2	1809.8
Iterative	176.9	176.9	176.9

Table 4B: Estimated costs, 30x30 Graph, 20% Variance on edge cost

Table 4B shows an estimate of the costs incurred by each algorithm on a 30x30 graph. We note the similarity of these estimates to the actual experimental results shown in Figure 6.

5. Experimental Analysis of Performance

We evaluated the algorithms in two stages. First, the algorithms implemented in EQUEL were run on the graphs and we obtained measurements of processing time. The experiments were repeated a number of times to arrive at average execution times. We also used Ingres in single-user mode to reduce overhead, due to contention for resources. We then designed a simulation using our algebraic cost models to verify that the time measurement was a reasonable comparator. The simulation took the number of iterations from the execution trace of the EQUEL programs to predict the execution-time. With our algebraic cost models and simulation we were able to predict actual execution time within ten percent. We report the results from the EQUEL implementation in this section. We also evaluate three different versions of the A* algorithm to determine the effect of different estimator functions and implementation decisions.

5.1. Comparative Performance Evaluation in Synthetic Grid

Grid Benchmark Computation: The benchmark path computation is based on an undirected graph with cycles for single-pair path computations. The node pairs and graphs for benchmark computations are shown in Figure 4.

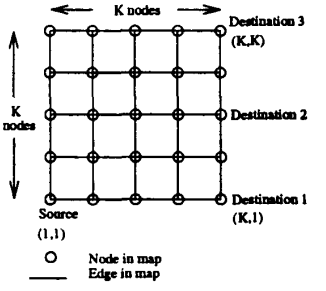

Figure 4: Synthetic graphs and node pairs

The synthetic graph represents two-dimensional grids with 4 neighbor nodes. The grid includes k*k nodes, with k nodes along each row and each column, and with edges connecting adjacent nodes along rows and columns. We chose three node pairs for path computation: diagonally opposite nodes, linearly opposite nodes and a random-node pair.

Two-dimensional grid graphs are typical of the navigation problems in an unstructured environment. These have been used in prior studies of single-pair path computation[24]. The choice of a synthetic graph also simplifies interpretation of results.

Candidate Algorithms: We chose three algorithms: Iterative, Dijkstra's, and A* (version 3).

Variable Parameters: The parameters for the first set of experiments included graph size, path length, and edge cost models. Experiments were carried out for the following graph sizes: 10*10, 20*20, and 30*30 node grids. We used three cost models for the edges: a uniform cost, a uniform cost with a small variance, and a skewed cost. The uniform-cost model assigns unit cost (i.e. 1) to each edge in the graph. A uniform cost with 20% variation assigns a cost of 1 + 0.2*U[0,1], where U[0,1] is a random number with uniform distribution between 0 and 1. This cost model will change the degree of backtracking required in the execution of estimator-based algorithms such as A* (version 3). Finally, the skewed-cost model assigns a small cost to the edges [(1, i), (1, i+1)] on the bottom of the grid and the edges [(k, i), (k, i+1)] on the

right side of the grid. This model eliminates backtracking from estimator-based A* (version 3), creating the best case for that version.

We have chosen problem instances of grid graphs and parameters, such as graph sizes and path length, based on the literature[13, 24]. The parameter choice of path length is motivated by our application. Other parameters including edge-cost models are exploratory, under the hypothesis that these will impact the performance of estimator-based path computation algorithms significantly.

5.1.1. Effect of Graph Size

The execution time for all the figures in this section represents a sum of the user time and the system time. The algorithms compute the path between diagonally opposite nodes (i.e. the longest path) to compare the worst-case performance of the various algorithms. The performance of A* (version 3), Dijkstra, and the Iterative BFS is shown by the graph in Figure 5. Table 5 gives the number of iterations for each of the algorithms for different graph sizes.

The graph size significantly affects the performance of A* (version 3) and Dijkstra's algorithms for computing the longest diagonal path. The number of iterations and execution times for A* (version 3) and Dijkstra grow linearly with the number of nodes in the graph. The iterative algorithm is affected to a lesser extent. Its execution time and number of iterations to be performed for computing a diagonal path grows sublinearly with the number of nodes in the graph.

