University of Bergen
Department of Informatics

Vaccination strategies based on graph centrality

Author: Haakon Osmundsen Benning

Supervisors: Fredrik Manne

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

November, 2022

Abstract

This thesis compares the ability of different vaccination strategies in limiting the peak infected, on a model for disease spread based on the SIR model, where we can set the underlying graph. We tested strategies based on Betweenness centrality, closeness centrality, and different k-neighbourhood based centralities. For the graphs, we used random, small world, and geometric graphs. We show that the diameter of the graphs is important for which strategies perform good and show that the strategies that considers the state of nodes (immune, infected, etc.) performs better than the ones that do not. We also give fast algorithms for calculating the k-neighbourhood and betweenness centrality on small sparse graphs.

Acknowledgments

I would like to thank my supervisor Fredrik Manne for his great advice and guidance on my thesis.

Haakon Osmundsen Benning
15 November, 2022

Contents

1 Introduction

March 2020 was the start of the covid-19 pandemic, now, two years later, the virus has lost its pandemic status in most countries. In accomplishing this, vaccines where the most important factor. But because of their limited supply, countries made different choices of how they should be distributed. In Norway and many other countries healthcare workers got priority, as they are both more exposed to the pandemic, and important for the maintaining the capacity of the healthcare system. The older age groups were also prioritized, as these have a higher likelihood of serious symptoms and death. While both groups where prioritized by most countries, other exposed groups like teachers where not always prioritized. During the pandemic there were those who argued that young people should be prioritized for vaccines as they were more likely to spread the virus since they tend to have more contacts. Now mutations of the virus give rise to new vaccines and ensuing discussions on how these should be distributed.

Thus, it is interesting to see how different vaccination strategies performs. In this thesis we investigate and evaluate different ways of allocating vaccines. There are different ways to measure how a strategy performs, we have chosen to use the peak number of infected. This is because one can assume that in a pandemic, everyone gets infected, and since the number of hospitalizations is proportional to the number of infected, minimizing the number of infected at any moment also minimizes the load on the healthcare system.

There are different ways of modelling pandemics, one of these is the SIR model introduced by of Kermack and McKendrick, where an individual can be in one of three groups, S – Susceptible, I – Infectious or R – Recovered/Resistant, and there is some chance for an individual to move from one group to the next (from S to I and from I to R). There are other models based on the same concept, such as SIS or SIRD. In general, this kind of model is called a compartmental model. All of these have an expression in the form of a differential equation, but this assumes a homogeneous mixing of the population, and thus does not allow for the ability to change the structure of the population or try different strategies based on structure. To circumvent this, one can use a graph for representing a population and their connection, as an underlying structure. Doing this gives the ability to test more complex population structures, and different strategies for vaccination/isolation.

For some intuition behind the strategies, we look at two examples, super spreaders and bridge-nodes. For super spreaders, that is vertices with high degree, it is easy to see that removing these can result in lowering the rate of spread, as if one of these vertices become infected there is a high probability that the neighbouring vertices will also become infected. As for bridge-nodes, that is a vertex whose removal would split the graph into two or more components. Removing these effectively isolates the components where there are no already infected vertices. Thus, one can isolate the spread to a smaller part of the graph.

Some of the strategies we have looked at are vaccinating based on the degree of vertices, the size of different k-neighbourhoods, betweenness centrality, closeness centrality, and random.

The graphs used are random connected graphs, small world graphs and geometric graphs. A problem with using many of these strategies in the real world is that it can be difficult to collect enough data to calculate the needed measures, where one of the most important reasons is privacy. For example, in the calculation of betweenness centrality one needs to know the whole graph. This is totally unrealistic as there is already concern about the amount of data existing apps made for limiting spread collects. Some of the centralities used has known approximations, for example betweenness centrality ((Riondato & Kornaropoulos, 2014), (Bader, Kintali, Madduri , & Mihail , 2007)), although these approximations use shortest path sampling, and still needs most of the graph structure. Approximations for some of the centralities could perform almost as well as what they approximate and could also elevate some of the privacy concern.

We will show the potential of some strategies to limit the peak.

One of the results we got is that for random and small world graphs, since the graph diameter is small ($\frac{\log(n)}{\log(n \cdot p)}$ for random, and $\log(n)$ for small world, with n number of vertices and p chance of an edge existing), it is unlikely that any strategy performs much better than random, at least using a realistic number of vaccines. While when we increase the diameter of the graphs using geometric graphs, we see a larger difference between strategies.

In Chapter 2 (Graphs), we define the graph notation that is used for this thesis. We introduce the different types of graphs that are used, some properties of these different types of graphs, and how the graphs are generated.

In Chapter 3 (Centralities), we introduce the centrality measures that are used, and how these are calculated. For the calculation of the centralities, we look at known algorithms for calculating the centralities, and introduce and compare these to our own algorithms, and show that our algorithms are faster on small and sparse graphs.

In Chapter 4 (Simulation), we introduce our implementation of the SIR model (with vaccinations). We show the parameters of the model and the vaccination strategies that are implemented based on the centralities from Chapter 3. We also go into the details of how the simulations are run in parallel, and some of the challenges around this.

In Chapter 5 (Results), we show the results from the different runs of the model from Chapter 4.

2 Graphs

2.1 Definitions

In the following we define the graph notation that is used in this thesis.

A graph $G(V, E)$ is a structure where V is the set of vertices and E the set of edges, with $|V| = n$ and $|E| = m$. Here $V = \{1,2, \cdots, n\}$ and an edge $e = (u, v)$ where $u, v \in V$, represents a connection between u and v. Two vertices u and v are neighbours if $(u, v) \in E$. G is an undirected graph, that is if u has v as a neighbour, v also has u as a neighbour. Thus (u, v) and (v, u) denotes the same edge.

An ordered sequence of vertices $P = \{v_1, v_2, \cdots, v_l\}$ is a path, if all consecutive vertices v_i and v_{i+1} are neighbours, and all vertices are unique, that is $v_i \neq v_j$ for all $v_i, v_j \in P, i \neq j$. The length of path P is $|P| - 1$, and the distance $dist(u, v)$ between to vertices u and v, is the length of the shortest path between v and u. If there is no path between v and u we let $dist(v, u) = \infty$. Also $dist(v, v) = 0$.

We call graph G connected if for every pair of vertices $v_i, v_j \in V$ there exists a path from v_i to v_j. If G is not connected, we can divide V into maximal subsets $C = \{c_1, c_2, \cdots, c_k\}$ such that each c_i is connected. We call C the connected components of G.

The neighbourhood of a vertex v, $N(v)$, is the set of all vertices that are neighbours of v, and the degree of v, $deg(v) = |N(v)|$, is the size of the neighbourhood of v. Let $deg_{max}(G)$ be the maximal degree of the vertices in the graph, i.e., $\max\{deg(v) | v \in V\}$.

The k-neighbourhood $N(k, v)$ of vertex v, is the set of vertices with distance at most k from v, that is $N(k, v) = \{u \in V \mid dist(v, u) \leq k\}$. Note that $N(1, v) = N(v) \cup \{v\}$. We also denote $N(1, v)$ by $N[v]$. We let $\eta(k) = \max_{v \in V} |N(k, v)|$ be the largest k-neighbourhood in the graph.

The eccentricity $\epsilon(v)$ of a vertex v is the length of a longest shortest path starting from v, i.e., $\epsilon(v) = \max_{u \in V} dist(v, u)$. The diameter, $diam(G)$, of graph G is the largest eccentricity over all vertices, i.e., $diam(G) = \max_{v \in V} \epsilon(v)$.

Let $L(k, v)$ be the k'th distance layer of vertex v, i.e., all vertices with distance exactly k from v, $L(k, v) = \{u \in V \mid dist(v, u) = k\}$. We define the density of a graph G as $dens(G) = \frac{m}{n}$. Let $BW(v)$ be the betweenness centrality and $CL(v)$ be the closeness centrality of vertex v, these are defined in Chapter 3.1.

The following table gives a summary of the graph notation.

Notation	Explanation

G	A graph
V	The set of vertices of G
E	The set of edges of G
n	$\|V\|$, the number of vertices in G
m	$\|E\|$, the number of edges in G
$dist(v,u)$	The distance between vertices v and u
$deg(v)$	The degree of v
$dens(G)$	The density of $G, \frac{m}{n}$
$N(v)$	The neighbourhood of v
$N(k,v)$	The k-neighbourhood of v
$\epsilon(v)$	The eccentricity of v
$diam(G)$	The diameter of graph G
$L(k,v)$	The k'th distance layer of v
$deg_{max}(G)$	The maximum degree of any vertex in G
$BW(v)$	The betweenness centrality of v, (defined in Chapter 3.1)
$CL(v)$	The closeness centrality of v, (defined in Chapter 3.1)

2.2 Graphs

We use different types of synthetically generated graphs in our experiments. In the following we define each of these graph classes and explain how we have generated them. The three main graph types are *random connected graphs* (RND), *small world graphs* (SWG), and *geometric graphs* (GEO or GEO_d where d is the lower bound for the diameter). We want to keep the graphs connected since we use them to model a pandemic where a disease spreads between vertices (persons) along edges (interactions). Thus, a disease cannot spread from one component to another.

2.3 Random connected graphs

Random graphs, or Erdős-Rényi graphs, are graphs where every edge has an equal chance of existing. In the paper *On the evolution of random graphs*, Erdős and Rényi show different properties of these graphs like the binomial distribution of vertex degree, and a $\frac{\ln n}{n}$ threshold for connectedness, that is, if each edge has a probability of existing higher than this value one would expect the graph to be connected. The diameter of random graphs is most probably $\frac{\log(n)}{\log\left(\frac{2n\cdot dens}{n-1}\right)}$ (Chung & Lu, 2001) if we keep the graphs sparse.

We construct random connected graphs by starting with $V = \{1,2,\dots,n\}$ and $E = \emptyset$. Then we add edges until $|E| = dens \cdot n = m$. There are two types of edges that can be added, these are edges that connecting two vertices of different connected components, and edges connect two vertices of the same connected component. Since we want the graph to be connected, we ensure that $n-1$ edges connecting two different connected components are picked, as then the graph will be connected. We do this by using a union-find data structure to keep track of the connected components and generate edges at random. We add the edge if it is either an edge connecting two connected components or if the number of edges we have left to pick is more than or equal the number of connected components left. Since we cannot add an edge twice, we create a recursive function R that, given a random start seed, generates a non-repeating sequence of values in the range $[0, \frac{n(n-1)}{2} - 1]$, and use a function M that takes a number from R and return an edge. We use

$$R(0) = seed, \qquad R(x+1) = (R(x) + prime)\%\, m,$$

where $prime$ is some prime number larger than m, and $seed$ some initial seed in the range $[0, m-1]$ chosen at random. Let $R.next$ be the next number in the sequence, that is, if we have $R(x) = y$, then $R.next = R(x+1)$. And,

$M\colon \left[0, \frac{n(n-1)}{2} - 1\right] \to (a, b)$, with $a < b$.

$$M(x) = \left(x - \frac{\delta(x)^2 - \delta(x)}{2}, \delta(x)\right), \delta(x) = \left\lfloor \frac{1 + \sqrt{1 + 8x}}{2} \right\rfloor$$

We show how we get these functions in Appendix 2. As an example of what different values for x gives, $M(0) = (0,1)$, $M(1) = (0,2)$, $M(2) = (1,2)$, that is, the function first returns all edges ending in 1, then all edges ending in 2 etc, where the edges are not self-edges i.e., (v, v).

ALGORITHM FOR GENERATING RANDOM CONNECTED GRAPHS

```
    Input: integers $n, m$
    Output: the edge list of a random connected graph with $n$ vertices and $m$ edges
1   $UF : union\ find$
2   $edgelist : Array[(int, int)]$          //the edge list for the output graph
3   $R : RandomGenerator$                  //random generator as described
4   $while\ |edgelist| \leq m :$
5       $v1,\ v2 = M(R.next)$              //generate edge between vertices $v1$ and $v2$
6       $if\ UF.find(v1) != UF.find(v2) \lor m - edges \geq UF.components$:
7           $edgelist \leftarrow (v1, v2)$  //append the edge to the list
8           $UF.join(v1, v2)$               //join the components if $v1$ and $v2$ are different
                                           //components, else this does nothing
9   $return\ edgelist$
```

The time complexity of the algorithm, if we assume the graph is connected after trying to add $O(m)$ edges, is $O(m \cdot \alpha(m, n))$, where α is the inverse Ackerman function. This is achieved using path compression and union-by-rank for the union-find data structure (Tarjan & Van Leeuwen, 1984). Although there is a possibility of the graph not being connected after $O(m)$ steps, this is highly unlikely. If this is the case, the algorithm has a time complexity of $O\left(n^2 \cdot \alpha(n^2, n)\right)$, as all edges incident to some vertex can come at the end of the random numbers generated by R. Since we guarantee that the graph is connected, the graph is not chosen totally random, as randomness would include non-connected graphs. But as a random graph is most probably connected when $dens(G) > \frac{(n-1)\cdot \ln n}{2n}$, as long as we choose m large enough this should not affect the randomness.

2.4 Small world graphs

Small world graphs are useful when modelling social networks. Their vertex degree distribution follows a power law distribution (see Figure 2), meaning that some small set of the vertices have a high degree, while the others have much lower degree. The diameter of small world graphs is also small, being proportional to the logarithm of the number of vertices, $O(\log(n))$ (see Figure 1).

For generation of small world graphs, we use the Stanford Snap module (Leskovec & Sosic, 2016) for python, which generates graphs using the Watts-Strogatz model. The function takes three parameters, the size of the graph, the outdegree (density), and a rewiring probability. The rewiring probability is the chance of an edge being disconnected from one of its vertices and connected to some other vertex at random, this is what creates the "long" edges which gives the graph its small-world behaviour. We let the rewiring probability be 0.0505, which is in the middle of the range 0.001 and 0.1 where the small world property emerges (Menezes, Kim, & Huang, 2017).

2.5 Geometric graphs

Geometric graphs are model entities in the physical domain where there typically is an edge between two vertices if they are sufficiently close to each other. For our simulations we generate geometric graphs by distributing some number of points n in a 2D plane. Then if the Euclidian distance between two points, x and y is less than or equal to some threshold we add an edge between x and y with probability q. We can set the lower bound d for the diameter of our geometric graphs, which allows us to look at the effect diameter has on the algorithms in Chapter 3, and the effect on disease spread and how different vaccination strategies depend on diameter in Chapter 5.

The generation of geometric graphs where we want a lower bound on the diameter d is simple. First create a plane with sides of length 1 and width $\alpha \leq 1$. Then the diagonal of this plane has length $\sqrt{\alpha^2 + 1}$. If we divide the diagonal into d parts of length r, and we let these d parts be edges between $d + 1$ vertices, we will get a graph with a diameter of at least d. If some of these points is missing while the graph is still connected, then this will result in a graph with higher diameter than d.

To generate a geometric graph, we place n points at random positions in the plane. Next, we choose a radius $r = \frac{\sqrt{\alpha^2+1}}{d}$. Each point represents a vertex of the graph, and an edge exists between two points x and y if their Euclidean distance is $\leq r$. This gives the desired diameter, as the length of the diagonal of the plane is $\sqrt{\alpha^2+1}$, and dividing this into lengths of size r, gives d parts. If each of these parts represent an edge, we get the desired diameter. Although the diameter is not guaranteed since the random points can be generated in such a way that the two point furthest away from each other is significantly closer than the diagonal of the plane. We show in Figure 1 that this way of generating geometric graphs does give d as a lower bound for the diameter. The problem with this approach is that we have no control over how many edges there are in the graph. To have better control over the number of edges, we introduce a probability q of an edge existing, between two points within distance r of each other. This gives us the ability to have the three parameters n, m and d (number of vertices, number of edges, and minimum diameter) when generating the graphs.

