Not Clique-bait: Algorithmic Approaches to the Maximum Clique Problem

Brendan Banfield
Aaron Banse
Kellen Knop
Sophie Quinn

Senior Integrative Exercise, Department of Computer Science, Carleton College
March 2025

1 Abstract

The maximum clique problem is a well-studied challenge in graph theory with applications in network analysis, bioinformatics, and social network modeling. This paper examines four algorithms for solving the problem, comparing their efficiency, accuracy, and scalability. We consider both exact and approximate approaches, discussing their computational complexity and practical applications. Experimental results are presented to evaluate performance across different graph structures. Through this analysis, we highlight the trade-offs between optimality and computational feasibility, providing insight into the strengths and limitations of these algorithms.

2 Background

The maximum clique problem is an NP-complete problem in graph theory that seeks to identify the largest clique of a given graph. Due to its computational complexity, solving this problem efficiently is a key challenge. This problem has applications over a variety of datasets, including social networks, web networks, and bioinformatics [4, 9]. We define a *graph* as an ordered pair $G = (V, E)$ where V is a set of nodes and E is a set of pairs $\{v, w\}$ with $v, w \in V$ representing edges between nodes (see Figure 1). A *complete graph* is a graph $G = (V, E)$ where $E = \{\{v, w\} \mid v, w \in V, v \neq w\}$ (see Figure 2). A *clique* of $G = (V, E)$ is a complete subgraph $C = (V', E')$, where $V' \subseteq V$, $E' \subseteq E$. See the nodes connected by orange edges in Figure 3. Notation note: in our pseudo-code, the neighbors of a node v are represented by $\Gamma(v)$.

2.1 Problem Statement

Given a graph G, the maximum clique problem aims to find the largest subgraph where all nodes are fully connected. We formalize the problem in the following formulation:

Input: An undirected, unweighted graph $G = (V, E)$.

Output: A complete subgraph $G' \subseteq G$, where $G' = (V', E')$ such that $V' \subseteq V$, $E' \subseteq E$, and $|V'|$ is maximized.

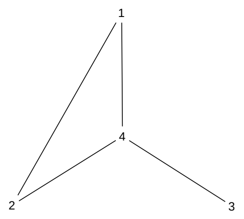

Figure 1: A graph on 4 nodes.

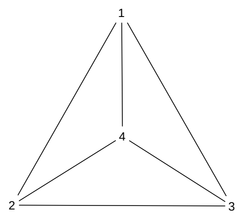

Figure 2: A complete graph on 4 nodes.

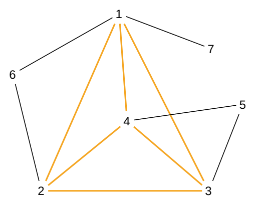

Figure 3: A graph on 7 nodes containing a maximum clique of size 4.

2.2 Algorithms

The max clique problem has been studied extensively, and there are many algorithmic solutions to it. Since the problem is NP-complete (and NP-hard in the decision-problem version), the runtime of exact solutions scales exponentially with graph size. Some exact solutions have been heavily optimized with heuristics or other methods, but still take a long time on large graphs due to this inherent limitation.

Approximation algorithms for the max clique problem have also been developed, which trade the guarantee of finding the correct solution for vastly improved runtime. These may find the largest clique, but they may also only find the second or third-largest clique. Depending on the application, approximations may be useful if simply finding any large clique is the goal.

2.2.1 Bron-Kerbosch

Bron-Kerbosch is a well-known and widely-used recursive backtracking algorithm [2]. Since Bron and Kerbosch's formulation in 1973, there have been newer versions that improve upon the original [3]. Instead of directly finding the maximum clique, this algorithm seeks to enumerate all of the *maximal* cliques. A maximal clique is one which is not contained in any larger clique (see Figure 4). Because the maximum clique by definition cannot be contained in any larger clique, it is always maximal. Therefore, it is relatively easy to modify this algorithm to return the maximum clique.

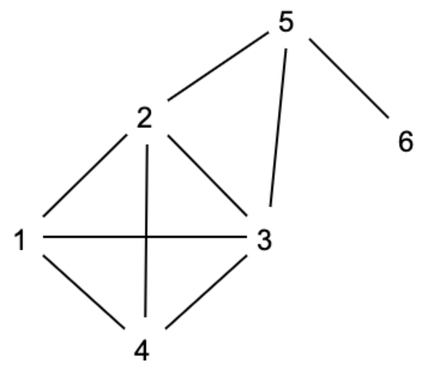

Figure 4: Graph with maximal cliques: {1,2,3,4}, {2,3,5}, and {5,6}.

The pseudo-code for both versions of Bron-Kerbosch is below in Algorithm 1. In summary, it works as follows. The main idea of the algorithm is to try all possible combinations of nodes (limited by existing edges) to find the maximal cliques. Three sets of nodes are maintained in the algorithm: R, P, and X. R gets extended by one node in each recursion and shrunk by one node on each backtrack. If R contains more than one node, it will always represent a clique. P is the set of candidate nodes that can extend R. X is the set of nodes that have already been tested by R.

In the basic version, the algorithm iterates over every order of possible combination of nodes in R (line 8). For each node in P, it gets added to R, and then the function is recursively called on the new R and the nodes of sets P and X which intersect with all nodes in R (line 9). Once P is empty, a clique has been found (line 2). If X is also empty, the clique is maximal.

Intuitively, since the recursive call uses only the nodes in sets P and X that intersect with the neighbors of the new node in R, all nodes in P and X are always connected to every node in

R. Thus, when P and X are both empty, the clique in R cannot get any larger. However, if X is not empty, the nodes in X are connected to R, forming a clique, but this clique has already been found.

