Exact Solution to the Clique Problem in Polynomial Time

Rosie Bartlett
lvff38@durham.ac.uk

July 11, 2022

Abstract

In this paper I will present the neighbourhood group algorithm for finding an exact solution to the clique problem in polynomial time dependant on the clique size.

1 The problem

The problem is henceforth defied as follows; for any arbitrary graph with n nodes, check for the existence of a clique of size k.

k will henceforth be used solely for the clique size. n refers to the number of nodes in the graph only when referred to in matters of time complexity

2 Algorithm

Please note here that I define $N(n)$ as the set of nodes adjacent to n.

I now present the Neighbour Search algorithm with notation `NeighbourSearch`(Q, I, t) with Q being a set of nodes making up a clique, I being the set of nodes that may be included in Q maintaining it being a clique, and t being $k - |Q|$.

The algorithm works by progressing through cliques, making them larger until $t = 0$ at which point a clique of size k has been found, or until the clique becomes maximal and $I = \emptyset$ at which point (assuming $t > 0$ otherwise the first condition applies) the algorithm will look for different cliques.

To calculate cliques, the algorithm starts with a node n_1 and gets $N(n_1)$. For all the cliques containing n_1, the nodes in any of the cliques must be adjacent to n_1 and thus in $N(n_1)$. We then go through each node $n_2 \in N(n_1)$. For n_1 and n_2 to be in a clique, any other nodes in the same clique must be in both $N(n_1)$ and $N(n_2)$ or $N(n_1) \cap N(n_2)$. This repeats, taking a node n from the set of available nodes I and updating the available set to $I \leftarrow I \cap N(n)$ until one of the conditions mentioned above is reached.

Algorithm 1: Neighbourhood Search Algorithm

```
1 Function NeighbourhoodSearch(Q, I, t): bool
2     foreach n ∈ I do
3         if NeighbourhoodSearch(Q ∪ {n}, I ∩ N(n), t − 1) then
4             return true
5         end
6     end
7     return false
```

Algorithm 2: Algorithm entry point

Data: $G = (V, E), k$
Result: boolean representing the existence of a clique of size k in G

```
1 foreach n ∈ V do
2 |   if NeighbourhoodSearch(Q ∪ {n}, I ∩ N(n), t − 1) then
3 |   |   return true
4 |   end
5 end
6 return false
```

2.1 Time complexity

IN PROGRESS

Firstly, the main loop (lines 2-17) runs n times. We then reach a conditional which we will assume the else condition (lines 5-16) since it has the larger time complexity and will give the worst case complexity for the algorithm. We then reach a second conditional (lines 6-15). This is here for optimisation. We will assume that this runs for each node for the worst case. The loop inside this condition (lines 7-14) runs at most $n-1$ times when the current node is connected to all other nodes. The while loop (lines 8-13) will run up to $n-2$ times in the case that n_1 and n_2 are connected to all the other nodes, each of which are connected to only n_1 and n_2. The section within the while loop (lines 10-12) will run in linear time since al operations can be performed within linear time by use of fixed length arrays for set operations.

We also precompute $N(n) \forall n \in V$ taking at most $O(n^3)$[1].

In total this gives $O(n^3) + O(n^4) = O(n^4)$ for the algorithm.

2.2 Lower bound

This algorithm has a lower bound of $\Omega(1)$ for a graph with one node. This is not very useful, so we instead consider the best case performance for a graph with n nodes.

In this graph, we would find the largest clique in the first iteration which would contain all the nodes in the graph giving the largest possible k value of n. This means that the first iteration would take linear time by the set operations. Any following iteration would take constant time since very little is actually run giving a total of $\Omega(n)$ for the lower bound. This is excluding the neighbourhood pre-computation.

3 Correctness

Let n_1 be a node in V. If we consider the possible cliques n is contained within, by definition all of these cliques must contain nodes within $N(n_1)$ otherwise n would not be adjacent to all nodes in the subgraph, and thus the subgraph would not be a clique. We now place a condition on the following section. If $|N(n_1)|$ is less than the size of the current largest clique, there is no possible way in which this node will give a larger clique and we can ignore it. If this is not the case we continue. By then taking $n_2 \in N(n_1)$, any clique that n_2 is contained within must contain nodes within $N(n_2)$ for the same reason. Therefore, any clique that contains both n and n_2 can only contain nodes within $N(n_1)$ and $N(n_1)$ or $N(n_1) \cap N(n_2)$ which we will call I.