We choose q the following way:

Let the area of the circle of radius r around a point be $C = \pi r^2$, and the point density of the plane be $D = \frac{n}{Area\ of\ plane} = \frac{n}{\alpha}$. Then $C \cdot D = \frac{n\pi r^2}{\alpha}$ is the estimated number of points within the radius r of a point. We want there to be $2 \cdot \frac{m}{n}$ edges incident to each vertex, and we also want there to be at least two points within each radius as this includes the point that we draw the radius from. From this we get $\gamma = 2 \cdot \frac{m}{n} + 1$ as the number of points we want within each radius. Let q be the chance of an edge existing. We then get $\gamma = \frac{n\pi r^2}{\alpha} \cdot q$ or $q = \frac{\gamma\alpha}{n\pi r^2}$. We also need some constraints for the input parameters.

We want r less than α and 1, since an r larger than any of these would always be outside the plane. Then $r \leq \alpha \leq 1$ gives the following constraint for d: $\sqrt{1 + \frac{1}{a^2}} \leq d$. We also want $0 < q \leq 1$, as q is a probability.

From this we get the following constraints: $\frac{2m+n}{n^2} \leq \frac{\pi(\alpha^2+1)}{\alpha d^2}$, $d \leq \sqrt{\frac{(\alpha^2+1)\pi n^2}{2m+n}}, and\ \alpha \leq \frac{z-\sqrt{z^2-4}}{2}$, where $z = \frac{d^2(2m+n)}{n^2\pi}$. In addition, for this additional constraint for α we fall back to the constraint of $\alpha \leq 1$ when $z^2 \leq 4$. In summary we get these constraints.

$$\sqrt{1 + \frac{1}{a^2}} < d < \sqrt{\frac{(\alpha^2+1)\pi n^2}{\alpha(2m+n)}}$$

$$\frac{2m+n}{n^2} \leq \frac{\pi(\alpha^2+1)}{\alpha d^2}$$

$$\alpha \leq \frac{z - \sqrt{z^2-4}}{2}$$

If these constraints are not met, we say the geometric graph does not exist. For example, the geometric graph with 1000 vertices, a density of 12, and a diameter of 128 does not exist.

Although we can choose α at will within the constraints, if we choose a value for α that is as high as possible, we get the plane to be as square-like as possible. We want to do this because when α is small, there is more overlap between the area of each of the circles formed by the points in the plane and the outside of the plane. One solution to this problem is to change the expression of the area of the plane from $1 \cdot \alpha$ to $\alpha + \pi r^2 + 2r\alpha + 2r$, that is, all the areas outside the plane that can overlap with the area of circles around the points. This has the effect of making the graph be closer to the desired number of edges but makes the graph structurally less like one with a higher value for α. Ignoring the overlap outside the plane gives a graph with a structure more like one of a higher value of α, and has the same effect as taking a $1 \times \alpha$ cut-out of a larger plane of points. We can see the effect of this decision in Figure 2 when comparing GEO_32 and GEO_128 when the density is 30. The α values for the two graphs are 1 and 0,31 respectively, and we can see that the number of vertices with a degree between 30 and 50 is significantly higher for GEO_128 than for GEO_32. For the same graph we also get slightly fewer edges than the aimed for density of 30. On average GEO_32 has 29,4 edges per vertex, and GEO_128 has 27,1 edges per vertex.

For the following algorithm for generating geometric graphs, we first check the constraints above and set a value for α, then generate n random points on the $1 \times \alpha$ plane. Then for all $1 \leq i < j \leq n$ if the Euclidian distance between points i and j is less than r we add the edge (i, j) to E with probability q.

ALGORITHM FOR GENERATING GEOMETRIC GRAPH

```
    Input: integers $n$, $m$, and $d$
    Output: the edge list of a geometric graph with $n$ vertices, $m$ edges and diameter of minimum $d$
1   $\boldsymbol{check\ constraints\ for\ n, m, and\ d}$
2   $\boldsymbol{points : Array[(float, float)]}$    //the array of all the points, indexed by their id
3   $\boldsymbol{edgelist : Array[(int, int)]}$    //the edge list for the output graph
4   $\boldsymbol{\alpha = \frac{z-\sqrt{z^2-4}}{2}\ or\ 1\ if\ z^2 \leq 4}$
5   $\boldsymbol{loop\ n\ times}$:
        $\boldsymbol{points \leftarrow (random, random \cdot \alpha)}$ //appends a new point to the array, where random gives a float between 0 and 1
7   $\boldsymbol{r = \sqrt{\alpha^2+1} \div d}$
8   $\boldsymbol{\gamma = 2 \cdot m \div n + 1}$
9   $\boldsymbol{q = \gamma \cdot \pi \cdot r \cdot r \cdot n \div \alpha}$
10  $\boldsymbol{for\ i :\ 1 \leq i \leq n-1}$:
11      $\boldsymbol{for\ j :\ i+1 \leq j \leq n}$:
12          $\boldsymbol{a, b = points[i], x, y = points[j]}$
13          $\boldsymbol{d1 = a - x, d2 = b - y}$
14          $\boldsymbol{if\ d1 \cdot d1 + d2 \cdot d2 \leq r \cdot r}$:
15              $\boldsymbol{add\ edge\ (i, j)\ to\ edgelist\ with\ probability\ q}$
16  $\boldsymbol{return\ edgelist}$
```

We only add edges when $i < j$ since q is the probability of an edge existing, and if we had looked at the edge twice (when $i > j$) we would have had to change q to compensate. Since the product $C \cdot D$ is the estimated value of how many points is in the radius of a vertex, some vertices will have higher degree than others. In addition to this being an estimate we also do not account for circles intersecting with the edges of the plane. These two factors combined makes the number of edges in the graph lower than m (especially when α is small), similarly to the small world graphs. Another way of generating a geometric graph could be to first count all the edges satisfying the radius and call this number δ and then add these edges with probability $\frac{m}{\delta}$. We chose not to do it this way, as the probability of an edge existing would not be the same between different graphs generated with the same inputs. Our algorithm has a time complexity of $O(n^2)$. This can be improved by creating $\frac{\alpha}{r^2}$ buckets ($\frac{1}{r} \cdot \frac{\alpha}{r}$) and adding the points to these buckets. Then for each point we only need to check the neighbouring 8 buckets, and the bucket containing the vertex for vertices within distance r. This would give an $O\left(\frac{n^2 r^2}{\alpha}\right)$ algorithm.

2.6 Properties of the graph types

In the following we show figures for diameter, degree distribution, and eccentricity distribution, for the following graph classes: SWG, RND, GEO_32, and GEO_128.

In Figure 1 we see the diameter of SWG, RND, GEO_32, and GEO_128, with densities 10, 20, and 30. For all the RND and SWG graphs, the diameter is small, and lies in the range $[6, 10]$. As can be seen, the GEO graphs have higher diameter than their lower bounds, and the diameter is relatively consistent when the number of vertices changes.

In Figure 2 we see the degree distribution of SWG, RND, GEO_32, and GEO_128, with $n = 10000$, and with densities of 10, 20, and 30.

In Figure 3 we see the eccentricity distribution of SWG, RND, GEO_32, and GEO_128, with $n = 10000$, and with densities of 10, 20, and 30. We see that for SWG and RND the eccentricities are centred on a few values. This is expected as the diameter of the graphs are low. For GEO_32 and GEO_128, we see that they both reach a "peak" value. For GEO_32 the value decreases after the peak, while the value stays constant for a while for GEO_128. This plateau is due to that GEO_128 has $\alpha = 0{,}31$, making the plane rectangular instead of square, as is the case for GEO_32.

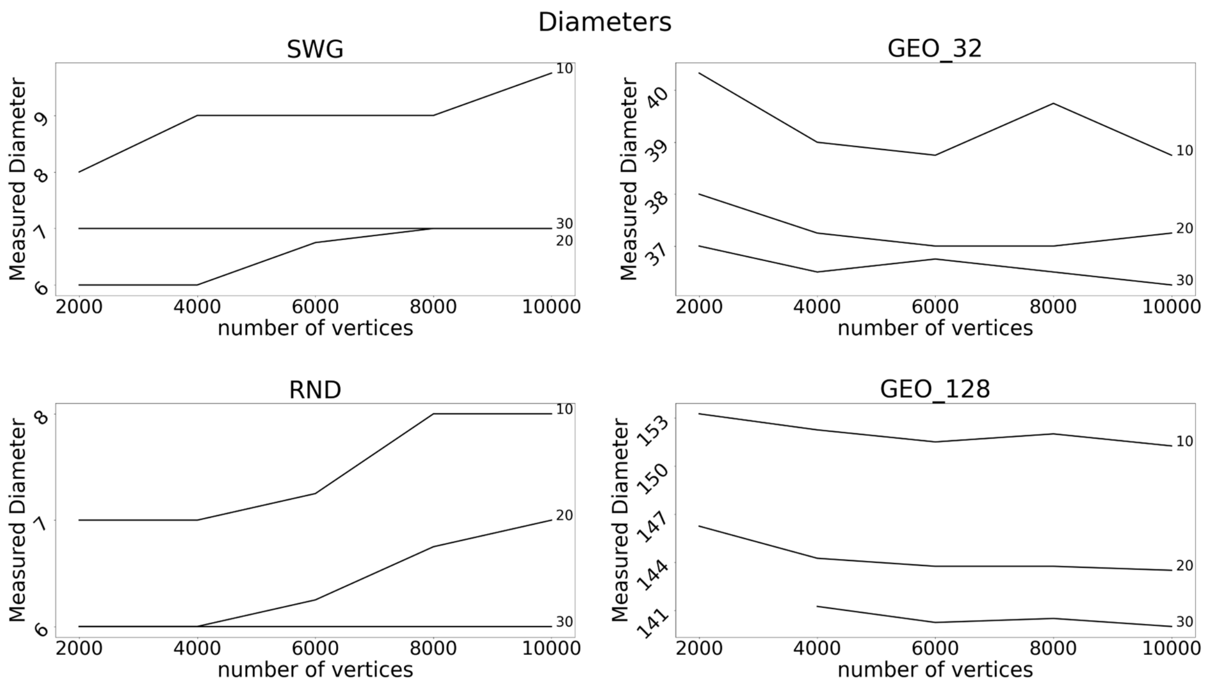

Figure 1 The diameter of SWG, RND, GEO_32, and GEO_128, with different numbers of vertices. The number on the right of each curve represents the density of the graphs. Note the geometric graph with diameter of 128, size 2000, and density 30 does not exist.

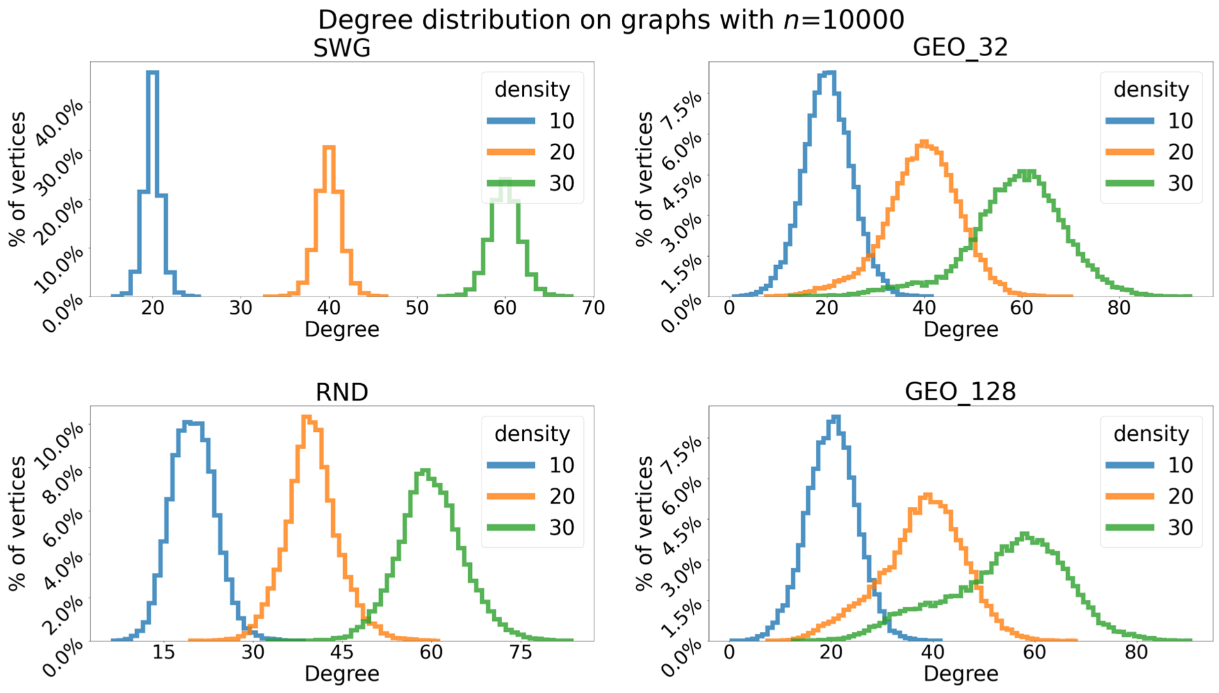

Figure 2 The degree distribution of SWG, RND, GEO_32, and GEO_128, where the number of vertices is 10000. The graphs have densities of 10, 20, and 30.

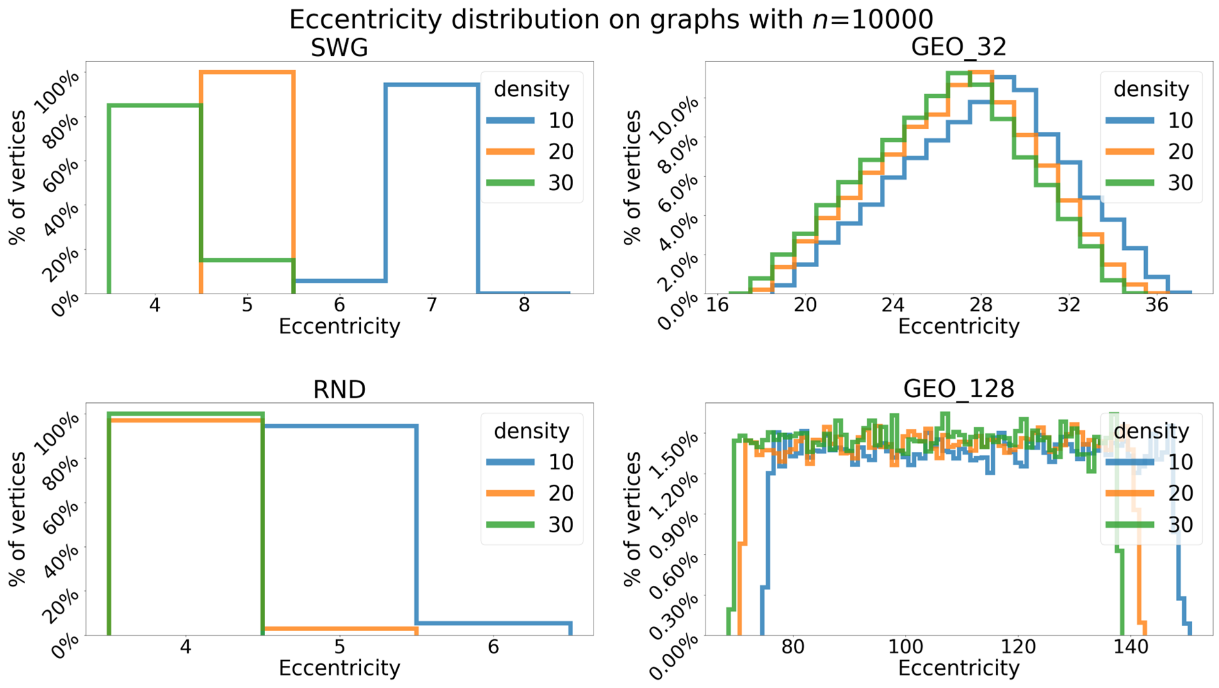

Figure 3 The eccentricity distribution of SWG, RND, GEO_32, and GEO_128, where the number of vertices is 10000. The graphs have densities of 10, 20, and 30.

