Proc. 1996 Australian New Zealand Conf. on Intelligent Information Systems, 18-20 November 1996, Adelaide, Australia. Editors, Narasimhan and Jain

A Dynamic Multi-source Dijkstra's Algorithm for Vehicle Routing

Peter W. Eklund, Steve Kirkby[1], Simon Pollitt
(peklund, skirkby, sepollit, @gisca.adelaide.edu.au)
Department of Computer Science, University of Adelaide, 5005, South Australia.
[1]National Key Centre for Social Applications of GIS, University of Adelaide, 5005, South Australia.

Abstract
This paper discusses the implementation of Dijkstra's classic double bucket algorithm for path finding in connected networks. The work reports on a modification of the algorithm embracing both static and dynamic heuristic components and multiple source nodes. The modified algorithm is applied in a 3D Spatial Information System (SIS) for routing emergency service vehicles.

The algorithm has been implemented as a suite of modules and integrated into a commercial SIS software environment. Genuine 3D spatial data is used to test the algorithm on the problem of vehicle routing and re-routing under simulated earthquake conditions in the Japanese city of Okayama. Coverage graphs were also produced giving contour lines joining points with identical travel times.

Keywords: Dijkstra's shortest path finding algorithm

INTRODUCTION

The paper reports the basis of a tool for emergency routing and response system to support emergency coordinators perform routing and analysis during civil emergency and rehearsal simulation. It also has potential applications in many other community services, for example public transport, road traffic and communication networks. To effectively distribute emergency services vehicles, an efficient routing process must be in place. Even in times of normal activity, routing and timing analysis should be performed to locate "hot-spots" (areas of slow response) and to look at ways of alleviating congestion.

This paper outlines a number of different implementations of Dijkstra's algorithm, and then describes the modifications used in this work. In particular, we report on a modification of the double bucket Dijkstra algorithm that embraces both static and dynamic heuristic components and multiple source nodes. This adaptation exists elsewhere in the literature but not in the combination we describe. Neither has such a configuration of Dijkstra's algorithm been applied to realistic data from a well-understood problem in 3D spatial information processing. A brief discussion of the algorithms performance on realistic test data is given, and the simulation and modelling results are presented.

DIJKSTRA'S SHORTEST-PATH FINDING ALGORITHM

A large amount of work has been done on finding shortest paths through abstract[2,3,4,5,6]. Road networks (like communications networks) are well-connected and Dijkstra's algorithm implementations have a lower computational complexity than suggested in [3,4].

Input to the algorithms is in the form of a digraph, $D=\{V,E\}$, where V represents the nodes (road intersections) and E represents the arcs (road segments). All arcs are directed, so bi-directional roads are represented using 2 arcs. Each node has associated with it a potential, $d(v)$ where $v \in V$, in this case representing the current shortest distance to the nearest source point.

There may be impedance on an arc corresponding to gradient or other flow limitations. Impedance can be both static or dynamic so node potentials are constantly changing. Node potentials are the time to traverse an arc. Other variables used by the algorithms are: n (number of nodes, $|V|$); m (number of arcs $|E|$); C (largest static travel time along a single arc); $S(v)$ (current status tag of node v); $P(v)$ (parent node of node v).

Labelling (or Scanning)

Any implementation of Dijkstra's algorithm depends heavily on the labelling (or Scan) function. Every implementation differs in the manner labelled nodes are selected. For each v, $S(v)$ has one of three values: unreached, labelled or scanned. Initially, $\forall v \in V$ (except source nodes), $v=\infty$ and $S(v)=$*unlabelled*. Source nodes have $d(v)=0$ and status *labelled*.

