Analysis of Graph Algorithms

We first analyze Breadth First Search. The analysis of the non-recursive version of Depth First Search is identical.

We first consider 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:

\[
\text{BFS}(G, s) \quad O(V)
\]

1. \(\text{for } v \in V \)
2. \(\text{do } \text{color}[v] \leftarrow \text{WHITE} \quad O(1)\)
3. \(Q \leftarrow \text{Make-Queue}() \quad O(1)\)
4. \(\text{color}[v] \leftarrow \text{GRAY} \quad O(1)\)
5. \(\text{Add}(Q, s) \quad O(1)\)
6. \(\text{while } Q \neq \emptyset \quad O(V)\)
7. \(\text{do } v \leftarrow \text{Pop}(Q) \quad O(1)\)
8. \(\text{color}[v] \leftarrow \text{BLACK} \quad O(1)\)
9. \(\text{for } u \in \text{Adjacent}[v] \quad O(E)\)
10. \(\text{do if } \text{color}[u] == \text{WHITE} \quad O(1)\)
11. \(\text{then } \text{color}[u] \leftarrow \text{GRAY} \quad O(1)\)
12. \(\text{Add}(Q, s) \quad O(1)\)

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

- The first point to consider is the complexity of the operations of the Queue data structure. If we use a Linked List with pointer to the tail node the Queue operations MakeQueue, Add, and Pop can be implemented efficiently in \(O(1)\).

- The other important point is that the body of the while loop, will be executed \(V\) times – the number of vertices in the graph. 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 coloring strategy – once a vertex enters the Queue it is colored GRAY which prevents it from entering the Queue twice. This mean that Line 7, Pop(Q), which is executed every time through the while loop is executed at most \(V\) times (once per vertex) after which the Queue is empty.

- The final point to note is that the for loop in Line 9 will execute at most \(E\) times. After all, the for loop simply looks at the adjacent edges of \(v\) and at most we may have to examine all edges in the graph.

- Clearly, the initialization step takes \(O(V)\) time since the loop is executed once per vertex and we do constant amount of work per vertex.

Overall the time complexity is
Individually we have

- **Init**(Lines 1 : 2) takes \( O(V) \) time
- **Setup**(Lines 3 : 5) takes \( O(1) \) time
- **Search**(Lines 6 : 12) can be divided into

\[
\text{Finish}(\text{Lines 7 : 8}) + \text{Explore}(\text{Lines 9 : 12})
\]

That is for each vertex \( v \) we perform a finish step (pop and mark), which takes \( O(1) \) time and we explore its neighbors, which takes at most \( O(E) \) times, i.e. at most all edges need to be explored. Thus per vertex we spend \( O(E) \) time for Finish and Explore, so for all vertices **Search**(Lines 6 : 12) takes \( O(V * E) \)

Finally, the BFS time complexity is

\[
\begin{align*}
\text{Init}(\text{Lines 1 : 2}) + \text{Setup}(\text{Lines 3 : 5}) + \text{Search}(\text{Lines 6 : 12}) &= O(V) + O(1) + O(V * E) \\
&= O(V + E)
\end{align*}
\]

or \( O(V * E) \) since this is the dominating term.

Our estimate of \( O(V * E) \) 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 \( E \approx V * V \) (actually \( E = V * (V - 1) / 2 \)), which implies that BFS is \( O(V * E) = O(V^3) \), i.e. not practical for large graphs.

*Breath First Search* Precise Analysis

We now consider a more accurate analysis of BFS. The overestimate in our analysis is in Line 9. 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 \( Adj_v \).

The more precise analysis breaks **Search**(Lines 6 : 12) by looking at the time spent to process each vertex during its Finish and Explore steps. Here is the while loop, unwound to show the time spent per vertex:

<table>
<thead>
<tr>
<th>Popped</th>
<th>Finish</th>
<th>Explore</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 )</td>
<td>( O(1) )</td>
<td>( Adj_{v_1} * O(1) )</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>( O(1) )</td>
<td>( Adj_{v_2} * O(1) )</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>( O(1) )</td>
<td>( Adj_{v_3} * O(1) )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( v_V )</td>
<td>( O(1) )</td>
<td>( Adj_{v_V} * O(1) )</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>( V * O(1) )</td>
<td>( \Sigma Adj_{v_i} * O(1) )</td>
</tr>
</tbody>
</table>

What remains is to see if we can provide an estimate for \( \Sigma Adj_{v_i} \). We claim that

\[
Adj_{v_1} + Adj_{v_2} + Adj_{v_3} + \cdots + Adj_{v_V} = 2E
\]

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 \( (a, b) \), so 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 \( b \).