3 Centralities

Centralities are used to describe properties of vertices that are important in some given context and are used to rank a vertex's relative importance in the graph. Some examples of centralities are betweenness, closeness and the vertex degree. In Chapter 4.3 we use different centralities to choose vertices to vaccinate in the simulations. Since centralities can be costly to compute it is important that this is done in a timely manner.

In this chapter we define and show algorithms for betweenness centrality, closeness centrality and k-neighbourhood on unweighted, undirected, and connected graphs.

3.1 Centralities used

Degree

The degree of a vertex v is defined as the number of incident vertices to v.

In the context of disease spread, vertices with high degree are the ones with a possibility of being "super spreaders", that is vertices that can infect many others in a short window of time. From this it is easy to see how these high degree vertices can be important in the spread of a pandemic.

k-Neighbourhood

The definition of the k-neighbourhood of a vertex $\boldsymbol{v}$ is:

$$\boldsymbol{N(k,v) = \{u \in V(G) | dist(v,u) \le k\}}$$

that is, all vertices within distance $\boldsymbol{k}$ of $\boldsymbol{v}$ in $\boldsymbol{G}$. The k-neighbourhood is not a centrality measure by itself, but it is included here since one can create different centralities measures based on it. For example, the k-neighbourhood, here the size of the 1-neighbourhood is equivalent to the degree of a vertex. Another use of the k-neighbourhood is the size of an intersection between the k-neighbourhood and some other set, for instance the set of infected vertices or the set of vaccinated vertices.

k-degree

The definition of the k-degree of a vertex v is $|N(k,v)|$, the size of the k-neighbourhood of v. The k-degree is a generalization of the normal degree of a vertex, as the regular vertex degree is equivalent to the 1-degree.

k-Neighbourhood intersections

The k-neighbourhood intersection is defined as $|N(k,v) \cap A|$, where $A \subseteq V(G)$ is a set of vertices with some property, for example the set of all infected vertices or the set of immune vertices. If $A = V(G)$ then this is equivalent to k-degree. The intersection gives the ability to ignore vertices that are predetermined to be less important.

Betweenness

The betweenness centrality of a vertex v is defined as:

$$\boldsymbol{BW(v)} = \sum_{\boldsymbol{a \neq v \neq b} \text{ \& } \boldsymbol{a,v,b \in V(G)}} \frac{\boldsymbol{\sigma_{ab}(v)}}{\boldsymbol{\sigma_{ab}}}$$

where $\sigma_{ab}(v)$ is the number of shortest paths between vertices a and b with v as an intermediate vertex, and σ_{ab} is the total number of shortest paths between a and b in the graph G.

In the contexts of betweenness, we can look at two types of vertices, leaves, and bridges. For leaf vertices, a vertex v where $deg(v) = 1$ (see vertex 3 in Figure 4), it is easy to see that the betweenness is zero as there are no paths going through it. Incident to every leaf node is always a bridge node, that is, a vertex which would create two or more connected components if it is removed (see vertex 1 in Figure 4). Let the set of these components be C, then we know that the betweenness of this vertex is at least $\sum |C_a| \cdot |C_b|$ where $C_a, C_b \in C$ and $C_a \neq C_b$, that is, the sum of the products of the pairs of the sizes of the components in C. Betweenness centrality is useful when determining vertices controlling information or disease spread on the paths going through them (Das, Samanta, & Pal, 2018).

Closeness

The definition of closeness centrality is:

$$\boldsymbol{CL(v)} = \sum_{\boldsymbol{u \in V(G)}} \frac{\boldsymbol{1}}{\boldsymbol{dist(v,u)}}$$

One observation with leaf vertices and closeness is that the leaf vertex has smaller closeness centrality than its incident vertex (unless it is also a leaf). Note that when v and u are in different components, $dist(v,u) \to \infty$, if we let the inverse $\frac{1}{dist(v,u)} \to 0$ then this results in the two vertices not having any effect on each other's closeness. Closeness centrality is useful for determining a vertex that can spread information or disease fast (Das, Samanta, & Pal, 2018).

Figure 4 shows an example graph together with a table of the values for betweenness, closeness and 2-neighbourhood. As previously noted, the k-neighbourhood is a set of vertices, and betweenness and closeness are both positive real numbers.

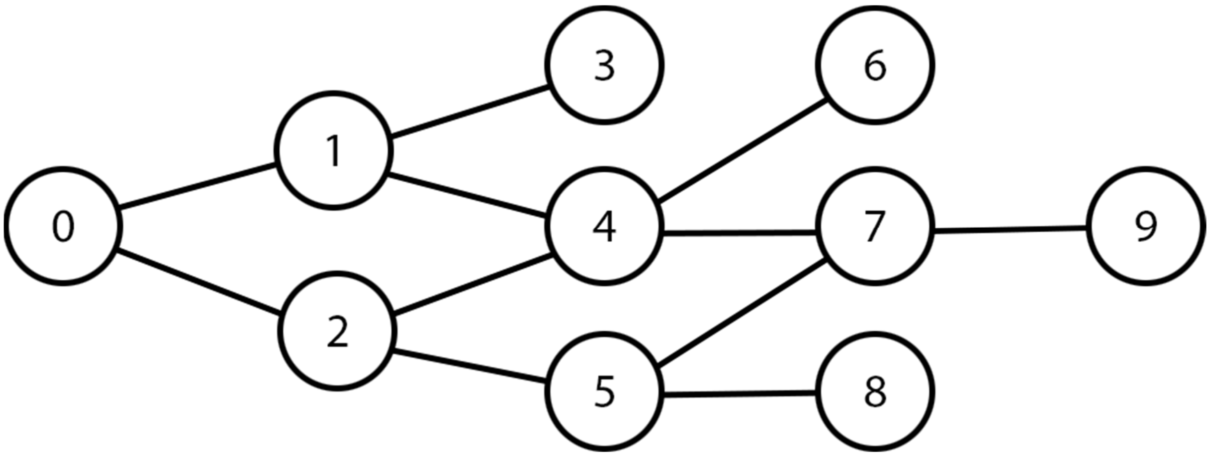

Figure 4: Example graph together with different centrality measures.

Vertex	Betweenness	Closeness	2-neighbourhood
0	3,33	4,75	{0 1 2 3 4 5}
1	9,67	5,42	{0 1 2 3 4 6 7}
2	9,00	5,67	{0 1 2 4 5 6 7 8}
3	0,00	3,70	{0 1 3 4}
4	17,00	6,33	{0 1 2 3 4 5 6 7 9}
5	8,65	5,42	{0 2 4 5 7 8 9}
6	0,00	4,08	{1 2 4 6 7}
7	12,33	5,67	{1 2 4 5 6 7 8 9}
8	0,00	3,70	{2 5 7 8}
9	0,00	3,83	{4 5 7 9}

3.2 Calculation of centralities

In this section we show the algorithms used in Chapter 5 to compute the different centralities. Because we work on relatively small graphs, there is a possibility of creating faster algorithms by lowering overhead or having better hardware utilization, despite sometimes introducing higher asymptotic time complexity. We present three algorithms that we have found to be efficient in practice. These are *k-neighbourhoods* (3.2.1) an algorithm for computing the k-neighbourhoods of all vertices in a graph, *distance-layers* (3.2.2) an algorithm for creating layers of vertices for each vertex where the different layers each contain all vertices at some distance from the vertex, and an algorithm for *betweenness* (3.2.4) that uses the *distance layers* algorithm. In Chapter 3.2.3 we analyse the algorithms *k-neighbourhoods* and *distance-layers* and compare *k*-neighbourhoods to similar algorithms. In Chapter 3.2.5 we compare Brandes's algorithms for betweenness to ours and show that ours is faster for small sparse graphs.

The bases for computing the different centralities are the two algorithms *k-neighbourhood* and *distance-layers*. The result from the *distance-layer*s algorithm is used for calculation of the betweenness and closeness centralities, while the other centralities, such as k-degree use the *k-neighbourhoods* algorithm.

An easy algorithm for computing closeness centralities is to first find the shortest distance between each pair of vertices in the graph, and then for each vertex sum the inverse distance from it to all the other vertices (see definition of closeness). Since this is an unweighted graph the easiest way of finding the shortest paths/distances is to use Breath First Search (BFS) starting from every vertex. For computing a k-neighbourhood we can similarly also use BFS from every vertex and stop after reaching distance k. For betweenness centrality the calculation becomes harder on unweighted graphs than for most weighted graphs. This is because there are often multiple shortest paths between two pairs of nodes in unweighted graphs. For example, in Figure 4, there are two shortest paths between vertex 0 and vertex 4 ($0 \to 1 \to 4$ and $0 \to 2 \to 4$). For weighted graphs, at least when the weights are mostly unique this is unlikely to happen. Because of this the shortest path algorithm we use needs to be able to reconstruct or save up to $O\left(3^{\frac{n}{3}}\right)$ shortest paths between each pair of vertices (upper bound shown in Appendix 1).

3.2.1 Algorithm for k-neighbourhoods

For the pseudo codes below, we use two data structures, $Array$ and Set. We use the shorthand of $id\ of\ v = v$ in both these data structures. $Array$ is an indexable collection of some data, where we can get the data from a given index. Set is an unorder collection of data, where we can check the inclusion or exclusion of some element. We also have some standard set operations with $A \cup B$ and $A \cap B$ being the union and intersection between A and B respectivly, and $A - B$ the difference between A and B (remove all elements of B from A). As for now the implementation of Set is not important. We get back to this in chapters 3.2.4 and 3.2.5.

There are several algorithms for finding the k-neighbourhood of a vertex. One can use breath first search and stop when reaching distance k, or a shortest paths algorithm and then make an array of vertices with distance $\leq k$. It is also possible to use the fact that the k-neighbourhood of a vertex, is the union of the (k-1)-neighbourhoods of the neighbours of v. That is, $N(k,v) = \bigcup_{u \in N(1,v)} N(k-1,u)$, where $N(0,u) = \{u\}$. This is the main idea for our algorithm.

ALGORITHM FOR CALCULATING K-NEIGHBOURHOODS

```
    Input: A graph $G$ and an integer $k$
    Output: An array of the k-neighbourhoods for each vertex of $G$
1   $prevLayer, nextLayer : Array\langle Set\rangle$
2   $for\ \forall\ v \in V(G)$:
3       $prevLayer[v] = \{v\}$
4       $nextLayer[v] = \{v\}$
5   $loop\ k\ times$:
6       $prevLayer, nextLayer = nextLayer, prevLayer$
        //saves us from creating a new nextLayer for each iteration.
7       $for\ \forall\ v \in V(G)$:
8           $for\ \forall\ u \in N(v)$:
9               $nextLayer[v] = nextLayer[v] \cup prevLayer[u]$
10  $return\ nextLayer$
```

3.2.2 Algorithm for distance-layers

The *distance-layers* algorithm builds on a similar idea as the *k-neighbourhoods* algorithm, but it instead creates an array of sets for each vertex. The length of the array for a vertex v is equal to the eccentricity of the vertex, and the set at index k of the array contains the vertices with distance exactly k to v. This is equivalent to the difference between the k-neighbourhood and the (k-1)-neighbourhood. Recall $L(k, v)$ being the set of vertices with distance exactly k from v, then $L(k, v) = \{u \in V(G) \mid dist(u, v) = k\} = N(k, v) - N(k - 1, v)$.

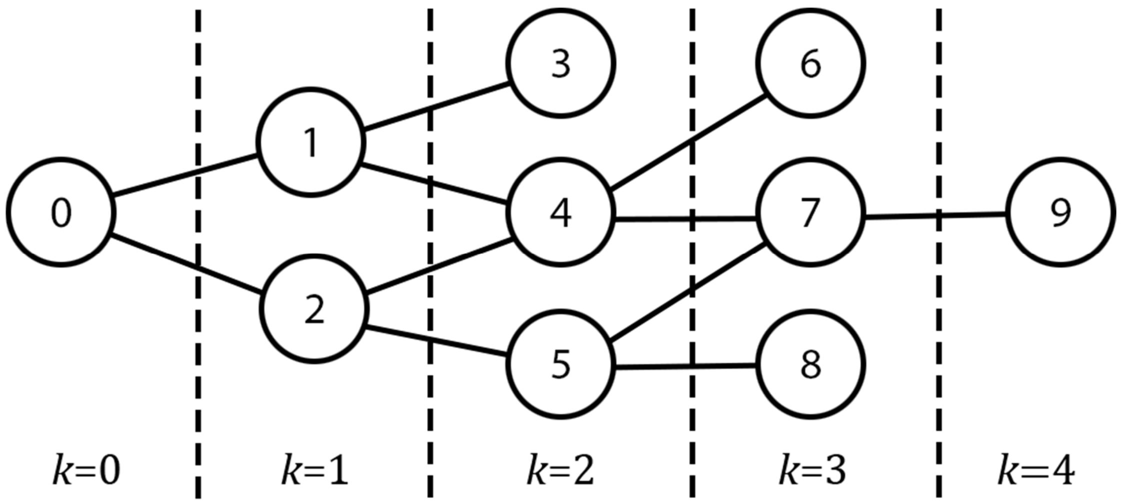

Figure 5: The distance layers of vertex 0

Figure 5 shows an example graph and the distance layers of vertex 0, where k is the number of each layer.

ALGORITHM FOR CALCULATING DISTANCE-LAYERS

```
     Input: A graph $G$
     Output: An array of arrays containing the set of vertices at distance $k$ from each vertex in $G$
1    $\boldsymbol{layers} : \boldsymbol{Array\langle Array\langle Set\rangle\rangle}$    //Indexed by [vertex][distance/layer]
2    $\boldsymbol{cumulative} : \boldsymbol{Array\langle Set\rangle}$    //Set of all vertices already in a lower layer
3    $\boldsymbol{notFinished} \leftarrow V(G)$    //Set of all unfinished vertices
4    $\boldsymbol{for} \forall \boldsymbol{v} \in \boldsymbol{V(G)}$:
5        $\boldsymbol{layers[0][v]} = \{\boldsymbol{v}\}$
6        $\boldsymbol{cumulative[v]} = \{\boldsymbol{v}\}$
7    $\boldsymbol{counter = 0}$    //Keeps count of the current distance
8    $\boldsymbol{while\ notFinished} \neq \emptyset$:
9        $\boldsymbol{counter \leftarrow counter + 1}$
10       $\boldsymbol{for} \forall \boldsymbol{v} \in \boldsymbol{notFinished}$:
11           $\boldsymbol{v\_layer = layers[counter][v]} = \{ \}$
12           $\boldsymbol{for} \forall \boldsymbol{u} \in \boldsymbol{N(v)}$ :
13               $\boldsymbol{v\_layer = v\_layer} \cup \boldsymbol{layers[counter - 1][u]}$
14           $\boldsymbol{v\_layer = v\_layer - cumulative[v]}$
15           $\boldsymbol{cumulative[v] = cumulative[v]} \cup \boldsymbol{layer}$
16           $\boldsymbol{if} |\boldsymbol{v\_layer}| == \boldsymbol{0}$:
17               $\boldsymbol{notFinished = notFinished} - \{\boldsymbol{v}\}$
18   $\boldsymbol{return\ layers}$
```

3.2.3 Analysis of k-neighbourhood and distance-layers

We now analyse the running time of the *k-neighbourhood* and *distance-layers* algorithms.

