Dijkstra algorithm, getting the shortest path in a graph

Luis Ricardo Montes Gomez
ID: 153788
luis.montesgz@udlap.mx

Octavio Palomeque Gasperin
ID: 151884
octavio.palomequegn@udlap.mx

February 23, 2019

1 Abstract

Dijkstra algorithm is useful when itâs necessary reach from one state to another with the lowest cost, there are many implementations of it. We had developed one that search the shortest path from a source node to any other. Then we get the times for 6,8,15,16,26,32,64,128,256 nodes and we plot them using Octave.

2 Introduction

In this document we are going to talk about Dijkstra Algorithm to try to parallelize it in order to topics seen in class. This algorithm is an implementation to find the shortest path between two nodes in a graph, this path is based in decisions taken with the information of each vertex weight that connects each node in the graph, in other words, the problem can be explain as follows:

Dijkstra algorithm is an implementation to find the shortest path between two nodes in a graph, it was proposed by Edsger W. Dijkstra in 1956. The original algorithm just find the shortest path between a node A and node B, but there are many variations of it, the most popular is one that finds all the shortest paths between a source node and any other node in the graph, producing a

shortest-path tree.

We need a directed graph $G = V, E$ with weights and V having the set of vertices. The special vertex s in V, where s is the source and let for any edge e in E, EdgeCost(e) be the length of edge e. All the weights in the graph should be non-negative.

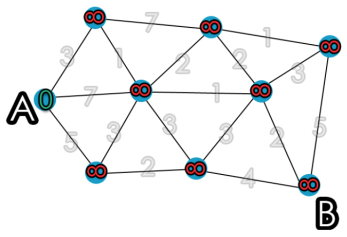

Figure 2: First Iteration.

In the Figure 3 we can see how the algorithm set the cost of travel from A to their neighbors.This is the first iteration , thatâs the reason that the intersections visited had the value of cost without any other value added.

3 Problem Description

The first step is set all intersections with âinfiniteâ just to note that this intersection has not been visited yet, then, at the first iteration, we look for the closest intersection and relabel the intersection with the weight of travel through that edge, and so on with the other intersections, then we move to the closest intersection and look for the closest intersection, if it isnât âinfiniteâ we check if the current intersection weight plus the cost of travel to that intersection is less than the current cost, if that is true we relabel the intersection with the sum and continue with the other intersections. With Figure 2 we illustrate how the algorithm works.

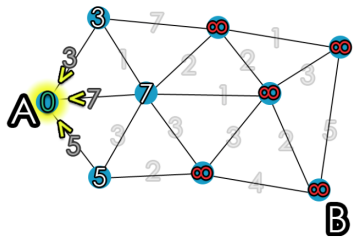

Figure 3: Second Iteration.

As can be seen, in Image 4 the algorithm move to the intersection with the lowest cost. After that itâs necessary to visit the neighbors intersections and label it with the actual intersection value plus the cost from travel to it. If we performed the previous operation we get one intersection with 4 and other with 10.

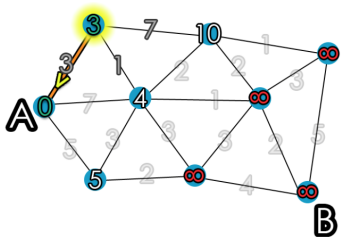

Figure 4: Fourth Iteration

The Image 5 shows the final result, after visit all the intersections and do the correct comparisons to determine if the path that we are following is really the shortest or the one with the lowest cost. Thatâs the reason because after get one result following the lowest costs, is necessary to return and verify what happen if we follow another path. Like if we go from âAâ to the intersection with a cost of five, then if we performed the sum of costs it return a greater cost from one intersection and the same value from the other one. The latest means that if we travel from that path the cost to achieve to the goal will be more expensive.

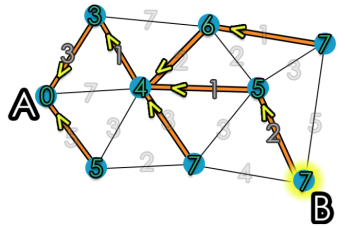

Figure 5: End of algorithm

4 Complexity of the algorithm

To calculate the Dijkstra algorithm complexity can be count based on the number of actions required to find a certain solution based in some conditions. The algorithm consist in $n-1$ iterations as maximum. In each iteration you add a vertex in to the set. In each iteration, the vertex with the lowest tag among those that are not in S_k must be identified, the number of this operations is bounded by $n-1$. Finally you have to perform a sum and a comparison to update the tag of each one of the vertex that are not in Sk_2. After each iteration are done at most $2(n-1)$ operations. The Dijkstra algorithm makes $O(n^2)$ operations to determine the longitude of the shortest path between two nodes of a simple graph. The complexity of the algorithm then could be defined by the next formula in case that we are not using a priority queue $O(|V|^2 + |A|)$

