

Fundamentals of Algorithms Lab Project Report



on Internet Routing

SESSION 2013-14

Project Topic

Internet Routing (Analysis and Implementation)

Team Member

Shivam Khandelwal [12103506]

Algorithms Covered

The following algorithms have been studied and implemented:

- Dijkstra's Algorithm
- Midpoint Routing Algorithm
- Two step Routing Algorithm
- Apex Angle Routing Algorithm
- Bowyer-Watson Algorithm
- Voronoi Algorithm (Overlay and Cells only)

Implementation

The routing and shortest-path search algorithms have been implemented in PHP, with a web interface in which a user can see the paths.

Graph is generated of N vertices which the user supplies randomly, and the algorithms are implemented on it dynamically.

The project was made online on a VPS, after installing Apache (2.2.15), PHP (5.3.3) at Port 80.

Importance of Routing Algorithms?

Routing is the process of selecting best paths in a network. In the past, the term routing was also used to mean forwarding network traffic among networks. However this latter function is much better described as simply forwarding. Routing is performed for many kinds of networks, including the telephone network (circuit switching), electronic data networks (such as the Internet), and transportation networks. The project is concerned with routing in electronic data networks using packet switching technology.

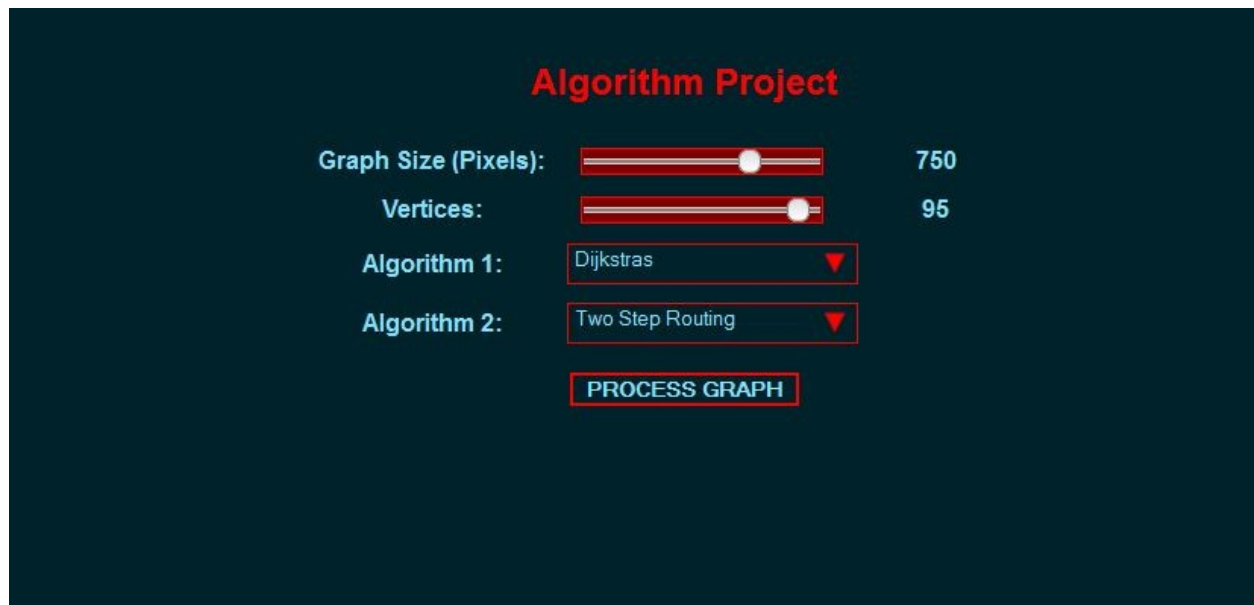
In packet switching networks, routing directs packet forwarding (the transit of logically addressed network packets from their source toward their ultimate destination) through intermediate nodes. Intermediate nodes are typically network hardware devices such as routers, bridges, gateways, firewalls, or switches. General-purpose computers can also forward packets and perform routing, though they are not specialized hardware and may suffer from limited performance. The routing process usually directs forwarding on the basis of routing tables which maintain a record of the routes to various network destinations. Thus, constructing routing tables, which are held in the router's memory, is very important for efficient routing. Most routing algorithms use only one network path at a time. Multipath routing techniques enable the use of multiple alternative paths.