The more efficient version of the algorithm cuts out many redundant iterations. Because a clique is a set of nodes, and thus unordered, a maximal clique only needs to be found once. Thus, we choose a pivot vertex before the for-loop iteration. In our implementation, the node from $P \cup X$ that has the most neighbors in P is chosen as the pivot. Then the loop only iterates over the nodes in P that are non-neighbors of the pivot (including the pivot itself). On the recursive steps of that node, any maximal clique including it's neighbors will be found, so there is no need to re-check the neighbors on a later iteration.

For example, in Figure 4, the basic version would iterate over every node in the for-loop. Thus it would find the clique {1,2,3,4} four times, the clique {2,3,5} three times, and {5,6} twice. However, if we instead choose node 2 as the pivot (since it has the most neighbors), the for-loop would only iterate over nodes 2 and 6 (non-neighbors of 2). Then, each clique will still be found, but only once.

Algorithm 1: Bron-Kerbosch

```
Function SolverBasic(R, P, X):
    if $P == \emptyset$ then
        if $X == \emptyset$ then
            Report $R$ as maximal;
        end
        return
    end
    for $v$ in $P$ do
        SolverBasic($R \cup v, P \cap \Gamma(v), X \cap \Gamma(v)$);
        $P \leftarrow P/v$;
        $X \leftarrow X \cup v$;
    end

Function SolverPivot(R, P, X):
    if $P == \emptyset$ then
        if $X == \emptyset$ then
            Report $R$ as maximal;
        end
        return
    end
    Choose pivot $u \in P \cup X$ with $max(\Gamma(u) \cap P)$ ;
    for $v$ in $P/\Gamma(u)$ do
        SolverPivot($R \cup v, P \cap \Gamma(v), X \cap \Gamma(v)$);
        $P \leftarrow P/v$;
        $X \leftarrow X \cup v$;
    end
```

2.2.2 Branch and Bound

Branch and Bound is another popular algorithm used to solve combinatorial optimization problems, including the maximum clique problem, by utilizing an upper bound and recursive branching. The version that was implemented was inspired by the one outlined in Konc and Janežič

(2007) [7]. See our pseudo-code in Algorithm 2.

Our approach begins with a greedy coloring algorithm to assign a numerical "color" (starting from 1) to each node in the graph such that no adjacent nodes have the same color. The colored vertices are stored in lists inside of a list C based on their colors such that a vertex colored k can be found inside list $C[k]$. Ordering the vertices in this way allows the algorithm to quickly access the node with the highest color. By summing the size of our current clique and the "color" of the node with the highest color, we get an upper bound, or a maximum estimate, for the size that our current clique could become if we continue to explore the branch. The size of the largest clique found so far works as a lower bound, which lets the algorithm know that it can prune any branch that does not provide an upper bound that is greater than or equal to the lower bound. There is no maximum clique in the beginning, so the lower bound starts at 0. Each time before we add a new node to the current clique, we use the equation in line 24 in the pseudo-code that compares the upper and lower bounds to decide whether to prune the branch or to continue exploring it.

The algorithm is more effective on dense graphs where the algorithm can set a large lower bound on the first few iterations, which leads to effective pruning. The more the algorithm can prune in linear time, the less it has to recurse in exponential time. The algorithm is less effective on sparse graphs; the algorithm cannot prune well when the nodes' colors are mostly the same. An example would be a graph where all of the cliques are the same size. If the upper bound isn't exclusively more than the size of the lower bound, the algorithm will still spend time running on it even though it won't result in a larger clique.

Algorithm 2: Branch and Bound

```
Graph ← $G = (V, E)$
$R \leftarrow V$
Function ColorSort($R$, $Graph$):
    $C \leftarrow \emptyset$
    for $Node$ $in$ $\|R\|$ do
        $k \leftarrow 1$;
        while $\|C\| < k$ do
            Append an empty list to C;
        end
        while $\Gamma(Node) \cap C[k-1] \neq Empty$ do
            $k \leftarrow k + 1$;
            while $\|C\| < k$ do
                Append an empty list to C;
            end
        end
        Append Node to C[k-1];
    end
    return $C$
Function MaxClique($R$, $C$, $Graph$, $Q$, $Qmax$):
    while $R \neq \emptyset$ do
        $p \leftarrow$ Node with the highest color;
        $C(p) \leftarrow$ The color of $p$;
        $R \backslash \{p\}$;
        if $\|Q\| + C(p) \geq \|Qmax\|$ then
            $Q \leftarrow Q \cup p$;
            if $R \cap \Gamma(p) \neq \emptyset$ then
                $R' \leftarrow R \cap \Gamma(p)$;
                $E' \leftarrow \{e \in E | e = (u, v) \text{ with } u, v \in R \cap \Gamma(p))\}$;
                $G' \leftarrow (R', E')$;
                $C' \leftarrow ColorSort(R', G')$;
                $Qmax \leftarrow$ MaxClique($R'$, $C'$, $G'$, Q, Qmax);
            end
            else if $\|Q\| > \|Qmax\|$ then
                $Qmax \leftarrow Q$;
                $Q \backslash \{p\}$;
                return $Qmax$
            end
            $Q \backslash \{p\}$;
        end
        else
            return $Qmax$
        end
    end
```

2.2.3 Genetic Algorithm

A genetic algorithm to approximate the max clique problem was introduced in Huang (2002) [6]. Genetic algorithms build up an initial "population" of potential solutions, then progress the population by combining good solutions and adding randomness to explore more possibilities.

Each member of the population, called a chromosome, represents a subset of the vertices of the graph. It is crucial to the algorithm that these subsets are always cliques, since they represent potential solutions.

See Algorithms 3-5 for pseudo-code of our implementation, described here. To generate a chromosome for the initial population, we use a greedy algorithm, outlined below. We pick a random vertex v, and for all neighbors of v, add it to the subgraph if the result is still a clique. The order we consider neighbors to add affects the final clique found, so order is randomized to explore more possibilities.