It is not guaranteed that I, along with n_1 and n_2, will be a clique however since nothing we have done thus far would allow us to assume this without loss of generality. We can assume though that any two unique nodes in I there is a path between them with a length of 2 through either n_1 or n_2. If we then go through each $n_3 \in I$ and take $\{n_1, n_2\} \cup (I \cap N(n_3))$ we end up with a clique since each node in I is at adjacent to both n_1 and n_2, and each node in $N(n_3)$ is adjacent to n_3 which means that each node in $I \cap N(n_3)$ is adjacent to n_1, n_2, n_3. This is an improvement since in I there were nodes that were 2 away, but by

[1] n repetitions of going through each edge for which there are at most $\frac{n(n-1)}{2}$ giving $O(n^3)$

constraining I by the neighbourhood of a node in I we remove any nodes in I that are not adjacent to the currently considered node in I giving a clique.
Unlike when working through $N(n_1)$, we can filter out some of the nodes in I as we go to increase speed since we're not considering the entirety of $N(n_3)$, but only finding connected sections in I.
This does not find all the cliques in the graph. It does however find the largest cliques given two adjacent nodes which allows us to then give $\omega(G)$ which would them allow us to state the existence of a k-clique in G.

4 Conclusion

This paper has presented a polynomial time algorithm for finding all cliques for an arbitrary undirected graph. There is room for improvement in the given algorithm. For example one might improve the number of iterations by not computing identical cliques; for example in C_3 each iteration, two nodes in $N(n)$ are considered, each functionally equivalent, and then the process repeats three times for equally functionally equivalent nodes. These optimisations, other than those given, are left for future research.

A Python implementation

Here a graph is represented as a dictionary with each key value pair being a node, the key being the node index, and the value being the nodes connected to the current node.

```
def NeighbourhoodGroup(graph: {int: {int}}) -> int:
  k = 0
  for n1 in graph:
    if len(graph[n1]) == 0:
      k = max(k, 1)
    elif len(graph[n1]) >= k:
      for n2 in graph[n1]:
        I = graph[n1].intersection(graph[n2])
        t = 0
        while len(I) != 0:
          n3 = list(I)[0]
          t = max(t, len(I.intersection(graph[n3])) + 1)
          I -= graph[n3].union({n3})
        k = max(k, t + 1)
  return k
```

Listing 1: Neighbourhood search algorithm for finding the maximal clique in a graph in $O(n^4)$ time. Written in Python 3.x

B Results

This appendix contains the results from testing the Python implementation of the algorithm given in appendix A. Each graph was run 5 times and the minimum, maximum, and mean times were recorded. Table 1 gives the complete list of graph types tested and their clique number. For each graph $n, m \in [5, 100]$. This does not list the given clique number by the algorithm. For each graph the algorithm gave the correct clique number.