Both *k-neighbourhood* and *distance-layers* algorithms use a “Set” data structure for storing the different distance layers, hence the size of the sets is at most $|V(G)| = n$. The choice of data structure for Set is important for the time complexity of the algorithms. Here we will look at two implementations of a set and their impact on the time complexity of the two algorithms. The two implementations are using hashsets and bitsets. Hashsets are hash maps where we only care whether a key is in use or not. Using this we can keep track of the inclusion of elements in the set, and it is also simple to compute the union between two sets as this can be done in $O(\eta(k))$ by adding the keys of the first set to the other. Bitsets are bit arrays where the i'th position in the array is a binary value representing either the exclusion or inclusion of vertex i in the set. Each set has n such binary values, one for each vertex, and therefor the union of two sets can be computed in time $O(n)$ using a bitwise or-operation over the two values. One can obtain a more efficient implementation by storing the bits in some larger structure such as a 64-bit integer and then use bitwise or-operations on these. Then a lookup of a specific value needs to use a binary mask in addition to the normal array lookup. Although the asymptotic time complexity does not get any better doing this, we only need $\frac{1}{64}$ as many or-operations compared to storing the bits individually.

A common expression for the complexity of both *k-neighbourhoods* and *distance-layers* using either of the set implementations is given by $O(it \cdot (n+m) \cdot op) = O(it \cdot m \cdot op)$ where it is the number of iterations of the outermost loop (line 5 in *k-neighbourhoods* and line 8 in *distance-layers*) and where op is the complexity of the union operation. The $n+m$ term, from lines 7 and 8 in *k-neighbourhoods*, and 10 and 12 in *distance-layers,* comes from iterating over all vertices and then all the vertices incident to them. Using BFS to compute the k-neighbourhood and distance layers both gives the same time complexities of $O(n \cdot (n + m)) = O(nm)$. Below is a table of the different complexities.

	Bitset	Hashset	BFS
k-neighbourhood	$k \cdot mn$	$k \cdot m \cdot \eta(k)$	mn
Distance layers	$diam(G) \cdot mn$	$diam(G) \cdot m \cdot \eta(k)$	mn

Apart from the time complexity there are three main properties of the implementations that impact their performances. These are, the amount of memory used, cache performance, and usage of the output.

We first consider memory usage. The hashset implementation for *k-neighbourhoods* keeps a hashset for each vertex containing the vertices in its k-neighbourhood. For *distance-layers* each vertex has an array of hashsets. The length of this is equal to the diameter of the graph. Since there is no overlap among the layers, together they contain exactly n vertices. The disadvantage of hashsets in the context of memory usage, is that their memory overhead can be large, dependent on how they are implemented. We use Java HashSet (java.util.HashSet) when comparing the different implementations. This implementation of HashSet uses memory for the buckets that stores the hash values in the form of linked lists. For the bitset we use Java bitsets (java.util.BitSet). With a bitset implementation there is no overhead, but every set needs $\frac{n}{64}$ 64-bit integers. This means that if a set has more than $\frac{n}{64}$ vertices in it, a bitset will use less space than a hash map. But for smaller sets, like a 1- or 2-neighbourhood, hash maps can use less memory. For the BFS implementation we can use an array of vertices for representing each of the k-neighbourhoods, this needs n arrays, each the size of its corresponding neighbourhood. For the *distance-layers*, we use an array of all the vertices sorted by their layers, and an array for keeping the indices of each layer. This uses $n + diam(G)$ integers. In addition, we need a queue and an array $dist$ of size n to do the BFS, but these can be reused for each vertex.

Secondly, we consider cache performance. Here there is little difference between *k-neighbourhood* and *distance-layers* when it comes to the different implementations, and therefore we only look at the cache performance of bitsets, hashsets, and BFS. The cache performance of hashsets is poor as the buckets consisting of linked lists requires non-sequential access of memory. Similarly, for BFS where checking whether a vertex has been visited is non-sequential on the $dist$ array, as one of some vertex v's neighbour's position in the $dist$ array is not necessarily in the same cache line as v's. For the bitset implementation the union operations have good cache performance and can also make use of larger registers supporting or-operations. Going back to the time complexity of the bitset and BFS implementations of the distance layers, the only difference is the $diam(G)$ term. But adding that the or-operations requires $\frac{n}{64}$ operations instead of n, we see that if the diameter of the graph is less than 64, the bitset implementation has fewer operations. Since we will mostly be using small diameter graphs this is significant.

Lastly, we look at how the output of the algorithms is to be used. There are two main uses of the output of these algorithms, iteration over the sets, and intersections between these sets and other sets. As described, we must consider three different types of outputs; hashsets, bitsets, and either arrays or arrays with an additional index array. For iteration both bitsets and arrays are much faster than hashsets, as again both the cache and space use of hashsets are poor. For intersections we have two cases, a "native" intersection, that is, an intersection between two sets of the same data structures (bitset, hashset, etc.), and a "non-native" between two different types of data structures. Comparing the "native" intersection between hashsets and between bitsets, the bitsets are again much faster, as they use bitwise and-operations in the same way as for the or-operations. For intersections between two unsorted arrays, we get an $O(\alpha_2 \log \alpha_1)$ operation (sort α_1 and do binary search lookups on the elements of α_2), where α_1 and α_2, ($\alpha_1 < \alpha_2$) are the sizes of the two arrays being intersected. As for the "non-native" intersections, mainly between a set and an array, it is usually better to iterate over the array, and do the lookup in the set, as the time use of a lookup is constant in both set implementations. This should be compared to arrays where a lookup is a linear operation.

With these three properties in mind, it is not surprising that the experimental results comparing the three methods used in computing k-neighbourhood and distance layers show that for small graphs the bitset implementation is the fastest. These results are presented in Chapter 3.3.1.

3.2.4 Algorithm for betweenness and closeness centrality

Calculation of closeness for a vertex v is easy when we have the distance-layers of v as it is simply the sum of the product between the size of each layer and the layer number of the layer, that is, $CL(v) = \sum_{1 \le l \le \epsilon(v)} |L(l,v)|/l$. This is because $L(l,v)$ contains all the vertices that are at distance l from v. And all of these contribute $1/l$ to the total closeness of v. Since there are $|L(l,v)|$ such vertices, therefor the total contribution for each layer is $|L(l,v)|/l$.

Calculation of betweenness is more difficult, as mentioned in Section 3.2 the algorithm for betweenness centrality needs to be able to recompute the multiple possible shortest paths between two vertices as saving all the paths is too costly as there can be exponentially many. A solution for this is *Brandes's algorithm* introduced in *A faster algorithm for betweenness centrality*, using BFS to count all the paths. We introduce an algorithm with some commonalities with *Brandes's algorithm*, but where we use *distance-layers* instead of BFS to count paths.

The main idea in this algorithm for betweenness centrality, is using the distance layers from the *distance-layers* algorithm to reconstruct all the shortest paths between all the pairs of vertices. First, we fix some vertex v, then we find all the paths with v as an endpoint. This is done by starting with the vertices in the distance layer furthest from v, and then dividing all the paths going through it and the one additional path where the vertex itself is an endpoint between the vertices it neighbours in the distance layer one closer to v.

ALGORITHM FOR CALCULATING BETWEENNESS CENTRALITY

```
    Input: A graph $G$ and the distance layers of $G$
    Output: An array containing the betweenness centrality of each vertex
1   $tempC : Array\langle float \rangle$
      //tracks how many paths runs through a node for each iteration of the outmost loop
2   $centralities : Array\langle float \rangle$
3   $tempC \leftarrow$ Array of length $V(G)$ with all values set to 0.0
4   $centralities \leftarrow$ Array of length $V(G)$ with all values set to 0.0
6   $for\ \forall v \in V(G)$:
7       $tempC \leftarrow$ set all values to 0.0
8       $for\ \forall i$ from $|layers|$ to $1$ :  //$i$ is the index of the current layer of the distance layers
9           $if\ layers[i][v] == null$: $continue$
10          $for\ \forall u \in layers[i][v]$:
11              $intersection = N(1, u) \cap layers[i-1][v]$
12              $inverseCardinality = 1/|intersection|$
13              $for\ \forall w \in intersection$ :
14                  $toNext = (tempC[u] + 1) \cdot inverseCardinality$
15                  $centralities[w] = centralities[w] + toNext$
16                  $tempC[w] = tempC[w] + toNext$
17  $return\ centralities$
```

Recall Figure 5, calculating the betweenness of vertex (0) starts with vertex (9) adding 1 to the paths going through vertex (7). Then (7) adds the number of paths going through it from the previous layer, 1, with the one path formed by itself, and divides these between its neighbours in the next layer (vertex (4) and vertex (5)). After we are done with the $k = 3$ layer vertex (3) has no path going through it, vertex (4) has 1 path from vertex (6) and $\frac{2}{2}$ from vertex (7) in a total for 2, and vertex (5) has $\frac{2}{2}$ from vertex (7) and 1 from vertex (5).

3.2.5 Analysis of betweenness and closeness centrality

We now analyse the time complexity and running time of *betweenness centrality* and *closeness centrality*.

First, we consider the *closeness centrality* algorithm. There are two implementations, one using BFS and one using *distance-layers*. The BFS implementation has a time complexity of $O\big(n \cdot (n + m)\big) = O(mn)$, while the *distance-layers* implementation has complexity $O\big(diam(G) \cdot n \cdot (n + m)\big) = O(diam(G) \cdot nm)$ (the complexity of *distance-layers*). In the case that the distance layers (or BFS) are precomputed these complexities become $O(n)$ and $O\left(n^2\big(\deg_{max} G + diam(G)\big)\right) = O(n^3)$ for the BFS and *distance-layers* implementations respectively. For the complexity of *betweenness centrality,* we compare our algorithms, with that of Brandes.

For the complexity of our algorithm for betweenness, we start with noticing that the combination of lines 8, 9, and 10 is equivalent to a for-loop over V, as the distance layers don't share any vertices. There is one difference from a normal iteration over V, in that this iteration is an $O(diam(G) \cdot n)$ operation. Lines 11 to 16 are essentially iterating over the neighbours of each vertex. In combination this gives an $O(n^2 diam(G) + n^2 + nm)$, since $n < m$ this is $O(nm \cdot diam(G)$. This assumes we are given the distance layers, if we are not, the complexity falls back to that of the *distance-layers* algorithm ($O(diam(G) \cdot nm)$ using the BitSet implementation). In comparison, Brandes's algorithm has complexity $O(nm)$.

When comparing these algorithms experimentally, since our algorithm for betweenness and closeness both use *distance-layers*, and both Brandes's algorithm and the BFS based closeness algorithm uses BFS, we can compute both closeness and betweenness together.

3.3 Experimental results for computing centralities

In this section we show the experimental results when comparing the three implementations of *k-neighbourhoods* and compare our algorithm for betweenness and closeness to Brandes's algorithm. We run the algorithms on four different sets of graph types, small world (SWG), random connected (RND), and two sets of geometric graphs (GEO), one with diameter of 32 and one with diameter of 128. Each of these sets contains three graphs with the same number of vertices and the same density. The graphs can have number of vertices is 2000, 4000, 6000, 8000, and 10000, and density of 10, 20, and 30. (Except for GEO with 2000 vertices and density of 30, which does not exist). For each graph in these sets, we run the algorithm three times, and record the smallest time. Then we take the average of these numbers for the graphs of the same size and density. For hardware setup see Chapter 5.1.1.

3.3.1 Experimental results for k-neighbourhoods

In the following we compare the three different implementations of the *k-neighbourhoods* algorithm. That is using BFS, HashSet, and the BitSet, as explained in sections 3.2.1 and 3.2.3. The results are shown in figures 3, 4, 5, 6.

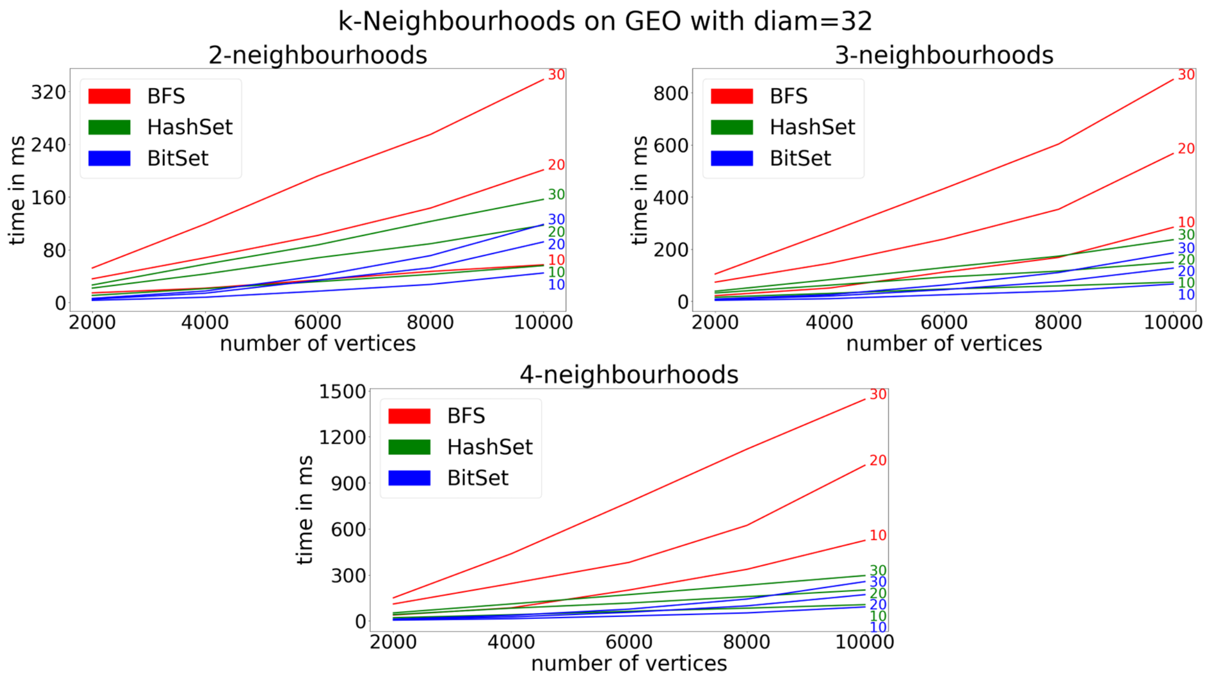

Figure 6 Timings for the k-neighbourhoods algorithm on geometric graphs with diameter of 32, and k=2,3,4. The numbers on the right of each curve represent the density of the graph.

