Lab 2

Sunday, 12 September 2021 8:49 pm

```
uint64_t dijkstaA(int **graph, int Ve, int source)
{
    uint64_t start = timeNow();

    // d keeps track of shortest distance of vertex from source
    // pi keeps track of parent of vertex
    int d[Ve], pi[Ve];

    // visited keeps track of whether vertex has been completely visited
    bool visited[Ve];

    // our priority queue for getting next shortest distance
    PriorityQueueArray prioQueue;

    for (int i = 0; i < Ve; i++)
    {
        // distance of source from source is 0
        if (i == source)
            d[i] = 0;
        else
            d[i] = INT32_MAX;

        pi[i] = -1;
        visited[i] = false;

        prioQueue.add(i, d[i]);
    }

    int debugA = 0;

    // continue until priority queue is empty, i.e. no more vertices to explore
    while (!prioQueue.isEmpty())
    {
        int *pair = prioQueue.pop();
        int u = pair[0], dist = pair[1];

        visited[u] = true;

        // loops through distance for all vertices from u by looking at the adj matrix
        // super inefficient for an adj matrix but it's how we keep track of edges
        for (int i = 0; i < Ve; i++)
        {
            int vertex = i;
            int distanceFromU = graph[u][i];

            // if vertex is connected, hasn't been visited and the (distance of u from source) + (distance of vertex from u)
            // is less than current distance of vertex from source
            if (distanceFromU != 0 && !visited[i] && (d[u] + distanceFromU) < d[i])
            {
                debugA++;
                // gets the shorter distance and updates the priority queue, distance array and parent array
                int shorterDistance = d[u] + distanceFromU;
                d[i] = shorterDistance;
                pi[i] = u;
                prioQueue.edit(i, shorterDistance);
            }
        }
    }

    uint64_t timeTaken = timeNow() - start;
    return timeTaken;
}
```

Let $V=|V|$, $E=|E|$

Looks neater

O(V)

Loop V times

O(v)

At most 3V times from observation

O(V) Iterate each item in array until vertex is found

$$T(V,E) = \underbrace{O(V)}_{\text{init}} + \underbrace{V \times O(V)}_{\text{pop}} + \underbrace{3V \times O(V)}_{\text{edit}}$$
$$= O(V) + O(V^2) + O(3V^2)$$
$$= O(V^2)$$

```
uint64_t dijkstaB(AdjacentList Adj, int Ve, int source)
{
    uint64_t start = timeNow();

    // d keeps track of shortest distance of vertex from source
    // pi keeps track of parent of vertex
    int d[Ve], pi[Ve];

    // visited keeps track of whether vertex has been completely visited
    bool visited[Ve];

    // our priority queue for getting next shortest distance
    PriorityQueueMinHeap prioQueue;

    for (int i = 0; i < Ve; i++)
    {
        // distance of source from source is 0
        if (i == source)
            d[i] = 0;
        else
            d[i] = INT32_MAX;

        pi[i] = -1;
        visited[i] = false;

        prioQueue.add(i, d[i]);
    }

    long debugB = 0;

    // continue until priority queue is empty, i.e. no more vertices to explore
    while (!prioQueue.isEmpty())
    {
        int *pair = prioQueue.pop();
        int u = pair[0], dist = pair[1];

        visited[u] = true;

        // loops through distance for all vertices from u by looking at the adj list
        // surprisingly this should be more efficient than our adj matrix
        for (int i = 0; i < Adj.list[u].size(); i++)
        {
            int *edge = Adj.list[u][i];
            int vertex = edge[0], distanceFromU = edge[1];

            if (!visited[vertex] && (d[u] + distanceFromU) < d[vertex])
            {
                debugB++;
                int shorterDistance = d[u] + distanceFromU;
                d[vertex] = shorterDistance;
                pi[vertex] = u;
                prioQueue.edit(vertex, shorterDistance);
            }
        }
    }

    uint64_t timeTaken = timeNow() - start;
    return timeTaken;
}
```

O(V)

Heapify doesn't conduct any swaps since all distances are infinity

Loop V times

$O(\log_2 V)$

Loop E times max

$O(V+\log_2 V)$

It's $O(V+\log_2 V)$ because we iterate through the array to find the vertex first then heapify

See PriorityQueueMinHeap.h

$$T(V,E) = \underbrace{O(V)}_{\text{init}} + \underbrace{V \times O(\log_2 V)}_{\text{pop}} + \underbrace{E \times O(V+\log_2 V)}_{\text{edit}}$$
$$= O(V) + O(V\log_2 V + EV + E\log_2 V)$$
$$= O(V\log_2 V + E\log_2 V + EV)$$
$$= O(V\log_2 V + VE)$$