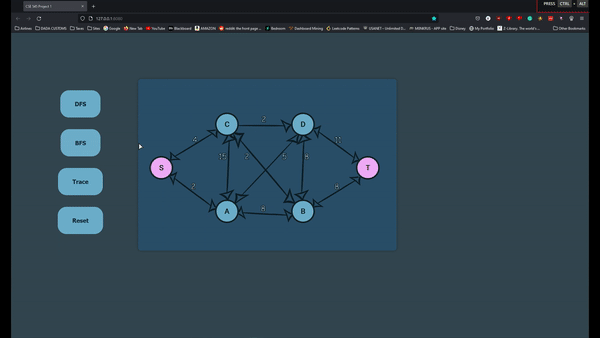
CSE 545

Daniel Maya Bonilla

Project 1

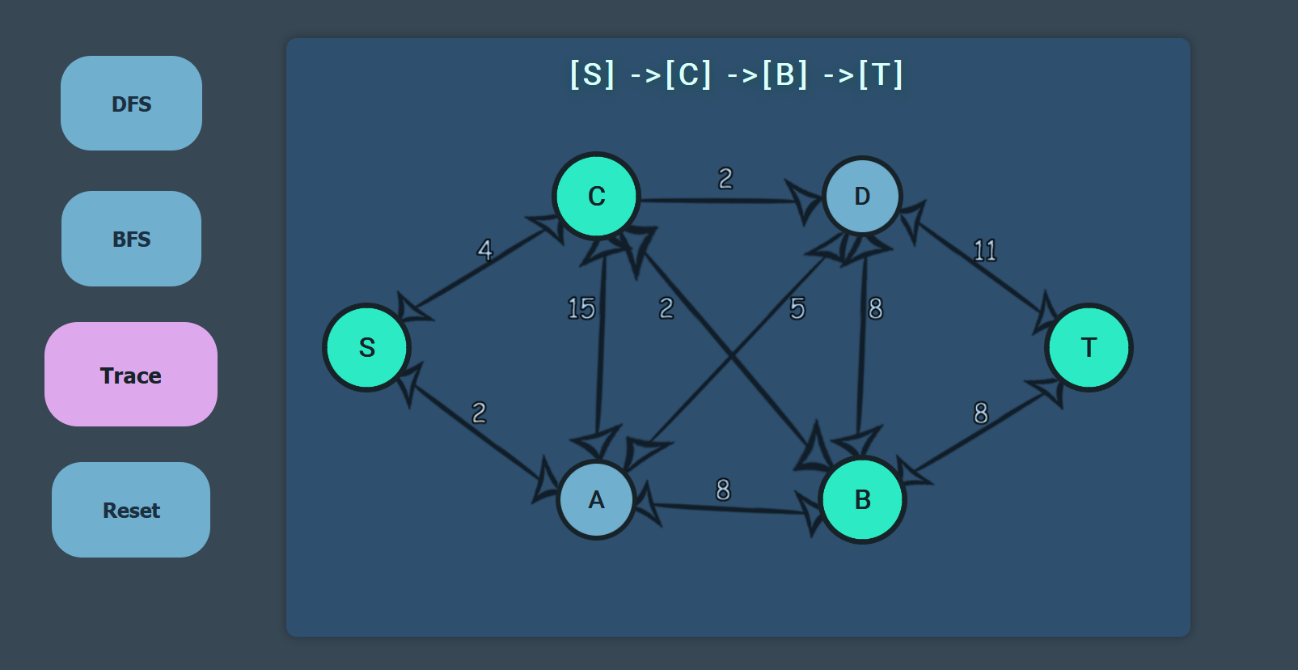
 Traversing Nodes Visualization Algorithms

**Part A:**

At first glance, when drawing a path, you might seem like the best approach is checking the first node and going with the lowest values you find. The problem is that there’s always the chance that you might’ve traversed a node where you left potentially a lower value.

Dijkstra’s’ algorithm solves this situation with a very interesting approach. However, in this case we are purely solving it by hand. From a top-down perspective, the intuitive way to solve it, is to draw a path from S to T, traversing the possible nodes we find, we will keep track of the sum of all the costs for the edges.

At the end we will end up with the best possible scenario of **[S -> C -> B -> T]**. Other than finding the best solution, we cannot guarantee our approach is the fastest but for that reason we will show other 2 important algorithms.