The *Scan* process has as input a partially processed digraph and a labelled node v. Scan processes each child node of the node v. Children are determined by following arcs from v. For each child, c, the current

0-7803-3667-4/96/$5.00 ©1996 IEEE

potential, $d(c)$, is compared against the time to the parent plus the time to travel from v to c. If the new potential is lower, $d(c)$ is updated, $P(c)$ is set to v and $S(c)$ is set to *labelled*. After all children of v are processed, $S(v)$ is set as *scanned*. This operation causes *unreached* and *scanned* status nodes to become *labelled*, indicating a visit to a new node or a re-visit of an already processed node via a shorter path. *Scan* is called continually until there are no labelled nodes, when the algorithm terminates.

Selection Methods

There are many different methods to select the next node passed to *Scan*[2]. The selection function determines the computational efficiency of the algorithm. The theory behind Dijkstra's algorithm dictates the labelled node with the lowest current potential should be the next node *Scanned*. Different implementations alter the manner in which this minimum potential node is located.

The *Linear Search* method is the simplest selection algorithm[2]. All *labelled* nodes are stored (unsorted) in a list that is linearly searched to find the minimum potential node. This is a highly inefficient implementation with $O(n^2)$ performance. This implementation is shown in Algorithm 1.

```
Dijkstra_Linear:
 INPUTS:
        V, the set of nodes in the network
        E, the set of arcs in the network
 OUTPUTS:
        P, the set of parent nodes on shortest paths to nearest
                source
 VARIABLES:
        List, an unordered list of labelled nodes
        Minimum_distance, the smallest distance label of any
                node in List.
        Minimum_node, the node with potential
                Minimum_distance
let source s ∈ V have d(s)←0, S(s)←labelled
let all other nodes v ∈ V have d(s)←∞, S(s)←unreached

Insert(List,s)
while ¬Empty(List) loop
 Minimum_dist←∞
 Mimimum_node←null
 for all v ∈ List loop
  if d(v)< Minimum_dist then
   Minimum_dist ←d(v)
   Minimum_node ←v
  end if
 end loop
 if Minimum_node ≠ null then
  Scan(Minimum_node)
  for all {w|(Minimum_node,w) ∈ E ∧ S(w)=labelled} loop
   Insert(List,w)
  end loop
  Remove(List, Minimum_node)
 else
  exit loop
 end if
end loop
```

Algorithm 1: Linear Search

A number of other selection methods are discussed in[2]. These are algorithms that refine the selection process from linear search (above) through single level and approximate buckets culminating in the double bucket.

The *Double Bucket* separates nodes into "buckets" to cut down the number of nodes to search. Two levels of buckets are created to further refine the selection process: high and low buckets. If the number of low level buckets is defined to be a constant B, the high level bucket i contains all nodes with integral potentials in the range $[iB, (i+1)B - 1]$. An index L is kept, indicating the lowest non-empty high level bucket. Nodes with potentials in the range $[LB,(L+1)B - 1]$ are distributed in the low level buckets. After every low level bucket is checked, i.e. all nodes in the bucket are scanned, L is incremented. Once all high level buckets are empty, the algorithm terminates. This ensures that nodes are scanned at most once, and results in a $O(m+n(B+C/B))$ algorithm[2]. This method is shown in Algorithm 2.

```
Dijkstra_Double_Bucket:
 INPUTS:
        V, the set of nodes in the network
        E, the set of arcs in the network
        C, max(|le|), ∀e ∈ E
        s, the source node, s ∈ V
        B, a constant used to determine number of buckets
                e.g. largest power of 2 <√C
 OUTPUTS:
        P, the set of parent nodes on shortest paths to nearest
                source
 VARIABLES:
        High, an array of Lists
        Low, an array of Lists

let source s ∈ V have d(s)←0, S(s)←labelled
let all other nodes v ∈ V have d(s)←∞, S(s)←unreached

Num←⌈(C+1)/B⌉
Low: array [0..B] of List
High: array [0..Num] of List

Insert(Low[0],s)
for L in 0 to Num loop
 for P in 0 to B loop
  while ¬Empty(Low[P]) loop
   Scan(Head(Low[P]))
   for all {w|(Head(Low[P]),w) ∈ E ∧ S(w)=labelled} loop
    if d(w)>((L+1)B - 1) then
     /* INSERT IN UPPER BINS */
     Insert(High[⌊d(w)/B⌋],w)
    else
     /* INSERT IN LOWER BINS */
     Insert(Low[d(w) - LB], w)
    end if
   end loop
   Remove(Low[L], Head(Low[L]))
  end loop
 end loop
end loop
```

Algorithm 2: Double Bucket

To implement the routing system, we begin with the Double Bucket implementation of Dijkstra's algorithm. Although the most complex method to code, the algorithm has good scalability and average case efficiency.

