International Journal of Advance Research in Science and Engineering
Vol. No.6, Issue No. 05, May 2017
www.ijarse.com
IJARSE
ISSN (O) 2319 - 8354
ISSN (P) 2319 - 8346

PREVENTION AGAINST HACKING USING DIJKSTRA ALGORITHM

Mr. Vijay Kumar Srivastava[1], Md. Umar[2], Mr. Kapil Mangla[3]

[1,2,3]Department of ECE, Satya College of Engg. & Tech., Palwal, (INDIA)

ABSTRACT

These days, accessing information and exchanging of data in business industry is increasing but it also Increases the risk of security. The state of the security on internet is bad and becomes worse. The explosive growth of internet has brought many good things, but there is also a dark side: Criminal hacker. The initial design for common communication protocols indicates that the technology was proposed to meet main requirements such as speed, performance, efficiency and reliability but security was not a concern at that stage. Hacking is the practice of modifying the features of system, in order to accomplish a goal outside of the creator's original purpose. Number of solutions is provided against hacking but they are unable to address those issues. This thesis provides security for entire infrastructure to protect against hacking. It generates the Dijkstra Algorithm and creates the confusion in front of hacker. Hacker cannot understand the current communication infrastructure and it is difficult for him to break the system easily.

Keywords*: *Dijkstra Algorithm, Hacking

I. INTRODUCTION

The world of internet is growing at an enormous pace and so is the concern of the security of the data over the internet. Since there isn't any restriction on the users who can access the internet, the vulnerability of the data over the internet is high. People can access someone else's data and manipulate it for their own good. Hacking is descriptive term used to describe the attitude and behavior of group of people who are greatly involved in technical activity which results in gaining unauthorized access. Hacking is a technique of maliciously attacking someone else's computer/network with an intention to steal or manipulate the data.

This proposed security approach is designed to eliminate the possibility of hacking by using trusted Graphs. These graphs are generated dynamically and would determine the amount of trust factor associated with each node. This paper aim to design a dynamic security approach that is mainly directed to defend hacking

II.DIJKSTRA'S ALGORITHM

This algorithm finds the shortest path from a source vertex to all other vertices in a weighted directed graph without negative edge weights.

Here is the algorithm for a graph G with vertices V = $\{v_1, \ldots v_n\}$ and edge weights w_{ij} for an edge connecting vertex v_i with vertex v_j. Let the source be v_1.

IJARSE
ISSN (O) 2319 - 8354
ISSN (P) 2319 - 8346

Initialize a set S = ∅. This set will keep track of all vertices that we have already computed the shortest distance to from the source.

Initialize an array D of estimates of shortest distances. D[1] = 0, while D[i] = ∞, for all other i. (This says that our estimate from v_1 to v_1 is 0, and all of our other estimates from v_1 are infinity.)

While S != V do the following:

1) Find the vertex (not is S) that corresponds to the minimal estimate of shortest distances in array D.

2) Add this vertex, v_i into S.

3) Recomputed all estimates based on edges emanating from v. In particular, for each edge from v, compute $D[i]+w_{ij}$. If this quantity is less than D[j], then set

$D[j] = D[i]+w_{ij}$.

Essentially, what the algorithm is doing is this:

Imagine that you want to figure out the shortest route from the source to all other vertices. Since there are no negative edge weights, we know that the shortest edge from the source to another vertex must be a shortest path. (Any other path to the same vertex must go through another, but that edge would be more costly than the original edge based on how it was chosen.)

Now, for each iteration, we try to see if going through that new vertex can improve our distance estimates. We know that all shortest paths contain sub paths that are also shortest paths. (Try to convince yourself of this.) Thus, if a path is to be a shortest path, it must build off another shortest path. That's essentially what we are doing through each iteration, is building another shortest path. When we add in a vertex, we know the cost of the path from the source to that vertex. Adding that to an edge from that vertex to another, we get a new estimate for the weight of a path from the source to the new vertex.

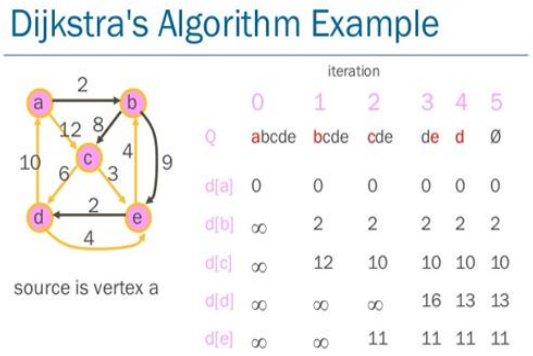
This algorithm is greedy because we assume we have a shortest distance to a vertex before we ever examine all the edges that even lead into that vertex. In general, this works because we assume no negative edge weights. The formal proof is a bit drawn out, but the intuition behind it is as follows: If the shortest edge from the source to any vertex is weight w, then any other path to that vertex must go somewhere else, incurring a cost greater than w. But, from that point, there's no way to get a path from that point with a smaller cost, because any edges added to the path must be non-negative.

