Delft University of Technology

Detecting Maximal Cliques Using Differential Equations

with applications in molecular docking

Author
Sofia Sarigiannidi

Supervisor
Yves van Gennip

Summary for General Audiences

This report explores two methods for finding a specific set of points in a graph. A graph is a set of points, called nodes, that can be connected by lines called edges. Each point has a weight, and we want to find a set where all the points are directly connected to each other while their total weight is as high as possible. The time it takes to find the exact solution to this problem grows as the amount of points in the graph grows. This means that, for large graphs, finding the specified set is (currently) practically impossible. Therefore, in this report, we explore methods that approximate the solutions to this problem.

The methods we develop are extensions of the Lokta–Volterra method introduced in [15]. This method, as well as its extensions, is based on a model that describes how animal populations compete with each other in nature (called the Lotka–Volterra system). Imagining each point in the graph as an animal population, the points that 'survive' this competition are the ones chosen by the methods. Unlike its extensions, the Lotka–Volterra method attempts to detect the largest set of connected points in a graph, instead of the one with the highest total weight. When the weight of all the points in a graph is zero, the methods introduced in this report are equivalent to the Lotka–Volterra method.

The two methods called the carrying capacity method and the competition method, work slightly differently based on how the weights of the points affect the competition between animal populations. In the carrying capacity method, the weight of a point determines how many of that type of animal can be sustained by their habitat. In the competition method, the weight determines how strongly that type of animal competes with (or 'kills off') others. The idea behind both these methods is that points with a higher weight will have a survival advantage.

The two new methods, the Lotka–Volterra method and an algorithm that repeatedly picks the point with the highest weight as long as all the points are still directly connected (called the greedy algorithm), are applied to 89 different graphs. These graphs have a specific structure related to the problem of molecular docking. This is the problem of predicting how two molecules will interact with each other and is equivalent to the problem of finding the

set of directly connected points with the highest total weight for appropriately chosen graphs. The choice of these molecular docking graphs is based on the structure of the interacting molecules and is described in [8].

After this application, we conclude that in most cases, the greedy algorithm returns the set of points with the highest weight and does so much faster than the other methods. However, we also see that when the greedy algorithm does not return the set with the highest weight, it typically returns the set with the lowest weight. In contrast, the competition method returns almost always either the highest or second highest weighted set. The carrying capacity method performs relatively poorly, only outperforming (by a little) the Lotka–Volterra method. Both the competition and carrying capacity methods have the longest processing time.

Summary for Peers

This report explores two heuristic methods for approximating the solution to the maximum-weighted clique problem. This problem involves identifying a set of nodes in a simple, undirected graph with weighted nodes and no loops, such that all nodes in the set are adjacent to each other (clique) and have the maximum total weight (maximum-weighted).

The methods we develop are extensions of the Lotka–Volterra method introduced in [15]. This method, as well as its extensions, is based on the dynamics of the generalized Lotka–Volterra model. Specifically, they associate each component of the solution of a mutually antagonistic Lotka–Volterra system with nodes in a given graph. Viewing each component of a solution as an animal population, the populations that 'survive' the antagonistic interactions correspond to the nodes selected by the methods. Unlike its extensions, the Lotka–Volterra method is developed to approximate the solutions to the maximum clique problem. The extensions, introduce weights to the nodes and, when the weight of all the nodes is zero, are equivalent to the Lotka–Volterra method.

The methods, called the carrying capacity method and competition method, differ in how they incorporate node weights into the dynamics of the animal populations. The carrying capacity method uses the weight of each node as a carrying capacity for the corresponding animal population. In the competition method, the weight of a node functions as a competitive coefficient for the corresponding animal population.

The two new methods, the Lotka–Volterra method and an algorithm that iteratively picks the node with the highest weight as long as the output is still a clique (the greedy algorithm) are applied to 89 different graphs. These graphs have a specific structure related to the problem of molecular docking. This is the optimization problem of predicting how two molecules will interact with each other and is equivalent to the problem of finding the maximum-weighted clique for appropriately chosen graphs. The choice of these molecular docking graphs is based on the structure of the interacting molecules and is described in [8].

After this application, we conclude that in most cases, the greedy algorithm returns the cliques with the highest weight and does so much faster than the other methods. However, we also see that when the greedy algorithm does not return the clique with the highest weight, it typically returns the clique with the lowest weight. In contrast, the competition method returns almost always either the highest or second highest weighted clique. The carrying capacity method performs relatively poorly, only outperforming (by a little) the Lotka–Volterra method. Both the competition and carrying capacity methods have the longest processing time.

Contents

Summary for General Audiences 1

Summary for Peers 3

1 Introduction 7

2 Detecting 'Heavy' Maximal Independent Sets 10
- 2.1 Maximum-Weighted Independent Sets 10
- 2.2 The Lotka–Volterra System 11
- 2.3 The Carrying Capacity Method 12
- 2.4 The Competition Method 13

3 Mathematical Analysis of the Methods 15
- 3.1 Mathematical Analysis of the Carrying Capacity Method 15
- 3.2 Mathematical Analysis of the Competition Method 20

4 Analysing the Methods 22
- 4.1 Analysis of the Carrying Capacity Method 22
- 4.2 Analysis of the Competition Method 25

5 Detecting 'Heavy' Maximal Cliques 28
- 5.1 Maximum-Weighted Cliques 28
- 5.2 Cliques and Independent Sets 28

6 Application to Molecular Docking 30
- 6.1 Molecular Docking 30
- 6.2 The Dataset 31
- 6.3 Applying the Methods 31
- 6.4 Results 32
  - 6.4.1 Weight-Based Performance Comparison 33
  - 6.4.2 Weight-Based Performance Rankings 35
  - 6.4.3 Time-Based Performance Comparison 36
  - 6.4.4 Discussion and Conclusion 38

7 Discussion 39

A Supplementary Proofs 41

B Code for the Carrying Capacity and Competition Method 46

B.1 Helpful Functions 46
B.2 The Methods 48
B.3 Code for Figures in Chapter 4 51
B.4 The greedy Algorithm 55
B.5 Application on Molecular Docking 55

C Tables 60

Bibliography 64

Chapter 1

Introduction

There are many important problems in Graph Theory for which we still have no efficient algorithms. One category for such problems are those which belong to the class NP (introduced in [1]). Problems in the class NP cannot (yet) be solved efficiently but their solution can be efficiently verified. By performing a task efficiently we mean here that it is done by an algorithm whose runtime grows (at most) polynomially with the size of the input. Some examples of NP problems include the travelling salesman problem [2] and those presented in [3].

A known NP problem of importance is the maximum-weighted clique problem. Given an undirected graph with weighted nodes, the solution to this problem is a set of nodes with maximum total weight (maximum-weighted), such that all nodes in the set are adjacent to each other (clique). A solution to the maximum-weighted clique problem for a given graph is also a solution to the maximum-weighted independent set problem for the complement of that graph. The solution to the maximum-weighted independent set problem is a maximum-weighted set of nodes, such that all nodes in that set are not adjacent to each other (independent).

Detecting maximum-weighted cliques and its complementary problem of detecting maximum-weighted independent sets, have many applications such as those mentioned in [4]. In coding theory, finding the largest code invariant under a given permutation group is equivalent to finding the maximum-weighted clique in a suitably constructed graph [5]. In operations research, a classical single-machine scheduling problem can be reduced to a version of the maximum-weighted clique problem for a job conflict graph [6]. In biochemistry, large-scale changes in the sequences of genomes can be modelled in terms of the maximum-weighted independent set problem in a graph whose nodes represent genome fragments and whose edges represent pairs of overlapping fragments, the weight of the nodes characterising the strength of the local similarity [7].

Another notable application in biochemistry is that of molecular docking [8], which we will delve deeper into.

Molecular docking is the problem of determining the optimal interaction between a ligand (small molecule) and a target receptor. By identifying which points in the molecules are significant to the binding process, a (node-weighted) 'binding interaction' graph can be made which represents their possible interactions. Determining the maximum-weighted clique of the 'binding interaction' graph is a solution to the problem.

There are many different algorithms to exactly solve the maximum-weighted clique problem, these include the branch-and-bound algorithms presented in [9], [10], [11]. Branch-and-bound is a well-known technique that iteratively divides the search space into subsets (branches) in a tree structure. It uses upper and lower bounds to prune (remove) non-promising branches, thereby reducing the number of subsets that need to be explored. The algorithm introduced in [9] employs a branch-and-bound approach to explore potential cliques in a graph while dynamically updating upper bounds to prune the search space and backtracking to navigate through different branches, ultimately finding the maximum-weight clique. In [10], the branching is guided by a weighted colour heuristic. This heuristic works by first assigning colours to the nodes such that no adjacent nodes share the same colour. Then, the sum of the maximum weights of each colour class becomes an upper bound used to remove branches from the search space. Finally, the algorithm introduced in [11] is a parallel branch-and-bound algorithm. This method examines potential cliques by branching on candidate nodes and calculating upper bounds to remove non-promising branches whose weights cannot exceed the current best weight. Each of the parallel processors independently handles different branches, the results are combined to find the clique with the highest total weight.

Since the maximum-weighted clique problem is in the class NP, all existing algorithms that solve it exactly do so very slowly for large graphs. Thus, many (fast) algorithms have been developed that approximate its solutions, for example, those presented in [12],[13],[14]. The algorithm introduced in [12], approximates the solution of the maximum-weighted clique in a graph by detecting independent sets with high weight in the complement of that graph. Starting with an independent set, the algorithm iteratively identifies nodes outside of the current set that, when added to it, increase the total weight of the set while still remaining independent. This process is repeated until no further improvements can be made. The heuristic introduced in [13] approximates the maximum-weight clique by utilising replicator dynamics. Initially, all nodes are assigned a probability of being included in the solution. The method iteratively updates these probabilities based on their fiting, determined by their weights and adjacency relationships. The process converges to a set of nodes with high probabilities, providing an approximation of the maximum-weight clique. The hybrid evolutionary algorithm of [14], models cliques as chromosomes

and explores the solution space through crossover, mutation operations, and selection. At each step, local heuristics (repair and replace) are utilized to extend the winning chromosomes into weighted cliques.

In this report, we introduce two new algorithms for approximating the solutions to the maximum-weighted independent set problem. We demonstrate that these methods consistently return maximal independent sets and identify potential biases they might have. Finally, by taking the complement of relevant graphs, we apply these methods to molecular docking. In this application we also compare the methods to the Lotka–Volterra method of [15] and a version of the greedy algorithm.

The new algorithms are generalised versions of the Lotka–Volterra method introduced [15]. This heuristic method was originally developed for the maximum (in number of nodes) independent problem. It is based on the ordinary differential equations known as the Lotka–Volterra system and always returns a maximal independent set. This method works by having each node in a graph correspond to a component of the fixed point of the Lotka–Volterra system. The nodes that correspond to a non-zero component are the output of the method. The two generalised versions incorporate the weight of each node in the graph as either a carrying capacity or a competition coefficient in the Lotka–Volterra system. In the molecular docking application, the Lotka–Volterra method is applied on the complement of each relevant graph.

Finally, the version of the greedy algorithm applied to molecular docking works by iteratively picking the node with the highest weight which is connected to all previously chosen nodes. This algorithm approximates the solution to the maximum-weighted clique problem and always returns maximal cliques.

Structure Overview

- Chapter 2 introduces the maximum-weighted independent set problem and two heuristic methods to approximate its solutions.
- Chapter 3 demonstrates that both heuristic methods return maximal independent sets.
- Chapter 4 discusses observations made when applying the methods.
- Chapter 5 introduces the maximum weighted clique problem and explains its connection to the maximum weighted independent set problem.
- Chapter 6 presents the results of applying the methods to molecular docking.
- Chapter 7 includes a discussion and conclusion to this report.

Chapter 2

Detecting 'Heavy' Maximal Independent Sets

In this chapter, we present two methods that approximate the solutions to the maximum-weighted independent set problem. These methods build upon the technique outlined in [15], extending its application to graphs whose nodes are weighted.

First, in Section 2.1 we define the maximum-weighted independent set problem. In Section 2.2 we introduce the system upon which the aforementioned methods are based. Finally, in Sections 2.3 and 2.4, we present each method individually and explain how is approximates the solutions of the maximum-weighted independent set problem.

2.1 Maximum-Weighted Independent Sets

In this section, we define the maximum-weighted independent set problem and outline the context in which the methods described in Sections 2.3 and 2.4 can be applied. We will be assuming the following for the rest of this chapter.

Consider a simple, connected undirected graph $G = (E, V)$ with n nodes and no loops. Here, E is the set of edges and V is the set of nodes of the graph (thus, $n = |V|$). Further, assume that each node $v_i \in V$ has a corresponding weight $w_i \geq 1$.

Now, define the vector $\boldsymbol{w}$, where for all $i \in [n] = \{1, 2, ..., n\}$ the entry w_i is the weight of node $v_i \in V$. Let W be the diagonal matrix with entries corresponding to the entries of the vector $\boldsymbol{w}$ (i.e. $W = \text{diag}(\boldsymbol{w})$). Furthermore, let A be the adjacency matrix corresponding to graph G, where $A_{ij} = 1$ if $\{v_i, v_j\} \in E$, and $A_{ij} = 0$ otherwise.

Given the above assumptions we define the following:

Definition 1. *A set S is an independent set in the graph $G = (E, V)$ if and only if $\forall u, v \in S \subset V : \{u, v\} \notin E$.*

Definition 2. *A set S is a maximal independent set in the graph $G = (E, V)$ if and only if S is independent in G and $\forall v \in V \backslash S : \exists u \in S$ such that $\{u, v\} \in E$.*

Definition 3. *A set S is a maximum-weighted independent set in the graph $G = (E, V)$ if and only if S is a maximal independent set in G and*

$$\sum_{v_k \in S} w_k = \max \Big\{ \sum_{v_k \in B} w_k \text{ with } B \text{ a maximal independent set} \Big\}.$$

Finally, we will call the sum of weights corresponding to the nodes of a set its weight. Also, we shall refer to sets with a relatively high weight as 'heavy'. And so, 'heavy' maximal independent sets approximate the weight of the maximum-weighted independent set. Note that 'heavy' is not a precise term and is only used here to differentiate and compare between sets on the basis of their weight.

2.2 The Lotka–Volterra System

The techniques outlined in the subsequent sections (Sections 2.3 and 2.4) for identifying 'heavy' maximal independent sets are related to the dynamics of the Lotka–Volterra system. Specifically, they are related to the trajectories of the multi-dimensional Lotka–Volterra system described by:

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{x} \circ (\mathbf{1} - M\boldsymbol{x}) \tag{2.1}$$

where $\circ$ denotes component-wise multiplication and $\mathbf{1}$ is the unit vector.

Note, for the methods described in Sections 2.3 and 2.4, matrix M and vector $\boldsymbol{x}$ have non-negative entries. For M this is by definition (as seen in Sections 2.3 and 2.4) and for $\boldsymbol{x}$ this is a consequence of the chosen initial conditions as proven in Lemma 2.

When $\boldsymbol{x}$ has non-negative entries, as is the case in this report, it can (and was originally developed to) represent animal populations [17], with M being the interaction matrix among these species. Since, in this case, all entries of M are non-negative, the model (2.1) represents a mutually antagonistic system, where interacting animal populations compete for their survival (e.g. through competing for resources).

The Jacobian at $\boldsymbol{x}$ of the system (2.1) is given by:

$$J(\boldsymbol{x}) = [\mathbb{I} - \text{diag}(\boldsymbol{x})M - \text{diag}(M\boldsymbol{x})] \tag{2.2}$$

where $\mathbb{I}$ is the identity matrix.
Equation (2.2) follows from the fact that for arbitrary $i, j \in [n]$

$$\begin{aligned} J(\boldsymbol{x})_{ij} &= \frac{d}{dx_j}\frac{dx_i}{dt} = \frac{d}{dx_j}\Big(x_i\big(1-(M\boldsymbol{x})_i\big)\Big) \\ &= \frac{d}{dx_j}\Big(x_i - x_i(M\boldsymbol{x})_i\Big) \\ &= \mathbb{1}_{\{i=j\}} - x_i M_{ij} - \mathbb{1}_{\{i=j\}}(M\boldsymbol{x})_i \end{aligned}$$

where $\mathbb{1}_{\{i=j\}}$ is equal to 1 when $i = j$ and 0 otherwise.

The definition of the Jacobian is provided here for completeness of the system definition, but it is only used in Chapter 3.

2.3 The Carrying Capacity Method

The first method for identifying 'heavy' maximal independent sets that is presented we call the carrying capacity method. That is because the weight of each node functions as its carrying capacity in the Lotka–Volterra system.