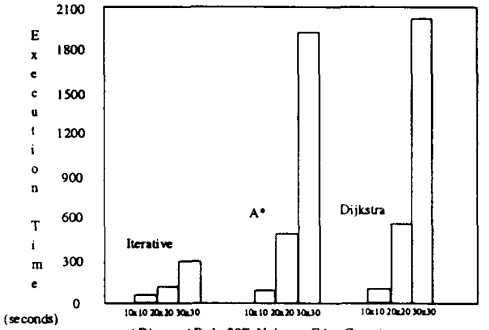

Figure 5: Effect of graph size on execution time

Algorithm / Graph Size	10 x 10	20 x 20	30 x 30
Dijkstra	99	399	899
A* (version 3)	85	360	838
Iterative	19	39	59

Table 5: Effect of Graph Size on Number of Iterations
(20% Edge Cost Variance, Diagonal Path)

5.1.2. Effect of Path Length

The effect of path length on the performance of the three algorithms is shown in figure 6 and table 6. Figure 6 shows the execution time of the algorithms for different path-length values. Table 6 gives the number of iterations for each of the algorithms for the different path lengths. As we can see in figure 6, A* (version 3) outperforms Iterative and Dijkstra's algorithm for horizontal paths. For the remaining two path lengths, the Iterative algorithm performs better.

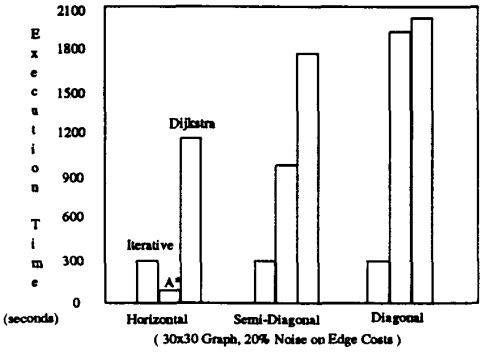

Figure 6: Effect of path length on execution time

Algorithm / Path	Horizontal	Semi-Diagonal	Diagonal
Dijkstra	488	767	899
A* (version 3)	29	407	838
Iterative	59	59	59

Table 6: Effect of Path Length on Number of Iterations
(20% Edge Cost Variance, 30x30 Graph)

5.1.3. Effect of Edge Cost Models

The effect of edge-cost models on the performance of the three algorithms is shown in figure 7 and table 7. Figure 7 shows the execution time of the algorithms for three cost models, a uniform cost, edge costs with 20% variance, and skewed edge-costs. Table 7 gives the number of iterations for each of the algorithms using the different edge cost models.

The execution cost incurred by Iterative BFS depends on the edge-cost model, even though the BFS algorithm does not take advantage of the cost information on the edges. This is due to the possibility of reopening a node and revising the path. Since the remaining two algorithms use edge costs to drive the search, it is expected that they are affected by variations in the edge-cost model. In the 20% edge cost variance model, A* (version 3) requires more backtracking (360 iterations vs. 85 iterations for uniform cost), and therefore has higher execution times. When the edge costs are skewed (i.e. the edges from the source to destination have a much lower edge cost), A* (version 3) and Dijkstra's algorithm perform very well. Backtracking is eliminated as the sub-path generated at every iteration is optimal.

