Exact Solution to the Clique Problem in Polynomial Time

Rosie Bartlett
lvff38@durham.ac.uk

July 12, 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 defined 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 $A(n)$ as the set of nodes adjacent to n. I also assume that no node is connected to itself. There exist trivial algorithms to filter out edges from a node to itself that run in faster time than the given algorithm.

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

2.1 Time complexity

I assume here that set operations take linear time. This can be implemented with two fixed length arrays trivially. I will also be ignoring times for function calls.

We first consider algorithm 2. This runs at most n times. We must now consider algorithm 1. Firstly the conditional on lines 2-4 takes constant time. The loop on lines 5-9 will run at most $n - k + t$ times since $|Q| = k - t$ giving $n - k + t$ not in Q which could be in I. The body of this loop takes linear time for the set operations and constant time for the comparison. Therefore, for a given function call it will take $f(t) = n(n - k + t)f(t - 1)$ time. However, we have a lower bound on the recursion depth of $k - 1$ since we only run the

Algorithm 1: Neighbourhood Search Algorithm

```
1  Function NeighbourhoodSearch(Q, I, t): bool
2      if t = 1 and I ≠ ∅ then
3          return true
4      end
5      foreach n ∈ I do
6          if NeighbourhoodSearch(Q ∪ {n}, I ∩ A(n), t − 1) then
7              return true
8          end
9      end
10     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  if k = 1 and |V| ≠ 0 then
2      return true
3  end
4  foreach n ∈ V do
5      if NeighbourhoodSearch({n}, A(n), k − 1) then
6          return true
7      end
8  end
9  return false
```

algorithm until $t = 1$ not $t = 0$. This means that in total we have

$$n \prod_{t=1}^{k-1} n(n-k+t) = O(n^{2k-1}) \tag{1}$$

2.2 Optimisations

There are probably a few optimisations we can make upon algorithm 1 and algorithm 2. One of these is that if $|A(n)| < k$ then we can ignore n. We do this since even if every node in $A(n)$ were adjacent to each other, the clique would be smaller than k and it would be use the algorithm to prove so.

It may also be useful to calculate $\Delta(G)$ to make sure that a clique of size k can theoretically exist in the given graph since $\omega(G) \leq \Delta(G)$. This optimisation would not change the time complexity of the algorithm, but would alter the lower bound to $\Omega(n)$ for $k > \Delta(G)$ giving time to calculate $\Delta(G)$.

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 $A(n_1)$ otherwise n would not be adjacent to all nodes in the subgraph, and thus the subgraph would not be a clique. By then taking $n_2 \in A(n_1)$, any clique that n_2 is contained within must contain nodes within $A(n_2)$ for the same reasons. Therefore, any clique that contains both n and n_2 can only contain nodes within $A(n_1)$ and $A(n_1)$ or $A(n_1) \cap A(n_2)$.

We now have a clique of size 2. If we continue this, selecting new nodes from the intersections of the neighbourhoods of the nodes already in the clique, we ensure that any node added to the clique is adjacent to the nodes already in the clique.

4 Conclusion

This paper has presented an exact algorithm for solving the clique problem in polynomial time for an arbitrary undirected graph. As noted in section 2.2, there is room for improvement. This paper is a proof of concept for the existence of a polynomial time algorithm for an NP-complete problem to show that P=NP. It is left to future research to optimise this algorithm.