The Carrying Capacity Method

> Given a simple, connected, undirected graph $G = (V, E)$ with n weighted nodes and no loops, define $\tilde{M} = \tau A + W^{-1}$ with A and W as described in Section 2.1 and $\tau > 1$.
> Let $\boldsymbol{x}(t)$ be the trajectory of the system (2.1) equipped with matrix $\tilde{M}$ and with initial condition at $\boldsymbol{x}(0) = \boldsymbol{x_0}$ such that $\forall i \in [n] : x_{0i} \in (0, w_i)$. Then the set $\{v_i \in V : x_i^* = w_i\}$ where $\boldsymbol{x}^* = \lim_{t\to\infty} \boldsymbol{x}(t)$, is an approximation to the maximum-weighted independent set G.

The existence of $\boldsymbol{x}^*$ and the fact that $\{v_i \in V : x_i^* = w_i\}$ is a maximal independent set are shown in Section 3.1.

As mentioned before, the carrying capacity method is based on the approach given in [15], with one key difference in the definition of the interaction matrix. In the original method the interaction matrix is defined as $\tau A + \mathbb{I}$ while in the the carrying capacity method, the interaction matrix is defined as $\tau A + W^{-1}$.

The purpose of this difference is the introduction of a new bias in the method, where instead of returning maximal independent sets with a relatively high number of nodes, it now returns 'heavy' maximal independent sets. To understand this difference, let us consider the component-wise Lotka–Volterra system (2.1) with interaction matrix $\tilde{M} = \tau A + W^{-1}$

$$\begin{aligned} \frac{dx_i}{dt} &= x_i\Big(1 - \big((\tau A + W^{-1})\boldsymbol{x}\big)_i\Big) \\ &= x_i\Big(1 - \frac{x_i}{w_i}\Big) - \tau x_i(A\boldsymbol{x})_i. \end{aligned} \tag{2.3}$$

Here we see that if we consider each component of $\boldsymbol{x}$ in (2.3) as an animal population, its corresponding weight functions as the carrying capacity of that population's habitat. The carrying capacity of a habitat is the maximum population size that can be sustained in that habitat. Therefore, a habitat with a high carrying capacity can sustain a larger population of a given animal population. Consequently, we hypothesize that a node with a higher weight is more likely to exhibit a trajectory that does not stabilize at zero. This is because, in nature, animal populations in habitats with high carrying capacities typically possess a survival advantage [16].

In contrast, for the original method of [15], each animal population would have the same carrying capacity, equal to one. This would imply that no animal population has a survival advantage based on its habitat. Instead, the driving force of the original model is the number of interactions a population has. The more antagonistic interactions a population has, the more likely it is that extinction will follow.

If our hypothesis is true, the maximal independent set generated by the carrying capacity method is expected to yield a 'heavy' maximal independent set. The extent of how 'heavy' this independent set is and the accuracy of the method in identifying the maximum-weighted independent set are investigated in Section 4.1.

2.4 The Competition Method

The second method for identifying 'heavy' maximal independent sets that is presented we call the competition method. That is because the weight of each node functions as a type of interactive competition coefficient in the Lotka–Volterra system.

The Competition Method

Given a simple, connected, undirected graph $G = (V, E)$ with n weighted nodes and no loops, define $\bar{M} = \tau AW + \mathbb{I}$ with A and W as described in Section 2.1, $\mathbb{I}$ the identity matrix and $\tau > 1$.
Let $\boldsymbol{x}(t)$ be the trajectory of the system (2.1) equipped with matrix $\bar{M}$ and with initial condition at $\boldsymbol{x}(0) = \boldsymbol{x_0}$ such that $\forall i \in [n] : x_{0i} \in (0, 1)$. Then the set $\{v_i \in V : x_i^* = w_i\}$ where $\boldsymbol{x}^* = \lim_{t\to\infty} \boldsymbol{x}(t)$, is the approximation to the maximum-weighted independent set G.

The existence of $\boldsymbol{x}^*$ and the fact that $\{v_i \in V : x_i^* = 1\}$ is a maximal independent set are shown in Section 3.2.

Similar to the carrying capacity method, the competition method differs from the method introduced in [15] in the definition of the interaction matrix. In this case, the difference is related to the influence of the adjacency matrix. Again, to

understand this difference, let us consider the component-wise Lotka–Volterra system 2.1 with interaction matrix $\bar{M} = \tau AW + \mathbb{I}$:

$$
\begin{aligned}
\frac{dx_i}{dt} &= x_i\Big(1 - \big((\tau AW + \mathbb{I})\boldsymbol{x}\big)_i\Big) \\
&= x_i(1 - x_i) - \tau x_i \sum_{j:A_{ij}\neq 0} w_j x_j. \qquad (2.4)
\end{aligned}
$$

If we now consider each component of $\boldsymbol{x}$ in (2.4) as an animal population, the competitive interaction coefficient between each population x_i and each population x_j it interacts with for $j \sim i$ is scaled by the weight of that animal population w_j.

The reasoning behind this method is that a population with 'heavy' neighbouring populations, that is neighbouring populations with a high interaction coefficients, are more prone to extinction. This leaves their neighbours with less competition and, thereby increasing their chances of survival. Thus, animal populations with high interaction coefficients are more likely to outcompete and reduce the populations they interact with, thereby increasing their own chances of survival.

In the original method of [15], each animal population had the same interaction coefficient (equal to τ). And so again, the driving force in the original model is the number of interactions a population has.

If the above hypothesis is true, the maximal independent set generated by the competition method is expected to yield a 'heavy' maximal independent set. The extent of how 'heavy' this independent set is, and the accuracy of the method in identifying the maximum-weighted independent set, will be investigated in Section 4.2.

Chapter 3

Mathematical Analysis of the Methods

In this chapter, we prove that the methods described in Chapter 2 consistently return maximal independent sets. This is covered in Section 3.1 for the carrying capacity method and in Section 3.2 for the competition method.

3.1 Mathematical Analysis of the Carrying Capacity Method

The carrying capacity method, introduced in Section 2.3, consistently returns a maximal independent set. This is a consequence of the following theorem.

Theorem 1. *Let $\tau > 1$ and $\tilde{M} = \tau A + W^{-1}$ with A and W as described in Section 2.1. Let $\boldsymbol{x}(t)$ be the trajectory of the system (2.1) equipped with matrix $\tilde{M}$ and with initial condition at $t = 0$ being $\boldsymbol{x_0}$ such that $\forall i \in [n] : x_{0i} \in (0, w_i)$. Then $\boldsymbol{x}^* := \lim_{t\to\infty} \boldsymbol{x}(t)$ exists, $\forall i \in [n] : x_i^* \in [0, w_i]$ and the set $\{v_i \in V : x_i^* = w_i\}$ is a maximal independent set.*

Proof. The proof of Theorem 1 follows from Lemmas 1, 2, 3, 4 and 5. □

For the rest of this section, assume that $\tilde{M} = \tau A + \mathrm{W}^{-1}$ with A and W as described in Section 2.1.

Lemma 1. *The solution to the initial value problem*

$$\begin{cases} \frac{d\boldsymbol{x}}{dt} = \boldsymbol{x} \circ (\mathbf{1} - M\boldsymbol{x}), \\ \boldsymbol{x}(0) = \boldsymbol{x_0} \end{cases}$$

exists and is unique.

Proof. Define the function $\boldsymbol{f} : \mathbb{R}^n \mapsto \mathbb{R}^n$ such that $\forall i \in [n]:$

$$f_i(\boldsymbol{x}) = x_i\Big(1 - \big(M\boldsymbol{x}\big)_i\Big) = x_i\Big(1 - \sum_{k=1}^{n} M_{ij}x_k\Big).$$

Let $i, j \in [n]$ be arbitrary. Then we have

$$\begin{aligned}\frac{\partial f_i}{\partial x_j} &= \mathbb{1}_{\{i=j\}} - \mathbb{1}_{\{i=j\}}\Big(\sum_{k=1}^{n} M_{ik}x_k\Big) - x_i\sum_{k=1}^{n}\mathbb{1}_{\{i=k\}}M_{ik}x_k \\ &= \mathbb{1}_{\{i=j\}} - \mathbb{1}_{\{i=j\}}\Big(\sum_{k=1}^{n} M_{ik}x_k\Big)_i - x_iM_{ii}x_i.\end{aligned}$$

Thus, it is clear (since it is the sum of continuous functions) that $\frac{\partial f_i}{\partial x_j}$ is continuous for all $i, j \in [n]$. This means that $\boldsymbol{f}$ is continuously differentiable. This implies (by [19]) that the initial value problem

$$\begin{cases} \frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}), \\ \boldsymbol{x}(0) = \boldsymbol{x_0} \end{cases}$$

has one and only one solution. $\square$

Lemma 2. *Let $\tau > 1$ and $\boldsymbol{x}$ be a solution of the system (2.1) equipped with matrix $\tilde{M}$ and with initial condition $\boldsymbol{x_0}$ such that $\forall i \in [n] : x_{0i} \in [0, w_i]$. Then, for all $t \geq 0$ it holds that $\forall i \in [n] : x_i(t) \in [0, w_i]$.*

Proof. Let $i \in [n]$ be arbitrary and assume that the conditions of Lemma 2 hold. If $x_i = 0$, then $\frac{dx_i}{dt} = 0$. Thus, for all $i \in [n]$, x_i cannot become negative for all $t \geq 0$, provided that $x_{0i} > 0$. If $x_i = w_i$, the following holds:

$$\begin{aligned}\frac{dx_i}{dt}\bigg|_{x_i=w_i} &= x_i[1 - (\tilde{M}\boldsymbol{x})_i]\bigg|_{x_i=w_i} = x_i[1 - \big((\tau A + W^{-1})\boldsymbol{x}\big)_i]\bigg|_{x_i=w_i} \\ &= x_i - \tau x_i\sum_{j\neq i} A_{ij}x_j - x_iW_{ii}^{-1}x_i\bigg|_{x_i=w_i} = x_i - \frac{1}{w_i}x_i^2 - \tau x_i\sum_{j\neq i} A_{ij}x_j\bigg|_{x_i=w_i} \\ &= w_i - \frac{w_i}{w_i}w_i - \tau w_i\sum_{j\neq i} A_{ij}x_j = -\tau w_i\sum_{j\neq i} A_{ij}x_j \leq 0.\end{aligned}$$

Thus, any coordinate of a trajectory cannot exceed its corresponding weight. Combining the above, we have that for all $i \in [n] : 0 \leq x_i \leq w_i$ provided that $x_{0i} \in [0, w_i]$. This completes the proof. $\square$

Lemma 3. *Let $\boldsymbol{x}$ be the solution of the system (2.1) equipped with matrix $\tilde{M}$ and with initial condition $\boldsymbol{x_0}$ such that $\forall i \in [n] : x_{0i} \in [0, w_i]$. Then $\boldsymbol{x}^* := \lim_{t\to\infty} \boldsymbol{x}(t)$ exists.*

Proof. Define the $(n+1)\times(n+1)$ matrix U such that $\forall i,j\in[n]$:

- $U_{ij}=\tilde{M}_{ij}-1$,
- $U_{(n+1)j}=U_{i(n+1)}=1$,
- $U_{(n+1)(n+1)}=0$.

Now let $\boldsymbol{y}$ be the solution of the replicator system (see Definition 8) on the simplex $\hat{S}_{(n+1)}=\{\boldsymbol{y}\in S_{(n+1)}:y_{(n+1)}>0\}$ where $S_{(n+1)}=\{\boldsymbol{y}\in\mathbb{R}^n:y_i>0 \text{ and } \sum_{i=1}y_i=1\}$ given by

$$\frac{dy_i}{dt}=-y_i\Big(\big(U\boldsymbol{y}\big)_i-\boldsymbol{y}^TU\boldsymbol{y}\Big). \tag{3.1}$$

Since the matrix $\tilde{M}$ is symmetric, so is the matrix U. Then, by Theorem 7 (found in Appendix A), $\boldsymbol{y}(t)$ converges to a fixed point $\boldsymbol{y}^{\boldsymbol{*}}$.

By Theorem 5 (found in Appendix A), the differentiable, invertible map given by $x_i=y_i/y_{n+1}$, maps the replicator system (3.1) onto the following system on $\{\boldsymbol{x}\in\mathbb{R}^n:\boldsymbol{x}>0\}$

$$\begin{aligned}\frac{dx_i}{dt}&=-x_i\Big(U_{i(n+1)}-U_{(n+1)(n+1)}+\sum_{j=1}^n\big(U_{ij}-U_{(n+1)j}\big)x_j\Big)\\&=-x_i\Big(-1-0+\sum_{j=1}^n\big(\tilde{M}_{ij}+1-1\big)x_j\Big)\\&=x_i\Big(1-\big(\tilde{M}\boldsymbol{x}\big)_i\Big).\end{aligned}$$

This is the Lotka–Volterra system (2.1) equipped with matrix $\tilde{M}$.

Now, let $\boldsymbol{x}$ be an arbitrary solution of the Lotka–Volterra system (2.1) equipped with matrix $\tilde{M}$. Then by the above, we have $\forall i\in[n]$:

$$\lim_{t\to\infty}x_i(t)=\lim_{t\to\infty}\frac{y_i}{y_{n+1}(t)}=\frac{y_i^*}{y_{n+1}^*}=:x_i^*.$$

Note that since $\boldsymbol{y}\in\hat{S}_n$ we have $y_{n+1}>0$.

And so, any solution of the Lotka–Volterra system (2.1) equipped with matrix $\tilde{M}$ converges to a fixed point. □

Lemma 4. *Let $\tau>1$ and $\boldsymbol{x}$ be the solution of system (2.1) equipped with matrix $\tilde{M}$ and initial condition $\boldsymbol{x}(0)=\boldsymbol{x_0}$. Then, there exists a set Ω with Lebesgue measure equal to zero, such that for all initial condition $\boldsymbol{x_0}\in\{(0,w_1)\times(0,w_2)\times \ldots\times(0,w_n)\}\backslash\Omega$, the limit $\boldsymbol{x}^*=\lim_{t\to\infty}\boldsymbol{x}(t)$ has entries x_i^* such that $x_i*=0$ or $x_i^*=w_i$*

Proof. Let $\tau > 1$ and $\boldsymbol{x}$ be the solution of system (2.1) equipped with matrix $\tilde{M}$ and initial condition $\boldsymbol{x}(0) = \boldsymbol{x_0}$. Let S denote the set of all stationary points of the system. Then, by Sard's Theorem [20], the Lebesgue measure of S is equal to zero.
Let $\boldsymbol{x}^* = \lim_{t\to\infty} \boldsymbol{x}(t)$ with $\boldsymbol{x}(0) \notin S$ be a stationary state of the system (we can do this by Lemma 3).
For the first part of the proof, suppose that $\forall i \in [n] : x_i^* \in (0, w_i)$. We will show that given this condition $\boldsymbol{x}^*$ is not stable.
Because x^* is a stationary point and $\forall i \in [n] : x_i^* \neq 0$, we have $\boldsymbol{x}^* = \tilde{M}^{-1}\mathbf{1}$. And so, using (2.2) the Jacobian of the system at $\boldsymbol{x}^*$ is given by:

$$J(\boldsymbol{x}^*) = [\mathbb{I} - \text{diag}(\boldsymbol{x})\tilde{M} - \text{diag}(\tilde{M}\boldsymbol{x})]\Big|_{\boldsymbol{x}=\boldsymbol{x}^*} = -\text{diag}(\boldsymbol{x}^*)\tilde{M}.$$

But then, since $\boldsymbol{x}^* > 0$, the matrix $\text{diag}(\boldsymbol{x}^*)^{1/2}$ exists. And so, since

$$\text{diag}(\boldsymbol{x}^*)\tilde{M} = \text{diag}(\boldsymbol{x}^*)^{1/2}\Big(\text{diag}(\boldsymbol{x}^*)^{1/2}\tilde{M}\text{diag}(\boldsymbol{x}^*)^{1/2}\Big)\text{diag}(\boldsymbol{x}^*)^{-1/2},$$

$J(\boldsymbol{x}^*)$ has the same eigenvalues as matrix

$$C(\boldsymbol{x}^*) = -\text{diag}(\boldsymbol{x}^*)^{1/2}\tilde{M}\text{diag}(\boldsymbol{x}^*)^{1/2}.$$