By the end, we will have determined all the shortest paths, since we have added a new vertex into our set for each iteration.

This algorithm is easiest to follow in a tabular format.

The adjacency matrix of an example graph is included below.

International Journal of Advance Research in Science and Engineering
Vol. No.6, Issue No. 05, May 2017
www.ijarse.com
IJARSE
ISSN (O) 2319 - 8354
ISSN (P) 2319 - 8346



Let a be the source vertex.

	a	b	c	d	e
a	0	10	inf	inf	3
b	inf	0	8	2	inf
c	2	3	0	4	inf
d	5	inf	4	0	inf
e	inf	12	16	13	0

Here is the algorithm:

Estimates	b	c	d	e
Add to Set				
a	10	inf	inf	3
e	10	19	16	3
b	10	18	12	3
d	10	16	12	3

We changed the estimates to c and d to 19 and 16 respectively since these were improvements on prior estimates, using the edges from e to c and e to d. But, we did NOT change the 10 because 3+12, (the edge length from e to b) gives us a path length of 15, which is more than the current estimate of 10. Using edges bc and bd, we improve the estimates to both c and d again. Finally using edge dc we improve the estimate to c.

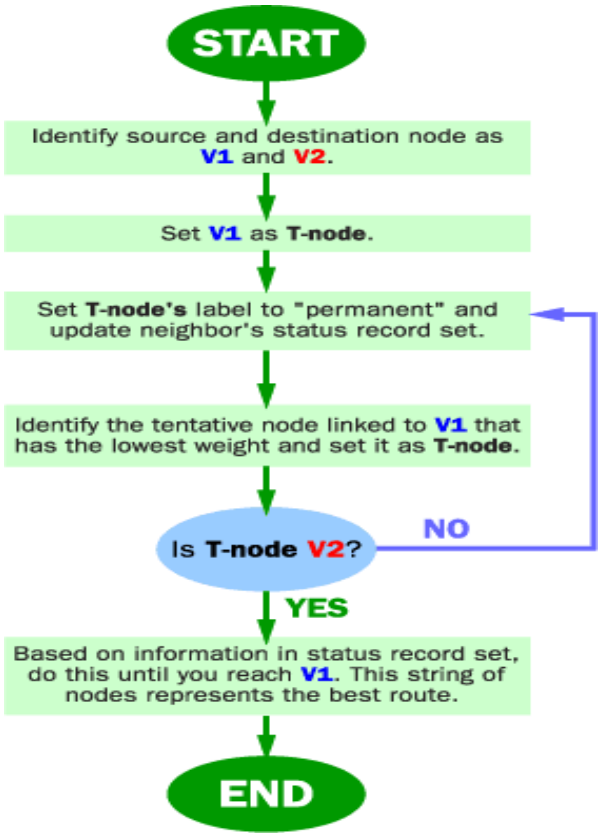
Now, we will prove why the algorithm works. We will use proof by contradiction. After each iteration of the algorithm, we "declare" that we have found one shortest path. We will assume that one of these that we have found is NOT a shortest path.

IJARSE
ISSN (O) 2319 - 8354
ISSN (P) 2319 - 8346

Let t be the first vertex that gets incorrectly placed in the set S. This means that there is a shorter path to t than the estimate produced when t is added into S. Since we have considered all edges from the set S into vertex t, it follows that if a shorter path exists, its last edge must emanate from a vertex outside of S to t. But, all the estimates to the edges outside of S are greater than the estimate to t. None of these will be improved by any edge emanating from a vertex in S (except t), since these have already been tried. Thus, it's impossible for ANY of these estimates to ever become better than the estimate to t, since there are no negative edge weights. With that in mind, since each edge leading to t is non-negative, going through any vertex not in S to t would not decrease the estimate of its distance. Thus, we have contradicted the fact that a shorter path to t could be found. Thus, when the algorithm terminates with all vertices in the set S, all estimates are correct. Try an example yourself on the graph with the following adjacency matrix using a as the source.

	a	b	c	d	e
a	0	3	4	inf	inf
b	inf	0	inf	6	8
c	inf	2	0	1	5
d	inf	inf	inf	0	2
e	inf	inf	inf	inf	0

III.Dijkstra Flow Chart



IJARSE
ISSN (O) 2319 - 8354
ISSN (P) 2319 - 8346

3.1 Graphs

This is mostly concerned with directed graphs. We adopt the point of view that every graph is directed, but sometimes the direction is not specified or may be ignored. Throughout, we use the terms *arc* and *edge* for the directed and undirected case, respectively. In this text we shall use both the natural intuitive geometric view of embedded graphs and the formal algebraic formulation.