In case of overlapping/equal routes, the following elements are considered in order to decide which routes get installed into the routing table (sorted by priority):

1. *Prefix-Length*: where longer subnet masks are preferred (independent of whether it is within a routing protocol or over different routing protocol)
2. *Metric*: where a lower metric/cost is preferred (only valid within one and the same routing protocol)
3. *Administrative distance*: where a lower distance is preferred (only valid between different routing protocols)

Routing, in a more narrow sense of the term, is often contrasted with bridging in its assumption that network addresses are structured and that similar addresses imply proximity within the network. Structured addresses allow a single routing table entry to represent the route to a group of devices. In large networks, structured addressing (routing, in the narrow sense) outperforms unstructured addressing (bridging). Routing has become the dominant form of addressing on the Internet. Bridging is still widely used within localized environments.

Website View of the project



The screenshot displays a web interface titled "Algorithm Project" in red text. Below the title, there are four configuration options, each with a label, a control element, and a value:

- Graph Size (Pixels):** A slider control with a white knob, set to the value **750**.
- Vertices:** A slider control with a white knob, set to the value **95**.
- Algorithm 1:** A dropdown menu showing "Dijkstras" with a red downward arrow.
- Algorithm 2:** A dropdown menu showing "Two Step Routing" with a red downward arrow.

At the bottom of the configuration section, there is a red button labeled **PROCESS GRAPH**.

Dijkstra's Algorithm

Let the node at which we are starting be called the initial node. Let the distance of node Y be the distance from the initial node to Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes unvisited. Set the initial node as current. Create a set of the unvisited nodes called the unvisited set consisting of all the nodes.
3. For the current node, consider all of its unvisited neighbors and calculate their tentative distances. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B (through A) will be $6 + 2 = 8$. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Select the unvisited node that is marked with the smallest tentative distance, and set it as the new "current node" then go back to step 3.

Code Snippet:

```
class Dijkstra {
    public static function addShortestPath(Graph $graph,
        Vertex $origin, Vertex $destination) {
        $visited = array();
        $distance = array();
        $previous = array();
        $vertices = $graph->getVertices();
        $n = count($vertices);
        foreach ($vertices as $vertex) {
            $distance[$vertex->key()] = INF;
            $previous[$vertex->key()] = NULL;
        }

        $distance[$origin->key()] = 0;

        while (count($vertices) > 0) {
            $min = INF;
            foreach ($vertices as $vertex) {
                $d = $distance[$vertex->key()];
                if ($d < $min) {
                    $min_vertex = $vertex;
                    $min = $d;
                }
            }
            unset($vertices[$min_vertex->key()]);
            $vertices = array_filter($vertices);
            if ($distance[$min_vertex->key()] == INF) {
                break;
            }
            foreach ($min_vertex->getNeighbors() as $neighbor) {
                $route = $distance[$min_vertex->key()]
                    + $min_vertex->distance($neighbor);
                if ($route < $distance[$neighbor->key()]) {
                    $distance[$neighbor->key()] = $route;
                }
            }
        }
    }
}
```

Reference:

Algorithm Course (10B11CI411)

http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Midpoint Routing Algorithm

```
1 calculate the coordinates of midpoint  $m$  of  $vt$ ;  
2 for each  $w \in N(v)$  {  
    // check whether  $t$  is a neighbor of  $v$   
3 if (  $w$  is the same node as  $t$  ) {  
4      $next(v)$  is set to  $w$ ;  
5     return;  
6 }  
7 update  $next(v)$  to  $w$  if  $w$  has a smaller  $d(w, m)$ ;  
8 }
```