Since $C(\boldsymbol{x}^*)$ is a symmetric and real matrix, all its eigenvalues are real. And so the point $\boldsymbol{x}^*$ is unstable if $C(\boldsymbol{x}^*)$ contains at least one strictly positive eigenvalue in its spectrum. That is equivalent to $C(\boldsymbol{x}^*)$ not being negative semi-definite.
Since $C(\boldsymbol{x}^*)$ is symmetric, to show that it is not negative semi-definite, it is sufficient to show that one of its principal minors of even order has a negative determinant (or of odd order has a positive determinant)[21].
Now consider a pair $k, l \in [n]$ with $A_{kl} = 1$. Then,

$$\begin{aligned}
C(\boldsymbol{x}^*)_{kk} &= -\frac{1}{w_k}x_k^*,\\
C(\boldsymbol{x}^*)_{ll} &= -\frac{1}{w_l}x_l^*,\\
C(\boldsymbol{x}^*)_{kl} &= -\tau(x_k^*)^{1/2}(x_l^*)^{1/2},\\
C(\boldsymbol{x}^*)_{lk} &= -\tau(x_l^*)^{1/2}(x_k^*)^{1/2}.
\end{aligned}$$

The corresponding 2x2 principal minor is given by

$$C(\boldsymbol{x}^*)_{kk}C(\boldsymbol{x}^*)_{ll} - C(\boldsymbol{x}^*)_{kl}C(\boldsymbol{x}^*)_{lk} = \frac{1}{w_k}\frac{1}{w_l}x_k^*x_l^* - \tau^2 x_k^*x_l^* = x_k^*x_l^*\Big(\frac{1}{w_k}\frac{1}{w_l} - \tau^2\Big).$$

Since $\tau > 1$ and $w_i \geq 1$ for all $i \in [n]$: $C(\boldsymbol{x}^*)_{kk}C(\boldsymbol{x}^*)_{ll} - C(\boldsymbol{x}^*)_{kl}C(\boldsymbol{x}^*)_{lk} < 0$. Thus $C(\boldsymbol{x}^*)$ is not negative semi-definite. Hence, $J(\boldsymbol{x}^*)$ has at least one positive

eigenvalue. This means $\boldsymbol{x}^*$ is unstable. By applying the Stable Manifold Theorem [22], there exists a stable manifold T of dimension strictly less than n. And so, if $\boldsymbol{x}_0 \notin T$, $\boldsymbol{x}^*$ doesn't converge to a steady state.
Now suppose that $\forall i \in [n] : x_i^* \in [0, w_i)$ and $\exists j \in [n] : x_j^* \in (0, w_j)$. Let $I = \{i \in [n] : x_i^* = 0\}$. By reordering the nodes, the Jacobian can (for any steady state $\boldsymbol{x}^*$) be rewritten as:

$$J(\boldsymbol{x^*}) = \begin{pmatrix} \big[\bar{J}(\boldsymbol{x^*})\big]_{i\notin I, j\notin I} & \big[-x_i^*\tilde{M}_{ij}\big]_{i\notin I, j\in I} \\ \big[0\big]_{i\in I, j\notin I} & \big[\mathrm{diag}(1-\sum_{k=1}^n \tilde{M}_{ik}x_k^*)\big]_{i\in I, j\in I} \end{pmatrix} \tag{3.2}$$

with $\Big(\bar{J}(\boldsymbol{x}^*)\Big)_{ij} = \mathbb{1}_{\{i=j\}} - x_i^*\tilde{M}_{ij} - \mathbb{1}_{\{i=j\}}\sum_{k\notin I}\tilde{M}_{ik}x_k^*$.
But, $\bar{J}(\boldsymbol{x^*})$ fulfils the assumptions of the first part of the proof and thus $\bar{J}(\boldsymbol{x}^*)$ has a positive eigenvalue. Since $J(\boldsymbol{x}^*)$ is a block triangular matrix and $\bar{J}(\boldsymbol{x}^*)$ is on its diagonal, all the eigenvalues of $\bar{J}(\boldsymbol{x}^*)$ are also eigenvalues of $J(\boldsymbol{x}^*)$. Thus $J(\boldsymbol{x}^*)$ has a positive eigenvalue. Therefore, $\boldsymbol{x}^*$ is unstable and there exists a stable manifold T of dimension strictly less than n. Finally, suppose $0 \le x_i^* \le w_i$ for all $i \in [n]$ with at least one component being strictly in $(0, w_i)$. Let $Q = \{i \in [n] : x_i^* = w_i\}$. Since $\boldsymbol{x}^*$ is a stationary point, for $i \in Q$:

$$\begin{aligned} &\frac{dx_i^*}{dt} = x_i^*[1 - (\tilde{M}\boldsymbol{x}^*)_i] = 0 \Rightarrow \\ &(\tilde{M}\boldsymbol{x}^*)_i = \big((\tau A + W)\boldsymbol{x}^*\big)_i = \tau(A\boldsymbol{x}^*)_i + \frac{1}{w_i}x_i^* = 1 \Rightarrow \\ &\tau(A\boldsymbol{x}^*)_i = 1 - \frac{1}{w_i}x_i^* = 1 - \frac{w_i}{w_i} = 0 \Rightarrow \\ &(A\boldsymbol{x}^*)_i = 0. \end{aligned} \tag{3.3}$$

So, $A_{ij} = 0$ for all $i \in Q$ and $j \notin I$. Thus, for $i \in Q$ and $j \neq i$, $j \notin I$:

$$\begin{aligned} [\bar{J}(\boldsymbol{x}^*)]_{ii} &= \mathbb{1}_{\{i=i\}} - x_i^*\tilde{M}_{ii} - \mathbb{1}_{\{i=i\}}\sum_{k\notin I}\tilde{M}_{ik}x_k^* \\ &= 1 - x_i^*(\tau A_{ii} + W_{ii}^{-1}) - \sum_{k\notin I}(\tau A_{ik} + W_{ik}^{-1})x_k^* \\ &= 1 - w_i x_i^* - w_i x_i^* - \tau\sum_{k\notin I}A_{ik}x_k^* \\ &= 1 - 2w_i^2 < 0 \end{aligned}$$

and

$$\begin{aligned} [\bar{J}(\boldsymbol{x}^*)]_{ij} &= \mathbb{1}_{\{i=j\}} - x_i^*\tilde{M}_{ij} - \mathbb{1}_{\{i=j\}}\sum_{k\notin I}\tilde{M}_{ik}x_k^* \\ &= -x_i^*(\tau A_{ij} + W_{ij}^{-1}) = -\tau x_i^* A_{ij} = 0. \end{aligned}$$

Thus, the Jacobian can be rewritten such that $\bar{J}(\boldsymbol{x}^*)$ is a block diagonal matrix. Specifically $\bar{J}(\boldsymbol{x}^*)$ can be rewritten as having $|Q|$ blocks of size 1 and

one block of size $n - |Q| - |I|$. But then the eigenvalues of the block of size $n - |Q| - |I|$ are also eigenvalues of $J(\boldsymbol{x}^*)$. And then since the block matrix of size $n - |Q| - |I|$ fulfils the assumptions of the first part of the proof it has a positive eigenvalue. Therefore, $\boldsymbol{x}^*$ is unstable and there exists a stable manifold T of dimension strictly less than n.
Since the finite union of sets with Lebesgue measure equal to zero has a Lebesgue measure equal to zero, the proof follows with $\Omega = S \cup T$. □

Lemma 5. *For $\tau > 1$, the set $\{v_i \in V : x_i^* = w_i\}$ where $\boldsymbol{x}^* := \lim_{t\to\infty} \boldsymbol{x}(t)$ with $\boldsymbol{x}(t)$ any solution of the system (2.1) equipped with matrix $\tilde{M}$, is a maximal independent set when $\boldsymbol{x}^*$ is stable.*

Proof. From Lemma 4, we know that any stable steady state $\boldsymbol{x}^*$, has entries x_i^* such that $x_i^* = 0$ or $x_i^* = w_i$. Define the set $Z = \{v_i \in V : x_i^* = w_i\}$.
Suppose $x_i^* = w_i$, then as shown by Equation (3.3), $x_i^*(1 - (\tilde{M}\boldsymbol{x}^*)_i) = 0 \Rightarrow (A\boldsymbol{x}^*)_i = 0$. This means that for all adjacent $j \sim i : x_j^* = 0$. This implies that Z is indeed an independent set in G.
Suppose that Z is not maximal. Then, there exists some $i \in [n]$ with $x_i^* = 0$ such that $x_j^* = 0$ for all $j \sim i$. And so, the lower right block of the Jacobian as seen in Equation (3.2) has an entry that lies in the diagonal of a block triangular matrix, which implies that there exists an eigenvalue λ such that :

$$\lambda = 1 - \sum_{k=1}^{n} \tau A_{ik} x_k^* = 1 > 0.$$

This implies that $\boldsymbol{x}^*$ is unstable. And so, by contradiction of the stability assumption, Z is maximal. □

3.2 Mathematical Analysis of the Competition Method

Similarly to the carrying capacity method, the competition method, as described in Section 2.4, consistently returns a maximal independent set. This stated in the following theorem.

Theorem 2. *Let $\tau > 1$ and $\bar{M} = \tau AW + \mathbb{I}$ with A and W as described in Section 2.1 and $\mathbb{I}$ the $n \times n$ identity matrix. Let $\boldsymbol{x}$ be the solution of the system (2.1) equipped with matrix $\bar{M}$ and with initial condition at $t = 0$ being $\boldsymbol{x_0} \in (0,1)^n$. Then $\boldsymbol{x}^* := \lim_{t\to\infty} \boldsymbol{x}(t)$ exists, $\forall i \in [n] : x_i^* \in [0,1]$ and the set $\{v_i \in V : x_i^* = 1\}$ is a maximal independent set.*

Proof. Let $\tau > 1$. Let $\boldsymbol{x}$ be the solution of the system (2.1) equipped with matrix $\bar{M}$ and with initial condition at $t = 0$ being $\boldsymbol{x_0} \in (0,1)^n$. Define

$\forall t \geq 0 : \boldsymbol{y}(t) := W\boldsymbol{x}(t)$. Then,

$$\begin{aligned}\frac{d\boldsymbol{x}}{dt} &= \boldsymbol{x} \circ \Big(1 - \bar{M}\boldsymbol{x}\Big) = \boldsymbol{x} \circ \Big(1 - (\tau AW + \mathbb{I})\boldsymbol{x}\Big) \\ &= W^{-1}\boldsymbol{y} \circ \Big(1 - (\tau A + W^{-1})\boldsymbol{y}\Big) = W^{-1}\boldsymbol{y} \circ \Big(1 - \tilde{M}\boldsymbol{y}\Big)\end{aligned}$$

which implies that

$$W^{-1}\frac{d\boldsymbol{y}}{dt} = W^{-1}\boldsymbol{y} \circ \Big(1 - \tilde{M}\boldsymbol{y}\Big) \iff \frac{d\boldsymbol{y}}{dt} = \boldsymbol{y} \circ \Big(1 - \tilde{M}\boldsymbol{y}\Big).$$

And also, $\boldsymbol{y_0} = W\boldsymbol{x_0}$ which implies that $\forall i \in [n] : y_{0i} \in (0, w_i)$. Then, by Theorem 1 we have that $\boldsymbol{y^*} := \lim_{t\to\infty} \boldsymbol{y}(t)$ exists and the set $\{v_i \in V : y_i^* = w_i\}$ is a maximal independent set. This implies that $\boldsymbol{x^*} := \lim_{t\to\infty} \boldsymbol{x}(t)$ exists, since

$$\lim_{t\to\infty} \boldsymbol{x}(t) = \lim_{t\to\infty} W^{-1}\boldsymbol{y}(t) = W^{-1} \lim_{t\to\infty} \boldsymbol{y}(t) = W^{-1}\boldsymbol{y^*}.$$

Additionally it implies that $\{v_i \in V : x_i^* = 1\}$ is a maximal independent set since

$$\{v_i \in V : y_i^* = w_i\} = \{v_i \in V : w_i x_i^* = w_i\} = \{v_i \in V : x_i^* = 1\}.$$

$\square$

Chapter 4

Analysing the Methods

In this chapter, we will outline some observations resulting from applying the methods described in the previous chapters. In Section 4.1, we will analyse the carrying capacity method, and in Section 4.2, the competition method.

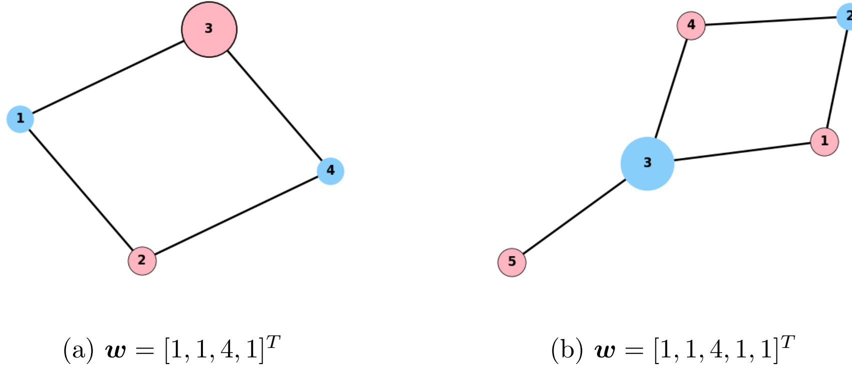
The following observations have been made after a limited number of tests. Each observation will be illustrated with concise yet representative examples. All results and figures presented in this chapter result from the implementation of the code provided predominantly in Appendix B.3 which itself uses code found in B.1 and B.2.

4.1 Analysis of the Carrying Capacity Method

First, we observe that the carrying capacity method has a bias towards nodes with lower degrees (number of neighbours). This means the algorithm tends to pick nodes with lower degrees, even if they are not part of a maximum-weighted independent set.

This observation is illustrated in Figure 4.1. Specifically, in Figure 4.1a the carrying capacity method correctly detects the maximum-weighted independent set $\{v_2, v_3\}$ with total weight equal to 5. But, when we add node v_5 as in Figure 4.1b, we see that the method returns the set $\{v_1, v_4, v_5\}$ with total weight equal to 3. This indicates a 'preference' for nodes with a lower degree.

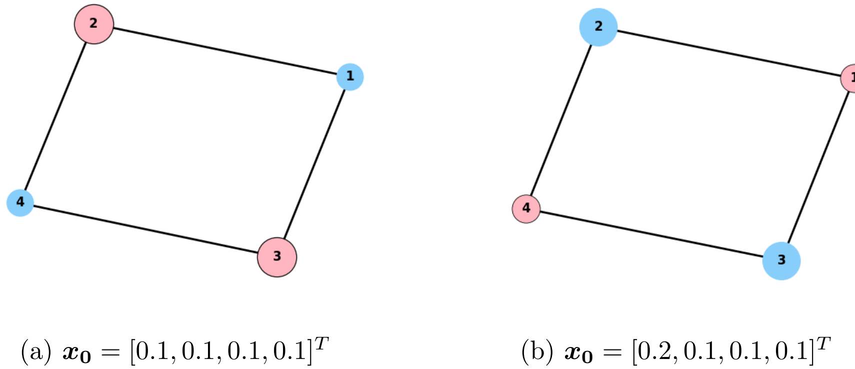
This bias is not surprising when one considers nodes as animal populations competing for resources. The more interactions a population has with others (i.e. the more neighbours a node has), the higher the risk of extinction. While a higher carrying capacity for a population offers a survival advantage, it does not always offset the impact of competition.



(a) $\boldsymbol{w} = [1, 1, 4, 1]^T$ (b) $\boldsymbol{w} = [1, 1, 4, 1, 1]^T$

Figure 4.1: Figure showing the maximal independent set (nodes in pink with black border) resulting from applying the carrying capacity method for the displayed graphs with initial conditions such that $\forall i : x_{0i} = 0.1$. The size of each node is determined by its weight.

The second observation we make about the carrying capacity method is related to the effect of the initial condition. Particularly, we observe a bias towards nodes with higher initial conditions. Consequently, the algorithm tends to pick a node v_i when x_{0i} is high, even if it is not part of a maximum-weighted independent set.



(a) $\boldsymbol{x_0} = [0.1, 0.1, 0.1, 0.1]^T$ (b) $\boldsymbol{x_0} = [0.2, 0.1, 0.1, 0.1]^T$

Figure 4.2: Figure showing the maximal independent set (nodes in pink with black border) resulting from applying the carrying capacity method for the displayed graphs with weight vector $\boldsymbol{w} = [1, 2, 2, 1]^T$.The size of each node is determined by its weight.

This observation is depicted in Figure 4.2. Here, any bias related to the degree of the nodes is eliminated by examining a regular graph (i.e. a graph whose nodes have all the same degree). Subsequently, in Figure 4.2a, it is clear that in the absence of bias stemming from initial conditions, the algorithm correctly identifies the maximum-weighted independent set. However, in Figure

4.2b, despite it not being part of the maximum-weighted independent set, the algorithm selects node v_1, which has a higher corresponding initial condition ($x_{01} = 0.2 > x_{0j}$ for all $j \neq 0$).