3.2 Vertex- and Edge-Oriented Definitions of Graphs

There are two almost equivalent definitions of graphs. The common one is the vertex-oriented approach, in which a graph is a pair G = (V, A), where V is the vertex (or node1) set, and A, the arc set, is a subset of V x V. For nodes u, v ∈ V. we use the notation uv to denote the element (u, v) of V x V , and say that the arc uv is oriented from u to v, The node u is called the tail of uv, written u = tail (uv). Similarly, v = head (uv) is the head of uv. We use tail (uv) whenever the graph G in question is not clear from the context. We use V (G) to denote the set of nodes of a graph G, and use |V (G)|, or sometimes just |G| to denote the number of nodes of G.

3.3 Paths, Cycles, Trees and Cuts

An x-to-y *walk* is a sequence of darts d_1…….. d_k such that tail(d_1) = x, head(d_k) = y and, for i =2…… k, head(d_{i-1}) = tail(di). A walk in which no dart appears more than once is called a *path*. An empty sequence represents the trivial path consisting of a single vertex. We use start (P) to denote the first vertex, x, of P and end (P) to denote the last vertex, y, of P. If additionally head(d_k) =tail(d_1) then the walk is a *cycle*. A walk is *simple* if no vertex occurs twice as the head of a dart in the walk.

.

3.4 Vector Spaces and Circulations

Let G = (V, A) be a directed graph. The *dart space* of G is $R^{Ax(\pm 1)}$, the set of vectors α that assign a real number α[d] to each dart d.

The *arc space* of G is the subspace of G's dart space satisfying ant symmetry with respect to rev. Namely, for every dart d, α[d] = α[rev(d)]. We call the vectors of the dart and arc spaces *dart vectors* and *arc vectors*, respectively.

3.5 Clockwise, Left and Right

A circulation ρ is *clockwise* if Φ[f] >0 for all faces f. A circulation ρ is *counterclockwise* if Φ[f] <0 for all faces f. Note that a circulation may be neither clockwise nor counterclockwise.

3.6 The Planar Dual

The *dual* of an embedded graph G = (Π, A) is the embedded graph G* =(Π*, A*). G is called the primal graph. Since (Π*)*= rev o rev o Π is the identity, which shows that the dual of the dual is the primal. If G is a planar connected graph, then so is G* By definition, the faces of G are the nodes of G*, and vice-versa. Note that according to our definition the set of darts of G and of G*is identical. Whenever it is not clear from the context we will explicitly specify whether we refer to a dart d in the primal graph G or in its dual G*.

IJARSE
ISSN (O) 2319 - 8354
ISSN (P) 2319 - 8346

3.7 Deletion and Contraction

In our algorithms we will use deletions of edges and contractions of (non-self-loop) edges. Intuitively, the result of contracting an edge uv in G is the graph G' in which the nodes u and v are identified with a single node. Formally, deleting an edge e is defined as deleting both darts of e.

3.8 Separators in Planar Graphs

Separators are the basis for most divide-and-conquer algorithms in planar graphs. In this method the graph is divided into two or more sub graphs. The problem is first solved recursively in each sub graph. Then the solutions are combined into a solution for the entire graph. The best known separator theorem for planar graphs was given by Lipton and Trajan. They proved that for any planar graph with n nodes, the vertex set can be partitioned into three sets A, B, C such that no edge connects A and B, neither A nor B contains more than 2n=3 nodes, and C contains at most 2p

2n nodes. The nodes of C are referred to in the literature as *separator*, *border* or *boundary* nodes.

The result can be stated in terms of weights, rather than number of nodes.

.