Two departures from the algorithm discussed in [2] were made. Firstly, the method in [2] assumes node potentials are integral. This is not the case in our system, but as the integral portion is of much higher significance than the decimal portion, it is safe to truncate the decimal portion *for the purpose of bin positioning only*. The decimal portion is retained and used for all other calculations. Secondly, our implementation required multiple source points to be supported. For example, there are several hospitals that are source points in our data. This feature is achieved by inserting all source nodes as labelled nodes in the first bucket before the first *Scan*. This has the effect of conceptually processing all source nodes concurrently. The algorithm produces a digraph. Each node in the digraph has a single arc to its immediate node predecessor on the shortest path.

Heuristics

What distinguishes our work is that we have multiple source points and static and dynamic components of the potential values. A number of factors were taken into consideration when calculating the static travel time along road segments:

1. *Segment length*: the distance between the points in three dimensions.
2. *Gradient impedance*: the percentage gradient was calculated using the formula:

$$g = 100 * \Delta z / \sqrt{\Delta x^2 + \Delta y^2}$$

An impedance value was assigned using Table 1 (devised using the information in [1]).

Gradient (%)	Impedance (sec/100m)
0≤...≤5	0
5≤...≤7	4
7≤...≤10	8

TABLE 1: Impedance vs. Gradient

3. *Travelling speed*: the speed of travel along a road is based on the number of lanes in the road. The figures used are given in Table 2.

Number of Lanes	Speed (km/h)
1	20
2	50
4	80

TABLE 2: Road Width vs. Speed

In addition to the static factors on travel time, it is important to account for the previous travel path at the time a node is being scanned. This takes into account any different paths to a point that have been calculated at an earlier time. For example, it is harder to turn across a four-lane highway that to turn into a one-lane side street. It may therefore be possible to get to a point in a city more quickly by a longer route that minimises turns across on-coming traffic. Impedance values should reflect this fact. The factors used to calculate these impedances are:

1. number of lanes in the previous road segment (segment just traversed);
2. number of lanes in the new road segment (segment about to be traversed);
3. turn angle between old and new road segments

The impedances used are given in Table 3, remembering that traffic in Japan travels on the left hand side of the road so the impedances are biased for that direction.

Old Width	New Width	Left turn	Right turn	Straight
1	4	4	10	0
1	2	2	6	0
1	1	1	1	0
2	4	4	10	0
2	2	2	6	0
2	1	2	6	0
4	4	4	10	0
4	2	2	10	0
4	1	2	10	0

TABLE 3: Dynamic Impedances

RESULTS

A number of different scenarios were implemented based on data from Okayama City in Japan. This model takes into account current tectonic plate lines that can be used to simulate breaks in the road traffic network under earthquake conditions. The test data sets are: an undamaged road network; a slightly damaged network (small earthquake); and a highly damaged network (large earthquake). Damage to the network is implemented by breaking a number of road segments around the fault lines and recomputing the shortest path coverage (see Figures 1,2 and 3).