Next, we move onto the main body of the algorithm, where we will iteratively improve the population until progress stagnates. First we select two parents from the population with probability proportional to their fitness value, the square root of their clique size. Taking the root of clique size "normalizes" probabilities of selection, making them more similar. Without it, bigger cliques would be disproportionately selected, and population diversity would diminish.

We next combine the two parents via crossover, where we swap parallel random segments from the two parents. We then mutate to add diversity. Since these operations do not preserve clique-ness, we perform a local optimization to extract and maximize a clique from the given subgraph. We first remove smallest-degree vertices until our subgraph is a clique, then add as many vertices to the clique as we can while maintaining clique-ness.

We now have two child cliques produced from the parents. The algorithm makes at most one replacement in the population for each iteration, so we select the best child, i.e. the child with the largest clique size, and compare it to the parents. If the child is larger than its most similar parent, replace that parent with the child. If not, check the less similar parent. If this fails, we check the smallest clique in the population to replace it with, but if the child is still not larger, then we do not make a replacement for this iteration, and add to the stagnancy count.

In the case that we do make a replacement, we will reset the stagnancy count. Once it reaches 50, the algorithm stops and returns the largest clique in the population.

2.2.4 Simulated Annealing

The simulated annealing algorithm presented in Geng et al. (2007) differs from the other algorithms in that it solves the decision version of the problem [5]; given a graph G and a clique size k, the algorithm searches exclusively for a clique of size k, and will never incidentally discover a smaller clique.

Simulated annealing is a random algorithm technique that uses an objective function as a measurement of how good the current state is (see Algorithm 6.) The objective function should be minimized, and reaching 0 means a perfect solution. At each time step, the algorithm will choose a small change to make to the state, and consider what would happen if the change was made. If the change decreases the objective function (i.e. improves the solution) then the change will be made. If it instead increases the objective function, the change will happen only sometimes, with a probability proportional to the *temperature*, a variable which starts high and decreases exponentially over time. This allows the algorithm to make changes that are bad in the short term, but allow it to escape local objective function minima in search of the global optimal solution. The temperature lowering over time results in the algorithm behaving more greedily the longer it runs, ensuring it will reach local minima more frequently. The algorithm

Algorithm 3: Genetic Algorithm

```
Function Solver($G$, $n$, $p_m$, $num\_cuts$):
    $P \leftarrow$ GeneratePopulation($G, n$)
    $stagnancy \leftarrow 0$
    while $stagnancy < 50$ do
        $Par_1, Par_2 \leftarrow$ SelectParents($P$)
        $C_1, C_2 \leftarrow$ Crossover($Par_1, Par_2, num\_cuts$)
        $C_1 \leftarrow$ Mutate($C_1, p_m$)
        $C_2 \leftarrow$ Mutate($C_2, p_m$)
        $C_1 \leftarrow$ LocalOptimization($C_1, p_m$)
        $C_2 \leftarrow$ LocalOptimization($C_2, p_m$)
        $P \leftarrow$ Replace($P, C_1, C_2$)
    end
Function GeneratePopulation($G$, $n$):
    $P \leftarrow \emptyset$ for $i \leftarrow 1$ to $n$ do
        Select a random vertex $v_0 \in V(G)$.
        $C \leftarrow \{v_0\}$
        $A \leftarrow \{v \in V(G) \mid (v_0, v) \in E(G)\}$
        while $A \neq \emptyset$ do
            Select a random vertex $w \in A$.
            if $\forall v \in C, (v, w) \in E(G)$ then
                $C \leftarrow C \cup \{w\}$
            end
            $A \leftarrow A \setminus \{w\}$
        end
        $P \leftarrow P \cup \{C\}$
    end
    return $P$
    Function SelectParents($P$):
        $P_A \leftarrow$ array with all elements of $P$
        $W \leftarrow$ array of size $|P_A|$
        for $i \leftarrow 1$ to $|W|$ do
            $W[i] \leftarrow \sqrt{|P_A[i]|}$
        end
        $Par_1, Par_2 \leftarrow$ two random elements of $P_A$ using probability weights in $W$,
          $Par_1 \neq Par_2$.
        return $Par_1, Par_2$
```

Algorithm 4: Genetic Algorithm contd.

```
Function Crossover($Par_1, Par_2, num\_cuts$):
    $B_1, B_2 \leftarrow$ bitarray representations of $Par_1, Par_2$
    Select a random set of non-intersecting intervals $I$ over $[1, |B_1|]$
    for each $[i_0, i_f]$ in $I$ do
        $cut\_temp \leftarrow B_1[i_0$ to $i_f]$
        $B_1[i_0$ to $i_f] \leftarrow B_2[i_0$ to $i_f]$
        $B_2[i_0$ to $i_f] \leftarrow cut\_temp$
    end
    $C_1, C_2 \leftarrow$ set representations of $B_1, B_2$
    return $C_1, C_2$
a
Function Mutate($C, p_m$):
    $r \leftarrow Random([0, 1])$
    if $r < p_m$ then
        $i \leftarrow RandomInt([1, |V(G)|])$ if $v_i \in C$ then
            $C \leftarrow C \setminus \{v_i\}$
        end
        else
            $C \leftarrow C \cup \{v_i\}$
        end
    end
    return $C$
Function LocalOptimization($G, C$):
    while $C$ is not a clique do
        Find $v \in C$ s.t. $\{w \in C \mid \{v, w\} \in E(G)\}$ is minimal.
        $C \leftarrow C \setminus \{v\}$
    end
    Let $V_r$ be a random ordering of $V(G) \setminus C$.
    for $v \in V_r$ do
        if $\forall w \in C, \{v, w\} \in E(G)$ then
            $C \leftarrow C \cup \{v\}$
        end
    end
    return $C$
```

Algorithm 5: Genetic Algorithm contd.

```
Function Replace($P, Par_1, Par_2, C_1, C_2$):
    $C' \leftarrow \max(C_1, C_2, \text{key} = |C|)$
    $R_1 \leftarrow \min(Par_1, Par_2, \text{key} = hamming(C', Par))$
    $R_2 \leftarrow \max(Par_1, Par_2, \text{key} = hamming(C', Par))$
    if $|C'| > |R_1|$ then
        Replace $R_1$ with $C'$ in $P$.
    end
    else if $|C'| > |R_2|$ then
        Replace $R_2$ with $C'$ in $P$.
    end
    else
        $P_{min} \leftarrow \min(P, \text{key} = |P_i|)$ if $|C'| > |P_{min}|$ then
            Replace $P_{min}$ with $C'$ in $P$.
        end
        else
            $stagnancy \leftarrow stagnancy + 1$
        end
    end
```

will continue running until the temperature reaches some final temperature value. The initial temperature, final temperature, and the decay rate are parameters.