Finally, we observe that the carrying capacity method appears to have a bias against nodes with a low weight, instead of having a bias for nodes with a high weight. The algorithm tends not to pick node v_i when w_i is low. The nodes chosen by the algorithm are from the set of those which are left over after eliminating the lower-weighted ones, even if a low-weighted node is part of the maximum-weighted independent set.

This observation is illustrated in Figure 4.3. Here, the algorithm returns the set $\{v_2, v_4\}$ with a total weight of 4, instead of the maximum-weighted independent set $\{v_1, v_5\}$ or $\{v_1, v_6\}$ with a total weight of 5. Again, viewing each node as an animal population, this can be explained by noting that the populations corresponding to nodes v_5 and v_6 will quickly die out due to their low carrying capacity, even though they are part of a maximum-weighted independent set.

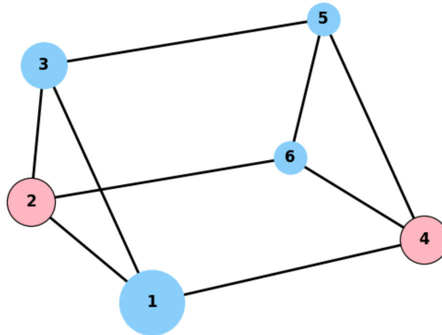


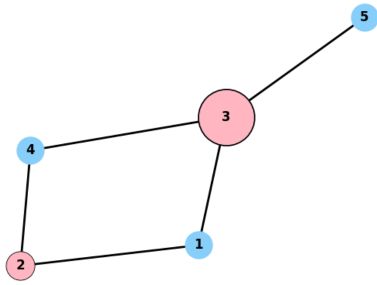
Figure 4.3: Figure showing the maximal independent set (nodes in pink with black border) resulting from applying the carrying capacity method for the displayed graphs with initial condition such that $\forall i : x_{0i} = 0.1$ and weight vector $\boldsymbol{w} = [4, 2, 2, 2, 1, 1]^T$. The size of each node is determined by its weight.

All the aforementioned biases of the carrying capacity method influence the accuracy of the method in finding maximum-weighted independent sets.

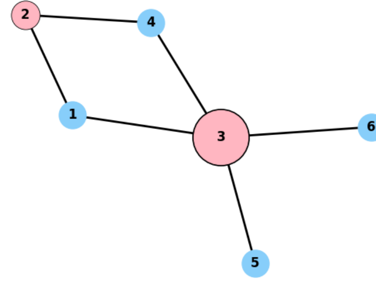
4.2 Analysis of the Competition Method

After implementing the competition method, we observed that any bias present in the carrying capacity method is either absent or weaker in the competition method.

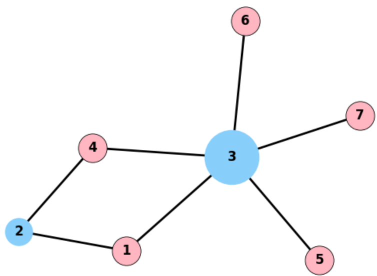
First, unlike the carrying capacity method, the competition method does not show a strong bias towards nodes with low degrees. This is illustrated in Figure 4.4. Specifically, in Figure 4.4a, we see that the competition method identifies the maximum-weighted set $\{v_2, v_3\}$, whereas, for the same graph, the carrying capacity method does not, as seen in Figure 4.1b. Additionally, Figures 4.4b, 4.4c, and 4.4d demonstrate that the competition method continues to correctly identify the maximum-weighted independent set without a clear bias towards nodes with lower (or higher) degrees.



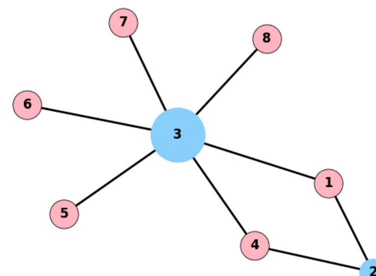
(a) $\boldsymbol{w} = [1, 1, 4, 1, 1]^T$



(b) $\boldsymbol{w} = [1, 1, 4, 1, 1, 1]^T$



(c) $\boldsymbol{w} = [1, 1, 4, 1, 1, 1, 1]^T$

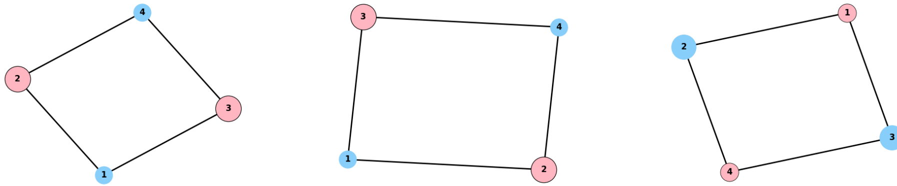


(d) $\boldsymbol{w} = [1, 1, 4, 1, 1, 1, 1, 1]^T$

Figure 4.4: Figure showing the maximal independent set (nodes in pink with black border) resulting from applying the competition method for the displayed graphs with initial conditions such that $\forall i : {x_0}_i = 0.1$. The size of each node is determined by its weight.

This observation has only been tested on a small scale, so we cannot definitively conclude that the competition algorithm generally lacks such a bias. However, we can conclude that the competition method does not exhibit a lower degree bias to the extent the carrying capacity method does.

Secondly, the competition method does have a bias towards nodes with a higher initial condition, but this bias is weaker when compared to the carrying capacity method. This is illustrated in Figure 4.5. Here we see that, unlike the carrying capacity method, the competition method correctly identifies the maximum-weighted set in the case illustrated by Figures 4.5a and 4.2b. Furthermore, the competition method accurately identifies the maximum-weighted independent set until the initial condition x_{01} exceeds 0.5, at which point the bias towards higher initial conditions becomes significant.



(a) $\boldsymbol{x_0} = [0.2, 0.1, 0.1, 0.1]^T$ (b) $\boldsymbol{x_0} = [0.5, 0.1, 0.1, 0.1]^T$ (c) $\boldsymbol{x_0} = [0.6, 0.1, 0.1, 0.1]^T$

Figure 4.5: Figure showing the maximal independent set (nodes in pink with black border) resulting from applying the competition method for the displayed graphs with weight vector $\boldsymbol{w} = [1, 2, 2, 1]^T$. The size of each node is determined by its weight.

Finally, the competition method does not appear to have a bias against nodes with low weight. This is demonstrated in Figure 4.6, where the competition method correctly identifies the maximum-weighted independent set. In contrast, as shown in Figure 4.3, the carrying capacity method fails to do so.

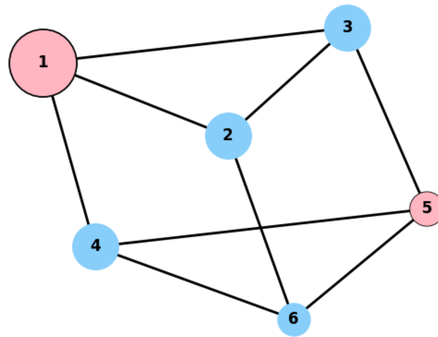


Figure 4.6: Figure showing the maximal independent set (nodes in pink with black border) resulting from applying the competition method for the displayed graphs with initial condition such that $\forall i : x_{0i} = 0.1$ and weight vector $\boldsymbol{w} = [4, 2, 2, 2, 1, 1]^T$. The size of each node is determined by its weight.

In conclusion, the competition method appears to perform better than the carrying capacity method. However, the competition method does have a bias for nodes with higher initial conditions which influences its ability to find maximum-weighted independent sets. Other biases might also be present but more research needs to be done on this topic.

Chapter 5

Detecting 'Heavy' Maximal Cliques

This chapter delves into how we can utilize the carrying capacity method and the competition method to approximate the solutions of the maximum-weighted clique problem. In Section 5.1 we define the maximum-weighted clique problem. In Section 5.2 we discuss how the maximum-weighted clique problem is related to the maximum-weighted independent set problem and explain why the carrying capacity method and the competition method are indeed suitable for detecting cliques.

5.1 Maximum-Weighted Cliques

Given the assumptions made in Section 2.1 we define the following:

Definition 4. *A set S is a clique in the graph $G = (E, V)$ if and only if $\forall u, v \in S \subset V : \{u, v\} \in E$.*

Definition 5. *A set S is a maximal clique in the graph $G = (E, V)$ if and only if S is clique in G and $\forall v \in V \backslash S : \exists u \in S$ such that $\{u, v\} \notin E$.*

Definition 6. *A set S is a maximum-weighted clique in the graph $G = (E, V)$ if and only if S is a maximal clique in G and*

$$\sum_{v_k \in S} w_k = \max\Big\{ \sum_{v_k \in B} w_k \textit{ with } B \textit{ a clique}\Big\}.$$

5.2 Cliques and Independent Sets

To understand the connection between the maximum-weighted clique problem and the maximum-weighted independent set problem, we need the following definition:

Definition 7. *The complement of the graph* $G = (V, E)$ *is the graph* $\bar{G} = (\bar{V}, \bar{E})$ *such that:*

- $\bar{V} = V$ *and*
- *For any pair of distinct vertices* $v, u \in V : \{v, u\} \in \bar{E} \iff \{v, u\} \notin E$.

With this definition, we can state the following theorem that connects cliques to independent sets.

Theorem 3. S *is an independent set in a simple, undirected graph* $G = (V, E)$ *with no loops if and only if* S *is a clique in the complement of* G.

Proof. Let S be an independent set in a simple, undirected graph with no loops $G = (V, E)$. And let $\bar{G} = (\bar{V}, \bar{E})$ be the complement of G. Then, since S is an independent set: $\forall u, v \in S \subset V : \{u, v\} \notin E$. But this implies that $\forall u, v \in S \subset \bar{V}\{u, v\} \in \bar{E}$. And so S is a clique in $\bar{G} = (\bar{V}, \bar{E})$.
The other way follows from the fact that the complement of the complement of G is again G ($\bar{\bar{G}} = G$). □

Corollary 1. S *is a maximum-weighted independent set in a simple, undirected graph* $G = (V, E)$ *with no loops if and only if* S *is a maximum-weighted clique in the complement of* G.

Thus, by Corollary 1, finding (an approximation to) the solution of the maximum-weighted clique problem is analogous to finding (an approximation to) the solution of the maximum independent set problem. This means that we can utilize the carrying capacity and the competition methods for detecting 'heavy' maximal cliques, provided that we implement them on the complement of the graph we are investigating.

Chapter 6

Application to Molecular Docking

In this chapter, we apply (and compare the performance of) the carrying capacity and competition methods to molecular docking. First, in Section 6.1 we introduce the problem of molecular docking. Section 6.3 outlines how the carrying capacity and competition method as well as a version of the greedy algorithm and the Lotka–Volterra algorithm of [15] are applied to molecular docking. Finally, in Section 6.4 the results of this application are discussed.

6.1 Molecular Docking

The final aim of this report is to apply the developed methods to molecular docking. Molecular docking is the problem of 'predicting the optimal interaction of two molecules, typically a small-molecule ligand and a target receptor' [8]. This has important applications in the field of biology most notably in assisting drug design [23].

As demonstrated in [8], molecular docking can be reduced to the maximum-weighted clique problem. This implies that determining the optimal interaction between two given molecules is equivalent to solving the maximum weighted clique problem for an appropriately chosen graph. Constructing these molecular docking graphs is a multi-step process. First, we need to identify which points (called pharmacophore points) in each molecule are involved in the binding interaction. Then each node in the graph is a possible interaction between the pharmacophore points of the different molecules. Two nodes have an edge between them if and only if the interactions they represent are physically compatible (meaning they can simultaneously occur). Finally, the weight of each node is determined by how strong is the interaction between the pharmacophore points.

Since the carrying capacity and competition method can approximate the solutions to the maximum-weighted clique problem, they can also approximate the solutions of molecular docking. In the following sections, these methods are applied to a set of molecular docking graphs.

6.2 The Dataset

In this section, we give some details about the database used for the application described in the following sections.

Our dataset consists of 89 out of the 447 files of the 'docking weighted graphs' of [24]. Due to the low computational power available at the beginning of this research, the chosen graphs are some of the smallest ones (in terms of their file size) from the entire 'docking weights graphs'.

All the molecular docking graphs are simple, undirected, node-weighted and without loops. The number of nodes of the 89 chosen graphs ranges from 17 to 2230 nodes, with an average number of nodes in a graph being 872. The number of edges ranges from 0 to 250139, with an average number of edges being 27727. The weights of the nodes range from 3937 to 208660 with the average weight being 49291.

6.3 Applying the Methods

In this section, we describe how we will apply (and compare) different algorithms for identifying ('heavy') maximal cliques to molecular docking graphs.

First, we note that molecular docking graphs are simple, undirected, node-weighted and without loops. Thus, in order to apply the carrying capacity and competition methods on a molecular docking graph, it is sufficient to divide the complement of the graph into connected components, apply the methods on each component and add the solutions to a single set. This is a result of the following theorem.

Theorem 4. *Let $G = (V, E)$ be a simple, undirected graph with no loops. Suppose that G had $N \geq 1$ distinct connected components $G_i = (V_i, E_i)$ for $i \in N$. Also, suppose that $\forall i \in N : S_i$ is an independent set in G_i. Then $S := \cup_{i=1}^{N} S_i$ is an independent set in G.*

Proof. Let $G = (V, E)$ be a simple, undirected graph with no loops and suppose that G can be divided into $N \geq 1$ distinct connected components $G_i = (V_i, E_i)$ for $i \in N$. Also, suppose that $\forall i \in N : S_i$ is an independent set in G_i. Now define $S := \cup_{i=1}^{N} S_i$ and let $u, v \in S$ be arbitrary. Then there are two cases we need to consider.

The first case is that both u and v belong to the same independent set S_i for some $i \in N$. In that case, it immediately follows that $\{u, v\} \notin E$.

In the other case, u and v belong to two different independent sets S_u and S_v for some $u, v \in N$ with $u \neq v$. But then u and v belong to the distinct set of vertices V_u and V_v of two unconnected components of G. And thus, again $\{u, v\} \notin E$.

Since in both cases for arbitrary $u, v \in S$ it holds that $\{u, v\} \notin E$, S is an independent set in $G = (V, E)$.

□

Besides the carrying capacity and competition methods we also apply the Lotka–Volterra method from [15]. This method is equivalent to both the carrying capacity and competition method with $W = \mathbb{I}$. Thus, to apply this method to molecular docking graphs we shall apply the competition method with weight vector $\boldsymbol{w} = \mathbf{1}$.

The final method we implement to molecular docking graphs is a version of the greedy algorithm. This version starts with the heaviest node in the given graph and iteratively picks the highest-weighted node that is connected to all previously picked nodes. The code for this algorithm can be found in Appendix B.4.

For the implementation of the methods based on the Lotka–Volterra system (carrying capacity, competition and Lotka–Volterra methods) we pick $\boldsymbol{x_0}$ to be within $(0, 1)^n$ and proportional to the corresponding weight vector $\boldsymbol{w}$. The reason for this is to give the Lotka–Volterra method a bias for 'heavier' nodes while still being able to compare the different methods for the same initial conditions. Research into which initial conditions are the most suitable for each different method is outside the scope of this report.

6.4 Results

In this section, we present the results of applying the carrying capacity, competition, Lotka–Volterra and greedy methods as described in Section 6.3.

The methods are applied on the set 'docking weighted graphs' of [24]. The code for this application and the raw results can be found in Appendix B.5 and C respectively.

Note that in this section and the accompanying figures, 'Method Weight' refers to the total weight of the output set produced by the specified method, while 'Method Time' indicates the duration required to complete the computation.

6.4.1 Weight-Based Performance Comparison

In this subsection, we discuss how the different methods compare based on their ability to detect 'heavy' cliques. We will call this ability their 'weight-based' performance.

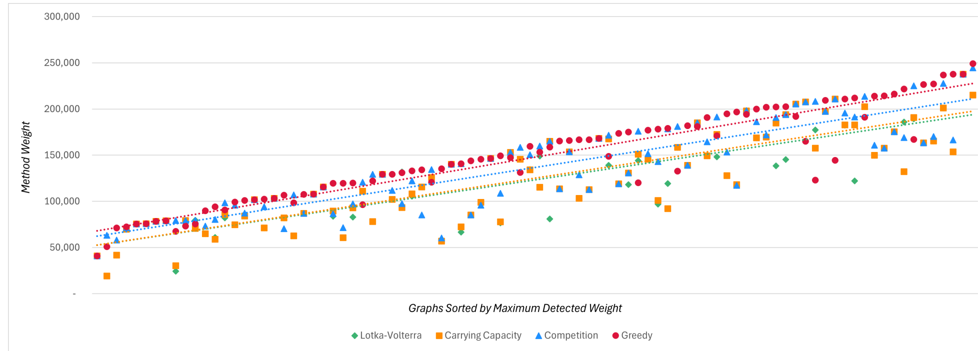


Figure 6.1: Comparison of the weights of the cliques detected by each method [Lotka-Volterra (green rhombus), carrying capacity (orange square), competition (blue triangle), and greedy (red circle)] across various graphs. The y-axis represents the weight of the clique detected by each method. For each graph, the highest weight detected by any method is identified, and these highest weights are sorted in ascending order to determine the placement of the graphs on the x-axis.