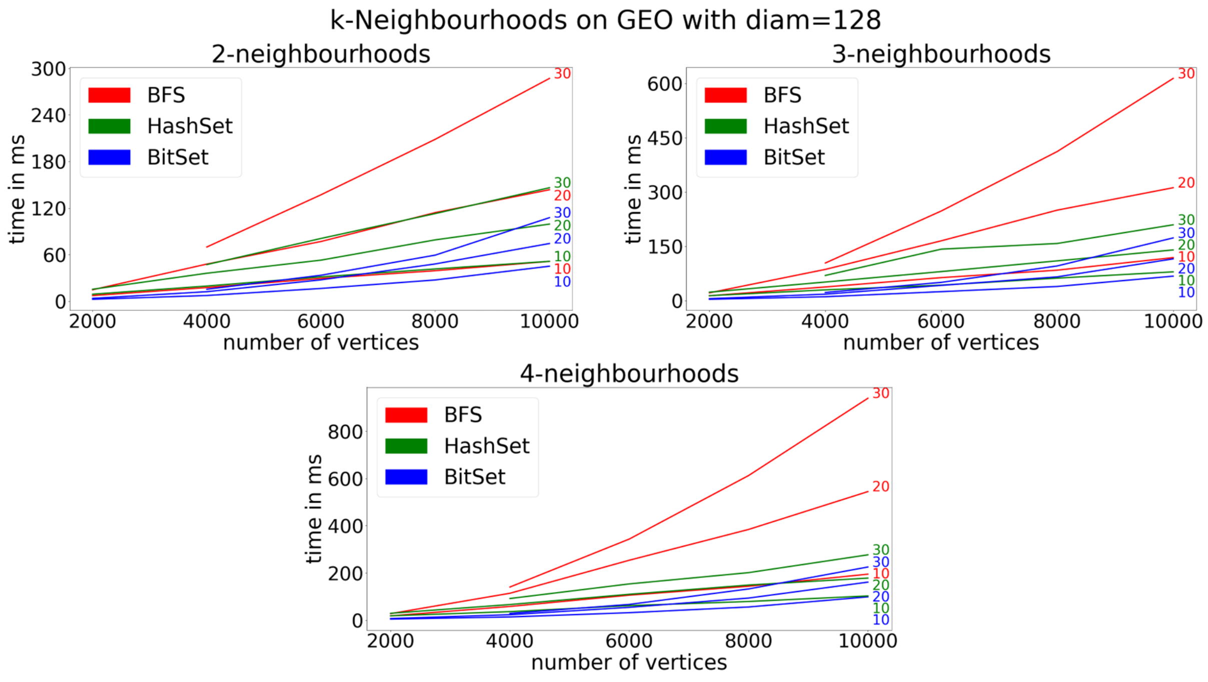

Figure 7 Timings for the k-neighbourhoods algorithm on geometric graphs with diameter of 128, and k=2,3,4. The numbers on the right of each curve represent the density of the graph. Note: the geometric graph of size 2000 with density of 30 does not exist for geometric graphs with diameter 128.

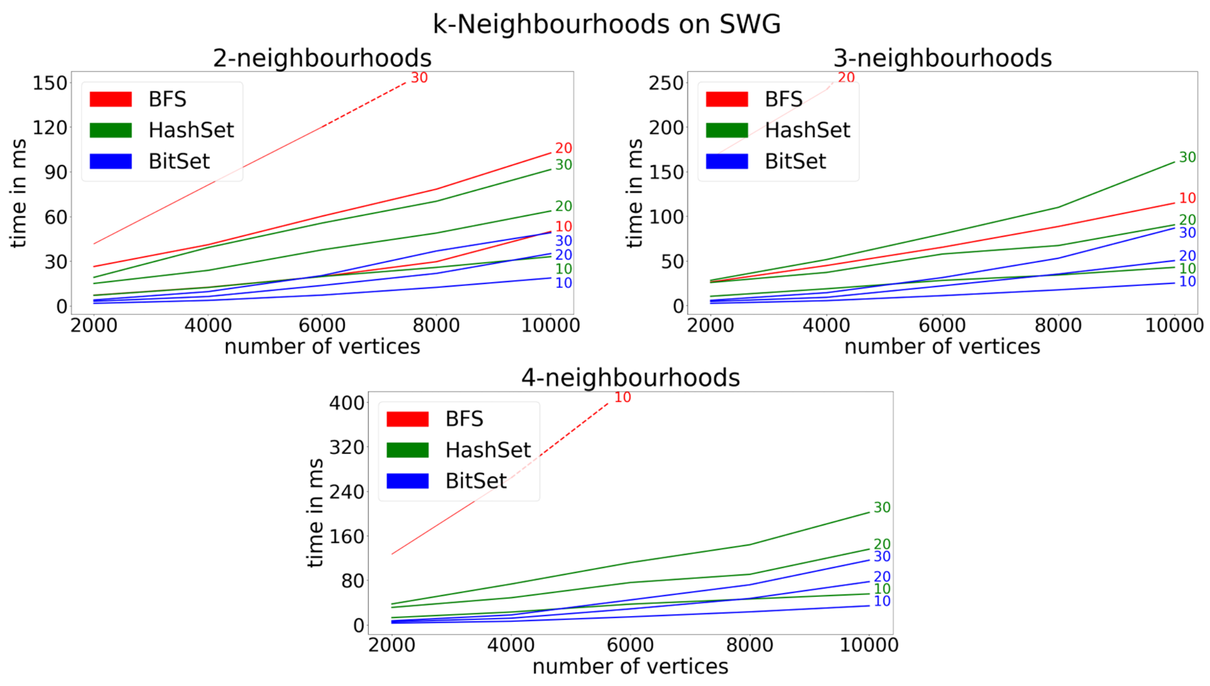

Figure 8 Timings for k-neighbourhoods algorithm on small world graphs, with k=2,3,4. The numbers on the right of each curve represent the density of the graph. The dotted lines represent curves going far outside the average values. Note: For k=3 and density of 30 the starting value is 2303, for k=4 and densities 20 and 30 the starting values are 9277 and 13551 respectively.

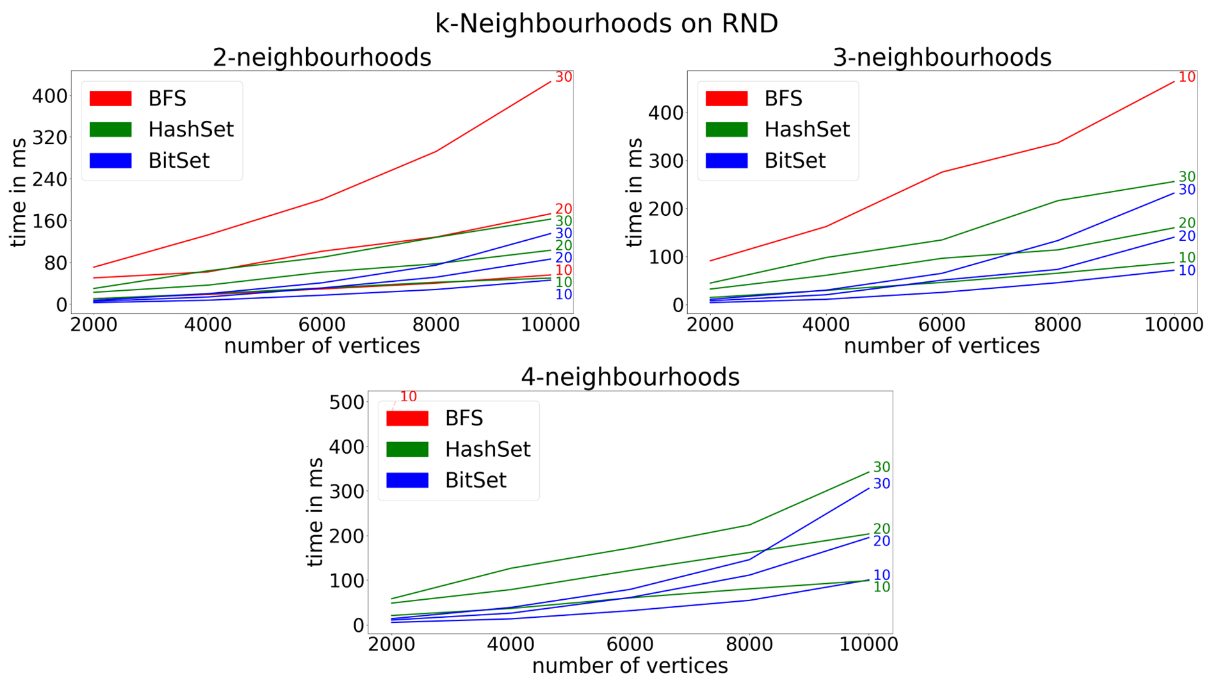

Figure 9 Timings for k-neighbourhoods algorithm on small world graphs, with k=2,3,4. The numbers on the right of each curve represent the density of the graph. The dotted lines represent curves going far outside the average values. Note: For k=3 and density of 30 the starting value is 14610, for k=4 and densities 20 and 30 the starting values are 28921 and 34127 respectively.

From Figure 6, Figure 7, and Figure 8 we see that for GEO_32, GEO_128, and SWG, the BitSet implementation is the fastest followed by the HashSet implementation, while the BFS implementation only performs close to the other implementations when the density of the graphs is 10. In fact, for SWG with $k = 3{,}4$ the BFS implementation is at least 30 times slower for the values that are not shown in the figure. Similarly to Figure 8, Figure 9 showing the RND runs, also show that the BFS implementation is orders of magnitude slower than HashSet and BitSet. But, unlike for GEO_32, GEO_128, and SWG, for RND the HashSet implementation is sometimes faster by a small margin.

BitSet does not use vectorized instructions and registers like AVX. Therefor the speedup of the BitSet implementation compared to the two others comes from better cache usage and that the $k \cdot n$ in the complexity of the BitSet implementation is a $k \cdot \frac{n}{64}$ as the BitSet does the merge of two BitSets in $\frac{n}{64}$ operations. There is also less overhead with the BitSet implementation being only an array of longs (64-bit integers) compared to the HashSet implementation that has buckets, linked lists, and stores both the values, and their hashvalues in these linked lists.

Overall, we see that the BitSet implementation performs best, and therefore we will use this for the *k-neighbourhoods* algorithm.

Note: For some reason the BitSet does not use vectorized instructions (AVX), although it should be easy to change. Preliminary results for a vectorized implementation using the Java jdk.incubator.vector, a non-final java module. Shows a speedup from 5 to 20 times compared to the non-vectorized BitSet.

3.3.2 Experimental results for betweenness and closeness

In the following we compare the two different algorithms for betweenness and closeness. That is using our implementation for betweenness and closeness, using BitSets, and Brandes's algorithm, as explained in Section 3.2.4. The results are shown in Figure 7.

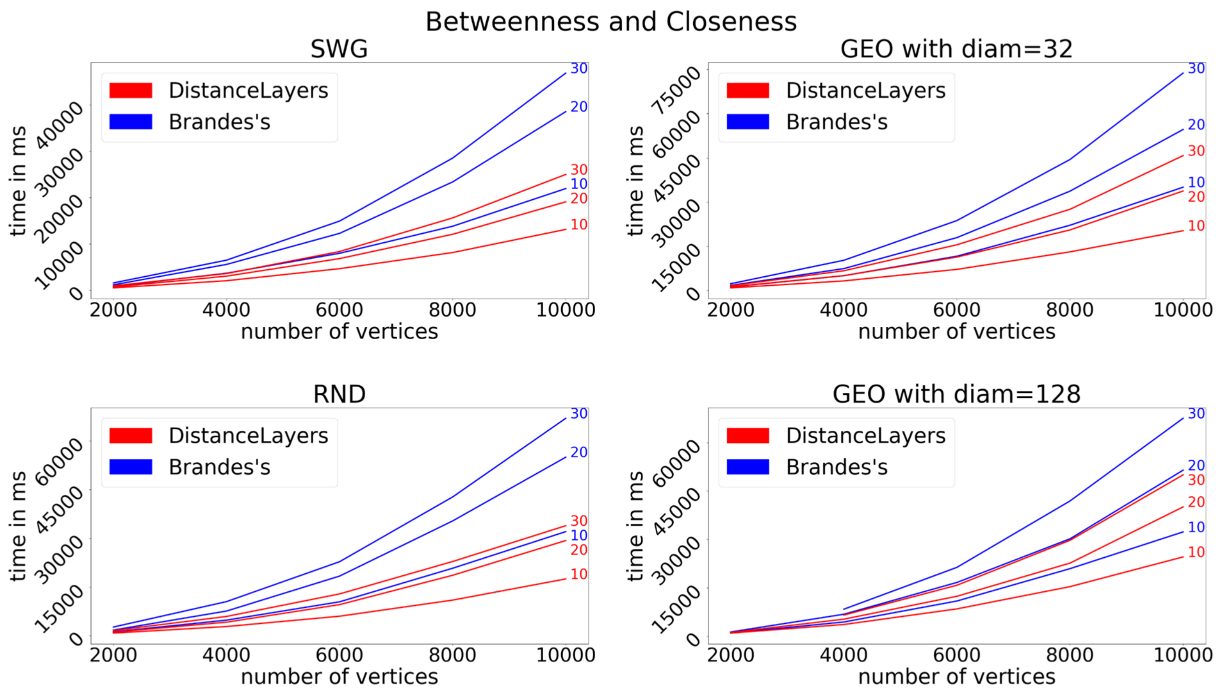

Figure 10 Experimental timings for calculation for Betweenness and closeness using the two different implementations. The numbers on the right of each curve represent the density of the graph.

In Figure 10 we see that the distance layers approach is faster than Brandes's algorithm, and where only for GEO_128 we see the Brandes's algorithm on graphs with a diameter of 20 coming close to taking the same time as our algorithm on graphs with a diameter of 30. Because of this we use our algorithm for Betweenness and Closeness in Chapter 5.

4 Simulation

In this chapter we describe how our simulations are done. We start by describing the parameters to the simulation in Section 4.1. Then in Section 4.2 we (show) the different vaccination strategies we will use. Lastly in Section 4.3 we describe how the simulations are structured.

We let a node, representing an individual, be a vertex with 6 fields (+ ID field) wrapped around it, these fields are:

Field	Type	Explanation
ID	int	Id of the vertex corresponding to the node
$vaccinated$	$bool$	Whether the node is vaccinated
$immune$	$bool$	Whether the node is immune
$beenInfected$	$bool$	Whether the node is, or has been infected
$counter$	int	If (Infected), this is the time until the node is no longer infected
$current$	$bool$	Whether the node is infected in the current step
$next$	$bool$	Whether the node is infected in the next step

A node is immune if it has either been infected and is no longer infected, or if it has been vaccinated. That is, a node that is vaccinated is both vaccinated and immune, while a node that has gained immunity from previously being infected is only immune.

For the simulation there are 7 parameters, the vaccination strategy, the graph, how many to vaccinate each timestep, the length of an infection, the chance of infection, the length of the simulation, and how many nodes are infected initially.

A simulation is initialized by infecting N_{init} nodes at random (described in Section 4.3). After the initialization the simulation runs for T steps, with a step described in Section 4.3. We want to be able to run the simulations in parallel, because of this, we precompute the betweenness centrality, the closeness centrality, and the k-neighbourhoods, and share these between the threads.

The code is written in Java, giving us descent speed and good modularity. Adding new vaccination strategies is simple and is done by creating a new class with a function for choosing nodes based on their status in the simulation. Then these choices are vaccinated.

On average one simulation takes 0.01 seconds. This of course depends on how many vertices and edges are in the graph, and what strategy is used.

4.1 Parameters

For the simulation we have 7 parameters:

Vaccination strategy (S)

The strategy we follow for choosing which nodes to vaccinate, a list of the strategies, and explanations for them is given in Chapter 4.2

Infection probability (p)
The probability of a node infecting a neighbouring node for each timestep

Length of infection (L)
L is the number of steps a node is infected

Vaccination fraction (N)
The number of nodes to vaccinate each step of the simulation, given as a fraction of the total number of nodes (n), that is we try to vaccinate $n \cdot N$ each step.

Graph parameters (G, n, $dens$, $diam$)
The parameters for the input graph G, with n number of vertices, and $dens$ number of edges divided by the number of vertices $\frac{m}{n}$. If the G is a geometric graph we have the additional d being the diameter. Although the graphs are generated independently of the simulations, as described in Chapter 2, it is useful to think of n, $dens$, and $diam$ as parameters.

Simulation Steps (T)
The number of steps the simulation lasts.

Initial Infected (N_{init})
The set of nodes to infect when initializing the simulation.

4.2 Vaccination strategies

Here we have the different strategies that we will compare in Chapter 5, the values for betweenness centrality, closeness centrality and the k-neighbourhoods are precomputed. These precomputations are done in parallel. The precomputed values do not change during the simulation. We also do not vaccinate infected nodes, or nodes that are already immune.