Code Snippet:

```
class Midpoint {  
    public static function addPath(Graph $graph, Vertex $source, Vertex $destination) {  
        $current = $source;  
        while ( !($current->isEqual($destination)) ) {  
            $midpoint = Midpoint::getMidpoint($current, $destination);  
            $neighbors = $current->getNeighbors();  
            $min_distance = INF;  
            $next = $current;  
            foreach ( $neighbors as $neighbor ) {  
                if ($neighbor->isEqual($destination)) {  
                    $graph->addPathEdge( new Edge($current, $neighbor) );  
                    return;  
                }  
                $distance = $neighbor->distance($midpoint);  
                if ($distance < $min_distance) {  
                    $next = $neighbor;  
                    $min_distance = $distance;  
                }  
            }  
            $graph->addPathEdge( new Edge($current, $next) );  
            $current = $next;  
        }  
    }  
  
    private static function getMidpoint(Vertex $s, Vertex $t) {  
        $s_coords = $s->coords();  
        $t_coords = $t->coords();  
        $x = ( $s_coords["x"] + $t_coords["x"] ) / 2;  
        $y = ( $s_coords["y"] + $t_coords["y"] ) / 2;  
        return new Vertex($x, $y);  
    }  
}
```

Reference:

http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5470471

Two step Routing Algorithm

```
1 for each  $w$  in  $N(v)$  {  
2   if (  $w$  has a smaller  $d(w, t)$  than  $d(v, t)$  ) {  
3      $sum = d(v, w) + d(w, t)$ ;  
4     if (  $w$  has a smaller  $sum$  than previous ) {  
5        $next(v) = w$ ;  
6     }  
7   }  
8 }
```

Code Snippet:

```
class TwoStep{  
  
    public static function addPath(Graph $graph, Vertex $vertex_start, Vertex $vertex_finish) {  
        error_reporting(E_ALL);  
        ini_set('display_errors', '1');  
        $vertices = $graph->getVertices();  
        $gridsize = $graph->size;  
  
        $neighbors = $vertex_start->getNeighbors();  
        $start_distance = $vertex_start->distance($vertex_finish);  
        $min_distance = INF;  
        $min_vertex = null;  
        foreach ($neighbors as $neighbor) {  
            if ($neighbor->isEqual($vertex_finish)) {  
                $graph->addPathEdge( new Edge($vertex_start, $neighbor) );  
                return;  
            } else {  
                $distance = $neighbor->distance($vertex_finish);  
                if ($distance < $start_distance){  
                    $sum = $distance + $neighbor->distance($vertex_start);  
                    if ( $sum < $min_distance){  
                        $min_distance = $sum;  
                        $min_vertex = $neighbor;  
                    }  
                }  
            }  
        }  
        $graph->addPathEdge( new Edge($vertex_start, $min_vertex) );  
        TwoStep::addPath($graph, $min_vertex, $vertex_finish);  
    }  
}
```

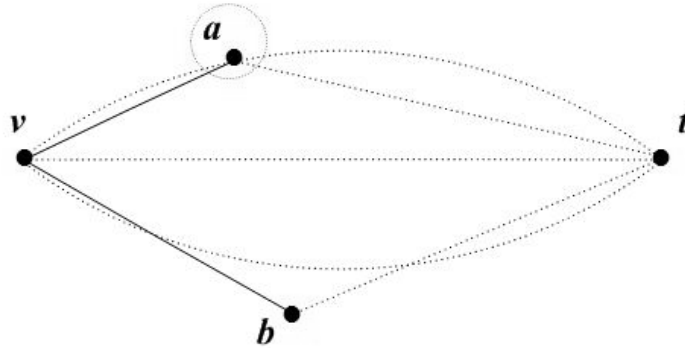
Reference:

<http://sydney.edu.au/engineering/it/~weisheng/papers/TwoStpRouting.pdf>

<http://staff.scem.uws.edu.au/~weisheng/talks/EllipseArcRouting.ppt>

Apex Angle Routing Algorithm

- We call $\angle vwt$ the apex angle for the current processing node v , a w in $N(v)$, and the destination t . The basic idea of Apex Angle Routing is to maximize this apex angle in each routing step.
- Apex Angle Routing essentially restricts its searching area of a w in $N(v)$ to the area enclosed by the two arcs with vt as the chord, i.e., it increases this area gradually until the first w in $N(v)$ is encountered.