First, the carrying capacity, competition, Lotka–Volterra and greedy method are applied to the 89 graphs described in Section 6.2. The weight of the clique they detect is calculated and the highest detected weight for each graph is determined. Then Figure 6.1 is obtained by sorting all highest detected weights in ascending order and having each vertical line include the weight detected by each method for a single graph.

In Figure 6.1, we observe that the greedy algorithm most frequently identifies the clique with the highest relative weight. Following this, the competition method ranks as the second-best performer. The carrying capacity and Lotka–Volterra methods both perform relatively poorly, with the Lotka–Volterra method being slightly worse. Additionally, we notice that for graphs with lower detected weights, the performance differences between the methods are smaller than for graphs with higher detected weights. This trend is not surprising, considering that graphs with lower weights typically have fewer nodes, leading to fewer cliques and, consequently, fewer options for the methods to choose from. While this is not always the case, it could explain the observed trend.

Since the Lotka–Volterra method generally has the lowest weight-based performance, we will use its results as a baseline to compare the other methods. Figures 6.2 and 6.3 are obtained by dividing the weight of the output of each

(depicted) method with the weight of the output of the Lotka–Volterra method for the same graph. These results are represented on the y-axis. Each vertical line represents a graph. These graphs are sorted so that the output relating to the competition and greedy method is monotonically decreasing for figures 6.2 and 6.3 respectively.

As expected, Figure 6.2 shows that the competition method typically returns cliques with higher (or equal) weights compared to those returned by the carrying capacity method for the same graphs. Similarly, Figure 6.3 demonstrates that the greedy algorithm generally achieves higher weight-based performance. However, comparing Figures 6.2 and 6.3 we see that the greedy algorithm more frequently returns cliques with weights lower than those produced by the Lotka–Volterra method, in contrast to the competition method. We will revisit this observation in Subsection 6.4.2.

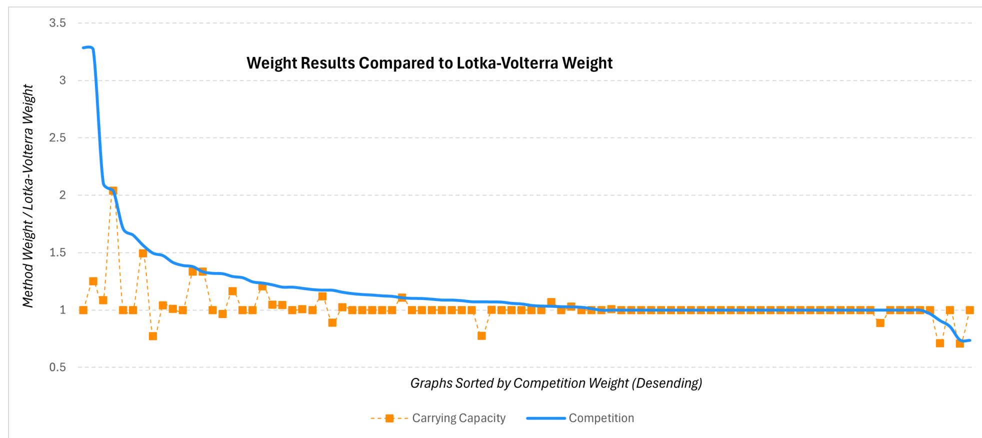


Figure 6.2: Figure depicting the weight of the cliques detected by the competition (blue line) and carrying capacity (orange square) methods, each divided by the weight of the clique detected by the Lotka–Volterra method for the same graph (y-axis). The graphs are sorted by the weight of the clique detected by the competition method (x-axis).

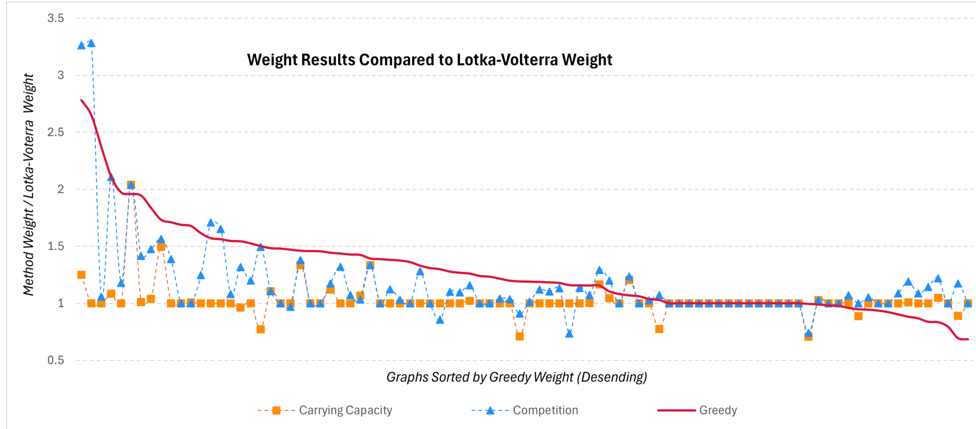


Figure 6.3: Figure depicting the weight of the cliques detected by the competition (blue triangle), carrying capacity (orange square) and greedy (red line) methods, each divided by the weight of the clique detected by the Lotka–Volterra method for the same graph (y-axis). The graphs are sorted by the weight of the clique detected by the greedy method (x-axis).

6.4.2 Weight-Based Performance Rankings

In this subsection, we again look into the weight-based performance of the different methods. This section will specifically focus on ranking and discussing their performance in different contexts.

First, Figure 6.4 depicts how often each method returns a clique with the first, second, third or fourth highest weight. It is clear that, as we have seen before, the greedy algorithm returns the 'heaviest' clique the most often with the competition method second most often. The carrying capacity and Lotka–Volterra method both rank most often 3rd in their weight-based performance.

Now we take a closer look into the best and worst performances depicted in Figure 6.5. Specifically, in Figure 6.5a, we observe that following the Lotka–Volterra method, the greedy algorithm most frequently exhibits the poorest weight-based performance, while the competition method exhibits this least often. This suggests that although the greedy algorithm often has the best weight-based performance (as seen in Figure 6.5b), it is relatively unreliable, as its performance tends to be worse than that of other methods when it does poorly. In contrast, the competition method consistently ranks either best or second best in its weight-based performance (as seen in Figure 6.4), making it the most reliable method for detecting 'heavy' maximal cliques.

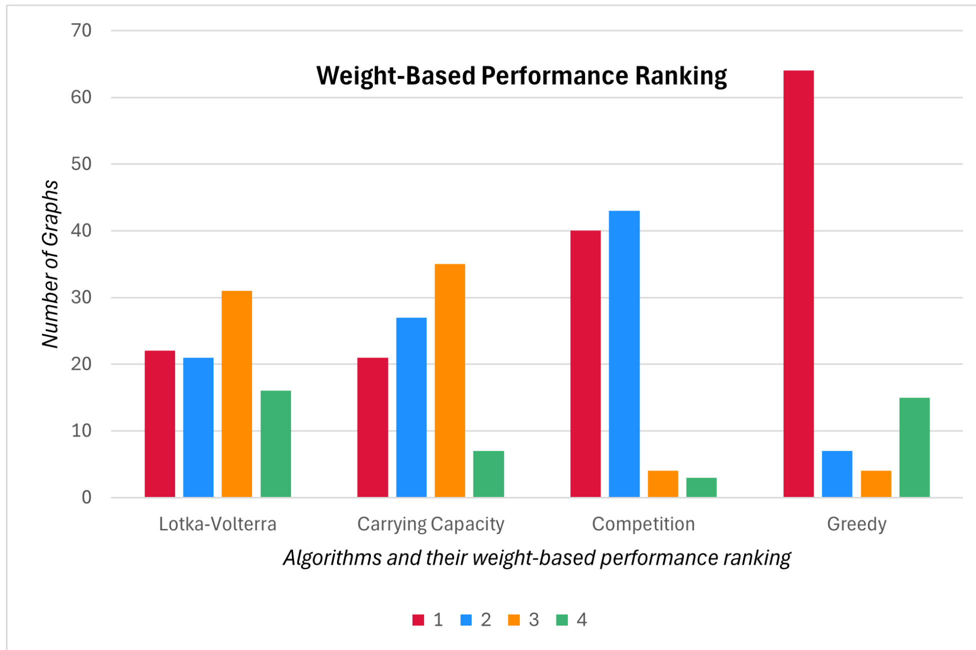
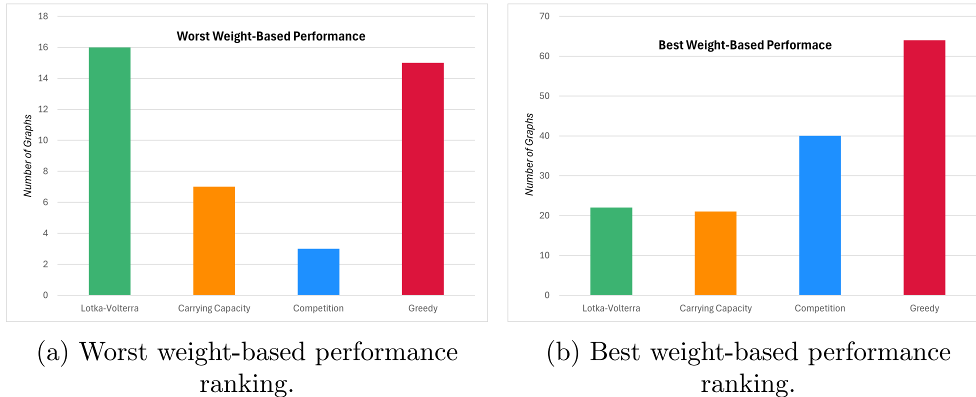


Figure 6.4: Figure depicting for how many graphs (y-axis) each method (Lotka–Volterra, carrying capacity, competition and greedy) ranks 1st (red), 2nd (blue), 3rd(orange) and 4th(green) in weight-based performance.



(a) Worst weight-based performance ranking.

(b) Best weight-based performance ranking.

Figure 6.5: Figure depicting for how many graphs (y-axis) each method each method (Lotka–Volterra, carrying capacity, competition and greedy) has the worst/best comparative weight-based performance.

6.4.3 Time-Based Performance Comparison

In this subsection, we will discuss the time-based performance (or processing time) of each method.

In Figure 6.6 we see how the processing time of each method relates to the number of nodes in the graph. The greedy algorithm has the lowest processing

time, which is very close (but not equal to) zero for all the graphs we applied it on. The method with the second lowest processing time is the Lotka–Volterra method. Both methods developed in this research (carrying capacity and competition) have the highest processing time with little difference in the trend in which processing time grows with the number of nodes.

All the methods have polynomial time complexity. This means that the time it takes for a method to return a clique is, at most, a polynomial function of the number of nodes in the graph. This is a direct result of the implementation of the methods and is illustrated by the fitted lines (dotted) in Figure 6.6. These lines are fourth-order polynomials which have an R^2 value of 0.8125 and 0.9063 for the competition and carrying capacity method respectively. Here R^2 is the coefficient of determination, which means that it having a value close to 1 is indicative of a proper fitting. Since all the methods have a polynomial time complexity, they are significantly faster than any current exact algorithm for detecting the maximum weighted clique for large graphs, as this problem is in the class NP.

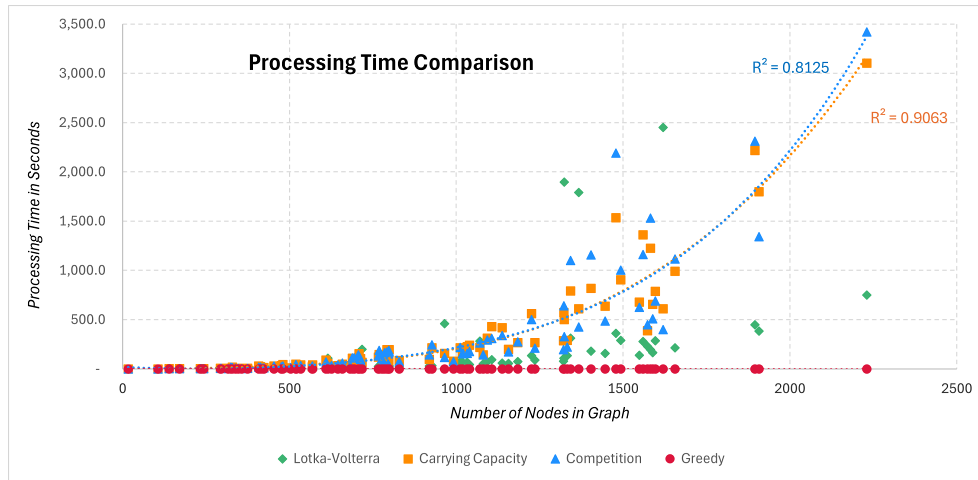


Figure 6.6: Figure depicting the processing time (in seconds) of each method [Lotka–Volterra (green rhombus), carrying capacity (orange square), competition (blue triangle), and greedy (red circle)] for a given graph (x-axis) as a function of the number of nodes in that graph (y-axis). The dotted lines correspond to the polynomials $2x^4(10^{-10}) - 3x^3(10^{-7}) + 0.0005x^2 - 0.1827x + 19.794$ (blue) and $3x^4(10^{-11}) + 3x^3(10^{-7}) - 0.0003x^2 + 0.2079x - 29.212$ (orange)

6.4.4 Discussion and Conclusion

Finally, we end this section with the conclusions made after applying the carrying capacity, competition, Lotka–Volterra, and greedy methods to the 'docking weighted graphs' dataset.

In the weight-based performance comparison, the greedy algorithm emerged as the most effective method for identifying the heaviest cliques across the majority of graphs, followed by the competition method. The carrying capacity and Lotka–Volterra methods, however, consistently showed poorer performance, with the Lotka–Volterra method ranking lowest. Interestingly, while the greedy algorithm often achieved the highest weight-based performance, it also demonstrated greater variability, occasionally performing worse than the Lotka–Volterra method. In contrast, the competition method exhibited more consistent performance, ranking either first or second almost always.

The time-based performance comparison revealed that the greedy algorithm not only performed well in the detection of 'heavy' cliques but also excelled in processing time, consistently requiring by far the least time across all graphs. The Lotka–Volterra method followed, with slightly higher processing times. The methods developed in this research, carrying capacity and competition, demonstrated higher processing times, though these times increased polynomially with the number of nodes, which is an improvement to exact algorithms for solving the maximum weighted clique problem.

In conclusion, the greedy algorithm performs best in regards to speed and ability to detect 'heavy' cliques but it also exhibits the highest variability in its results. The competition method, while slower, offers more reliable performance across different scenarios.

Chapter 7

Discussion

In this report, we present two heuristic methods for approximating the solutions to the maximum weighted clique problem. Both methods are based on the dynamics of the Lotka–Volterra system and are named the carrying capacity and competition methods because of their biological interpretation of the weight of nodes in a graph.

The two heuristics are an extension of previous work in [15] for the problem of maximum independent sets. And, we show that for appropriately chosen initial conditions, they consistently return maximal cliques.

The developed methods are applied to the problem of molecular docking, which is equivalent to detecting maximum weighted cliques on relevant graphs. And so, the carrying capacity and competition methods, as well as the method they are based on (the Lotka–Volterra method) and the greedy algorithm, are applied on 89 such graphs. This application reveals the following observations.

The greedy algorithm consistently outperforms others in identifying the 'heaviest' cliques, followed by the competition method. The carrying capacity and Lotka–Volterra methods generally show poorer performance, with the latter ranking lowest. Notably, while the greedy algorithm exhibits the best weight-based performance, it also shows greater variability, occasionally underperforming compared to the Lotka–Volterra method. The competition method, in contrast, demonstrates more consistent performance, often ranking first or second.

The Greedy algorithm also excels in terms of processing time, requiring the least time across all graphs. The Lotka–Volterra method follows, with slightly higher processing times. The carrying capacity and competition methods, despite higher processing times, exhibit polynomial time complexity, indicating significant speed advantages over exact algorithms.

In conclusion, this report delves into two heuristic algorithms for the maximum weighted clique problem, demonstrating their potential through both theoretical analysis and practical application. While the greedy algorithm remains more effective in many cases, the competition method's consistency offers a valuable alternative.