This algorithm takes as parameters a graph G with adjacency matrix A_G, clique size k, initial temperature τ_0, final temperature τ_f, and decay rate α. It creates a set of vertices C of size k. It gradually tries to make this set contain a clique by swapping vertices in C for vertices not in C. The objective function is the number of missing edges between vertices in the "clique" C:

$$f(A_G, C) = \frac{1}{2} \sum_{v_1, v2 \in C} (1 - A_G(v_1, v_2)) \tag{1}$$

where the leading $\frac{1}{2}$ accounts for summing over both the pair v_1, v_2 and v_2, v_1. At each time step, the algorithm picks random vertices $v_1 \in C$ and $v_2 \notin C$. Let c_1 be the objective function value if no swap is made, and c_2 be the objective function value after swapping v_1 with v_2, and let $\Delta c = c_2 - c_1$. If $\Delta c \leq 0$, the swap would improve the clique (or leave it the same), so we always make it. If instead $\Delta c > 0$, we make the swap with probability $p = e^{-\Delta c/\tau}$ for current temperature τ. This probability is lower when Δc is higher, meaning the swap is more harmful to the connectedness of C, and lower when the temperature is low, near the end of the algorithm.

Algorithm 6: Simulated Annealing

```
Function NumMissingEdges($A_G$, $C$):
    $numMissingEdges \leftarrow 0$;
    for each unique pair of vertices $v_1, v_2$ in $C$ do
        if $A_G(v_1, v_2) = 0$ then
            $numMissingEdges \leftarrow numMissingEdges + 1$;
        end
    end
    return $numMissingEdges$
Function Solver($G$, $k$, $\tau_0$, $\tau_f$, $\alpha$):
    $\tau \leftarrow \tau_0$
    $C \leftarrow$ the $k$ highest-order vertices of $G$;
    while $\tau > \tau_f$ do
                                        ▷ Pick two vertices and consider swapping them
        $v_1 \leftarrow$ a random vertex in $C$
        $v_2 \leftarrow$ a random vertex not in $C$
                                        ▷ Check if swapping the vertices improves the graph
        $c_1 \leftarrow$ NumMissingEdges($A_G, C$)
        $C \leftarrow C \cup \{v_2\} \setminus \{v_1\}$
        $c_2 \leftarrow$ NumMissingEdges($A_G, C$)
        $\Delta c \leftarrow c_2 - c_1$
        $i \leftarrow 0$
        while $i < 8n$ and $\Delta c > 0$ do
            $C \leftarrow C \setminus \{v_2\}$
            $v_2 \leftarrow$ a random vertex not in $C$
            $C \leftarrow C \cup \{v_2\}$
            $c_2 \leftarrow$ NumMissingEdges(
```

3 Methods

3.1 Analyzing algorithm performance

We tested our algorithms on three different datasets, explored in Section 3.2. Both exact and approximation algorithms were tested for runtime, and approximation algorithms were additionally tested for accuracy, since they are not guaranteed to produce the correct solution.

To test for accuracy, there were two main metrics we considered. The first option is the average ratio of the size of the algorithm's solution to the size of the actual maximum clique. This gives a detailed view of how good the solutions it produces are, on average. The second, which we selected, is the percentage of the time the algorithm outputs the actual max clique size. In cases where our algorithms perform poorly, this metric is not as useful, and the first may be more appropriate. However, the approximation algorithms we implemented frequently found the correct max clique, making the second metric a good option.

On the graphs we tested on, we measured number of vertices and edge density to see how they affected algorithm performance. Edge density is the proportion of possible edges that are present in the graph:

$$\text{Edge Density} = \frac{|E|}{\frac{|V|(|V|-1)}{2}} = \frac{2|E|}{|V|(|V|-1)}. \tag{2}$$

3.2 Graph datasets

To get results on runtime and successes for our algorithms, we ran tests on three different types of graph datasets: DIMACS graphs, protein product graphs, and randomly-generated graphs.

The Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) is a collaboration between the academy and industry to explore theoretical graph theory problems [1]. They host yearly/semi-yearly challenges that focus on specific topics. We used a selection of benchmark datasets produced by the second challenge from 1992-3 that focused on NP-hard problems, including the Maximum Clique problem [8]. These datasets are often used as benchmarks in academic papers, and were particularly useful to us, as many of them have known solutions. We tested on 25 DIMACS graphs, listed in Table 1 in Results. For these graphs, we cut off runtimes after 2 hours.

In order to see how our algorithms performed on a real-world application of this problem, we tested on 10 protein product graphs created by Depolli et al. [4]. For a more in-depth explanation of how these graphs were created, see Section 3.3. The runtime cutoff for these tests was 4 hours.

Lastly, to gather data with specific metrics, we created datasets of randomly-generated graphs. To create one graph, we specified a number of nodes and a certain edge probability. Then, for each possible pair of nodes, an edge would be populated with that probability. To create a dataset, we made 100 graphs of various edge sizes with random edge probabilities. We originally tested on graphs of size 25, 50, 100, 200, and 400 to get a variety of sizes, but not getting so large that runtime cutoff would always be exceeded. We later tested on sizes 60, 75, and 85 to find the best graph size where we could see the curves in runtime without having runtimes that were either negligible or too long. We collected additional data about each graph, especially edge density. In order to gather a large amount of data, we made the runtime cutoff 10 minutes for these graphs.