Finally, we get

\[
\text{Search}(\text{Lines 6 : 12}) = V * O(1) + \Sigma Adj_{v_i} = V * O(1) + 2E * O(1) = O(V) + O(E) = O(V + E)
\]
The analysis of Prim’s algorithm is almost identical to the analysis of BFS. The only difference comes from the fact that we use a PriorityQueue, so the complexity of the operations Pop and Add is no longer $O(1)$. Instead

- if we use a LinkedList to implement the PriorityQueue the complexity of Add is $O(1)$ (just add to the end or to the front), but the complexity of Pop is $O(n)$ since we have to traverse the whole list to find the smallest item.

- if we use a BinarySearchTree to implement the PriorityQueue the complexity of both Add and Pop becomes $O(\log n)$

The second option is better so in our analysis we will use $O(\log n)$ for the Queue operations. Effectively, this means that in the algorithm analysis, which is essentially the analysis of BFS, we need to multiply by a factor of $\log n$.

Here is the pseudocode for Prim’s Algorithm:

```
PRIM(G, s)
1   for v ∈ V
2       do dist[v] ← ∞
3           parent[v] ← NIL
4           color[v] ← GRAY
5       dist[s] ← 0
6   Q ← MAKE-PRIORITY-QUEUE()
7   for v ∈ V
8       do ADD(Q, s)
9   while Q ≠ ∅
10      do v ← POP(Q)
11         color[v] ← BLACK
12      for u ∈ Adjacent[v]
13         do if color[u] ≠ BLACK and dist[u] > weight(u, v)
14            then dist[u] ← weight(u, v)
15            parent[u] ← v
```

The analysis is as follows:

- **Init(Lines 1 : 5)** takes $O(V)$ — process $V$ vertices with $O(1)$ per vertex

- **Setup(Lines 6 : 8)** takes $O(V \times \log V)$ — add $V$ vertices to the queue which costs $O(\log V)$ per vertex

- **Search(Lines 9 : 17)** — we use the same per-vertex analysis as we did in BFS; however, this time Finish will take $O(\log V)$ because Pop is costlier and similarly Explore will take $O(\log V)$ per adjacent vertex because Add is $O(\log V)$.

Here is the table from BFS for the while loop analysis adapted to Prim’s Algorithm to reflect
Finally, we get

\[
\text{Search}(\text{Lines } 9 : 17) = V \cdot O(\log V) + \sum \text{Adj}_{v_i} \cdot O(\log V) = \\
V \cdot O(\log V) + 2E \cdot O(\log V) = (V + 2E) \cdot O(\log V) = O((V + E) \cdot \log V)
\]

Overall, for \textit{Prim’s Algorithm} we get

\[
\text{Init}(\text{Lines } 1 : 5) + \text{Setup}(\text{Lines } 6 : 8) + \text{Search}(\text{Lines } 9 : 17) = \\
O(V) + O(V \cdot \log V) + O((V + E) \cdot \log V)
\]

which is \(O((V + E) \cdot \log V)\), since this is the dominating term.

**Note:** As we discussed, \textit{Line} 14 represents a relaxation step for vertex \(u\). This means that \textit{Line} 14 is not \(O(1)\), since the \texttt{PriorityQueue} may need to be re-organized. A simple strategy, if we use a \texttt{BinarySearchTree}, is to delete the vertex and re-insert it which will cost \(O(\log V)\).

**Note:** An implementation of \texttt{PriorityQueue} using a data structure called \texttt{FibonacciHeap} allows for \textit{Line} 14 to be implemented in \(O(1)\) time (amortized). Will this change the complexity of \textit{Prim’s Algorithm}?

The same analysis also applies to \textit{Dijkstra’s Algorithm}, since the two algorithms only differ in what value they compute for the \textit{dist}[] field. Thus, both algorithms are \(O((V + E) \cdot \log V)\).