5 Procedural implementation of Dijkstra's algorithm

5.1 Coding

The first function is basically the Dijkstra solution and would be our core function of the program that follows the logic previously mentioned in the explanation of the algorithm.

```
int costs[MAX][MAX];
matrixFill(MAX, costs);
int resulting[MAX];

//Print Matrix
/*for (int i = 0; i < MAX; i++){
        for (int j = 0; j < MAX; j++){
                printf("%d ",costs[i][j]);
        }
        printf("\n" );
}*/
bool visited[MAX];
        clock_t begin_clock = clock();
        for (int i = 0; i < MAX; i++){
        resulting[i] = INT_MAX;
        visited[i] = false;
}
//set the resulting[0][0] node as 0
resulting[0] = 0;
for (int i = 0; i < MAX-1; i++){
        int min = minDistance(resulting, visited);
        visited[min] = true;
        for (int j = 0; j < MAX; j++){
                if(!visited[j] && costs[min][j] && resulting[min] != INT_MAX
resulting[min] + costs[min][j] < resulting[j]){
                        resulting[j] = resulting[min] + costs[min][j];
                }
        }
}
```

Additionally to this we have also the function that calculate the minimum distance, this in addition just to show how it works.

```
int min = INT_MAX, min_index;
for (int v = 0; v < MAX; v++)
        if (visited[v] == false && resulting[v] <= min)
                min = resulting[v], min_index = v;
        return min_index;
```

5.2 Results of the implementation

As can be seen in Figure 7 the time of execution increase while the number of nodes increase, this incrementation is linear so as more nodes you add into your problem to solve more time consumption will require.

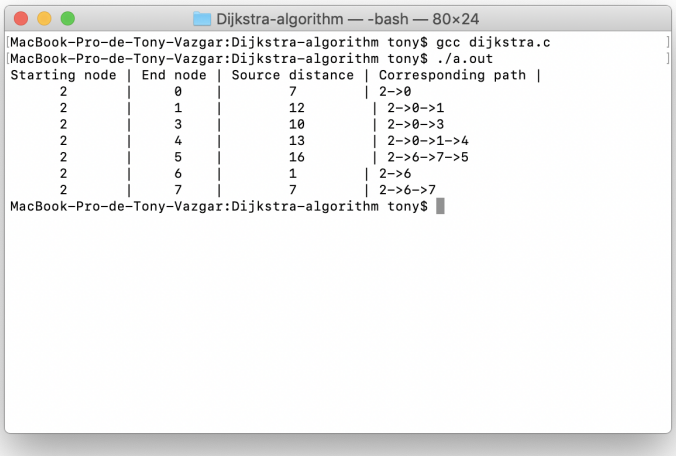

Figure 6: Results after executing the code

6 Parallel Implementation

6.1 Algorithm Description

As we saw in class for the parallel implementation of the Dijkstra problem we used the library called OpenMP inherent to gcc, using a technique that is included in this library called pragmas that give the instruction to the

compiler to send several tasks in the implementation to separated processors in the computer giving the opportunity to reduce the time consumption in relation to how many instructions are possible to be parallelized, this is what we call scalability.

In addition to this, is important to mention that our implementation is parallelized in an automatic way, which adds the parallelization in every sector that is possible, the most common one is in the cycles of the code.

Using 2 simple instructions into the code, when the compiler compiles the code adding the instruction to use openMP it will start using several processors to do the task.

```
#include <omp.h>

#pragma omp parallel num_threads(4)
#pragma omp for
```

6.2 Results of the parallel implementation

The parallel implementation shows that just with some simple implementation of pragmas the efficiency of the algorithm increase considerably good. The reduction is considerable from the first processors to the second one, therefore as you can see in the figure 9, even if the addition of a second processor into the analysis creates a reduction of time consumption is not the same with a third or fourth processor because there's no more threads to paralleliced. In other words, the scalability of the problem got to the top of the possibilities, in other circumstances or other kind of problems where more parallelization possibilities can be applies this could increase with a third or more processors.