Future research should focus on refining these methods and understanding where their application is most appropriate. This can be done by investigating if there are specific types of graphs for which the competition and carrying capacity perform better and are more suitable. This research can also include if there are types of graphs for which the greedy algorithm performs worse and if there is overlap in the aforementioned graph types. Additionally, research can be done as to what extent (if any) the range of weights affects the performance of the algorithms. Since the methods are based on a biological model (the logistic growth mutually antagonistic Lotka–Volterra model), biological observations can be potentially used to improve them and understand why the competition method performs, in general, better than the carrying capacity method. Finally, and maybe most interestingly, research can be done to understand the effect the choice of initial conditions has on the performance of the methods. This can include determining how (if possible) to optimise the choice of initial conditions.

Appendix A

Supplementary Proofs

In this appendix, we present additional theorems and lemmas, along with their proofs, utilized in the report. All the following content is based on theorems and exercises found in [18].

Definition 8. *For an $n \times n$ matrix U, the replicator system on the simplex $S_n = \{\boldsymbol{y} : \sum_{i=1}^{n} y_i = 1$ and $\forall i \in n : y_i \geq 0\}$ is given $\forall i \in [n]$ by*

$$\frac{dy_i}{dt} = y_i\Big(\big(U\boldsymbol{y}\big)_i - \boldsymbol{y}^T U \boldsymbol{y}\Big). \tag{A.1}$$

Lemma 5 (Exercise 4.12 [18])**.** *Let $\boldsymbol{f}, \boldsymbol{g} : \mathbb{R}^n \to \mathbb{R}^n$ be continuous functions with g strictly positive. Then the solutions of $\frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}, t)$ and $\frac{d\boldsymbol{y}}{dt} = \boldsymbol{g}(\boldsymbol{y}, t)\boldsymbol{f}(\boldsymbol{y}, t)$ can be transformed into each other by a strictly monotonic change in the time scale $\phi(t)$.*

Proof. Let $\boldsymbol{g}$ be a strictly positive function. Let $\boldsymbol{x}$ be the solution of

$$\frac{dx(t)}{dt} = \boldsymbol{f}\big(\boldsymbol{x}(t), t\big)$$

with $\boldsymbol{f}$ an arbitrary continuous function. Define the map $\boldsymbol{y}(t) = \boldsymbol{x}(\phi(t))$ where $\phi(t) = \int \boldsymbol{g}\big(\boldsymbol{y}(s), s\big) ds$ with the initial condition $\phi(t_0) = t_0$.

Then, we have

$$\begin{aligned}
\frac{d\boldsymbol{y}(t)}{dt} &= \boldsymbol{x}'\big(\phi(t)\big)\phi(t)' \\
&= \boldsymbol{f}\big(\boldsymbol{x}(\phi(t)), t\big)\frac{d}{dt}\Big(\int \boldsymbol{g}\big(\boldsymbol{y}(s), s\big) ds\Big) \\
&= \boldsymbol{f}\big(\boldsymbol{y}(t), t\big)\boldsymbol{g}\big(\boldsymbol{y}(t), t\big).
\end{aligned}$$

Since $\boldsymbol{g}$ is strictly positive, ϕ is monotonic. Therefore, $\phi(t)^{-1}$ exists and is monotonic as well. And so, the other way directly follows.

□

Lemma 6 (Exercise 7.12 [18]). *The addition of a constant c to the jth column of a matrix U does not change (A.1) on S_n.*

Proof. Consider the replicator system (A.1) equipped with matrix U.

Let $\bar{U}$ be the matrix obtained by adding a constant c to the j-th column of U:

$$\bar{U}_{ik} = \begin{cases} U_{ik} + c & \text{if } k = j, \\ U_{ik} & \text{otherwise.} \end{cases}$$

Then for all $i \in [n]$ we have:

$$(\bar{U}\boldsymbol{x})_i = \sum_{k=1}^{n} \bar{U}_{ik} x_k = \sum_{k=1}^{n} (\bar{U}_{ik} + c\ \mathbb{1}_{k=j}) x_k = (U\boldsymbol{x})_i + c x_j \tag{A.2}$$

and

$$\begin{aligned} \boldsymbol{x}^T \bar{U} \boldsymbol{x} &= \sum_{i=1}^{n} \sum_{k=1}^{n} x_i \bar{U}_{ik} x_k = \sum_{i=1}^{n} \sum_{k=1}^{n} x_i (U_{ik} + c\ \mathbb{1}_{k=j}) x_k \\ &= \boldsymbol{x}^T U \boldsymbol{x} + \sum_{i=1}^{n} x_i c x_j = \boldsymbol{x}^T U \boldsymbol{x} + c x_j. \end{aligned} \tag{A.3}$$

Note that the last equality we used that, since $\boldsymbol{x} \in S_n$, $\sum_{i=1}^{n} x_i = 1$.

Substituting (A.2) and (A.3) into the replicator system we get:

$$\begin{aligned} \frac{dx_i}{dt} &= x_i \left((\bar{U}\boldsymbol{x})_i - \boldsymbol{x}^T \bar{U} \boldsymbol{x} \right) \\ &= x_i \left((U\boldsymbol{x})_i + c_j x_j - (\boldsymbol{x}^T U \boldsymbol{x} + c_j x_j) \right) \\ &= x_i \left((U\boldsymbol{x})_i - \boldsymbol{x}^T U \boldsymbol{x} \right). \end{aligned}$$

□

Theorem 5 (Theorem 7.5.1 [18]). *For $n \geq 2$, there exists a differentiable, invertible map from $\{\boldsymbol{y} \in S_n : y_n > 0\}$ with $S_n = \{\boldsymbol{y} \in \mathbb{R}^n : y_i \geq 0$ and $\sum_{i=1}^{n} y_i = 1\}$ onto $\mathbb{R}^{n-1}_{\geq 0} = \{\boldsymbol{x} \in \mathbb{R}^{n-1} : x_i \geq 0\}$ mapping the trajectories of the n-dimensional replicator system*

$$\frac{dy_i}{dt} = y_i \Big(\big(U\boldsymbol{y}\big)_i - \boldsymbol{y}^T U \boldsymbol{y} \Big)$$

onto the trajectories of the $(n-1)$-dimensional Lotka–Volterra system

$$\frac{dx_i}{dt} = x_i \Big(r_i + \sum_{j=1}^{n-1} M_{ij} x_j \Big)$$

with $r_i = U_{in} - U_{nn}$ and $M_{ij} = U_{ij} - U_{nj}$.

Proof. Define the map from $\{\boldsymbol{y} \in S_n : y_n > 0\}$ onto $\{\boldsymbol{x} \in \mathbb{R}^n_{\geq 0} : x_n = 1\}$ by

$$x_i = \frac{y_i}{y_n} \quad \text{for } i \in [n].$$

By Lemma 6 we can add the constant $-U_{nj}$ to the jth column of U and not change the n-dimensional replicator system. And so, without loss of generality, we assume that the last row of U consists of only zeros.

Then we have

$$\begin{aligned}\frac{dx_i}{dt} &= \frac{d}{dt}\Big(\frac{y_i}{y_n}\Big) = \frac{1}{y_n}\frac{dy_i}{dt} + y_i \frac{d}{dt}\Big(\frac{1}{y_n}\Big) \\ &= \frac{1}{y_n}\Big(y_i\big((U\boldsymbol{y})_i - \boldsymbol{y}^T U\boldsymbol{y}\big)\Big) - \frac{y_i}{y_n^2}\Big(y_n\big((U\boldsymbol{y})_n - \boldsymbol{y}^T U\boldsymbol{y}\big)\Big) \\ &= \frac{y_i}{y_n}\Big((U\boldsymbol{y})_i - (U\boldsymbol{y})_n\Big) = x_i\Big((U\boldsymbol{y})_i - (U\boldsymbol{y})_n\Big).\end{aligned}$$

But since the last row of U consists of only zero, we have $\big(U\boldsymbol{y}\big)_n = 0$. This implies

$$\frac{dx_i}{dt} = x_i\big(U\boldsymbol{y}\big)_i = x_i \sum_{j=1}^{n} U_{ij} y_j = x_i\Big(\sum_{j=1}^{n} U_{ij} x_j\Big) y_n.$$

Since $y_n > 0$, from Lemma 5 we have

$$\frac{dx_i}{dt} = x_i\Big(\sum_{j=1}^{n} U_{ij} x_j\Big).$$

Finally, since $x_n = 1$

$$\frac{dx_i}{dt} = x_i\Big(r_i + \sum_{j=1}^{n-1} U_{ij} x_j\Big),$$

where $r_i = U_{in}$.
Going from the $(n-1)$-dimensional Lotka–Volterra system to the n-dimensional replicator system we utilize the inverse of the map given previously. To determine the inverse map, first note that since $\boldsymbol{y} \in \hat{S_n}$,

$$y_i = \frac{y_i}{\sum_{j=1}^{n} y_i}.$$

Then using the previously defined map, we have

$$y_i = \frac{x_i/y_n}{\sum_{j=1}^{n} x_i/y_n} = \frac{x_i}{\sum_{j=1}^{n} x_i}.$$

The rest of the proof follows in an analogous way.

□

Lemma 7 (Exercise 7.5.2 [18]). *Suppose all r_i are equal in the n-dimensional Lotka–Volterra system*

$$\frac{dx_i}{dt} = x_i\big(r_i + \sum_{j=1}^{n} U_{ij}x_j\big) \qquad on\ \mathbb{R}_{\geq 0}. \tag{A.4}$$

Then $y_i = x_i / \sum_{j=1}^{n} x_j$ satisfies the replicator system (A.1) on S_n.

Proof. Let $\boldsymbol{x}(t)$ be the trajectory of the n-dimensional Lotka–Volterra system (A.4). Define the map $y_i = x_i / \sum_{j=1}^{n} x_j$ for all $i \in [n]$.

Then $\forall i \in [n] : y_i > 0$ and $\sum_{i=1}^{n} y_i = \big(\sum_{i=1}^{n} x_i / \sum_{j=1}^{n} x_j\big) = 1$. So, $y_i = x_i / \sum_{j=1}^{n} x_j$ maps $\mathbb{R}_{\geq 0}$ onto S_n.

Now we will show that $\boldsymbol{y}(t)$ with entries y_i as given by the given map, satisfies the replicator system

$$\begin{aligned}\frac{dy_i}{dt} &= \frac{d}{dt}\Big(\frac{x_i}{\sum_{j=1}^{n} x_j}\Big) = \Big(\sum_{j=1}^{n} x_j\Big)^{-1}\frac{dx_i}{dt} - x_i\Big(\sum_{j=1}^{n} x_j\Big)^{-2}\sum_{j=1}^{n}\frac{dx_j}{dt}\\ &= \frac{x_i}{(x_1 + \ldots + x_n)}\Big(r_i + \sum_{j=1}^{n} U_{ij}x_j\Big) - \frac{x_i}{(x_1 + \ldots + x_n)^2}\sum_{j=1}^{n} x_j\big(r_i + \sum_{k=1}^{n} U_{ik}x_k\big).\end{aligned}$$

But, since all r_i are equal we can say that $\forall i \in [n] : r_i = r$. Then we have

$$\begin{aligned}\frac{dy_i}{dt} &= \frac{x_i}{(x_1 + \ldots + x_n)}\Big[r + \sum_{j=1}^{n} U_{ij}x_j - \frac{1}{(x_1 + \ldots + x_n)}\Big(r\sum_{j=1}^{n} x_j + \sum_{j=1}^{n}\sum_{k=1}^{n} x_j U_{jk} x_k\Big)\Big]\\ &= y_i\Big[r\Big(1 - \frac{x_1 + \ldots + x_n}{x_1 + \ldots + x_n}\Big) + \sum_{j=1}^{n} U_{ij}x_j - \frac{1}{(x_1 + \ldots + x_n)}\sum_{j=1}^{n}\sum_{k=1}^{n} x_j U_{jk} x_k\Big]\\ &= y_i\Big[\sum_{j=1}^{n} U_{ij}x_j - \frac{1}{(x_1 + \ldots + x_n)}\sum_{j=1}^{n}\sum_{k=1}^{n} x_j U_{jk} x_k\Big]\\ &= y_i\Big((x_1 + \ldots + x_n)\sum_{j=1}^{n} U_{ij}y_j - (x_1 + \ldots + x_n)\sum_{j=1}^{n}\sum_{k=1}^{n} y_j U_{jk} y_k\Big)\\ &= y_i\Big(\big(U\boldsymbol{y}\big)_i - \boldsymbol{y}^T U \boldsymbol{y}\Big)(x_1 + \ldots + x_n).\end{aligned}$$

And then by Lemma 5 it follows that

$$\frac{dy_i}{dt} = y_i\Big(\big(U\boldsymbol{y}\big)_i - \boldsymbol{y}^T U \boldsymbol{y}\Big).$$

□

Theorem 7 (Theorem 19.2.1 [18]). *Each solution of the replicator system (A.1) on S_n equipped with a symmetric matrix converges to a fixed point.*

The full proof can be found on pages 252-253 of [18]. The following are additional calculations that can assist with that proof.

First, Define $P(\boldsymbol{x}) = \Pi_{i=1}^n x_i^{p_i}$. Then we have

$$\begin{aligned}
P^{-1}\big(\frac{d}{dt}P\big) &= P^{-1}\sum_{i=1}^n p_i\ x_i^{p_i-1}\frac{dx_i}{dt}\big(x_1^{p_1}\dots x_{i-1}^{p_{i-1}}x_{i+1}^{p_{i+1}}\dots x_n^{p_n}\big)\\
&= P^{-1}\sum_{i=1}^n p_i\ x_i^{p_i}\Big(\big(M\boldsymbol{x}\big)_i - \boldsymbol{x}^T M\boldsymbol{x}\Big)\big(x_1^{p_1}\dots x_{i-1}^{p_{i-1}}x_{i+1}^{p_{i+1}}\dots x_n^{p_n}\big)\\
&= P^{-1}\sum_{i=1}^n p_i\Big(\big(M\boldsymbol{x}\big)_i - \boldsymbol{x}^T M\boldsymbol{x}\Big)\Pi_{j=1}^n x_j^{p_j}\\
&= P^{-1}\sum_{i=1}^n p_i\Big(\big(M\boldsymbol{x}\big)_i - \boldsymbol{x}^T M\boldsymbol{x}\Big)P\\
&= \sum_{i=1}^n p_i\Big(\big(M\boldsymbol{x}\big)_i - \boldsymbol{x}^T M\boldsymbol{x}\Big)\\
&= \boldsymbol{p}^T M\boldsymbol{x} - \boldsymbol{x}^T M\boldsymbol{x}
\end{aligned}$$

Additionally, define $L(\boldsymbol{x}) = -\sum_i^n p_i \log\left(\frac{x_i}{p_i}\right)$. Then we have,

$$\begin{aligned}
\frac{d}{dt}L(\boldsymbol{x}) &= -\sum_i^n p_i\frac{1}{x_i}\frac{dx_i}{dt} = -\sum_{i=1}^n p_i\Big(\big(M\boldsymbol{x}\big)_i - \boldsymbol{x}^T M\boldsymbol{x}\Big)\\
&= \boldsymbol{x}^T M\boldsymbol{x} - \boldsymbol{p}^T M\boldsymbol{x} = \boldsymbol{x}^T M\boldsymbol{x} - \boldsymbol{p}^T M\boldsymbol{p} + \boldsymbol{p}^T M\boldsymbol{p} - \boldsymbol{x}^T M\boldsymbol{p}\\
&= \boldsymbol{x}^T M\boldsymbol{x} - \boldsymbol{p}^T M\boldsymbol{p} - \sum_{i=1}^n x_i[(M\boldsymbol{p})_i - \boldsymbol{p}^T M\boldsymbol{p}]
\end{aligned}$$

Note that for the above we use that $\boldsymbol{x}^T M\boldsymbol{p} = \boldsymbol{p}^T M\boldsymbol{x}$ which follows from the fact that M is symmetric.

Appendix B

Code for the Carrying Capacity and Competition Method

This appendix includes all the Python code used for this report.

Section B.1 includes all the imported libraries and functions that are needed to define each method. In Section B.2 we present the functions for implementing the carrying capacity and competition methods. Section B.3 includes the code specifically used for the Figures seen in Chapter 4. The code for implementing the greedy algorithm for maximum weighted cliques is found in Section B.4. Finally, Section B.5 includes the code used for the application on molecular docking (in Chapter 6).