3.3 Calculating Protein Structural Similarity

A paper by Depolli et al. [4] introduced a method for calculating protein structural similarity by finding the maximum clique of a protein product graph, described below. They implement and test max clique algorithms highly optimized for solving on these graphs, utilizing parallelization over multiple threads.

To find the similarity between two proteins, we first encode them as protein graphs. See Figure 5 for a visual representation of the following explanation. Each node has a spatial coordinate and represents the geometric center of an amino acid along the protein backbone. Nodes are grouped into 5 colors based on their physiochemical properties, which allows us to equate nodes with similar function. An edge exists between two nodes if they are less than $15\mathring{A}$ apart. These protein graphs represent an abstracted version of the protein's structure.

Once we have two protein graphs G_1, G_2, we create a product graph P, encoding structural similarity information, as follows. The product graph has nodes (v_1, v_2) for all $v_1 \in G_1, v_2 \in G_2$ where v_1 and v_2 have the same color. Two nodes $(v_1, v_2), (w_1, w_2)$ have an edge between them if $(v_1, w_1) \in E(G_1)$, $(v_2, w_2) \in E(G_2)$ and $|\text{distance}(v_1, w_1) - \text{distance}(v_2, w_2)| < 0.5\text{Å}$. Thus, a pair of connected nodes represent a structurally matching pair of nodes in G_1 and G_2, so a clique on the product graph is a matching region between G_1 and G_2. The paper uses the solution of the max clique problem on this graph to determine protein similarity.

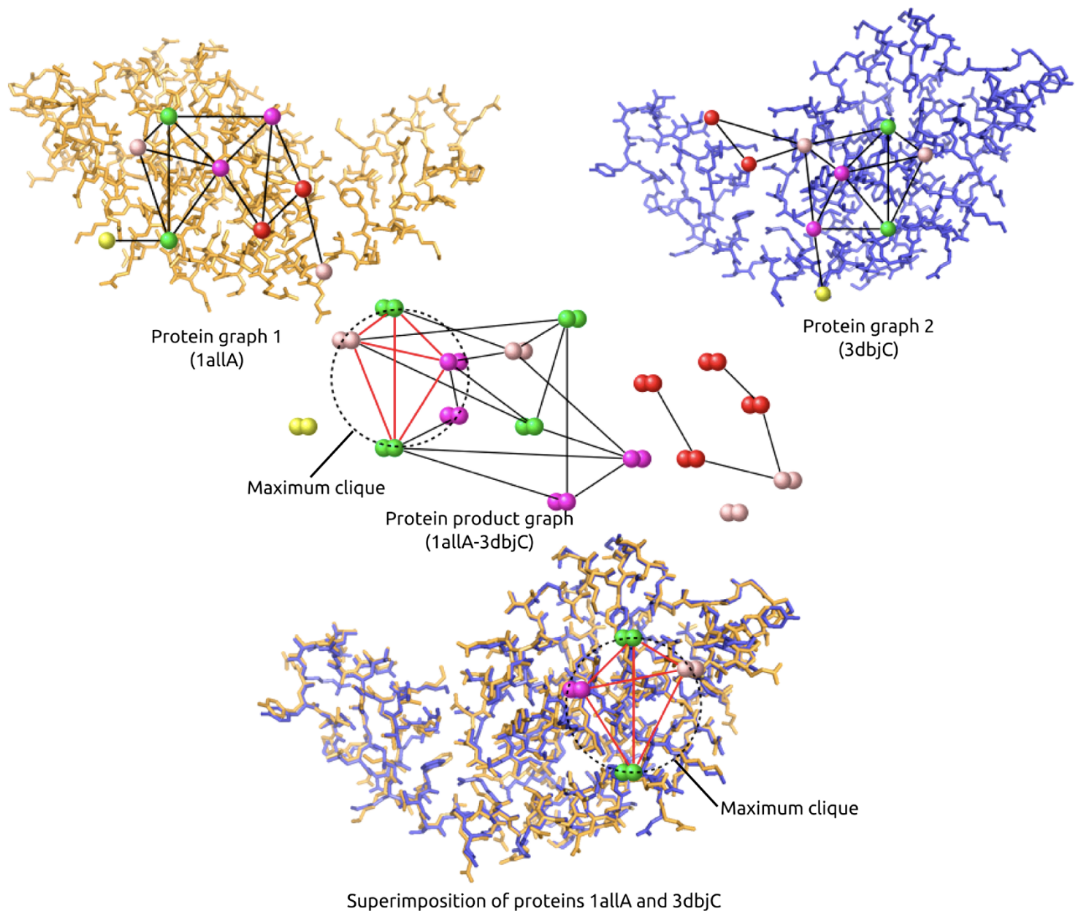

Figure 5: Protein structural comparison. Proteins are first converted to protein graphs (top). For clarity, only a section of each protein graph is shown. Vertices are colored according to their physicochemical properties, i.e., acceptor (red), donor (green), $\pi - \pi$ stacking (pink), and aliphatic (yellow). A protein product graph is constructed from both protein graphs (center). A maximum clique of four vertices connected with red edges indicates the similarity between proteins. Finally, the two compared proteins are superimposed (bottom) according to the best alignment of vertices represented by the maximum clique. (Figure and figure caption from Depolli et al. Figure 2 [4])