5.2. Comparative Performance on Minneapolis Road Map

The Minneapolis road map data consisted of 1089 nodes and 3300 edges that represented highway and freeway segments for a 20-square-mile section of the Minneapolis area. The data about each segment includes x and y position of the two nodes, average speed for the segment, average occupancy, and road type. For these preliminary

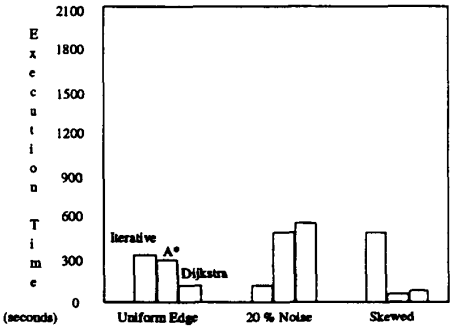

Figure 7: Effect of edge cost models on execution time

Algorithm / Cost	Uniform Cost	20% Variance	Skewed
Dijkstra	399	399	48
A* (version 3)	189	360	38
Iterative	39	39	56

Table 7: Effect of Edge Cost Models on Iterations
(20x20 Graph, 20% Edge Cost Variance, Diagonal Path)

experiments, we used only the distance between edges as the edge cost. Edges that connected freeway segments were one-way, making the resulting graph directed as opposed to undirected. The road segments were obtained by digital imaging of the area followed by regional identification. The map is shown in Figure 8. The more dense center region is the downtown Minneapolis region. In this region, the highways and freeways are not parallel to the x or y axis. The outlying areas do show a more gridlike pattern of roads, except where lakes interrupt in the lower left corner and where the Mississippi river flows north to southeast in the upper right quadrant of the map.

Table 8 and figure 9 show the number of iterations and execution times of the various algorithms for 4 different paths. The paths that we considered included two diagonal paths (from A to B and from C to D) and two short paths (from D to G and E to F). The results are consistent with our predictions from the synthetic grid and the algebraic simulations. The graph size is slightly larger (1089 nodes) compared to the 30x30 synthetic grid (900 nodes). The iterative algorithm must perform approximately the same number of iterations, regardless of path length. The cost per iteration will change depending on amount of backtracking that must be done. In the diagonal case, the path from point A to point B is against the slope of the downtown area, resulting in more backtracking. The lower cost for the C to D path may result from less backtracking due to the downtown area grid being almost parallel to the path. The iterative algorithm is superior to the estimator-based algorithms when the number of iterations is high. With a smaller number of iterations (i.e. shorter path length), the estimator-based algorithms clearly outperform the iterative algorithm. The path from D to G required only 17 iterations for the optimal A* algorithm, resulting in a cost that is 95% smaller than that of the iterative algorithm. Similar results were obtained for the second short path from E to F.

Figure 8: Minneapolis Road Map

Algorithm / Path	A to B	C to D	G to D	E to F
Iterative	55	51	55	41
A* (version 3)	453	266	17	64
Dijkstra	1058	1006	105	307

Table 8: Effect of Path length and orientation on Iterations
(Data from Minneapolis)

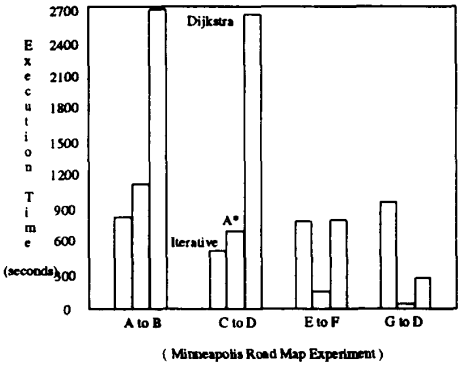

Figure 9: Minneapolis Road Map Results

5.3. Analysis of Design Decisions in Implementing A*

We examined two design issues in implementing the estimator-based A* algorithms: frontierSet management and choice of estimator function.

FrontierSet operations account for most of the work in each iteration of estimator-based path-computation algorithms. We examine two implementations of the frontierSet: as an independent relation, and as an attribute in the nodes relation. The frontierSet can be managed as an independent relation. Addition of new reachable nodes can be implemented by insert operations, with deletion of an unexplored node implemented by a delete operation. Selection of the best node can be implemented by a scan of the frontierSet. This implementation requires adjustment of the index on the part of relation R, representing the frontierSet. Alternatively, the frontierSet can be implemented by adding an attribute status to each node in the node relation. The attribute status can take four values: open, closed, current, and null. Addition of reachable nodes can be implemented by a replace operation,

which changes the status of newly reachable nodes to open. Deletion of an explored node can be implemented by a replace operation, which changes the status attribute to closed. Selection of the best node in the frontierSet could be implemented by a scan. It has been argued that the latter implementation is cost effective, due to the smaller overhead of index maintenance[24]. We use the QUEL[24] command REPLACE instead of APPEND and DELETE in implementing all the algorithms in section 3.

