Analysis of Graph Algorithms

We first analyze *Breadth First Search* with a *rough analysis* of the algorithm in order to develop some intuition. We then build on this analysis to provide a more accurate estimate.

Breadth First Search Rough Analysis

Here is the pseudocode for the algorithm along with the estimated time complexity for each line:

	procedure `BFS`(G, src):			
1	**for** $v \in V$:	V times		
2	unmark v		$O(1)$	
3	reset *parent*		$O(1)$	
4	reset *distance*		$O(1)$	
5	reset *src* distance	$O(1)$		
6	mark *src*	$O(1)$		
7	$Q \leftarrow$ `Make-Queue`(src)	$O(1)$		
8	**while** $Q \neq \emptyset$:	V times		
9	$v \leftarrow$ `Pop`(Q)		$O(1)$	
10	**for** $u \in Adjacent[v]$:		E_i times	
11	**if** u is unmarked:			$O(1)$
12	update u			$O(1)$
13	mark u			$O(1)$
14	`Add`(Q, u)			$O(1)$

The time complexity estimates in the pseudocode above come from the following observations:

- First consider the complexity of the *Queue* operations. If we use a *Linked List* with pointer to the *tail node* the `Queue` operations `MakeQueue`, `Add`, and `Pop` can be implemented efficiently in $O(1)$.
- $O(V)$`, Lines 1-5, Init:`
  Initialization is $O(V)$ since the loop is executed once per vertex and we do constant amount of work per vertex.
- $O(1)$`, Lines 6-7, Setup:`
  This is $O(1)$ given the previous note about *LinkedList* as a choice for *Queue*.
- `Line 8, While Loop:`
  The `while` loop is executed V times. This may not be clear immediately, but it follows from the fact that each vertex will enter the `Queue` *exactly once* and will leave the `Queue` *exactly once*. This is ensured by the marking strategy – once a vertex enters the `Queue` it is marked which prevents it from entering the `Queue` twice.
- $O(V)$`, Line 9, Finish:`
  Line 9, $Pop(Q)$, which is $O(1)$, is executed V times by the `while` loop (once per vertex) after which the `Queue` is empty.
- $O(V \cdot E)$`, Lines 10-14, Explore:`
  The `for` loop in Line 10 will execute at most E times. The `for` loop simply looks at the adjacent edges of v, so at most we may have to examine all edges in the graph. The work inside the `for` loop is $O(1)$ and since these lines are repeated V times by the **while** loop, the total is $O(V \cdot E)$.

Overall the time complexity is:

$$Init(Lines\ 1:5) + Setup(Lines\ 6:7) + Finish(Line\ 9) + Explore(Lines\ 10:14)$$

$$O(V) + O(1) + O(V) + O(V \cdot E) = O(V \cdot E)$$

Our estimate of $O(V \cdot E)$ suggests that the algorithm is impractical for *dense* graphs. If the graph is fully connected, i.e. every vertex is connected to every other vertex, then we can estimate that $E \approx V \cdot V$ (actually $E = V \cdot (V-1)/2$), which implies that *BFS* is $O(V \cdot E) = O(V^3)$, i.e. not practical for large graphs.

Breadth First Search Precise Analysis

We now consider a more accurate analysis of *BFS*. The overestimate in our analysis is in *Line* 10. Clearly, we do not need to explore all edges in the graph for each vertex. Instead, for each vertex v we only explore the adjacent edges for this vertex which is some number $Adj_v = E_v$.

The precise analysis breaks $Explore(Lines\ 10:14)$ by looking at the time spent to process each vertex during its *Finish* and *Explore* steps. Here is the `while` loop, shown as the individual V cycles along with work per vertex:

Popped	#Adjacent	Work per Adjacent Lines 11-14
v_1	E_{v_1}	$O(1)$
v_2	E_{v_2}	$O(1)$
v_3	E_{v_3}	$O(1)$
$\cdots$	$\cdots$	$\cdots$
v_V	E_{v_V}	$O(1)$
Total	$\sum E_{v_i} \cdot O(1)$	

What remains is to see if we can provide an estimate for $\sum E_{v_i}$. We claim that

$$E_{v_1} + E_{v_2} + E_{v_3} + \cdots + E_{v_V} = 2E$$

Even though we do not know the individual terms in the above summation we actually know the overall value of the summation itself. This value is just $2E$, since every time we look at an adjacent vertex we effectively look at one of the edges (a, b). Also, each edge (a, b) is looked at *twice* — once from the point of view of vertex a and once from the point of view of vertex b (we are assuming undirected graph).

Finally, we get

$$Explore(Lines\ 10:14) = \sum E_{v_i} \cdot O(1) = 2E \cdot O(1) = O(E)$$

Note that we do not multiply by V for the `while` loop, since the individual cycles of the `while` loop are already taken into account in the rows of the table.