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}}, k: int) -> bool:
  def inner(Q: {int}, I: {int}, t: int) -> bool:
    if t == 1 and len(I) != 0:
      return True
    for n in I:
      if inner(Q.union({n}), I.intersection(graph[n]), t - 1):
        return True
    return False
  if k == 1 and len(graph) != 0:
    return True
  for n in graph:
    if inner({n}, graph[n], k - 1):
      return True
  return False
```

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

```
def NeighbourhoodGroup(graph: {int: {int}}, k: int) -> bool:
  def inner(Q: {int}, I: {int}, t: int) -> bool:
    if t == 1 and len(I) != 0:
      return True
    for n in I:
      if inner(Q.union({n}), I.intersection(graph[n]), t - 1):
        return True
    return False
  if k == 1 and len(graph) != 0:
    return True
  for n in graph:
    if len(graph[n]) + 1 >= k:
      if inner({n}, graph[n], k - 1):
        return True
  return False
```

Listing 2: The more optimised version of the previous implementation

B Results

This appendix contains the results from testing the Python implementation of the algorithm given in listing 2. This only contains times. The algorithm gave the expected answer for all tested graph. The code for running these benchmarks along with the code for checking the algorithm and raw data in CSV format gives the expected value are available in this repository in the `benchmark.py` and `test.py` files respectively