Betweenness centrality (BC)
Vaccinates nodes in order of their betweenness centrality in decreasing order.

Closeness centrality (CC)
Vaccinates nodes in order of their closeness centrality in decreasing order.

Random (R)
Vaccinates non-immune nodes at random.

k-Neighbourhood (k-NH)

Vaccinates vertices according to the size of their k-neighbourhood.

Infected in k-Neighbourhood (k-INH)
Vaccinates the nodes according to how many infected are in their k-neighbourhood. The size of the k-neighbourhood is used as a tiebreaker.

Infected in k-Neighbourhood negative vaccinated (k-INH-VAC)
Vaccinated the nodes according to the number of infected nodes that are in their k-neighbourhood. Where the size of the k-neighbourhood minus the number of vaccinated nodes is used as a tiebreaker.

Infected in k-Neighbourhood negative immune (k-INH-IMM)
Same as "Infected in k-Neighbourhood negative vaccinated" but using immune instead of vaccinated.

k-Neighbourhood ignoring immune (k-NH-iIMM)
Unlike the other strategies, this one does not use precomputed values. The k-neighbourhood is recalculated each step but where the immune nodes are ignored. That is, the k-neighbourhoods are computed on the graph where the vertices corresponding to immune nodes are removed. Because this is recalculated during the simulation, this strategy does not have any part that can be precomputed. After this the nodes are ranked in the same way as "k-Neighbourhood" but using the newly calculated k-neighbourhood instead of the precomputed one.

4.3 Simulation steps

The simulation is done in steps, where a step is equivalent to some fixed time period in a real pandemic. Each step consists of three phases, the incrementing phase, the infection phase, and the vaccination phase. Infecting the initial nodes $P_i \in N_{init}$, is done by setting $P_i.counter = L, P.current = true$ and $P_i.beenInfected = true\ \forall P_i \in N_{init}$. To save time, if there are no infected nodes left after some step, we skip the rest of the steps.

In the incrementing phase we want to check what nodes go from infected to immune. We do this by iterating over every node P and decrement its counter (how long a node has left as infected if it is infected). If $P.counter > 0$ we set $P.current = P.next,\ P.next = true,$ that is, the node is infected in the next step. If $P.counter \leq 0$, P is either just done with being infected, or has not been infected. If $P.current$ we set $P.immune = true$. Then we set $P.current = P.next,\ P.next = false$.

In the infection phase we want to see which nodes become infected in the next step. We do this by iterating over all the nodes, and if a node is infected, we try to infect each of its neighbours with a probability p. If the neighbour is either already infected or is immune, we skip it. If a neighbouring node P becomes infected, we set $P.counter = L$, $P.next = true$ and $P.beenInfected = true$.

Lastly, in the vaccination phase we use the vaccination strategy S and choose $n \cdot N$ nodes to vaccinate. If $n \cdot N$ is more than there are candidates for nodes to vaccinate, we vaccinate all the candidates. The vaccination of a node P is done by setting $P.immune = true$ and $P.vaccinated = true$.

During each step we can record different data, for example how many newly infected nodes, how many nodes are infected in total, etc. This gives us a lot of flexibility for how and what we want to show as results without needing to rerun the simulations.

5 Results

In this chapter we precent the results from the simulation and vaccination strategies described in Chapter 4.

5.1 Setup

5.1.1 Hardware

For experiments, we used an Intel Core i9-9900k CPU @ 3.60GHz and 32 GB of DDR4 memory running at 2666 MT/s. We are using Java 19 (build 19+36-2238), on Windows 10.

5.1.2 Parameters

In Chapter 4.1 we introduced the 7 input parameters for the simulation, the vaccination strategy S, the infection probability p, the length of infection L, the vaccination fraction N, the simulation steps T, the initially infected N_{init}, and the graph G. With the parameters of G being, the number of vertices n, the density $dens$, and for geometric graphs the lower bound for the diameter $diam$ (written as GEO_$dens$). To limit the number of simulations we need to run, we fix some of these parameters.

For T, the number of steps in the simulation, we set this to 300. Since we only look at the peak value, we can safely do this as we found in practice that no peak happened after 300 steps. We fix N_{init}, the set of nodes to initially infect, to have size 1, we found that higher values for this parameter ended in the graphs being infected fast, especially for RND and SWG where the diameter is small. We chose the same initial node to infect for the i'th run of each strategy but made sure that the node picked for the i'th run and $(i+1)$'th run were different. As we have 20 runs for each of the input parameters, we do this by selecting the node with id $\left\lfloor \frac{i}{19} \cdot (n-1) + 1 \right\rfloor$ (with $0 \leq i \leq 19$). This gives the 20 nodes perfectly spread on the range $[1, n]$. Since the graphs have random indices, this results in the picks being random, but consistent between the strategies. Lastly, the parameters p (infection probability) and L (length of infection) are set such that the infection does not die of too quick, but also so that it does not spread too fast. For an example of how different values of p and L look, see Figure 11. We let $p = 1{,}5\%$ and $L = 8$. We also fix n, the size of the graphs, to 10000, as this value is the highest where the simulations can still be run in a feasible amount of time. We also want n to be large since smaller values gives more noise in the results.

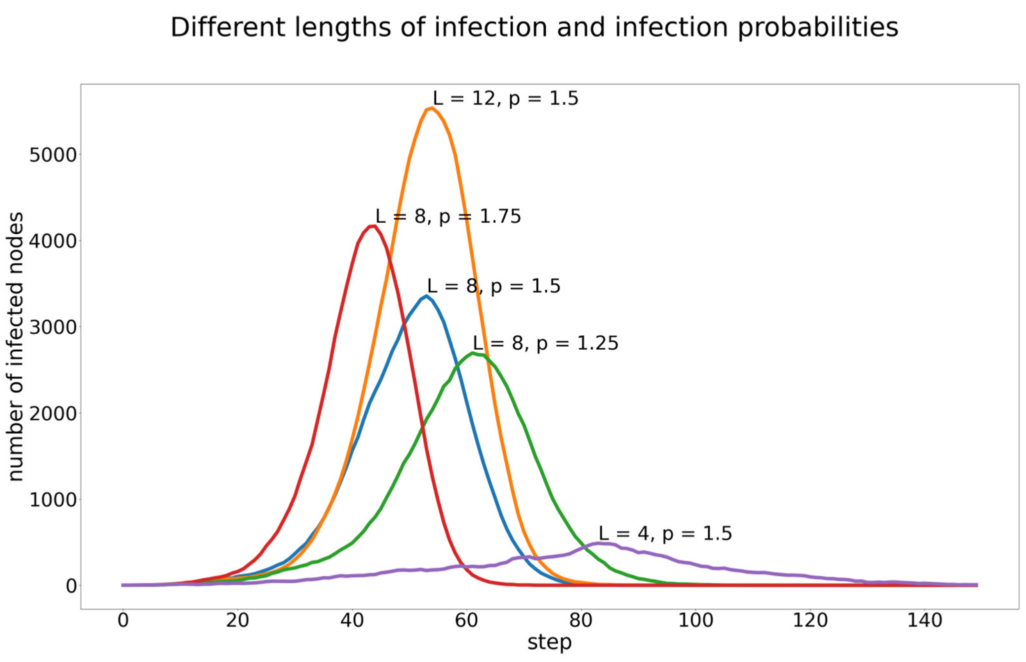

Figure 11 How the number of infected looks for different values of L and p. The graph used is SWG with n=10000 and density = 14

For the variable parameters, that is strategy S, vaccination fraction N, and the rest of the graph parameters $dens$ and $diam$ for geometric graphs, we use all the strategies described in Chapter 4.2 (with $k \in \{1,2,3,4\}$ for the strategies that take an input k), for a total of 23 different strategies.

For the vaccination fraction N, we used numbers close to those of the Norwegian covid-19 vaccination program (FHI, 2021), where the average vaccination each day between January 1st and October 1st is $0{,}5\%$ of the total population ($5{,}4$ million) with a peak number of vaccinated in one day being $2{,}3\%$. As our vaccine gives total immunity, we chose to halve this, therefor N ranges from $0{,}25\%$ to 1%, with steps of $0{,}125\%$ and a total on 7 different values.

For the two graph parameters, we choose the lower bound for the diameter $diam$ to be 32 (GEO_32) and 128 (GEO_128), this gives us 4 types of graphs, SWG, RND, GEO_32, and GEO_128. Lasty, the density parameter $dens$, we don't want this to be too high as then the spread is fast (as p is the chance for each neighbour), for the same reason we don't want the value to be to small (≤ 4 or 6) as then there is little to no spread, therefor we let $dens \in \{6, 8, 10, 11, 12, 13, 14\}$ for RND and GEO_32, $dens \in \{8, 10, 11, 12, 13, 14\}$ for SWG, and $dens \in \{10,11,12,13,14\}$ for GEO_128.

5.1.3 Visualization

Because some runs of the simulation stop early, in that there is little to no spread, as is the case for many of the runs where the density is low, we remove these to get cleaner data. If we don't remove these, there is more noise introduced, as these types of runs are randomly spread out over the strategies (see Table 1 for numbers on how many runs were removed). We remove all runs where the peak is less than 1% of the total population.

	SWG	RND	GEO_32	GEO_128
Min	4-NH-iIMM: 13,0%	3-NH-iIMM: 17,0%	2-NH-iIMM: 15,7%	2-INH: 26,5%
Max	1-INH-VAC: 14,5%	2-INH: 18,7%	1-INH-IMM: 17,9%	3-NH-iIMM: 28,2%
Avg	13,8%	17,8%	17,1%	27,4%

Table 1 Shows the maximum and minimum of runs removed for each of the 4 graph types. Data on the form (strategy: 'amount removed') where amount removed is the % of the total number of runs removed.

In Figure 12 we see an example of how the performance of a strategy is presented. Where dark green is good (close to the best performing strategy), white is bad (more than $x\%$ higher than the best performing strategy), and grey means that there are no results for the input because the runs are removed. The title of the figure is in the order of; strategy S, then the graph type (SWG, RND, GEO_32, and GEO_128), and the number of vertices (usually 10000). The x-axis has the values of the densities, and the y-axis the vaccination fraction. There is a colour bar on the right of the figure mapping the distance from the best to its corresponding colour.

To get the value $v_{S,dens,N}$ (the value used in the figures) for a coordinate in the grid $(dens, N)$, we calculate the normal of the runs, $x_{S,dens,N}$, for that coordinate over all the strategies S. We take the best of these values $x^*_{dens,N}$, and then calculate the distance as $v_{S,dens,N} = \frac{x_{S,dens,N}}{x^*_{dens,N}} - 1$, as $x^*_{dens,N} \leq x_{S,dens,N}$, this value is always positive. If $v_{S,dens,N} > c$, we set $v_{S,dens,N} = c$, were c is some cut-off value (in Figure 12 $c = 1\%$). We set c such that as much of the colour range (dark green to white) is used, while cutting of outliers (these become white in the figures). We use $c = 2\%$ for RND, $c = 3\%$ for SWG, $c = 10\%$ for GEO_32, and $c = 12\%$ for GEO_128.

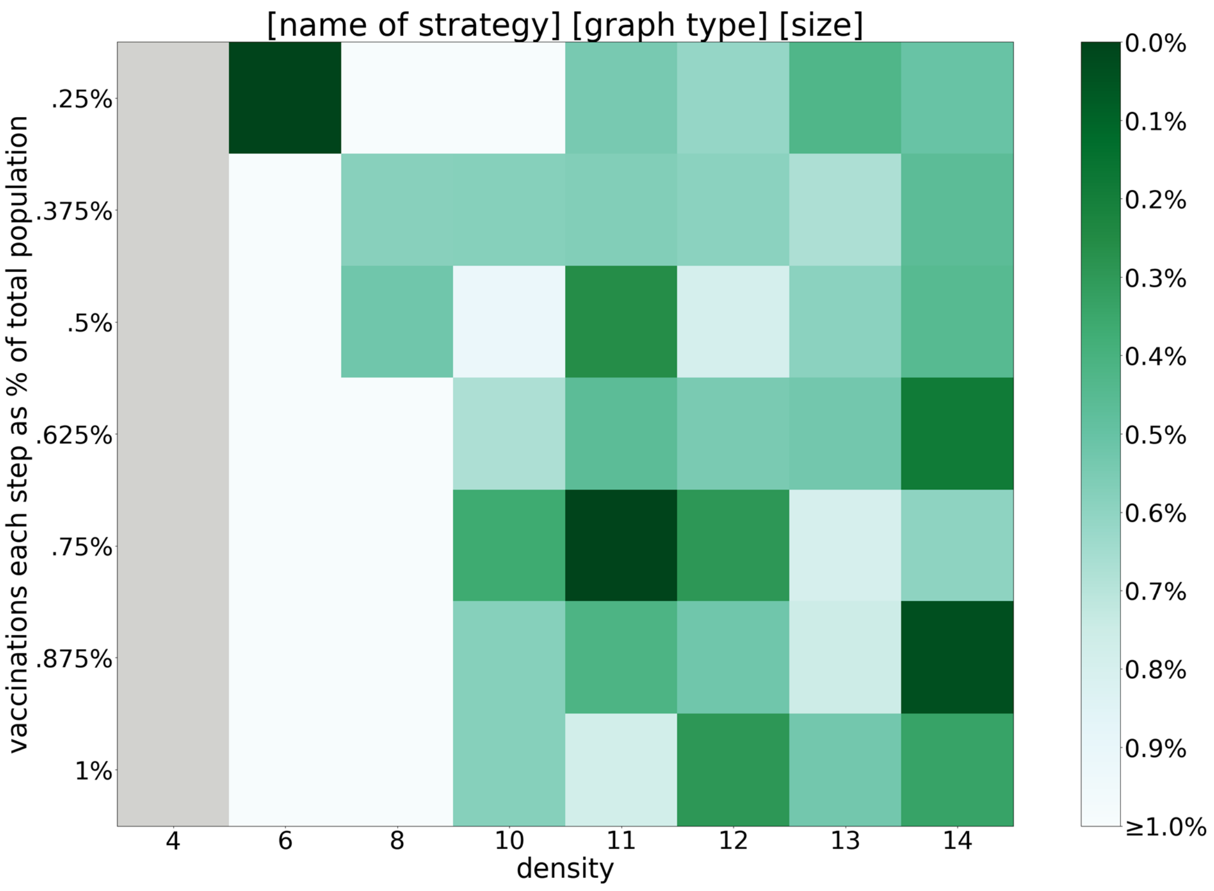

Figure 12 An example of a result figure, the title of the figure is the 3 parameters for the simulation, namely the strategy used, the type of graph used, and the size of the input graphs. For the colours in the grid, dark green is good, white is bad, and grey means there are no runs where the spread is sufficient.

For simplicity we want to have some measure that ranks the strategies on a graph class in a natural way. To do this, we let $v'_{dens,N} = \frac{\sum v_{S,dens,N}}{|S|}$ $v'_{dens,N} = \sum_{\forall dens,\forall N} v_{S,dens,N} /|S|$, be the average value for the grid coordinate $(dens, N)$ (see Figure 14 for an example). Letting $rank(S')$ be the rank of strategy S', we define $rank(S')$ as the sum of the distances from the average value to the value of S' over all the densities and vaccination fractions, that is, $rank(S') = \sum_{\forall dens,\forall N}\left(v'_{dens,N} - v_{S',dens,N}\right)$. If some strategy performs bad compared to the average performance, this value can be negative. Another way to define rank is to only sum the positive values (where the strategy performed better than average), instead of all the values, this gives a similar order to the definition we use.

The $rank$ measure has some practical properties, for one, it lets us compensate for that some of the densities and vaccination fractions it might be easier to have a good value. For example, with RND it becomes easier to have a good value with higher densities, as seen in Figure 14. It will also rank both a strategy that performs good over many of the densities and vaccination fractions, and a strategy performing very good in just a small amount of the densities and vaccination fractions similarly.