B.1 Helpful Functions

This section includes most functions that are used in the implementation of the methods (excluding the methods themselves). All other sections refer back to functions (and use libraries) in this section.

```
import networkx as nx
import numpy as np
from scipy import integrate
import warnings
import matplotlib.pyplot as plt


# Function used to visualise graphs with weighted nodes

def visualize_graph(graph: nx.Graph, weights=None, colored_nodes=None):
    if colored_nodes is None:
        node_colors = 'lightskyblue'
```

```
    else:
        default_color = 'lightskyblue'
        node_colors = [default_color] * len(graph.nodes())
        for node in colored_nodes:
            node_colors[list(graph.nodes()).index(node)] = 'lightpink'

    node_sizes = [700] * weights
    labels = {i: i + 1 for i in graph.nodes()}

    pos = nx.spring_layout(graph)
    for node in graph.nodes():
        node_color = node_colors[list(graph.nodes()).index(node)]
        node_size = node_sizes[list(graph.nodes()).index(node)]

        if node in colored_nodes:
            nx.draw_networkx_nodes(graph, pos, nodelist=\
                [node], node_color='black', node_size=node_size)

        nx.draw_networkx_nodes(graph, pos, nodelist=[node],\
            node_color=node_color, node_size=node_size*(1/1.1))

    nx.draw_networkx_edges(graph, pos, width=2)
    nx.draw_networkx_labels(graph, pos, labels,\
        font_size=12, font_weight='bold')
    plt.axis('off')
    plt.show()


# Function used to check if a given set is a maximal independent set

def is_maximal_independent_set(graph, independent_set):
    complement_set = [node for node in graph.nodes() if node \
                      not in independent_set]

    independence_property = all(not graph.has_edge(u, v) for u in
                            independent_set for v in independent_set)

    maximality_checks = [not all(not graph.has_edge(u, v) for v in \
                        independent_set) for u in complement_set]
    maximality_property = all(maximality_checks)

    return independence_property and maximality_property
```

```
# Function used to check if a given set is a clique

def is_clique(graph, node_set):
    node_set = set(node_set)
    for node in node_set:
        neighbors = set(graph.neighbors(node))
        if not node_set - {node} <= neighbors:
            return False
    return True


# Function which divides a graph into connected components

def divide_into_subgraphs(graph):
    connected_components = list(nx.connected_components(graph))
    subgraphs = [graph.subgraph(component).copy() for \
                 component in connected_components]
    return subgraphs


# Function to calculate the total weight of a set

def SetWeight(weights,Set):
    return sum(weights[node-1] for node in Set)
```

B.2 The Methods

The following code includes a function for the implementation of each of the heuristic methods.

```
# Function implementing the Carrying Capacity Method

def Carrying_Capacity(G: nx.Graph, w: np.ndarray, x0: np.ndarray, tau: float):

    with warnings.catch_warnings():
        warnings.simplefilter("ignore")

        #Define Matrices
        W = np.diag(1/w)
        A = nx.to_numpy_array(G)
        M=tau*A+W

        # Define the generalized differential equations and the Jacobian
        f = lambda t, x: x - np.dot(np.dot(np.diag(x), M), x)
```

```
    J = lambda t, x: np.identity(len(A)) - \
        (np.dot(np.diag(x), M) + np.diag(np.dot(M, x)))

    # Set up the ODE solver
    ode_solver = integrate.ode(f, J)
    ode_solver.set_integrator('dopri5')

    # Stop integration when all nodes converge to 0 or their weight
    def solout(t, x):
        c=0
        for i in range(0,len(x)):
            if (x[i] < 1e-5) | (x[i] > w[i] - 1e-5):
                c+=1
        return -1 if c==len(x) else 0

    ode_solver.set_solout(solout)

    # Integrate until output 0 or weight for each element
    while True:
        t_end = 1e9
        ode_solver.set_initial_value(x0, 0)
        y = ode_solver.integrate(t_end)

        if solout(t_end, y) == -1:
            break
        x0 = y

    y = np.transpose(y)

    # Determine variables that do not converge to zero
    nodes_arr = list(G.nodes())
    max_independent_set = \
        [nodes_arr[i] for i in range(len(G)) if y[i] > w[i] - 1e-5]


if is_maximal_independent_set(G, max_independent_set):
    return max_independent_set
else:
    print('The algorithm did not return MIS.')
```

```
# Funtion implementing the Competition Method

def Competition(G: nx.Graph, w: np.ndarray, x0: np.ndarray, tau: float):

    with warnings.catch_warnings():
        warnings.simplefilter("ignore")

        #Define Matrices
        W =np.diag(w)
        G = nx.to_numpy_array(G)
        A = np.dot(U, W)
        M = tau * A + np.identity(len(A))

        # Define the generalized differential equations and the Jacobian
        f = lambda t, x: x - np.dot(np.dot(np.diag(x), M), x)
        J = lambda t, x: np.identity(len(A)) - \
            (np.dot(np.diag(x), M) + np.diag(np.dot(M, x)))

        # Set up the ODE solver
        ode_solver = integrate.ode(f, J)
        ode_solver.set_integrator('dopri5')

        # Stop integration when all nodes converge to 0 or 1
        def solout(t, x):
            return -1 if np.all((x < 1e-5) | (x > 1 - 1e-5)) else 0

        ode_solver.set_solout(solout)

        # Integrate until binary output
        while True:
            t_end = 1e9
            ode_solver.set_initial_value(x0, 0)
            y = ode_solver.integrate(t_end)

            if solout(t_end, y) == -1:
                break

            x0 = y

        y = np.transpose(y)

        # Determine variables that converge to one
        nodes_arr = list(G.nodes())
        max_independent_set = \
            [nodes_arr[i] for i in range(len(G)) if y[i] > 1 - 1e-5]
```

```
    if is_maximal_independent_set(G, max_independent_set):
        return max_independent_set
    else:
        print('The algorithm did not return MIS.')
```

B.3 Code for Figures in Chapter 4

All the Figures of Chapter 4 result from the following code.

```
# For Figure 4.1a

nodes=[0,1,2,3]
edges=[(0,1),(1,3),(3,2),(2,0)]

G=nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)

w=np.array([1,1,4,1])
x0=np.array([0.1,0.1,0.1,0.1])
tau=1.1

Set=Carrying_Capacity(G, w, x0, tau)
visualize_graph(G,w,Set)


# For Figure 4.1b

nodes=[0,1,2,3,4]
edges=[(0,1),(1,3),(3,2),(2,0),(4,2)]

G=nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)

w=np.array([1,1,4,1,1])
x0=np.array([0.1,0.1,0.1,0.1,0.1])
tau=1.1

Set=Carrying_Capacity(G, w, x0, tau)
visualize_graph(G,w,Set)
```

```
# For Figure 4.2

nodes=[0,1,2,3]
edges=[(0,1),(1,3),(3,2),(2,0)]

G=nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)

w=np.array([1,2,2,1])
x0_a=np.array([0.1,0.1,0.1,0.1])
x0_b=np.array([0.2,0.1,0.1,0.1])
tau=1.1

Set=Carrying_Capacity(G, w, x0_a, tau)
visualize_graph(G,w,Set)

Set=Carrying_Capacity(G, w, x0_b, tau)
visualize_graph(G,w,Set)


# For Figure 4.3 and 4.6

nodes=[0,1,2,3,4,5]
edges=[(0,1),(2,0),(0,3),(3,4),(3,5),(1,2),(4,5),(2,4),(1,5)]

G=nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)

com_U=nx.complement(G)

w=np.array([4,2,2,2,1,1])
x0=np.array([0.1,0.1,0.1,0.1,0.1,0.1])
tau=1.1

Set=Carrying_Capacity(G, w, x0, tau)
visualize_graph(G,w,Set)

Set=Competition(G, w, x0, tau)
visualize_graph(G,w,Set)


# For Figure 4.4a
```

```
nodes=[0,1,2,3,4]
edges=[(0,1),(1,3),(3,2),(2,0),(4,2)]

G=nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)

w=np.array([1,1,4,1,1])
x0=np.array([0.1,0.1,0.1,0.1,0.1])
tau=1.1

Set=Competition(G, w, x0, tau)
visualize_graph(G,w,Set)


# For Figure 4.4b

nodes=[0,1,2,3,4,5]
edges=[(0,1),(1,3),(3,2),(2,0),(4,2),(5,2)]

G=nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)

w=np.array([1,1,4,1,1,1])
x0=np.array([0.1,0.1,0.1,0.1,0.1,0.1])
tau=1.1

Set=Competition(G, w, x0, tau)
visualize_graph(G,w,Set)


# For Figure 4.4c

nodes=[0,1,2,3,4,5,6]
edges=[(0,1),(1,3),(3,2),(2,0),(4,2),(5,2),(6,2)]

G=nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)

w=np.array([1,1,4,1,1,1,1])
x0=np.array([0.1,0.1,0.1,0.1,0.1,0.1,0.1])
tau=1.1
```

```
Set=Competition(G, w, x0, tau)
visualize_graph(G,w,Set)


# For Figure 4.4d

nodes=[0,1,2,3,4,5,6,7]
edges=[(0,1),(1,3),(3,2),(2,0),(4,2),(5,2),(6,2),(7,2)]

G=nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)

w=np.array([1,1,4,1,1,1,1,1])
x0=np.array([0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1])
tau=1.1

Set=Competition(G, w, x0, tau)
visualize_graph(G,w,Set)


# For Figure 4.5

nodes=[0,1,2,3]
edges=[(0,1),(1,3),(3,2),(2,0)]

G=nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)

w=np.array([1,2,2,1])
x0_a=np.array([0.2,0.1,0.1,0.1])
x0_b=np.array([0.5,0.1,0.1,0.1])
x0_c=np.array([0.6,0.1,0.1,0.1])
tau=1.1

Set=Competition(G, w, x0_a, tau)
visualize_graph(G,w,Set)

Set=Competition(G, w, x0_b, tau)
visualize_graph(G,w,Set)

Set=Competition(G, w, x0_c, tau)
visualize_graph(G,w,Set)
```

B.4 The greedy Algorithm

This section includes code from implementing the greedy Algorithm which picks nodes in a graph based on their weight.

```
# Function Implementing the greedy Algorithm for Maximum Weighted Cliques

def greedy(G, weights):

    # Sort nodes by their weights in descending order
    nodes_sorted_by_weight = \
        sorted(G.nodes, key=lambda node: weights[node-1], reverse=True)

    # Add nodes to a clique by the greedy method
    clique = []

    for node in nodes_sorted_by_weight:
        # Check if adding the node to the current clique keeps it a clique
        if is_clique(G, clique + [node]):
            clique.append(node)

    return clique
```

B.5 Application on Molecular Docking

This section includes the code for applying the different methods to molecular docking. The output of this code is a data frame which includes the results of the different methods for graphs from 'Docking weighted graphs' [24].

```
import pandas as pd
import time
import os


# Defining a DataFrame to store the results

columns=['File_Name','Amount_Nodes','Graph','Complement_Graph',/
'LV_Weight','LV_Time','CarryingCapacity_Weight','CarryingCapacity_Time',/
'Competition_Weight','Competition_Time','greedy_Weight','greedy_Time']
Results = pd.DataFrame(data=None, columns=columns)



# Read the file

folder_path = './docking_weighted_graphs'
```

```
csv_files = sorted([f for f in os.listdir(folder_path) if \
    f.endswith('.int2')], key=lambda f: \
    os.path.getsize(os.path.join(folder_path, f)))

for file in csv_files:
    file_path = os.path.join(folder_path, file)

    df = pd.read_csv(file_path, delimiter=" ")
    print(f"Reading {file}")

    column_names = df.columns
    Amount_Nodes=column_names[2]
    print('The number of nodes is '+str(column_names[2]))



#Define the Graph

    nodes=[]
    edges=[]
    weights=np.array([])

    for i in range(0, len(df)):
        if df['p'][i]=='n':
            nodes.append(df['edge'][i])
            weights=np.append(weights,int(df[column_names[2]][i]))
        else:
            edges.append((df['edge'][i],int(df[column_names[2]][i])))

    G=nx.Graph()
    G.add_nodes_from(nodes)
    G.add_edges_from(edges)

    com_G=nx.complement(G)



# Check that Graph is connected

    if nx.is_connected(G)==False:
        Graph= 'Not Connected'
    else:
        Graph='Connected'
    if nx.is_connected(com_G)==False:
```

```
        Complement_Graph='Not Connected'
    else:
        Complement_Graph='Connected'



# Define helpful variables

    new_weights=weights/1000
    weight_proportional_initial_condition=0.01*new_weights
    tau=1.1



# Initialize variables for the Results

    LV_Weight=0
    CarryingCapacity_Weight=0
    Competition_Weight=0
    greedy_Weight=0

    LV_Time=0
    CarryingCapacity_Time=0
    Competition_Time=0
    greedy_Time=0

    Set_LV=[]
    Set_CarryingCapacity=[]
    Set_Competition=[]
    Set_greedy=[]



# The Lotka-Volterra Algorithm

    t0=time.time()
    for U in divide_into_subgraphs(com_G):
        w = np.array([new_weights[node-1] for node in U.nodes()])
        x0 = [weight_proportional_initial_condition[node-1] \
            for node in U.nodes()]
        Set_LV.extend(Competition(U,np.ones(len(x0)),x0,tau))
    t1=time.time()

    LV_Weight=SetWeight(weights,Set_LV)
    LV_Time=(t1-t0)
```

```
# The Carrying Capacity Method

    for U in divide_into_subgraphs(com_G):
        w = np.array([new_weights[node-1] for node in U.nodes()])
        x0 = [weight_proportional_initial_condition[node-1] \
            for node in U.nodes()]
        Set_CarryingCapacity.extend(Carrying_Capacity(U,w,x0,tau))
    t2=time.time()

    CarryingCapacity_Weight=SetWeight(weights,Set_CarryingCapacity)
    CarryingCapacity_Time=(t2-t1)



# The Competition Method

    for U in divide_into_subgraphs(com_G):
        w = np.array([new_weights[node-1] for node in U.nodes()])
        x0 = [weight_proportional_initial_condition[node-1] \
            for node in U.nodes()]
        Set_Competition.extend(Competition(U,w,x0,tau))
    t3=time.time()=

    Competition_Weight=SetWeight(weights,Set_Competition)
    Competition_Time=(t3-t2)



# The greedy Algorithm

    Set_greedy.extend(greedy(G,new_weights))
    t4=time.time()

    greedy_Weight=SetWeight(weights,Set_greedy)
    greedy_Time=(t4-t3)



# Build and Return Data Frame

    data=[file,Amount_Nodes,Graph,Complement_Graph,LV_Weight,LV_Time,\
        CarryingCapacity_Weight,CarryingCapacity_Time,Competition_Weight,\
```

```
        Competition_Time,greedy_Weight,greedy_Time]

fn='Results.xlsx'
Results.to_excel(fn)
```

Appendix C

Tables

In this appendix, we include all the tables with the results of the implementation described in Chapter 6.

Here we use the following abbreviations:

- LV: Lotka–Volterra method
- CC: carrying capacity method
- Com: competition method

The files mentioned in the following tables (under 'File Name') are taken from the 'Docking weighted graphs' of [24].

Note that the runtime of each method seen in the following tables is given in seconds. Additionally, if the runtime of a method is less than 0.0008 seconds, the code returns a runtime of zero.