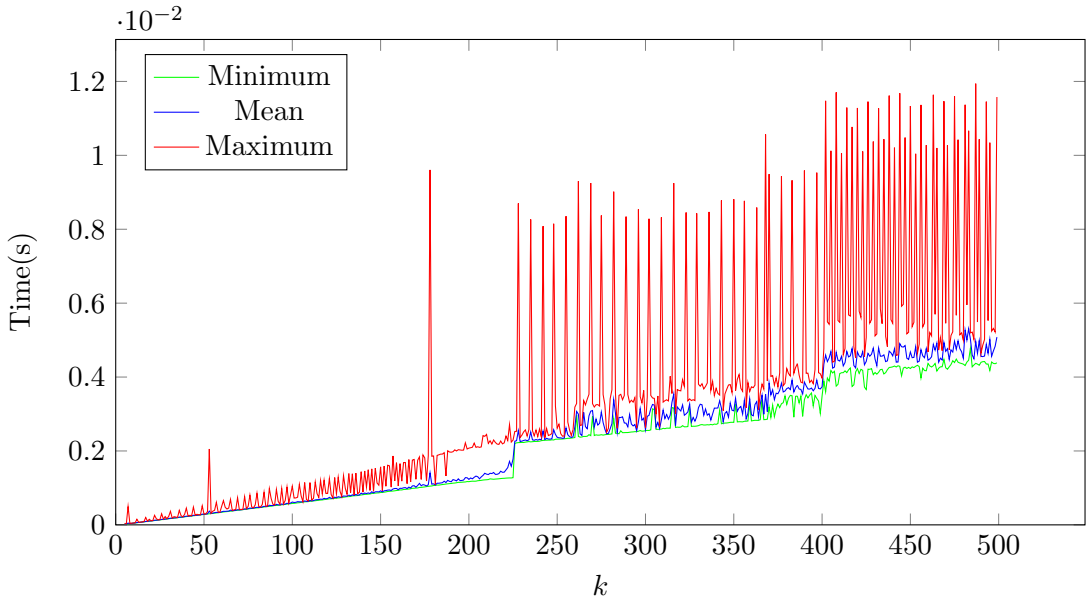

Figure 1: Times for `NeighbourhoodSearch`(K_{500}, k) using 25 repetitions

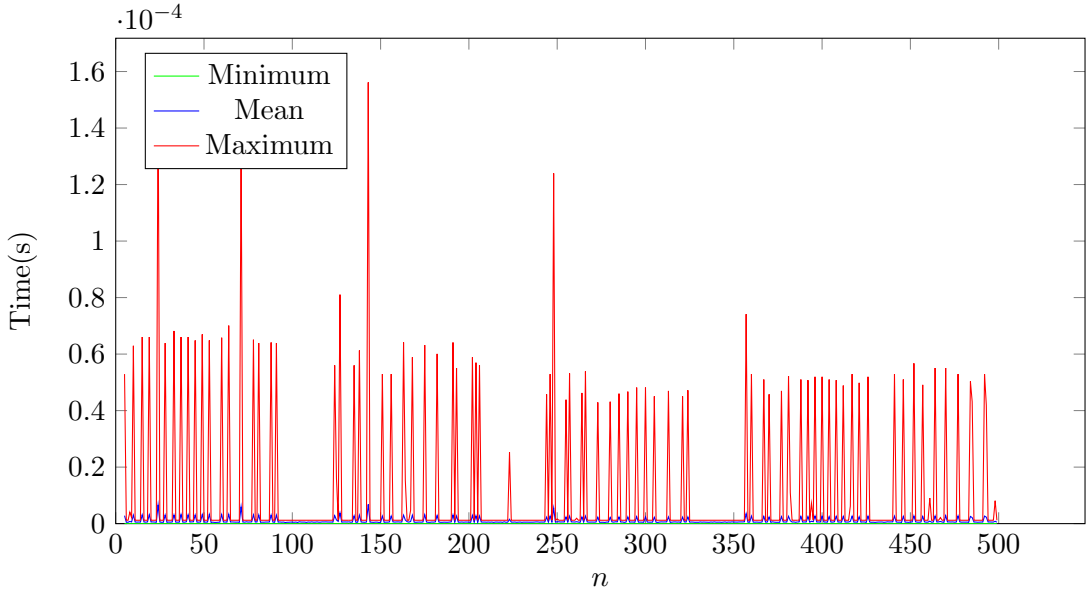

Figure 2: Times for `NeighbourhoodSearch`$(C_n, 2)$ using 25 repetitions. Here I start at C_3

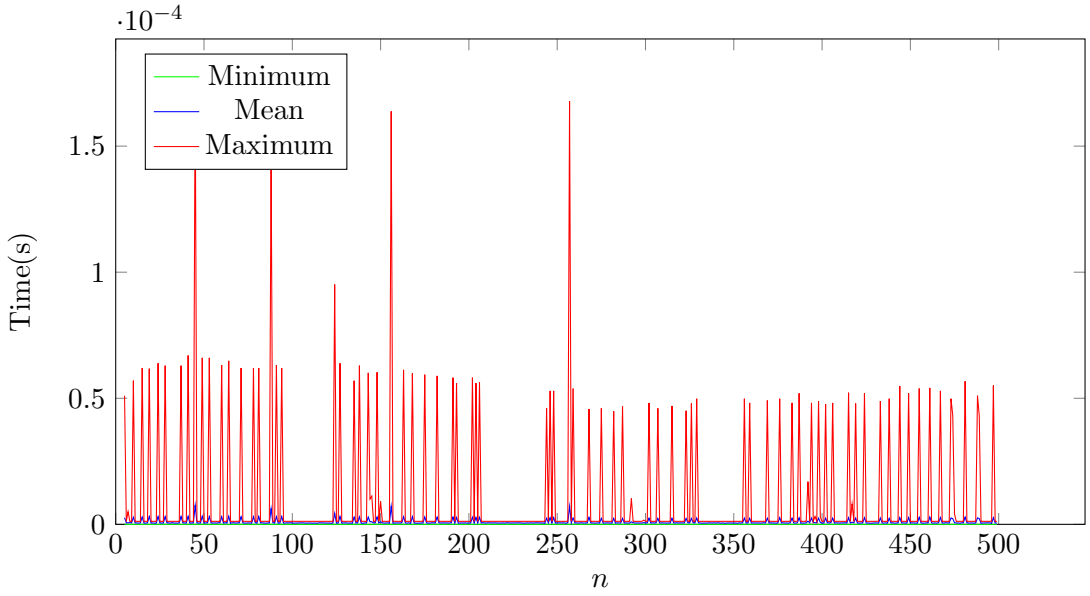

Figure 3: Times for `NeighbourhoodSearch`$(L_n, 2)$ using 25 repetitions. Here I start at L_3

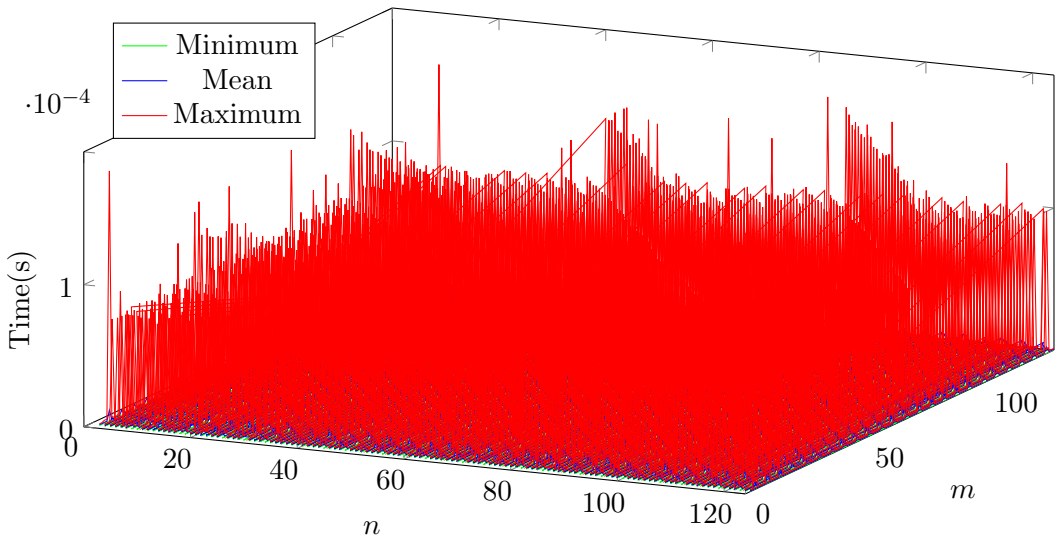

Figure 4: Times for `NeighbourhoodSearch`$(K_{n,m}, 2)$ using 25 repetitions. Here I start at $K_{2,2}$

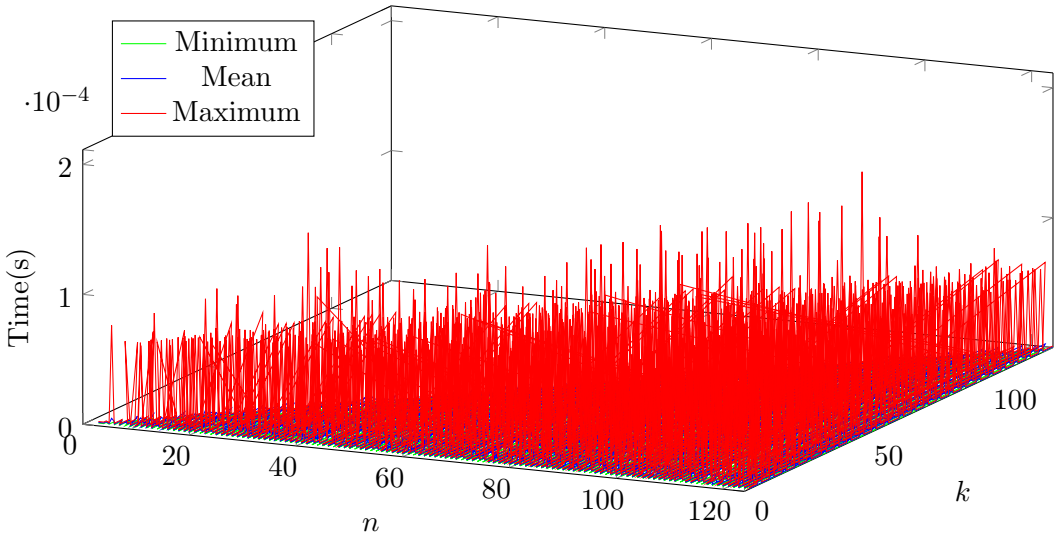

Figure 5: Times for `NeighbourhoodSearch`(K_n, k) using 25 repetitions. Here I start at $K_{2,2}$ and constrain k to $2 < k \leq n$.