4 Results

Graph	$\|V\|$	Density	Max Clique	B-K (s)	B&B (s)	Gen. Alg.	Sim. An.
c-fat200-1	200	0.077	12	0.0019	0.004	100%	100%
c-fat500-1	500	0.036	14	0.0085	0.0096	100%	100%
johnson16-2-4	120	0.765	16	16	4.1	100%	100%
johnson32-2-41	496	0.879	16	Timed out	Timed out	100%	100%
keller4	171	0.649	11	15	5.5	100%	100%
keller5	776	0.752	27	Timed out	Timed out	20%	10%
keller6	3361	0.818	55	Timed out	Timed out	0%	0%
hamming10-2	1024	0.990	512	Timed out	320	100%	90%
hamming8-2	256	0.969	128	Timed out	1.7	100%	100%
san200_0.7_1	200	0.700	30	Timed out	7.0	100%	0%
san400_0.5_1	400	0.500	13	Timed out	7.6	0%	0%
san400_0.9_1	400	0.900	100	Timed out	Timed out	100%	0
sanr200_0.7	200	0.697	18	340	160	20%	90%
sanr400_0.5	400	0.501	13	140	200	10%	0%
san1000_0.5	1000	0.502	15	Timed out	1300	0%	0%
brock200_1	200	0.745	21	2100	740	0%	20%
brock400_1	400	0.748	27	Timed out	Timed out	0%	0%
brock800_1	800	0.649	23	Timed out	Timed out	0%	0%
p_hat300-1	300	0.244	8	0.21	1.2	80%	100%
p_hat500-1	500	0.253	9	2.3	13	100%	100%
p_hat700-1	700	0.249	11	12	53	10%	50%
p_hat1000-1	1000	0.245	10	60	290	100%	90%
p_hat1500-1	1500	0.253	12	690	3600	0%	0%
MANN_a27	378	0.990	126	Timed out	Timed out	0%	0%
MANN_a45	1075	0.996	345	Timed out	Timed out	0%	0%

Table 1: Teset results on DIMACS Graphs for Bron-Kerbosch, branch and bound, genetic algorithm, and simulated annealing. The exact algorithms (B-K and B&B) were run once each. The approximate algorithms show the percentage of time they found the max clique over 10 trials. The time cutoff was 2 hours (7,200 s).

Graph	$\lvert V \rvert$	Density	M.C.	B-K (s)	B&B (s)	GA (s)	GA %	SA (s)	SA %
1allA_3dbjC_41	451	0.97	356	–	–	1.1	0%	310	0%
1f82A_1zb7A_5	655	0.97	500	–	–	1.3	100%	54	100%
1KZKA_3KT2A_78	271	0.99	247	0.68	8.6	0.22	100%	1.8	100%
2FDVC_1PO5A_83	750	0.96	556	16	4.1	1.9	100%	81	100%
2UV8I_2J6IA_13107	200	0.86	69	–	–	0.81	100%	16	90%
2W00B_3H1TA_10858	346	0.91	143	–	–	1.4	90%	76	90%
2W4JA_2A2AD_0	563	0.98	447	–	–	1.3	100%	22	100%
3HRZA_2HR0A_476	905	0.94	563	–	–	3.9	100%	29	90%
3P0KA_3GWLB_0	138	0.94	89	6.0	20	0.36	100%	1.4	100%
3ZY0D_3ZY1A_110	61	0.98	52	0.0015	0.041	0.12	100%	0.027	100%

Table 2: Test results on 10 protein product graphs. M.C. stands for maximum clique. B-K and B&B are the runtimes of Bron Kerbosch and Branch and Bound on one trial, respectively. A '–' means a trial timed out, with a time cutoff of 4 hours (14,400 s). GA (s) and SA (s) are the average runtimes over 10 trials of genetic algorithm and simulating annealing, respectively. The % columns represent the percent of runs over 10 runs that the maximum clique was successfully found by genetic and simulating annealing.

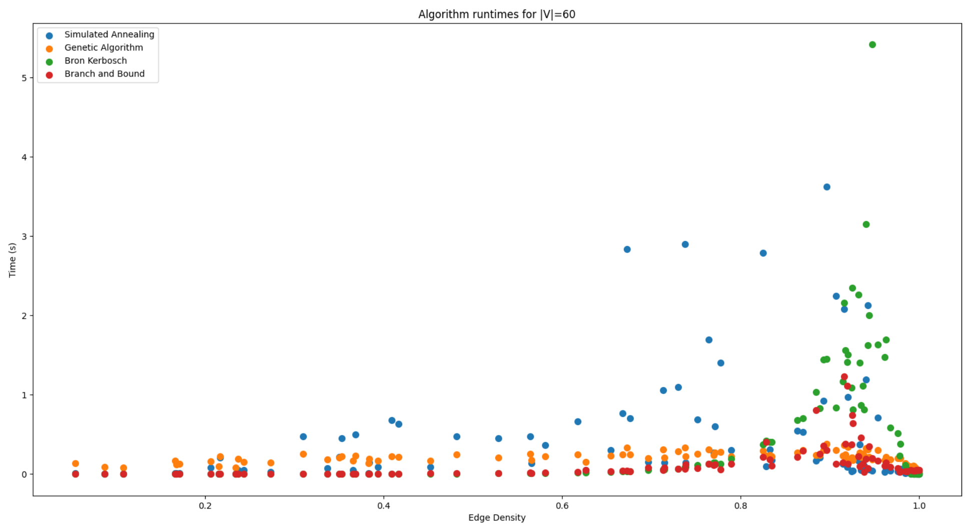

Figure 6: Runtimes of all algorithms on random graphs with $|V| = 60$. Both Bron-Kerbosch and Branch and Bound have runtimes that increase with high edge density but then decrease at very high density nears 100%.