Code Snippet:

```
class ApexAngle {
    public static function addPath(Graph $graph, Vertex $source, Vertex $destination) {
        $current = $source;
        while ( !($current->isEqual($destination)) ) {
            $st_angle = ApexAngle::getAngle($current, $destination);
            $min_difference = 2 * pi();
            foreach ($current->getNeighbors() as $neighbor) {
                $neighbor_angle = ApexAngle::getAngle($current, $neighbor);
                $difference = abs( $st_angle - $neighbor_angle );
                if ($difference > pi() ) {
                    $difference = abs( (2 * pi()) + $neighbor_angle);
                }
                if ( $difference < $min_difference ) {
                    $min_difference = $difference;
                    $next = $neighbor;
                }
            }
            $path = new Edge($current, $next);
            $graph->addPathEdge($path);
            $current = $next;
        }
    }

    public static function getAngle(Vertex $source, Vertex $dest) {
        $s_coords = $source->coords();
        $t_coords = $dest->coords();
```

Reference:

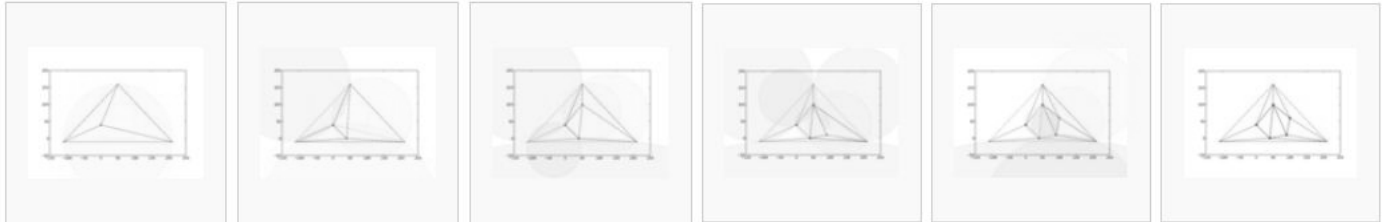
<http://sydney.edu.au/engineering/it/~weisheng/papers/TwoStpRouting.pdf>

<http://staff.scem.uws.edu.au/~weisheng/talks/EllipseArcRouting.ppt>

Bowyer-Watson Algorithm

In computational geometry, the Bowyer–Watson algorithm is a method for computing the Delaunay triangulation of a finite set of points in any number of dimensions. The algorithm can be used to obtain a Voronoi diagram of the points, which is the dual graph of the Delaunay triangulation.

The Bowyer–Watson algorithm is an incremental algorithm. It works by adding points, one at a time, to a valid Delaunay triangulation of a subset of the desired points. After every insertion, any triangles whose circumcircles contain the new point are deleted, leaving a star-shaped polygonal hole which is then re-triangulated using the new point.



First step: insert a node
in an enclosing
"super"-triangle

Insert second node

Insert third node

Insert fourth node

Insert fifth (and last)
node

Remove super-triangle
edges

Code Snippet:

```
class Deluanay {
    public static function triangulate(Graph $graph) {
        error_reporting(E_ALL);
        ini_set('display_errors', '1');
        $graph->resetEdges();
        $graph->resetTriangles();
        $graph->resetPath();
        $vertices = $graph->getVertices();
        $gridsize = $graph->size;
        $outlier_1 = new Vertex($gridsize/2, -($gridsize * 10000));
        $outlier_2 = new Vertex(-($gridsize * 10000), $gridsize * 10000);
        $outlier_3 = new Vertex($gridsize * 10000, $gridsize * 10000);
        $super = new Triangle($outlier_1, $outlier_2, $outlier_3);
        $triangle_buffer = array();
        $triangle_buffer[] = $super;
        foreach ($vertices as $vertex) {
            $edge_buffer = array();
            foreach ($triangle_buffer as $key => $triangle) {
                if (isset($triangle_buffer[$key]) && $triangle->isInCircle($vertex)) {
                    $t_edges = $triangle->getEdges();
                    foreach ($t_edges as $t_edge) {
                        $edge_buffer[] = $t_edge;
                    }
                    unset($triangle_buffer[$key]);
                }
            }
        }
    }
}
```