Replacing in the earlier ananysis, we get:

$$Init(Lines\ 1:5) + Setup(Lines\ 6:7) + Finish(Line\ 9) + Explore(Lines\ 10:14)$$

$$O(V) + O(1) + O(V) + O(E) = O(V + E)$$

This is much better than the first estimate. If the graph is fully connected, in which case $E \approx V^2$, we get that *BFS* is $O(V + V^2)$ or $O(V^2)$, not $O(V^3)$.

Prim's Algorithm Precise Analysis

The analysis of Prim's algorithm is almost identical to the analysis of *BFS*. The only difference comes from the fact that we use *PriorityQueue*, so the complexity of the operations is no longer $O(1)$.

There are two operations to consider: `Pop-Min` and `Decrease-Key`. The second operation should rearrange the *PriorityQueue* after the *comparison value/key* is updated. In *Prim* this value is the `distance` field of a vertex.

Here are a few possibilities for *PriorityQueue* and the complexity of operations:

	`Make-PQ`	`Pop-Min`	`Decrease-Key`
BST	$O(n \log n)$	$O(\log n)$	$O(\log n)$ but need to `Remove`+`Add`
Heap	$O(n)$	$O(\log n)$	$O(\log n)$ only `Push-Up`
Fibonacci Heap	$O(n)$	$O(\log n)$	$O(1)$
LinkedList	$O(n)$	$O(n)$	$O(1)$

The *LinkedList* is given as an exercise. In principle, *PriorityQueue* may manage the items by storing them in a *sorted* or *unsorted* linked list, although this may not be ideal implementation in most cases.

The analysis below will be based on the *Heap* option. Effectively, this means that in the algorithm analysis, which is essentially the analysis of *BFS*, we need to multiply by a factor of $\log V$.

```
procedure Prim(G, src):
 1  for v ∈ V:                                              V times
 2      unmark v                                                       O(1)
 3      reset parent                                                   O(1)
 4      reset distance                                                 O(1)
 5  reset src distance                                      O(1)

 6  PQ ← Make-PQ(V(G))                                      O(V)

 7  T ← Make-Graph(V(G))                                    O(1)

 8  while Q ≠ ∅:                                            V times
 9      v ← Pop-Min(Q)                                                 O(log V)
10      mark v                                                         O(1)

11      Add(T, edge(u, v))                                             O(1)

12      for u ∈ Adjacent[v]:                                           E times
13          if u is unmarked and dist[u] > weight(v → u):                          O(1)
14              dist[u] ← weight(v → u), i.e. Decrease-Key                         O(log V)
15              update parent                                                      O(1)
```

The analysis is as follows:

- $O(V)$, `Lines 1-5, Init:`
  Same as in BFS.
- $O(V)$, `Lines 6-7, Setup:`
  Building a *Heap* with all vertices is $O(V)$ and building an empty *Graph* is $O(1)$.
- `Line 8, While Loop:`
  Executes V times. The *PriorityQueue* starts with all vertices and one vertex is popped per cycle.
- $O(V \log V)$, `Lines 9-11, Finish`
  Same as BFS but removing from the *PriorityQueue* is $O(\log V)$.
- $O(E \log V)$, `Lines 9-11, Finish`
  Similar to BFS in the table below.

The table from *BFS* for the `while` loop analysis is adapted to *Prim* to reflect the log factor per queue operation:

Popped	#Adjacent	Work per Adjacent, Lines 13-15 dominated by Line 14:`Decrease-Key`
v_1	E_{v_1}	$O(\log V)$
v_2	E_{v_2}	$O(\log V)$
v_3	E_{v_3}	$O(\log V)$
$\cdots$	$\cdots$	$\cdots$
v_V	E_{v_V}	$O(\log V)$
Total	$\sum E_{v_i} \cdot O(\log V)$	

Finally, we get

$$Explore(Lines\ 12:15) = \sum E_{v_i} \cdot O(\log V) = 2E \cdot O(\lg V) = O(E \log V)$$

and overall

$$Init(Lines\ 1:5) + Setup(Lines\ 6:7) + Finish(Lines\ 9:11) + Explore(Lines\ 12:15)$$

$$O(V) + O(V) + O(V \log V) + O(E \log V) = O((V + E) \log V)$$

The same analysis also applies to *Dijkstra's Algorithm*, since the two algorithms only differ in what value they compute for the *dist*[] field. Thus, both algorithms are $O((V + E) \log V)$ if *Heap* is used.

Note: *Line* 14 requires special attention, since it depends on the choice of *PriorityQueue*. As an exercise repeat the analysis for the various *PriorityQueue* options.