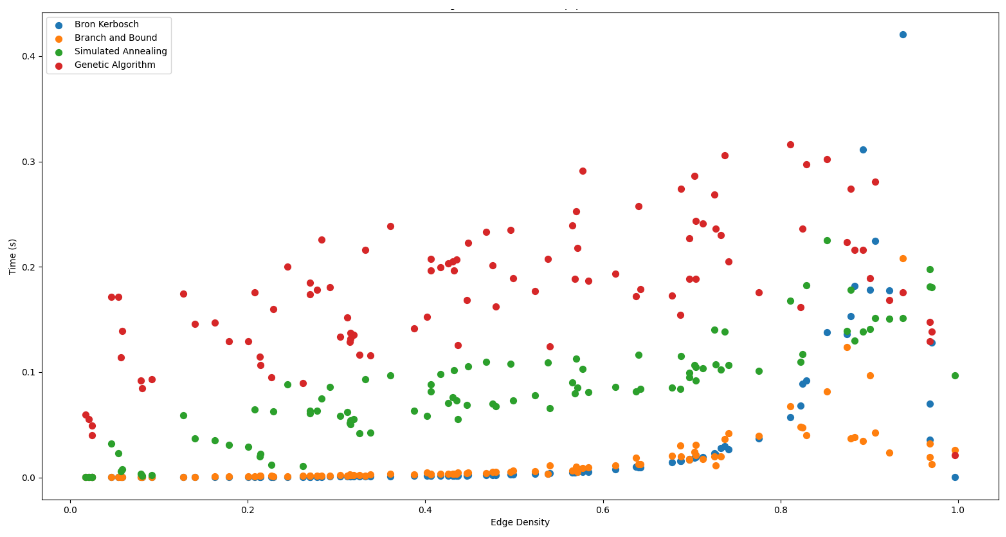

Figure 7: Runtimes of all algorithms on random graphs with $|V| = 50$. On these small graphs, the random algorithms tend to be slower due to high overhead, but exhibit an approximately linear runtime, while the exact algorithms are exponential.

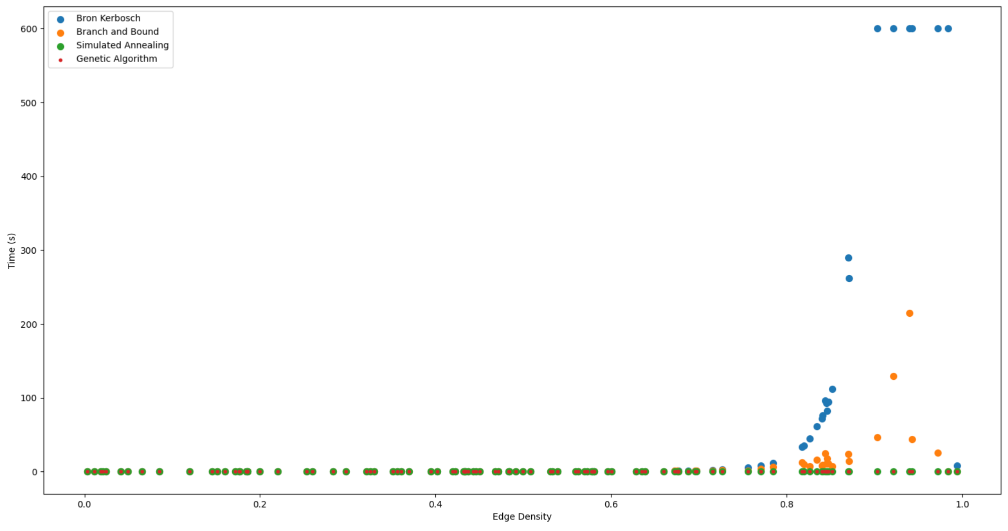

Figure 8: Runtimes of all algorithms on random graphs with $|V| = 100$. On graphs of this size, the random algorithms have negligible runtimes while the exact algorithms are exponential.

5 Discussion

5.1 Overview

Each algorithm we implemented performed similarly to results described in their respective papers. Since we wrote each algorithm in Python, our runtimes were a few times slower than each paper's implementation, written in faster languages like C and C++. Still, our algorithm's runtimes scaled with graph size at the same rate as described in each paper.

As expected, our exact algorithms had exponential time complexity, and our approximation algorithms had near-linear time complexity on the graphs we tested.

5.2 Algorithm Comparisons

5.2.1 Exact Algorithms

The Bron-Kerbosch algorithm is heavily dependent on both number of nodes and edge density because the recursive call happens inside a for-loop. We expect Bron-Kerbosch to perform better when there are fewer nodes and/or the graph is sparse. This is because if a graph is sparser, the maximal cliques will be smaller, and thus the recursion depth will be lower. If there is a low recursion depth on average, the number of iterations (so number of nodes) can be much higher. Additionally, if there are not many nodes (and so not many for-loop iterations), the recursion depth is still low even if it is high-edge density, since the depth is capped by the number of existing nodes.

Because the branch and bound algorithm uses recursion to explore different combinations of vertices, it gains a runtime advantage when it can minimize the amount of recursion it has to do by pruning, preventing it from spending time exploring nonproductive branches. Pruning is most effective when a relatively high lower bound (compared to the rest of the cliques in the graph) is set early on so that it can prune most branches in the graph. The algorithm is least effective when there are multiple cliques containing the same, high-colored node because then the algorithm will recurse into all of those branches instead of pruning them.

One interesting feature of the runtime plots is that as edge density increases, Bron-Kerbosch and Branch and Bound runtimes initially increase exponentially, but then decrease for nearly fully connected graphs. Since Bron-Kerbosch enumerates all maximal cliques, this is likely because very high density results in a small number of maximal cliques. More specifically, because the nodes are highly inter-connected, the pivot node at each stage will have few non-neighbors, meaning that the for-loop will have few iterations. For Branch and Bound, the cause is likely that the algorithm is more likely to find a large clique quickly, allowing it to prune branches very aggressively with bounding.

Taking a closer look, we expected Branch and Bound to be slower than Bron-Kerbosch because Branch and Bound has a worst-case time complexity of $O(2^{|V|})$, while Bron-Kerbosch is only $O(3^{\frac{|V|}{3}})$. However, in practice, Branch and Bound performed better than Bron-Kerbosch on very large or dense graphs (see Figure 8 or Table 1.) This is likely because Branch and Bound utilizes pruning during its running, whereas Bron-Kerbosch will always find all maximal cliques. Thus it will not end more quickly if a large clique is found early on.