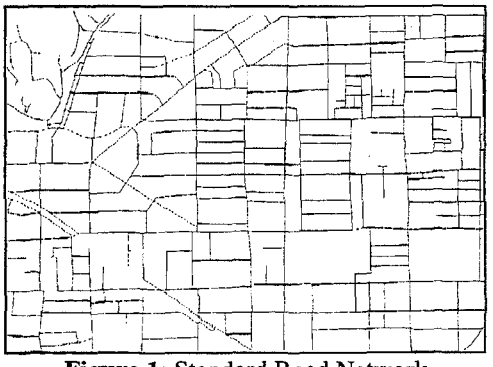

Figure 1: Standard Road Network

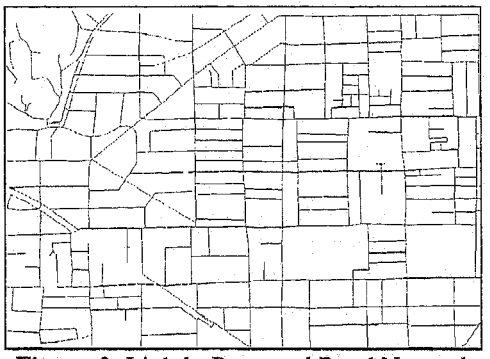

Figure 2: Lightly Damaged Road Network

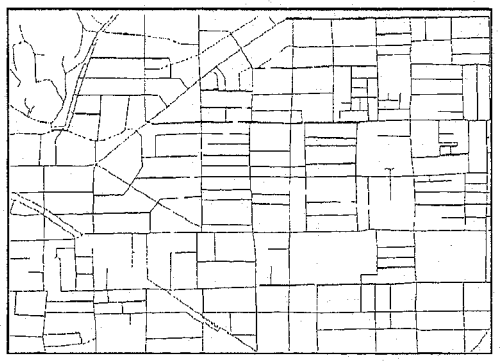

Figure 3: Highly Damaged Road Network

Coverage layers, showing all road segments travelled along in any evaluated shortest path route, are produced for each scenario. Inspection shows little difference between undamaged and lightly damaged scenarios - the algorithm finds nearby routes well. However, the coverage changes greatly for the highly damaged network - the algorithm must search for longer routes around the broken areas. The longest / slowest paths were also reported and are summarised in Table 4 (see Figures 4,5 and 6).

Layer	Longest Path	Slowest Path
Standard	3.078km	13.104min
Light Damage	3.078km	13.104min
Heavy Damage	3.185km	26.456min

TABLE 4: Timing Results

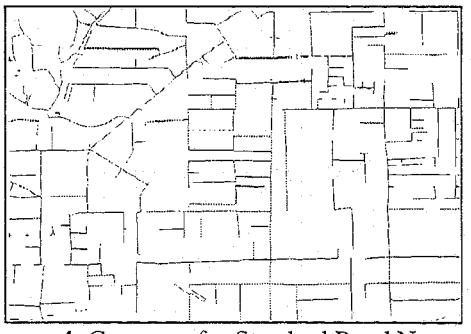

Figure 4: Coverage for Standard Road Network

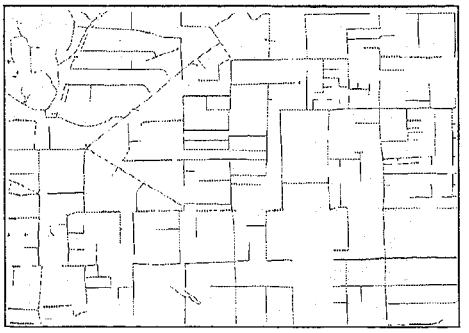

Figure 5: Coverage for Lightly Damaged Road Network

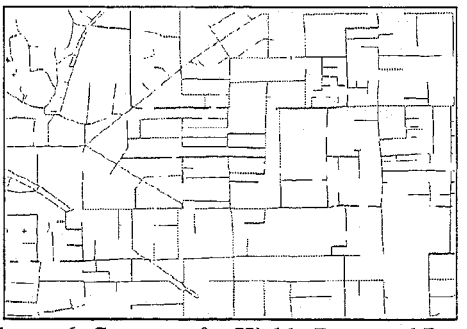

Figure 6: Coverage for Highly Damaged Road Network

The final analysis performed is the construction of time contours. The contour lines join points requiring equal travel time from the nearest source. The timing results from Table 4 lead us to believe that time contours should be similar for the undamaged and lightly damaged networks - this is the case as shown in Figures 7 and 8.