Graph G	Clique number $\omega(G)$
K_n	n
$K_{n,m}$	2
C_n	2
L_n	2

Table 1: Graph types tested and their respective clique number

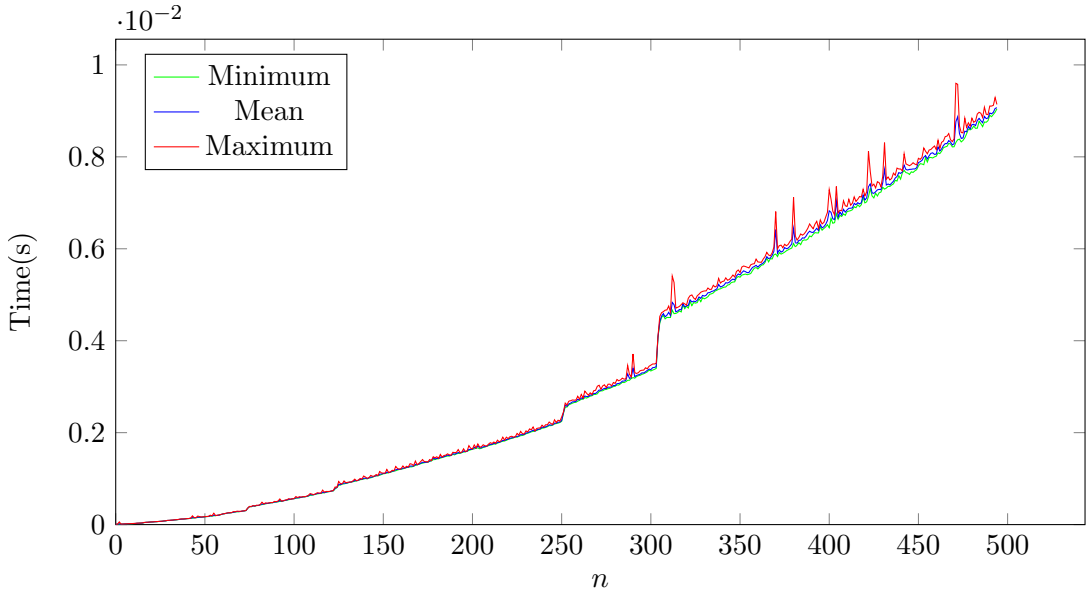

Figure 1: Time for complete graphs K_n

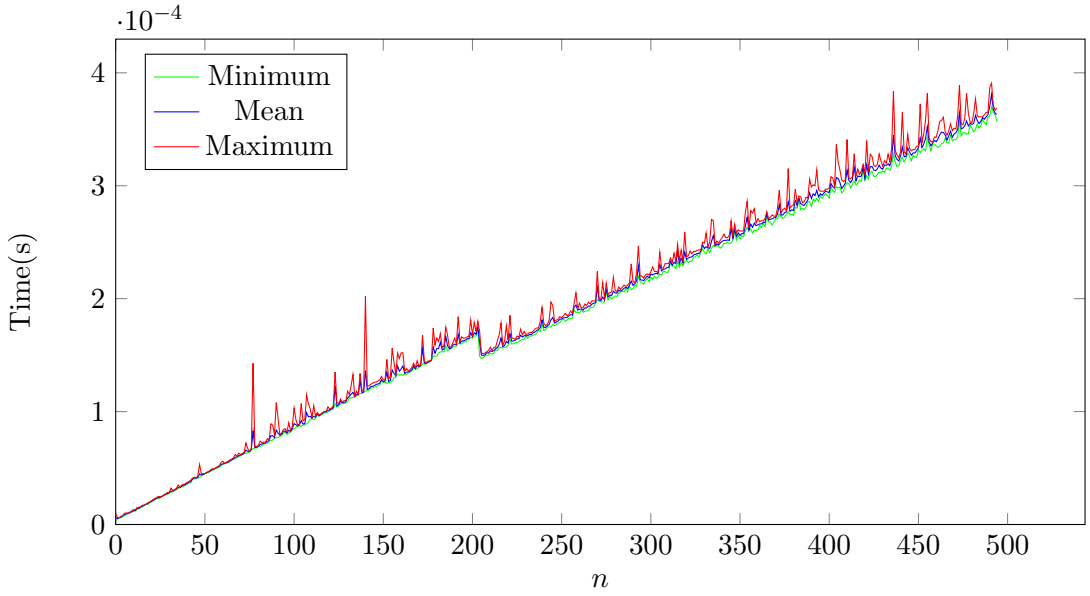

Figure 2: Time for cycle graphs C_n

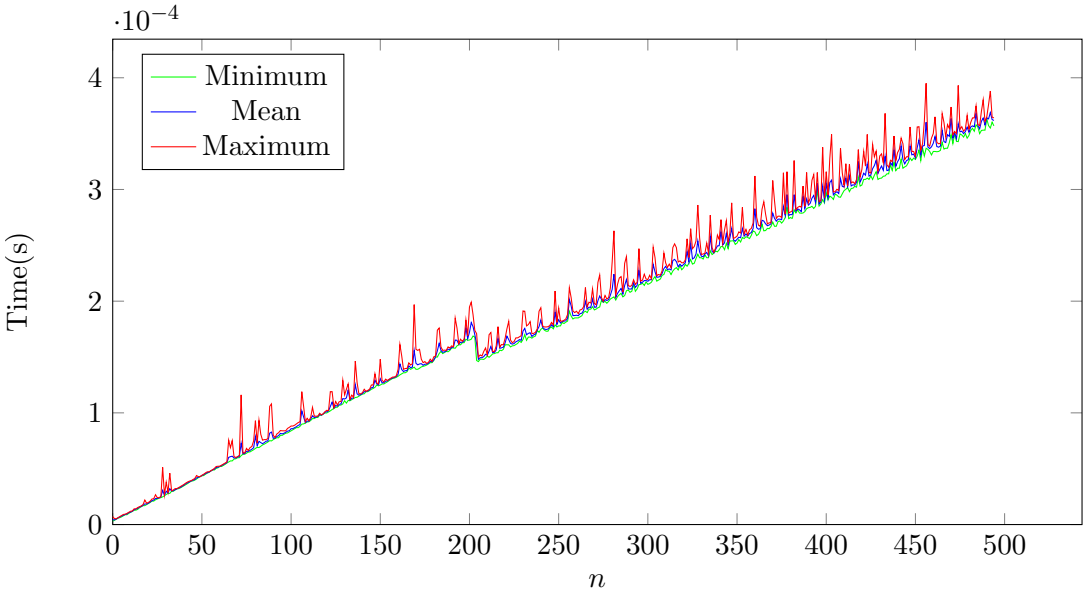

Figure 3: Time for linear graphs L_n