For each of the graph classes we give an “overview” figure, showing the performance of R (random), CC (closeness centrality), 1-NH (1-neighbourhood), and BC (betweenness centrality). We will give a figure for the average performance over all the strategies, and table of the $rank$ for each of the strategies, and finally a figure of the four best performing strategies according to $rank$.

5.2 Results for RND

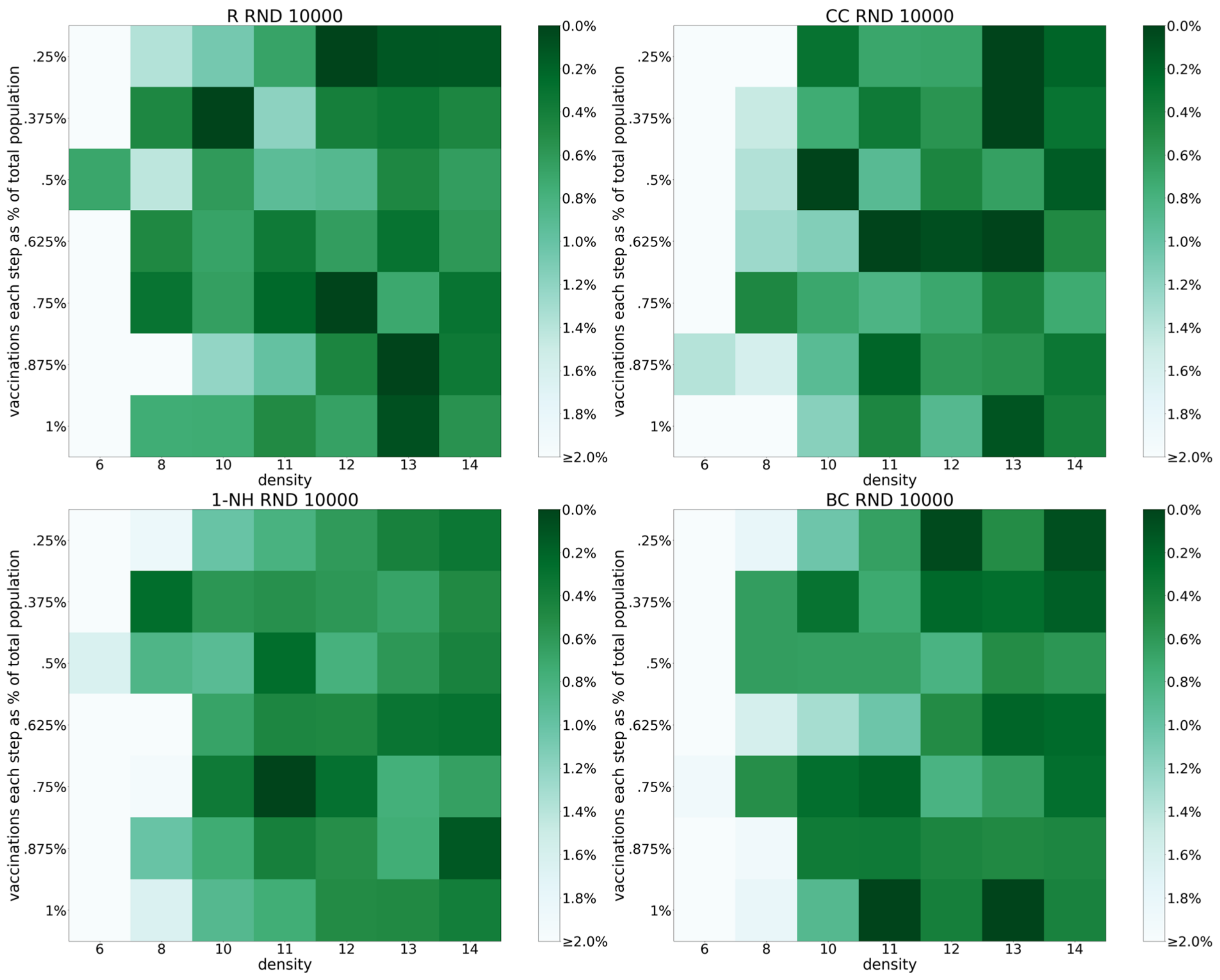

Figure 13 An overview of 4 different strategies (Random (R), closeness (CC), 1-neighbourhood (1-NH), and betweenness (BC)) on RND with 10000 *vertices.*

In Figure 13 we see the effectiveness the 4 overview strategies. Of these, R has the highest $rank$ value, and 1-NH the lowest. We can see that all these strategies perform bad on the low densities (6 and 8).

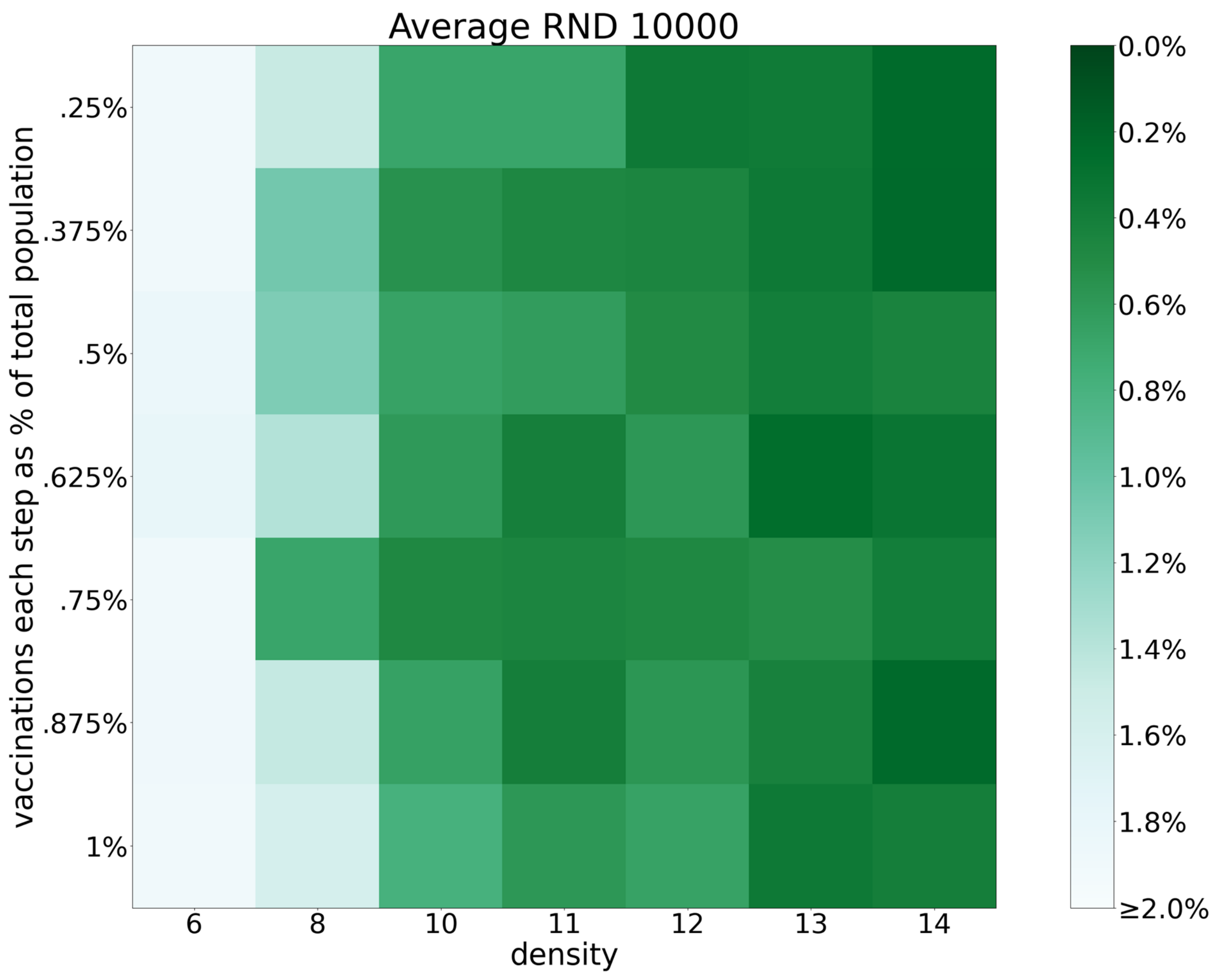

Figure 14 The average performance of the strategies on RND

In Figure 14 we see the average performance over all the strategies. Clearly, the figure shows that it is easier to come close to the best strategy when the density increases. Therefor a strategy that is good when the density is low, is in some sense "better" than one that performs good when the density is high. Likewise, if a strategy performs bad when the density is high, it is "worse" than one that performs bad when the density is lower.

Strategy	Rank	Strategy	Rank
1-INH	2.77	4-NH	-0.37
4-INH-IMM	1.13	BC	-0.39
2-NH	0.88	1-INH-IMM	-0.39
1-INH-VAC	0.85	3-INH	-0.44
4-INH	0.63	2-INH-VAC	-0.47
2-INH-IMM	0.60	4-INH-VAC	-0.52
R	0.55	4-NH-iIMM	-0.55
1-NH-iIMM	0.40	2-INH	-0.84
3-INH-VAC	0.25	CC	-0.86
3-INH-IMM	0.22	3-NH	-1.29
2-NH-iIMM	0.01	1-NH	-1.79
3-NH-iIMM	-0.35		

Table 2 The $rank$ of the strategies for RND

In Table 2 we see the $rank$ value for the strategies on RND. We can see that 1-INH and 4-INH-IMM have good $rank$. We can also see that R has a high $rank$, indicating that it can be hard to do much better than random.

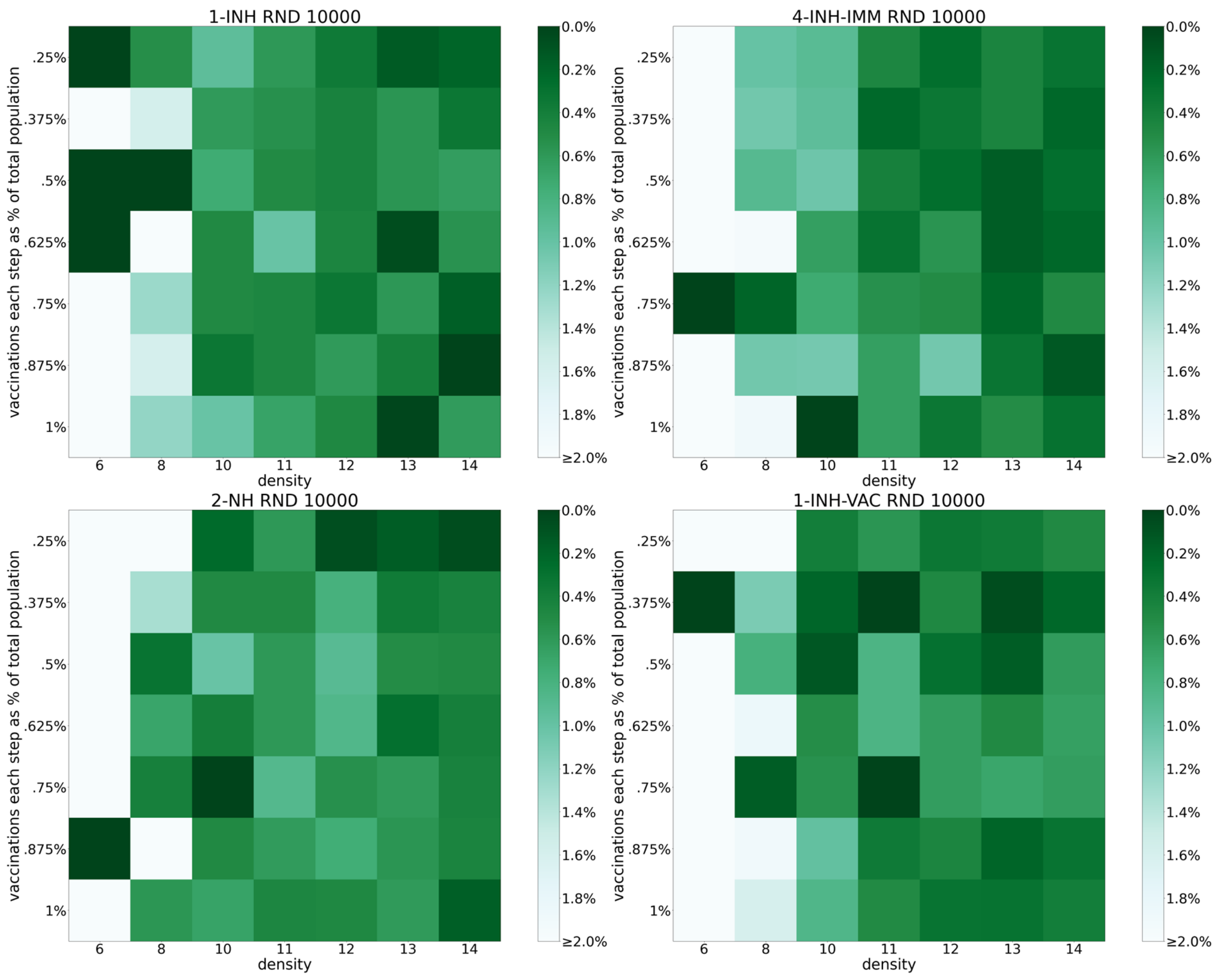

Figure 15 The 4 best strategies according to $rank$, in order from left to right, and top to down.

Figure 15 shows the 4 best strategies for RND. We see that the reason for the high $rank$ of 1-INH is its performance on the low density, low vaccination part of the figure, as well as average performance on the rest of the figure. The other three strategies look similar, and (as does those in the overview Figure 13).

5.3 Results for SWG

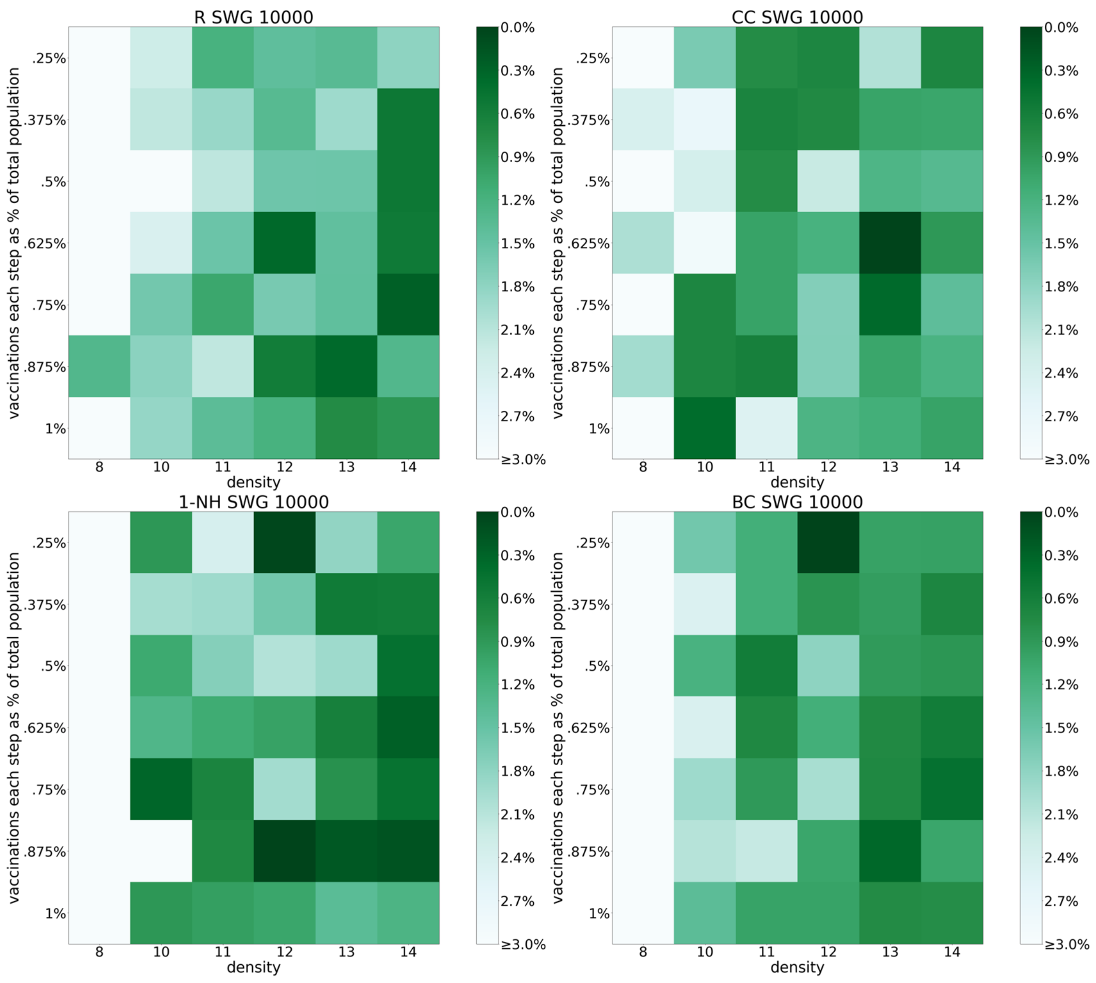

Figure 16 An overview of 4 different strategies (Random (R), closeness (CC), 1-neighbourhood (1-NH), and betweenness (BC)) on SWG with 10000 *vertices.*

Figure 16 we see the effectiveness the 4 overview strategies. Of these, 1-NH has the highest $rank$ value, and R the lowest, although they are both low. Similarly, to SWG, we see that these strategies does not perform good on the low densities (8).