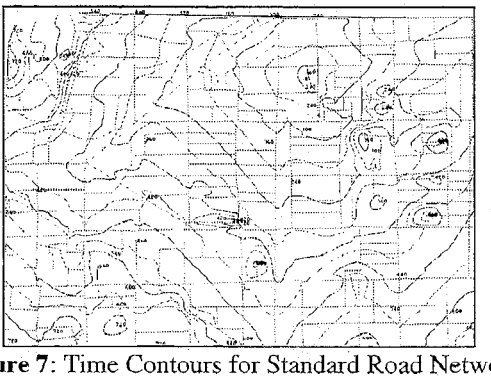

Figure 7: Time Contours for Standard Road Network

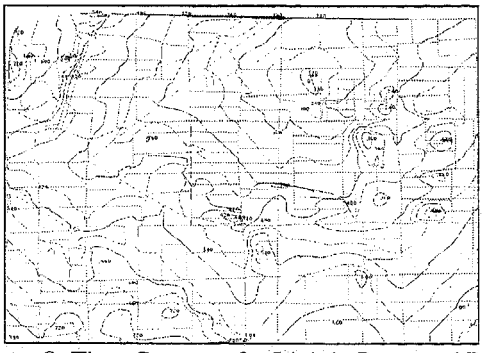

Figure 8: Time Contours for Lightly Damaged Road Network

The highly damaged network has a greatly different time contour graph (see Figure 9). The area where all the contours are very closely spaced corresponds to the location of the road breaks. Closely spaced contours indicate rapidly changing values and the results are those one would expect - to traverse that part of the network takes considerable time. These graphs indicate the success of the shortest path finding method, as network disruptions are avoided.

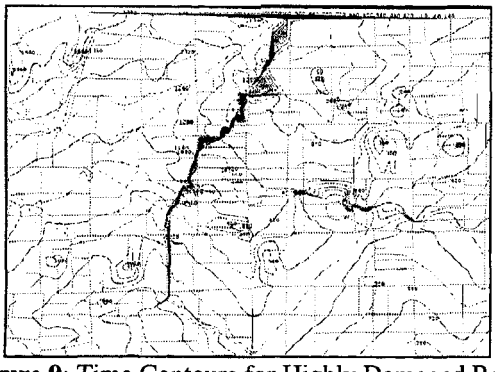

Figure 9: Time Contours for Highly Damaged Road Network

CONCLUSIONS

The direct correlation between the road networks (damaged and undamaged), the coverage layers and the time contour graphs indicate the success of the routing engine. The modified Dijkstra's algorithm, taking account of both static and dynamic search heuristics and using multiple sources, is highly efficient. A shortest path coverage for our test data is computed in under three seconds (not including data transfer time between the SIS package and the routing engine) with algorithm complexity $O(m+n(B+C/B))$.

The routing engine is currently being updated to include a number of other heuristics (including power-to-weight ratios), and to become part of a fire fighting management system. Other applications under consideration include integration into an urban modelling environment to simulate and analyse various resources, both civil and emergency services, and to assist in their management.

REFERENCES

1. AASHTO, "Handbook on Highway Design", Web Document, http://www.gis-trans.com/.
2. B.V. CHERKASSY, A.V. GOLDBERG, T. RADZIK, "Shortest Path Algorithms: Theory and Experimental Evaluation" Technical Report, Stanford University, 1993.
3. R.B. DIAL, "Algorithm 360: Shortest Path Forest with Topological Ordering", Communications of the ACM, 12(11), November 1969.
4. E.W. DIJKSTRA, "A Note on Two Problems in Connection with Graphs", Numerical Mathematics, (1):269-271, 1959.
5. L.R. FORD, Jr, D.R. Fulkerson, "Flows in Networks", Princeton University Press, 1962.
6. G. GALLO, S. PALLOTINO, "Shortest Path Algorithms", Annals of Operations Research, (13):3-79, 1988.