3.9 Lengths and Shortest Paths

Graphs can be augmented with different kinds of attributes. One of the most common attributes is lengths (sometimes called weights). A length assignment is a function length from the set of darts to the real numbers. Very often we talk of arc lengths, in which case the length of the dart a+ is length(a) and the length of a is infinite. In general, length assignments may be negative, although in various circumstances only non-negative lengths are considered. When no confusion arises we may refer to lengths of the corresponding arcs instead of the darts. We extend the notation to sets of darts (and to paths in particular). For a set D of darts, the length (or weight) of D is length (D) =P

3.10 Dijkstra's Algorithm

Dijkstra's algorithm is the common name for an algorithm for computing single source shortest paths in a directed graph with non-negative arc lengths. It is named after E. J. Dijkstra, although several very similar algorithms were independently discovered and published in the second half of the 1950's by Leyzorek, Gray, Johnson, Ladew, Meaker, Petry, and Seitz and by Dantzig . See Schrijver for a historical survey.

The procedure UPDATEHEAP (Q, s, d) decreases the key of element s in the heap Q to be d (or inserts the element if it is not already in the heap). The procedure EXTRACTMIN (Q) returns the

Algorithm 1 Dijkstra (G, s)

1: for all v € V (G): d(v) ←1

2: d(s) ←0

3: initialize an empty heap Q

4: UPDATEHEAP (Q, s, d(s))

5: S ←Φ

IJARSE
ISSN (O) 2319 - 8354
ISSN (P) 2319 - 8346

6: **while** S # V (G)

7: v ←EXTRACTMIN (Q)

8: ACTIVATE (Q,G, v, d)

9: S ←S U{v}

Element with minimum key in the heap Q.

For an arc vu, *relaxing* vu is an attempt to decrease the distance label of u by considering a path that gets to u through v using vu . The procedure ACTIVATE relaxes all the arcs whose tail is v.

Algorithm 2 Activate (Q,G,v,d)

Relaxes all the arcs in G whose tail is v in the heap Q using the labels d(.)

1: **for** each arc vu €E(G)

2: d(u) ←min{d(u), d(v) + l(vu)}

3: UPDATEHEAP (Q, u, d(u))

In Dijkstra's algorithm each node v is extracted from the heap exactly once. At that time the arcs emanating from v are relaxed. It follows that each arc is relaxed exactly once. Dijkstra's algorithm runs in O ((n + m) log n) time if elementary data structures such as a binary heap are used, and in O(n log n + m) time when implemented with Fibonacci heaps . It is easy to augment the algorithm to produce a shortest-path tree within the same running time.

3.11 Price Functions and Reduced Lengths

A price function is called *feasible* if it induces nonnegative reduced lengths on *all* darts of G.

Idea appeared much earlier, for example in Ford and Fulkerson's algorithm for minimum-cost flow.

Let G be a directed graph with arc lengths (⌊).

A *price function* is a function Φ from the nodes of G to the reals. For a dart (or arc) d, the *reduced length with respect to* Φ is

$$_{\Phi}(d) = ⌊ (d) + \Phi(tail(d)) - \Phi(head(d)).$$

For any nodes s and t, for any s-to-t path P, the reduced length ⌊ $_{\Phi}$ (P) of P is

$$_{\Phi}(P) = \Sigma ⌊ (d) + \Phi(tail(d)) - \Phi(head(d)).$$
$$= \Phi(Start(P) - \Phi(end(P) + \Sigma⌊ (d)$$
$$= ⌊ (P) + \Phi(P) - \Phi(t).$$

3.12 Circulations and Shortest Paths

In this section we discuss a relation between circulations in a directed planar graph G with arc capacities c, and shortest paths in the dual graph G*, where the length of a dart d in G* is taken to be its capacity in G

Namely,lengthG* (d) = c[d]

IJARSE
ISSN (O) 2319 - 8354
ISSN (P) 2319 - 8346

REFERENCES

Journal Paper:

[1] IJARCCE Prevention against Hacking using Trusted Graph by Yash Sanzgiri.

[2] P. Elias, A. Feinstein, and C. Shannon. A note on the maximum flow through a network. *IEEE Transactions on Information Theory*, 2(4):117–119, 1956.

Books:

[1] Hacking and securing iOS Applications by Jonathan Zdziarski.

[2] Hacking Web Apps by Mike Shema.

[3] Efficient Algorithms for Shortest-Path and Maximum-Flow Problems in Planar Graphs by Shay Mozes.

Theses:

[1] Using Dijkstra Algorithm in calculating alternative shortest paths for public Transportation shortest paths for public transportion with transfers and walking by Haitham Latif Hassan AL-Tameemi.

[2] Fourth International World wide Confer-ence proceedings (1995), no.1, pp.461-471.PHAM, V.A. and Karmouch**.**

[3] G. Borradaile. Exploiting Planarity for Network Flow and Connectivity Plobrems. Brown University, 2008.