Reference:

[http://en.wikipedia.org/wiki/Bowyer–Watson_algorithm](http://en.wikipedia.org/wiki/Bowyer%E2%80%93Watson_algorithm)

http://iss.ices.utexas.edu/?p=projects/galois/benchmarks/delaunay_triangulation

Voronoi Algorithm

Voronoi diagram is a way of dividing space into a number of regions. A set of points (called seeds, sites, or generators) is specified beforehand and for each seed there will be a corresponding region consisting of all points closer to that seed than to any other. The regions are called Voronoi cells. It is dual to the Delaunay triangulation.

It is named after Georgy Voronoi, and is also called a Voronoi tessellation, Voronoi decomposition, a Voronoi partition, or a Dirichlet tessellation (after Peter Gustav Lejeune Dirichlet). Voronoi diagrams can be found in a large number of fields in science and technology, even in art, and they have found numerous practical and theoretical applications.

(The result obtained from Bowyer-Watson algorithm described above is passed into Voronoi code, as it required input in this type of form.)

Code Snippet:

```
class Voronoi {
    //Delaunay triangulation as input
    public static function addCells(Graph $graph) {
        $triangles = $graph->getTriangles();
        foreach ($triangles as $triangle) {
            foreach ($triangles as $neighbor) {
                if ($triangle->sharesEdge($neighbor) ) {
                    $cell_edge = new Edge($triangle->c_circumcenter, $neighbor->c_circumcenter);
                    $graph->addPathEdge($cell_edge);
                }
            }
        }
    }
}
```

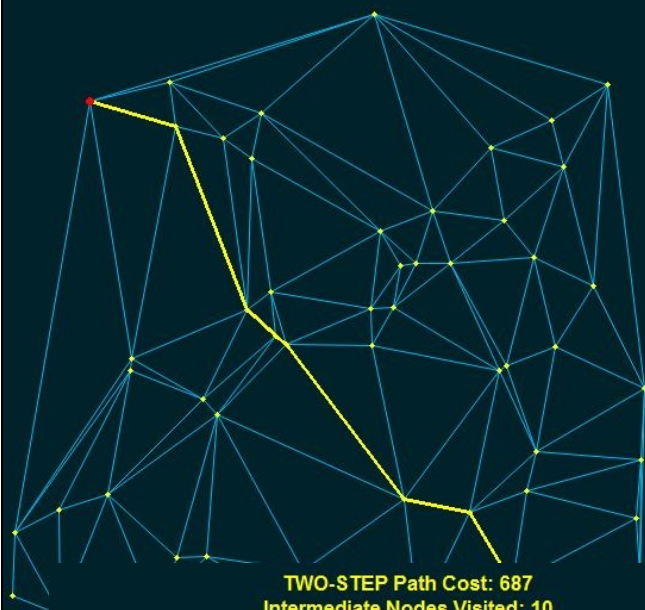
Reference:

http://en.wikipedia.org/wiki/Voronoi_diagram

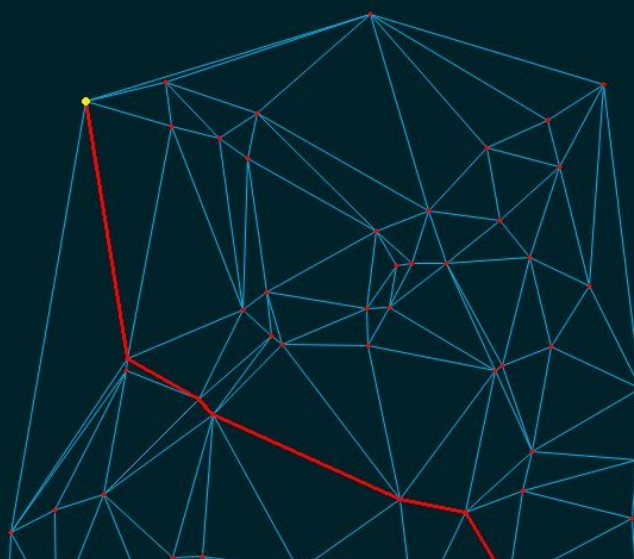
<http://datavoreconsulting.com/programming-tips/voronoi-diagrams-in-mathematica/>