File Name	# Nodes	LV Weight	LV Time	CC Weight	CC Time	Com Weight	Com Time	greedy Weight	greedy Time
CHEMBL522271_1s3bA_clq1.weighted_graph.int2	17	78997	0.007	78997	0.022	78997	0.02	78997	0
CHEMBL82293_2nnqA_clq1.weighted_graph.int2	106	40900	0.522	40900	0.572	40900	0.434	40900	0
CHEMBL169926_3bwmA_clq1.weighted_graph.int2	139	78309	1.316	78309	1.235	78309	1.047	78309	0
CHEMBL115510_2v3fA_clq1.weighted_graph.int2	170	75562	1.91	75562	1.946	75562	1.44	75562	0
CHEMBL212412_3nf7A_clq2.weighted_graph.int2	233	79574	0.97	70556	2.145	79574	1.93	75387	0
CHEMBL159758_3bgsA_clq2.weighted_graph.int2	237	82040	0.8	82040	2.79	70338	2.167	106560	0
CHEMBL8659_2nnqA_clq2.weighted_graph.int2	243	41692	0.967	41692	1.78	57905	2.094	71324	0
CHEMBL175362_3bgsA_clq2.weighted_graph.int2	293	110947	1.9	110947	7.42	120719	5.35	96256	0
CHEMBL43134_3bgsA_clq2.weighted_graph.int2	295	138989	1.53	138989	7.11	138989	6.21	181928	0.007
CHEMBL274669_2v3fA_clq3.weighted_graph.int2	316	84163	2.17	84163	5.544	87154	4.193	100834	0.8
CHEMBL274513_1s3bA_clq3.weighted_graph.int2	324	119254	2.299	92147	5.699	178397	16.944	179052	0
CHEMBL10131_2v3fA_clq3.weighted_graph.int2	329	74627	6.75	74627	17.01	95702	13.69	99269	0
CHEMBL229957_2oyuP_clq1.weighted_graph.int2	339	107942	7.9	107942	6.96	107942	4.78	107942	0
CHEMBL366356_3nf7A_clq2.weighted_graph.int2	347	79310	1.86	79310	5.21	79310	4.93	73167	0
CHEMBL32865_3nf7A_clq3.weighted_graph.int2	356	65041	2.51	65041	4.78	73142	4.085	89809	0
CHEMBL168389_3bwmA_clq2.weighted_graph.int2	374	69844	3.44	69844	8.19	71595	7.38	72433	0.001
CHEMBL196391_3nf7A_clq3.weighted_graph.int2	408	85106	9.87	85106	27.6	85106	19.53	143732	0
CHEMBL284846_1li4A_clq2.weighted_graph.int2	413	153589	3.96	153589	20.63	153589	16.4	166024	0
CHEMBL193759_3nf7A_clq3.weighted_graph.int2	419	83873	5.61	89565	12.821	86688	8.47	119451	0.008
CHEMBL228561_2oyuP_clq1.weighted_graph.int2	423	103344	20.81	103344	13.53	103344	10.6	103344	0
CHEMBL1393_2aa2A_clq2.weighted_graph.int2	453	165264	6.2	165264	28.49	170230	27.95	227238	0
CHEMBL607632_1li4A_clq3.weighted_graph.int2	476	168820	6.043	168820	33.14	186130	22.005	199802	0
CHEMBL187566_3l3mA_clq1.weighted_graph.int2	477	140223	43.98	140223	23.61	140223	17.32	140223	0.0075
CHEMBL1213968_1s3bA_clq3.weighted_graph.int2	480	145702	5.55	145702	43.37	158487	29.2	131381	0.008
CHEMBL372171_2b8tA_clq2.weighted_graph.int2	504	24247	5.92	30314	8.62	79156	14.72	67473	0
CHEMBL610384_1li4A_clq3.weighted_graph.int2	519	185140	8.3	185140	47.53	185140	42.67	180930	0.007
CHEMBL425913_2b8tA_clq2.weighted_graph.int2	520	19211	11.62	19211	15.434	63081	20.31	50898	0
CHEMBL1393_2am9A_clq2.weighted_graph.int2	534	153686	6.26	153686	39.19	166350	31.36	237699	0
CHEMBL6497_2b8tA_clq2.weighted_graph.int2	568	66757	9.34	72391	21.84	140757	31.11	140757	0
CHEMBL259330_2v3fA_clq5.weighted_graph.int2	569	82033	14.37	85675	39.11	98458	27.45	90537	0.008

Table C.1: Table including the total weight of the output and the computing time of each method (Lotka–Volterra, carrying capacity, competition and greedy) applied to the graph given by the named file.

File Name	# Nodes	LV Weight	LV Time	CC Weight	CC Time	Com Weight	Com Time	greedy Weight	greedy Time
CHEMBL357646_3nf7A_clq6.weighted_graph.int2	609	148843	15.95	153094	89.56	153094	70.08	147252	0.008
CHEMBL311091_3l3mA_clq1.weighted_graph.int2	615	129255	109.34	129255	55.28	129255	34.33	129255	0
CHEMBL278336_3ln1A_clq2.weighted_graph.int2	626	78137	10.94	78137	30.03	129178	33.73	122034	0
CHEMBL307647_3ny8A_clq2.weighted_graph.int2	645	62561	14.43	62561	34.17	106925	39.71	98276	0
CHEMBL1091982_3bwmA_clq4.weighted_graph.int2	659	82917	26.47	92818	48.847	97354	59.167	119702	0.008
CHEMBL307647_2vt4B_clq2.weighted_graph.int2	682	60651	14.58	60651	34.14	71518	34.05	119601	0
CHEMBL41799_1d3gA_clq3.weighted_graph.int2	688	145426	22.42	194008	108.33	194008	112.74	202565	0
CHEMBL960_1d3gA_clq3.weighted_graph.int2	694	134347	18.87	134247	108.91	150520	110.45	159458	0
CHEMBL459902_2nnqA_clq2.weighted_graph.int2	706	117997	16.61	130698	83.91	130698	140.89	175045	0
CHEMBL151125_1s3bA_clq7.weighted_graph.int2	709	210865	33.83	210865	155.43	210865	127.75	144302	0.008
CHEMBL288381_1d3gA_clq3.weighted_graph.int2	713	138404	20.3	184796	103.23	190828	98.31	202141	0
CHEMBL361708_3eqhA_clq1.weighted_graph.int2	718	101679	197.14	101679	65.46	101679	47.68	101679	0
CHEMBL51927_1d3gA_clq3.weighted_graph.int2	767	148046	24.83	172412	120.99	191379	131.164	171053	0.008
CHEMBL164976_1b9vA_clq4.weighted_graph.int2	769	182496	33.51	182859	147.46	195575	187.27	210865	0.009
CHEMBL99718_3ny8A_clq6.weighted_graph.int2	776	149273	41.64	149273	145.34	164410	103.59	190679	0.008
CHEMBL1393_3bqdA_clq2.weighted_graph.int2	781	163275	18.74	163275	98.84	163275	77.86	226574	0
CHEMBL48928_1d3gA_clq3.weighted_graph.int2	785	207749	27.8	207749	145.97	207749	163.92	164971	0.00091
CHEMBL320788_1b9vA_clq4.weighted_graph.int2	792	198215	35.9	197964	196.38	198215	172.74	194317	0.007
CHEMBL139461_1c8kA_clq2.weighted_graph.int2	795	61010	24.46	58847	51.89	80357	53.57	94150	0
CHEMBL99718_2vt4B_clq6.weighted_graph.int2	798	148912	37.48	115239	87.25	159743	189.02	153198	0.008
CHEMBL48531_1d3gA_clq3.weighted_graph.int2	799	237797	28.02	237797	194.95	237797	173.88	237797	0
CHEMBL221647_3hl5A_clq2.weighted_graph.int2	829	102002	22.87	102002	86.92	111725	76.26	129320	0
CHEMBL64396_2oyuP_clq2.weighted_graph.int2	919	97057	29.88	100898	98.54	143131	143.98	178226	0
CHEMBL370228_1d3gA_clq3.weighted_graph.int2	927	122239	33.368	182677	215.55	191259	243.52	211809	0.008
CHEMBL81536_2hv5A_clq1.weighted_graph.int2	965	115634	461.06	115634	158.27	115634	116.84	115634	0
CHEMBL12594_3ln1A_clq3.weighted_graph.int2	992	56996	41.99	56996	78.5	60379	76.07	135172	0
CHEMBL309415_1njsA_clq5.weighted_graph.int2	1012	177369	70.87	157654	213.39	208133	216.17	123038	0.008
CHEMBL138065_2hv5A_clq2.weighted_graph.int2	1018	103234	68.97	103234	170.19	128641	154.17	166631	0
CHEMBL1077990_2nnqA_clq3.weighted_graph.int2	1033	112633	68.64	112633	226.31	112633	158.29	166647	0.008
CHEMBL258191_2aa2A_clq2.weighted_graph.int2	1040	169687	45.26	169687	240.34	171919	175.41	201870	0

Table C.2: Table including the total weight of the output and the computing time of each method (Lotka–Volterra, carrying capacity, competition and greedy) applied to the graph given by the named file.

File Name	# Nodes	LV Weight	LV Time	CC Weight	CC Time	Com Weight	Com Time	greedy Weight	greedy Time
CHEMBL154623_1d3gA_clq4.weighted_graph.int2	1071	186047	282.39	132094	217.411	169180	263.42	221653	0.009
CHEMBL358216_1r9oA_clq2.weighted_graph.int2	1081	71339	51.84	71339	141.27	94199	141.66	102465	0
CHEMBL414824_2oyuP_clq3.weighted_graph.int2	1093	113489	75.64	113489	312.01	113489	293.94	165251	0.008
CHEMBL307602_3ln1A_clq3.weighted_graph.int2	1106	80954	94.88	165151	429.88	165151	310.119	158605	0.007
CHEMBL186380_3l3mA_clq2.weighted_graph.int2	1138	205246	60.99	205246	416.13	205246	340.11	191980	0
CHEMBL8381_2hv5A_clq2.weighted_graph.int2	1156	105619	55.98	108084	200.53	122123	172.26	133094	0.008
CHEMBL64396_2oyuP_clq3.weighted_graph.int2	1184	127801	72.83	127801	266.02	153386	272.96	194917	0.008
CHEMBL102179_3kbaA_clq2.weighted_graph.int2	1226	149780	134.45	149780	560.62	160713	499.42	213815	0
CHEMBL214784_1r9oA_clq2.weighted_graph.int2	1236	86799	88.35	86799	267.21	86799	210.39	107572	0.008
CHEMBL99369_1zw5A_clq2.weighted_graph.int2	1321	115675	78,932	115675	285.14	85218	194.65	134064	0
CHEMBL1214838_2aa2A_clq3.weighted_graph.int2	1322	215221	107.44	215221	573.02	244513	640.76	249057	0.007
CHEMBL277281_1vsoA_clq1.weighted_graph.int2	1323	146460	1895.13	146460	500.44	146460	327.39	146460	0
CHEMBL1223598_3pblA_clq5.weighted_graph.int2	1332	131769	134.41	93258	290.83	97582	220.61	131008	0.017
CHEMBL104592_2oyuP_clq4.weighted_graph.int2	1342	138924	311.23	167468	790.73	171671	1099	148597	0.009
CHEMBL77851_2fszA_clq1.weighted_graph.int2	1367	168164	1791.67	168164	611.24	168164	425.37	168164	0
CHEMBL241027_3pblA_clq5.weighted_graph.int2	1404	202591	180.4707403	202591	818.5670941	213536	1157.179631	191173	0.033998728
CHEMBL441663_2oyuP_clq3.weighted_graph.int2	1446	145684	156.39	145684	635.17	151638	485.52	177162	0.008
CHEMBL241455_3pblA_clq5.weighted_graph.int2	1478	144197	359.4183049	151051	1534.34986	175970	2191.453062	120259	0.010999918
CHEMBL394019_3pblA_clq5.weighted_graph.int2	1493	189221	288.0558505	190767	904.247247	225169	1002.79525	166927	0.022577286
CHEMBL436848_3bqdA_clq2.weighted_graph.int2	1549	157487	139.19	157487	678.42	157487	623.91	214321	0.0064
CHEMBL203094_1j4hA_clq2.weighted_graph.int2	1560	197360	276.52	197360	1362.67	197360	1160.42	209383	0
CHEMBL113762_1c8kA_clq5.weighted_graph.int2	1573	76780	230.24	77639	389.01	108657	447.82	149328	0.0254
CHEMBL100154_1zw5A_clq2.weighted_graph.int2	1582	125741	193.69	125741	1226.19	134556	1532.12	120702	0.008
CHEMBL280805_3kbaA_clq2.weighted_graph.int2	1589	119075	163.52	119075	655.59	119075	509.11	173590	0.008
CHEMBL115395_1c8kA_clq5.weighted_graph.int2	1597	99027	284.37	99027	785.73	95892	690.4	145603	0.02
CHEMBL108441_2azrA_clq1.weighted_graph.int2	1620	75799	2449.93	75799	608.57	75799	399.81	75799	0.008
CHEMBL1209236_2am9A_clq3.weighted_graph.int2	1656	158389	212.67	158389	990.99	181188	1114.74	132591	0.007
CHEMBL474268_1e66A_clq2.weighted_graph.int2	1895	201101	446.86	201101	2218.81	227558	2312.96	236760	0.009
CHEMBL1082743_1sj0A_clq2.weighted_graph.int2	1907	117295	382.5	118124	1798	117295	1343	196832	0.008
CHEMBL1083065_1e66A_clq3.weighted_graph.int2	2230	175342	748.03	175342	3103	175342	3420.63	216217	0.02423

Table C.3: Table including the total weight of the output and the computing time of each method (Lotka–Volterra, carrying capacity, competition and greedy) applied to the graph given by the named file.

Bibliography

[1] Cook, S. (1971). "The complexity of theorem proving procedures". Proceedings of the Third Annual ACM Symposium on Theory of Computing. pp. 151–158. doi:10.1145/800157.805047. ISBN 9781450374644. S2CID 7573663.

[2] Hoffman, K. L., Padberg, M., & Rinaldi, G. (2013). Traveling salesman problem. In Springer eBooks (pp. 1573–1578). https://doi.org/10.1007/978-1-4419-1153-7_1068

[3] Garey, M. R., Johnson, D. S., & Bell Laboratories. (1979). Computers And Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company. https://bohr.wlu.ca/hfan/cp412/references/ChapterOne.pdf

[4] Butenko, S. and Pardalos P. M.(2003). Maximum independent set and related problems, with applications [Ph.D. Dissertation]. University of Florida.

[5] Pardalos, P., & Xue, J. (1994). The Maximum Clique Problem. Journal of Global Optimization, pages 301-328.

[6] Szabó, S. (2021). A Clique Search Problem and its Application to Machine Scheduling. SN Operations Research Forum. https://doi.org/10.1007/s43069-021-00111-x

[7] Butenko, S., & Wilhelm, W. (2006). Clique-detection models in computational biochemistry and genomics. European Journal of Operational Research, pages 1–17. https://doi.org/10.1016/j.ejor.2005.05.026

[8] Banchi, L., Fingerhuth, M., Babej, T., Ing, C., & Arrazola, J. M. (2020). Molecular docking with Gaussian Boson Sampling. Science Advances, 6(23). https://doi.org/10.1126/sciadv.aax1950

[9] Östergård, P. R. (1999). A new algorithm for the Maximum-Weight clique problem. Electronic Notes in Discrete Mathematics, 3, 153–156. https://doi.org/10.1016/s1571-0653(05)80045-9

[10] Babel, L. (1994). A fast algorithm for the maximum weight clique problem. Computing, 52(1), 31–38. https://doi.org/10.1007/bf02243394

[11] Pardalos, P. M., Rappe, J., & Resende, M. G. C. (1997). An exact parallel algorithm for the maximum clique problem.

[12] Mannino, C. (1999). An augmentation algorithm for the maximum weighted stable set problem. Computational Optimization and Applications, 14(3), 367–381. https://doi.org/10.1023/a:1026456624746

[13] Stix, V., Pelillo, M., & Bomze, I. (2000). Approximating the maximum weight clique using replicator dynamics. IEEE Transactions on Neural Networks, 11(6), 1228–1241. https://doi.org/10.1109/72.883403

[14] Singh, A., & Gupta, A. K. (2006). A hybrid evolutionary approach to maximum weight clique problem. https://www.semanticscholar.org/paper/A-Hybrid-Evolutionary-Approach-to-Maximum-Weight-Singh-Gupta/fd2821ef0b4ed3471997751a49eec26751baa21c

[15] Mooij, N. (2022). Generating maximal independent sets using Lotka-Volterra dynamics [Master Thesis, Utrecht Univercity]. https://studenttheses.uu.nl/handle/20.500.12932/41651

[16] Griffen, B. D., & Drake, J. M. (2008). Effects of habitat quality and size on extinction in experimental populations. Proceedings - Royal Society. Biological Sciences/Proceedings - Royal Society. Biological Sciences, 275(1648), 2251–2256. https://doi.org/10.1098/rspb.2008.0518

[17] Murray, J. D. (1995). Mathematical Biology (Biomathematics, Vol 19) (Second, Corrected). Springer

[18] Hofbauer, J., & Sigmund, K. (1998). Evolutionary games and population dynamics. https://doi.org/10.1017/cbo9781139173179

[19] Braun, M. (1993). Differential Equations and Their Applications: An Introduction to Applied Mathematics. Springer.

[20] Brendon, G. E. (1993). Topology and Geometry. In Graduate Texts in Mathematics (139). Springer.

[21] Prussing, J.E. (1986). The principal minor test for semidefinite matrices. Journal of Guidance Control and Dynamics, 9, 121-122.

[22] Perko, L. (2001). Differential equations and dynamical systems. In Texts in applied mathematics (Vol. 7). Springer. https://doi.org/10.1007/978-1-4613-0003-8

[23] M. Chaudhary, K. Tyagi, A review on molecular docking and its applications. (2024). ResearchGate.

https://www.researchgate.net/publication/379938962_A_REVIEW_ON_MOLECULAR_DOCKING_AND_ITS_APPLICATION

[24] Rozman, K., Ghysels, A., Zavalnij, B., Kunej, T., Bren, U., Janežič, D., & Konc, J. (2024). Enhanced Molecular Docking: Novel algorithm for identifying highest weight K-Cliques in weighted General and Protein-Ligand graphs. Journal of Molecular Structure, 1304, 137639. https://doi.org/10.1016/j.molstruc.2024.137639