We expected there to be a greater difference between the runtimes of Bron-Kerbosch and Branch and Bound. Branch and Bound does tend to do run faster than Bron-Kerbosch due to its pruning, but it is not cutting the time down as much as we expected it to on larger graphs. This happened because we overestimated the effect that pruning would have relative to Bron-Kerbosch.

5.2.2 Approximation Algorithms

Both simulated annealing and the genetic algorithm performed quite well. In nearly all cases where the approximation algorithms did not find the maximum clique, the exact algorithms either timed out or took a very long time to find the clique. It is possible that the approximation algorithms also could have found theses cliques on larger graphs if they were given more time or modified slightly to account for the more challenging problem.

The genetic algorithm slightly outperformed simulated annealing on accuracy, though not without exceptions. However, it also had a runtime that was orders of magnitude smaller than simulated annealing on large graphs, in addition to not needing to take clique size as a parameter.

The runtime of simulated annealing is somewhat hard to analyze conventionally. In the case where it fails to find a clique, it will always take a constant number of iterations before halting, according to τ_0, τ_f, and α. Each iteration has an upper bound of $O(|V| \cdot k)$, but in practice this is a very poor bound. Unfortunately, tightening it depends heavily on the local geometry of the graph. If the algorithm is in a state where it can swap on vertex in C with another without changing the objective function, but not improve the graph with a single swap, it is likely to repeatedly switch the vertices back and forth. These switches can happen quickly and cause a large number of iterations to pass quickly, decreasing the algorithms runtime substantially. With only a slightly different geometry with a single local minima instead of a pair, runtime will dramatically increase.

The runtime of the genetic algorithm is less complex and depends less on the graph. It has a fixed number of generations in which it can fail to find an improvement before terminating. Each generation requires several sub-algorithms to be run. Most have linear complexity, except for the local optimization step, where a subgraph produced from crossover is optimized to become a clique. Local optimization consists of a reduction and expansion step. Given a subgraph X, both reduction and expansion complexity are bounded by $|X|^2$. However, these bounds are extremely loose in practice. The worst case for reduction is that we need to remove every vertex but one to become a clique, and this has a complexity of $|X|!$. Similarly, the worst case for expansion is that we start from a clique of size 1, meaning we have to iterate for each other vertex in the graph, giving $|X|!$ complexity. These are both equivalent to $|X|^2$ complexity.

It is important to note that the key operations in the reduction and expansion step, namely degree within X and checking if a node's neighbors contain X, respectively, utilize highly efficient bitwise operations. While this does not reduce complexity, it does likely contribute to the generally fast runtime of the algorithm.

Given the best and worst case of the most complex step, and the linear trend of runtime on the graphs we tested, it is likely that the practical time complexity of the genetic algorithm is somewhere between linear and quadratic. Given the high variance of runtime for different graphs of the same size, this is hard to determine. However, further testing on much larger graphs could reveal a clearer pattern.

6 Conclusion

In conclusion, graph characteristics such as size and edge density heavily impact the runtimes of these algorithms and should be taken into consideration when choosing the best algorithm to find the MaxClique on a dataset. On small graphs, one of the exact algorithms should be chosen to ensure accuracy and get a faster runtime. On very large graphs, it is likely that an approximation algorithm will be better. Bron-Kerbosch can execute in a short amount of time on very sparse, large graphs. However, real-world datasets like protein graphs are extremely dense and thus are more suited to approximations. Generally, the genetic algorithm was both faster and more accurate than simulated annealing, so would probably be the better choice between the two on most data.

Limitations and future work. Time was the biggest limiting factor for gathering our results. Especially in terms of comparing the exact algorithms, getting meaningful data on large datasets was prohibitively time-consuming.

One goal for future work would be to conduct more extensive testing, especially on larger graphs and graphs of various structures. We would like to quantify the types of graphs that each algorithm performs best on. This would be especially interesting on the approximation algorithms to see which types of graph structures or clique arrangements cause them to perform very poorly.

Another avenue for future work is to compare these algorithms on different types of real-world datasets. The protein product graphs were extremely dense graphs. Other types of datasets might produce datasets with different sizes, edge densities, and structures that would impact the algorithms' runtimes.

References

[1] Dimacs. `http://dimacs.rutgers.edu/`. Accessed: 2025-03-12.

[2] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[3] Frédéric Cazals and Chinmay Karande. A note on the problem of reporting maximal cliques. *Theoretical computer science*, 407(1-3):564–568, 2008.

[4] Matjaz Depolli, Janez Konc, Kati Rozman, Roman Trobec, and Dusanka Janezic. Exact parallel maximum clique algorithm for general and protein graphs. *Journal of chemical information and modeling*, 53(9):2217–2228, 2013.

[5] Xiutang Geng, Jin Xu, Jianhua Xiao, and Linqiang Pan. A simple simulated annealing algorithm for the maximum clique problem. *Information Sciences*, 177(22):5064–5071, 2007.

[6] Bo Huang. Finding maximum clique with a genetic algorithm. *Penn. State Harrisburg Master's Thesis in Computer Science*, 2002.

[7] Janez Konc and Dušanka Janezic. An improved branch and bound algorithm for the maximum clique problem. *proteins*, 4(5):590–596, 2007.

[8] Franco Mascia. Dimacs benchmark set. `https://iridia.ulb.ac.be/~fmascia/maximum_clique/DIMACS-benchmark`. Accessed: 2025-03-12.

[9] Ryan A Rossi, David F Gleich, Assefaw H Gebremedhin, and Md Mostofa Ali Patwary. What if clique were fast? maximum cliques in information networks and strong components in temporal networks. *arXiv preprint arXiv:1210.5802*, 2012.