<http://www.cs.cornell.edu/Info/People/chew/Delaunay.html>

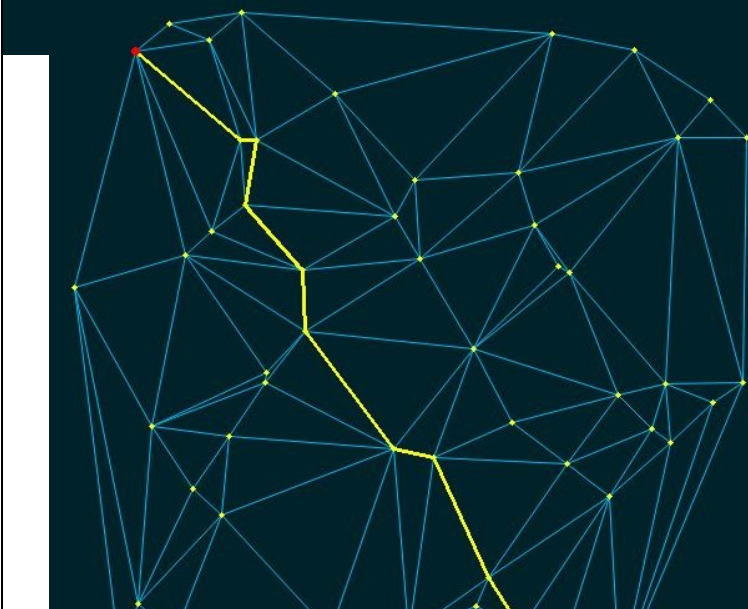
DIJKSTRAS Path Cost: 649
Intermediate Nodes Visited: 7



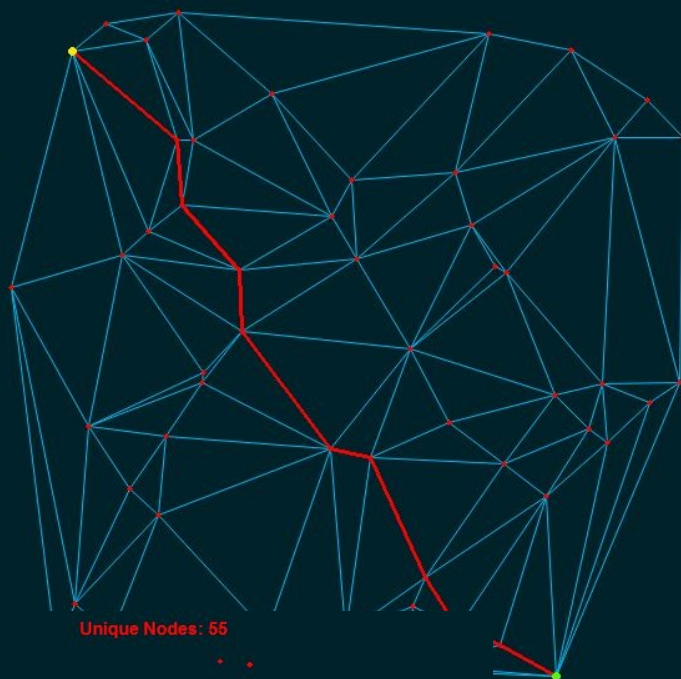
MIDPOINT Path Cost: 680
Intermediate Nodes Visited: 6



TWO-STEP Path Cost: 687
Intermediate Nodes Visited: 10



APEX-ANGLE Path Cost: 673
Intermediate Nodes Visited: 9



Unique Nodes: 55