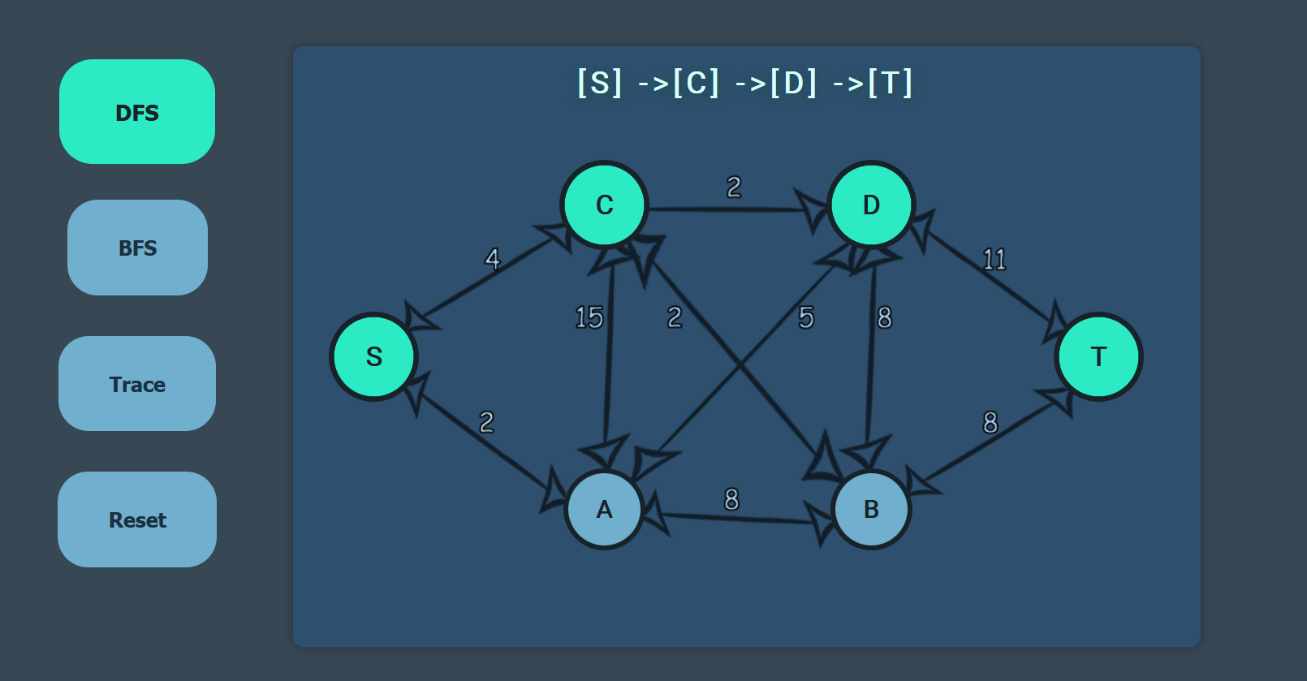


*Shortest Path – Computed by Hand and Represented Visually:*

**Part B:**

The next algorithms will be computed using JavaScript, the programming language doesn’t really provide much other than a comfortable environment for the programmer. In this case I used JavaScript so I could letter implement a better representation for how the algorithms are shown when traversing the graph. With a combination of CSS/Html we can create a simple way to represent our graph and compute it visually.

Let’s start with DFS, Depth First Search, as the name indicates this algorithm was about to show the expected. It will traverse the nodes it finds one at a time, until it reaches the deepest level (k) in a graph/tree or the result is found. If the result isn’t found and we reach the deepest level, we backtrack until traversal can continue to happen and explore new nodes. The implementation is straightforward, and this is the path it took to reach the end.



*DFS – Computed with JavaScript*

Before talking about the observations and time it took to complete the traverse, let’s quickly show the next algorithm. BFS, Breadth First Search, performs similarly to DFS, however, this time we are traversing every possible node at a time, then we add to a queue, and eliminate the duplicates. In a sense, we are searching for every level of the graph/tree first, then we proceed to explore the next one. We repeat this step until we reach the goal, or there’s no more levels to traverse. My first thought was that it would take longer to reach the destination node, however, I was surprised to find that it’s quite efficient at traversing. The first scenario I thought was having a much bigger and complex graph, and in instance BFS will traverse faster since it can cover much wider range. However, since the provided graph is not big enough, the results can’t be properly tested.

A picture containing diagram

Description automatically generated The number of nodes traversed is more than DFS, as we had to explore each one of them before reaching T. Therefore, the nodes traversed were: [ S -> A -> C -> B -> D -> T], there is some variance since the actual order can change depending on the runtime speed.

*BFS – Computed with JavaScript*

After running both algorithms over 100 instances, and taking the average of the 10 best runs, the conclusion shows a wide difference with margins of errors. Errors in this case are considered in the actual hardware these are run. Since the dataset for the graph is not big enough to determine a good runtime of both algorithms, they range from 0.01ms – 0.12ms. Depending on which one runs first, it might take more given the program has to load the dependencies. With a bigger data set these algorithms provide much better results and they are extremely efficient doing so.