Estimator functions are used to select the best node on the frontierSet to be explored in current iteration. A perfect estimator function helps the algorithm to discover the shortest path by exploring the minimum number of nodes in the graph[26]. We examine two estimator functions: euclidean distance and manhattan distance. Consider two nodes located at coordinates (x1, y1) and (x2, y2) respectively. The euclidean distance between the nodes is defined to be $\sqrt{((x1-x2)^2+(y1-y2)^2)}$. It always underestimates the cost of the shortest path between nodes. The manhattan distance between them is defined to be $(|x1-x2|+|y1-y2|)$. Manhattan distance is a perfect estimate of the length of the shortest path between nodes in grid graphs with a uniform cost model. The manhattan distance estimator function is not perfect on the Minneapolis data set, however, due to the non-uniform costs between edges. In fact, the manhattan distance on the Minneapolis data set is not always an underestimate, thus for this graph, use of the manhattan distance does not guarantee an optimal solution.

We examined three implementations of estimator based-algorithms in this experiment: A* version 1, A* version 2, and A* version 3. Version 1 implements the frontierSet as a separate relation and uses the euclidean estimator function. Version 2 implements the frontierSet via a status attribute in the node relation and uses the euclidean estimator function. Version 3 implements the frontierSet via a status attribute in the node relation and uses the manhattan estimator function.

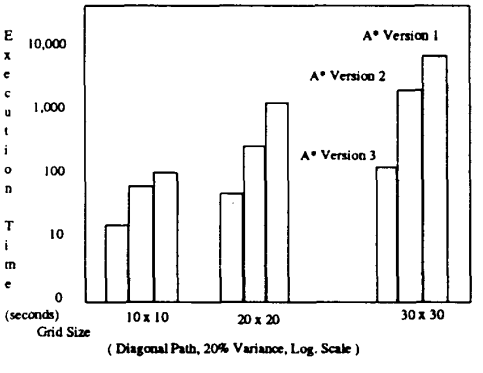

Figure 10: Effect of graph size on Execution Time of Versions

5.3.1. Effect of FrontierSet Management

The effect of frontierSet management can be examined by comparing the performances of version 1 and version 2. Version 1 and version 2 show quite different performance for different graph sizes as shown in Figure 10. As the graph size increases, the performance of A* version 1 becomes worse than version 2. This is attributed to the fact that APPEND and DELETE operations cost more than the REPLACE operation in Ingres [24]. Another reason for this difference in performance is that the A* version 2 updates the cost and flag of a neighboring node only if the original cost of that neighbor is greater than the cost of traveling through the "current" node. This step further combines the APPEND and DELETE in A* version 1 to a REPLACE in version 2.

Figure 11 shows that A* version 1 performs worse than version 2 for a uniform-cost grid. However, it does better than version 2 for a skewed graph, because of the higher initialization costs for version 2. A* version 1 expands nodes and appends them to the resultant relation as it goes along, unlike version 2, which begins by loading all neighbors into the resultant relation. Both versions perform worst on the graph with 20% variation in edge cost.

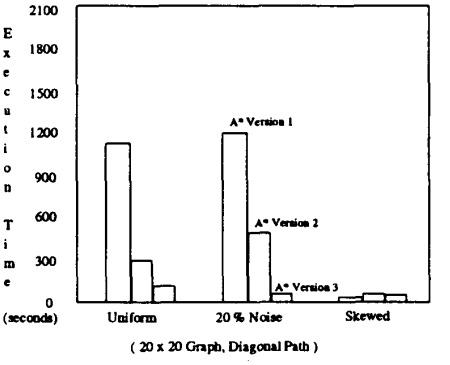

Figure 11: Effect of varying edge cost on Execution Time of Versions

We can see from figure 12 that the execution time for all versions increases as the path length increases. A* version 1 starts out much better than version 2, but for longer paths it falls behind. The poor performance of versions 2 in the straight-line path could be attributed to higher initialization costs.