7 Comparison between parallel and procedural implementation

As we can see in the next figures, the execution time of the algorithm grows as you add more nodes into the execution, starting with 50 and finishing with 500, the execution time is almost linear, therefore, once we add the pragmas into the implementation as we can see in the figure 9 the time

execution is reduced making it incredible faster as it was with the procedural implementation.

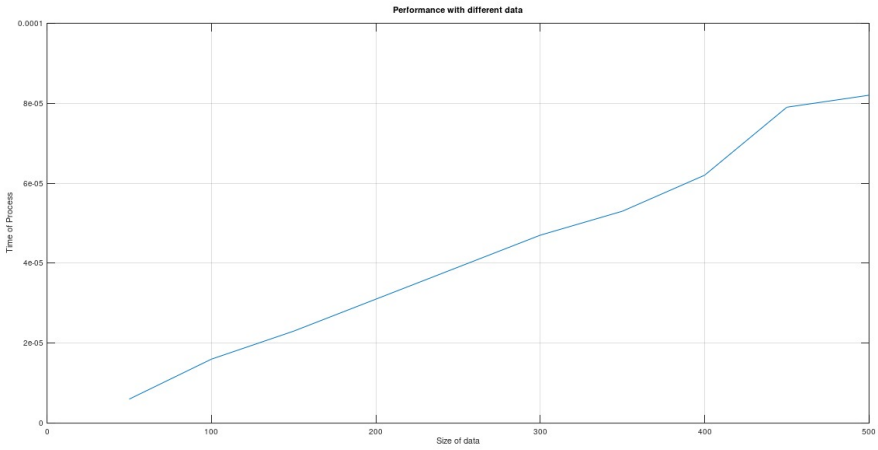

Figure 7: Chart that shows the execution time based in the amount of nodes in the graph.

The addition of the pragmas into our program shows us a big difference between the procedural and the parallel implementation, and this method used was quite simple to incorporate into our implementation. So the relation between the benefit that we got and the human-time required for the addition is incredible effective.

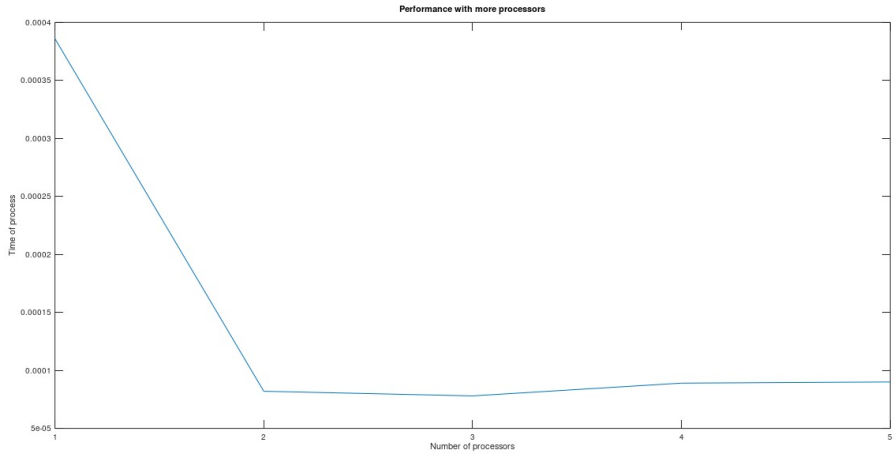

Figure 8: Chart showing the reduction in time execution after the parallelization of the algorithm.

8 Conclusions

After test with different data the Dijkstra Algorithm we can say that it's a good way to search the minimum path between two nodes, because the execution time it's not to high. For the main porpoise of this implementation, the algorithm is a good one to make it parallel, because it have many for loops and the search trough the paths could be faster if one processor search in different paths.

After we had the opportunity to do the implementation with parallel programming we obtained the results that we expected, after the implementation we saw an improvement in the time execution as is showed in the figure 9 that we saw previously.

Finally, we would like to add that as we saw in class, most of the applications don't take into consideration the use of multiple compute power from even as something as it could be a laptop, the lack of this implementation in several programs and applications create a waste of compute power making that most of the time we are just using one processors when is simple to add an automatic parallelization.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2009). *I*ntroduction to algorithms. MIT press.

[2] Brandes, U. (2001). A faster algorithm for betweenness centrality. *J*ournal of mathematical sociology, 25(2), 163-177.

[3] Johnson, D. B. (1973). A note on Dijkstra's shortest path algorithm. *J*ournal of the ACM (JACM), 20(3), 385-388.

[4] Goodrich, M. T., Tamassia, R. (2002). *A*lgorithm design. Wiely India.