5.3.2. Effect of Estimator Functions

The effect of estimator functions can be studied by comparing the performance of reducing execution time for most cases of path computation. Therefore, choosing a good estimator is of the utmost importance. The effect of estimator functions on performance can be studied by comparing the performances of version 2 and version 3 of A* in figures 10, 11, and 12. Figure 10 shows that the execution cost of version 3 does not grow as rapidly as the execution cost of version 2. For the 30x30 grid, version 3 performs ten times better than version 2. A* version 2 has a comparatively higher cost of execution in the the graph with the 20% edge-cost variation (figure 11), whereas version 3 has a comparatively higher cost in a uniform-cost graph. Similarly, in Figure 12 we observe that execution time for version 3 is almost linear to the increase in path length, whereas the execution time for version 2 increases very rapidly when compared to the increase in path length. Manhattan distance also outperforms euclidean distance for grid graphs.

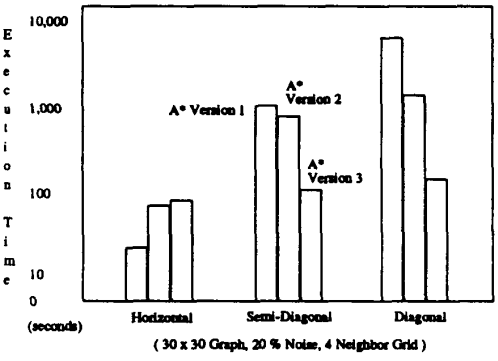

Figure 12: Effect of Path on Execution Time of Versions

6. Conclusions and Future Work

We have carried out algebraic and implementation based evaluations of the performance of three algorithms for computing the shortest path between a pair of points (source, destination). A* outperforms the iterative algorithm if the path[source, destination] is much smaller than the diameter of the graph, or if the edge-cost distribution is skewed. The iterative algorithm outperforms A* and Dijkstra's algorithms if the path[source, destination] is comparable to the diagonal of the graph with non-skewed edge-cost distributions. Estimator functions can reduce the number for nodes explored to provide satisfactory performance on graphs with hundreds of nodes. The performance of estimator-based algorithms is determined by the quality of the estimator as well as by the implementation decisions. Management of the frontierSet and the exploredSet are the key design decisions affecting performance. Algorithms such as A*, with the manhattan-distance estimator function are not guaranteed to be optimal. These algorithms were able to find a good path very quickly. In real applications such as the ATIS, the tradeoff between optimality and speed may allow for sub-optimal algorithms to speed the processing. Our future work will include analyzing the algorithms to find a way to characterize the tradeoff.

7. Acknowledgements

This work was begun to accommodate the needs of an advanced-traveller information system being designed for the GuideStar project. This work was supported by the Minnesota Dept. of Transportation and the Center for Transportation Studies at the University of Minnesota.

8. References

1. J. H. Rillings and R. J. Betsold, Advanced Driver Information Systems, *Trans. on Vehicular Technology* 40(1)IEEE, (February 1991).

2. G. F. King and T. M. Mast, Excess Travel: Causes, Extent and Consequences, *Transport. Res. Rec., 1111, TRB, Nat. Res. Council*, (1987).

3. J. Eder, Extending SQL with General Transitive Closure and Extreme Value Selections, *Trans. on Knowledge and Data Engineering* 2(4)IEEE, (1990.).

4. M. Mannino and L. D. Shapiro, Extensions to Query Languages for Graph Traversal Problems, *Trans. on Knowledge and Data Eng.* 2(3)IEEE, (Sept. 1990).

5. R. Agrawal, Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries, *Trans. on Software Eng.* 14(7)IEEE, (July 1988).

6. H. Lu, K. Mikkilineni, and J. P. Richardson, Design and Evaluation of Algorithms to Compute the Transtive Closure of a Database Relation, *Proc. Intl. Conf. on Data Eng.*, IEEE, (1987).

7. J. Han, G. Qadah, and C. Chaou, Processing and Evaluation of Transitive Closure Queries, *Conf. on Extending Database Technology*, EDBT, (1988).

8. H. V. Jagadish, R. Agrawal, and L. Ness, A Study of Transitive Closure As a Recursion Mechanism, *Proc. Conf. on Management of Data*, ACM, (1987).

9. H. Jagadish, A Compressed Transitive Closure Technique for Effective Fixed-Point Query Processing, *Proc. Intl. Conf. on Expert Database Systems*, Benjamin Cummings, (1989).

10. Shaul Dar and H. V. Jagadish, A Spanning Tree Transitive Closure Algorithm, *Proc. Intl. Conf. on Data Eng.*, IEEE, (1992).

11. R. Agrawal, A. Borgida, and H. V. jagadish, Efficient Management of Transitive Relationships in Large Data Bases, *Proc. Conf. on Management of Data*, ACM SIGMOD, (1989).

12. R. Agrawal and H. V. jagadish, Hybrid Transitive Closure Algorithms, *Proc. Int. Conf. Very Large Data Bases*, VLDB, (1990).

13. Y. Ioannidis, On the Computation of the Transitive Closure of Relational Operators, *Proc. Intl. Conf. on Very Large Data Bases*, VLDB, (1987).

14. P. Valduriez and S. Khoshafian, Transitive Closure of Transitively Closed Relations, *Proc. Itl. Conf. on Expert Database Systems*, Benjamin Cummings, (1989).

15. I. F. Cruz and T. S. Novell, Aggregate Closure: An Extension of Transitive Closure, *Intl. Conf. on Data Engineering*, IEEE, (1989).

16. Y. E. Ioannidis and R. Ramakrishnan, An Efficient Transitive Closure Algorithm, *Intl. Conf. on Very Large Data Bases*, VLDB, (1988).

17. B. Jiang, A Suitable Algorithm for Computing Partial Transitive Closures in Databases, *Intl. Conf. on Data Engineering*, IEEE, (1990).

18. F. Banchillon and R. Ramakrishnan, An Amateur's Introduction to Recursive Query Processing Strategies, *Conf. on Management of Data*, ACM SIGMOD, (1986).

19. F. Banchillon, Naive Evaluation of Recursively Defined Relations, *On Knowledge Base Management Systems - Integrating Database and AI Systems*, Springer Verlag (Ed. Brodie, Mylopolous), (1985).

20. P. Valduriez and H. Boral, Evaluation of Recursive Queries Using Join Indices, *Intl. Conf. Expert Database Systems*, (1986).

21. Y. Kusumi, S. Nishio, and T. Hasegawa, File Access Level Optimization Using page Access Graph on Recursive Query Evaluation, *Proc. Conf. on Extending Database Technology*, EDTB, (1988).

22. B. Jiang, I/O Efficiency of Shortest Path Algorithms: An Analysis, *Proc. Intl. Conf. on Data Eng.*, IEEE, (1992).

23. T. H. Cormen, C. E. Leiserson, and R. Rivest, Single source shortest paths (Ch#25), *Introduction to Algorithms*, The MIT Press, (1990).

24. R Kung, E. Hanson, Y. Ioannidis, T. Sellis, L. Shapiro, and M. Stonebraker, Heuristic Search in Data Base Systems, *Proc. Expert Database Systems*, Benjamin Cummings Publications, (1986).

25. R. Detcher and J. Pearl, *Generalized best-first strategies and the optimality of A**, UCLA-ENG-8219, University of California, Los Angeles ().

26. D. Galperin, On the optimality of A*, *Artificial Intelligence* 8(1) pp. 69-76 (1977).