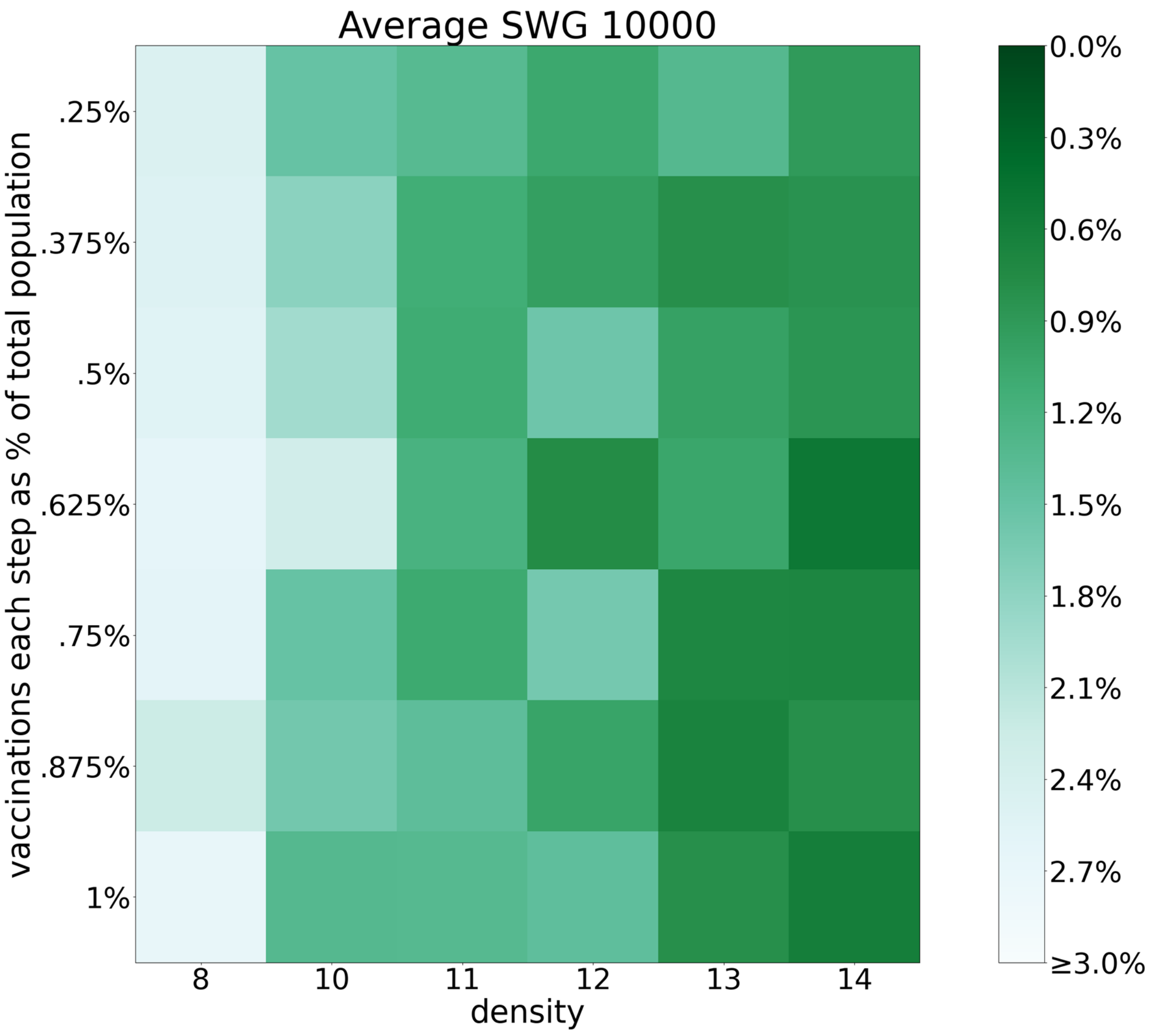

Figure 17 The average performance of the strategies on SWG

In Figure 17 we see the average performance of the strategies on SWG. Like for RND, the strategies on average perform bad on the lowest densities (here 8) and performs closer to the average as the density increases. We also see as the vaccination fraction increases it is easier to perform close to the average.

Strategy	Rank	Strategy	Rank
1-NH-iIMM	2.62	4-NH-iIMM	-0.07
2-NH-iIMM	2.28	1-NH	-0.24
3-INH-IMM	2.14	2-INH-IMM	-0.38
3-INH-VAC	1.67	3-INH	-0.67
3-NH-iIMM	1.38	BC	-0.70
3-NH	0.95	CC	-0.93
2-INH	0.79	1-INH-IMM	-1.32
4-INH-IMM	0.69	1-INH-VAC	-1.75
4-INH-VAC	0.67	1-INH	-2.10
2-NH	0.49	2-INH-VAC	-2.30
4-NH	0.22	R	-3.39
4-INH	-0.06		

Table 3 The $rank$ of the strategies for SWG

In Table 3 we see the ranks for the strategies on SWG. We can observe that k-NH-iIMM performs good, having good rank for $k \in \{1,2,3\}$, while for $k = 4$ the performance is close to average. This shows that the recalculation of the k-neighbourhoods can be important. This is also supported by that both 3-INH-IMM and 3-INH-VAC perform good, while 3-INH performs a bit worse than average. As these 3 strategies use the number of infected in the 3-neighbourhood as the first key in their sorting, and only differ by their second key. This second key being the size of the k-neighbourhood minus either the number of immune or vaccinated in the neighbourhood (0 for k-INH). For 3-INH-IMM and 3-INH-VAC the second key seems have a similar effect as recalculating the neighbourhoods (this holds for a lesser extent for 4-INH-IMM and 4-INH-VAC). Lastly, we see that R performs the worst, indicating that it is hard to do worse than random.

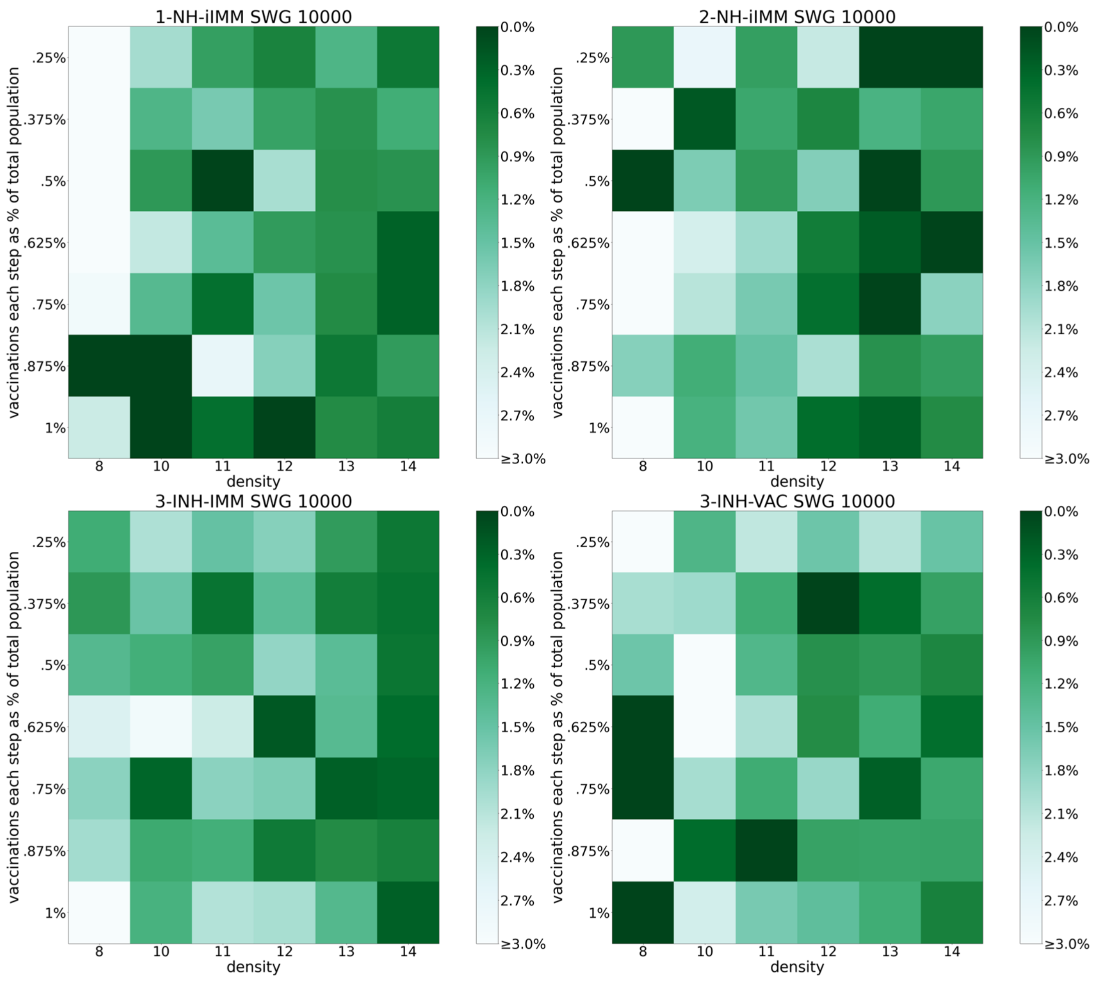

Figure 18 The 4 best strategies on SWG according to $rank$, in order from left to right, and top to down.

In Figure 18 we can see that 1-NH-iIMM, 2-NH-iIMM, and 3-INH-VAC have good rank because they have some very good performing configurations. While for 3-INH-IMM we see a more consistent performance over all the densities and vaccination fractions, and as the average is bad in the low densities this gives a good $rank$.

5.4 Results for GEO_32

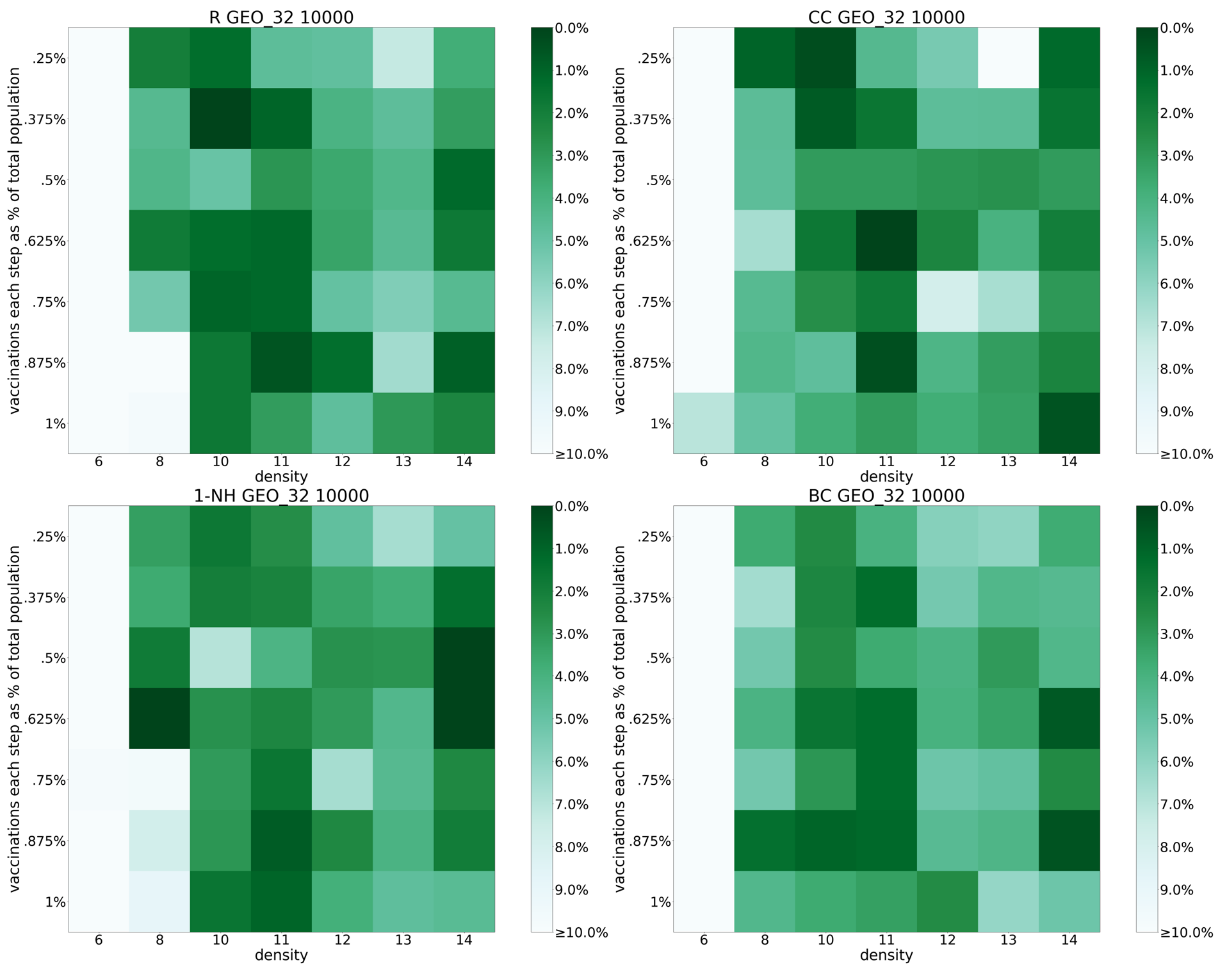

Figure 19 An overview of 4 different strategies (Random (R), closeness (CC), 1-neighbourhood (1-NH), and betweenness (BC)) on GEO_32 with 10000 *vertices.*

In Figure 19 we see the effectiveness the 4 overview strategies. Of these, CC has the highest $rank$ value, and BC the lowest, although they are both amongst the lowest $rank$s. Like for RND and SWG, we see bad performance on the low density (6).

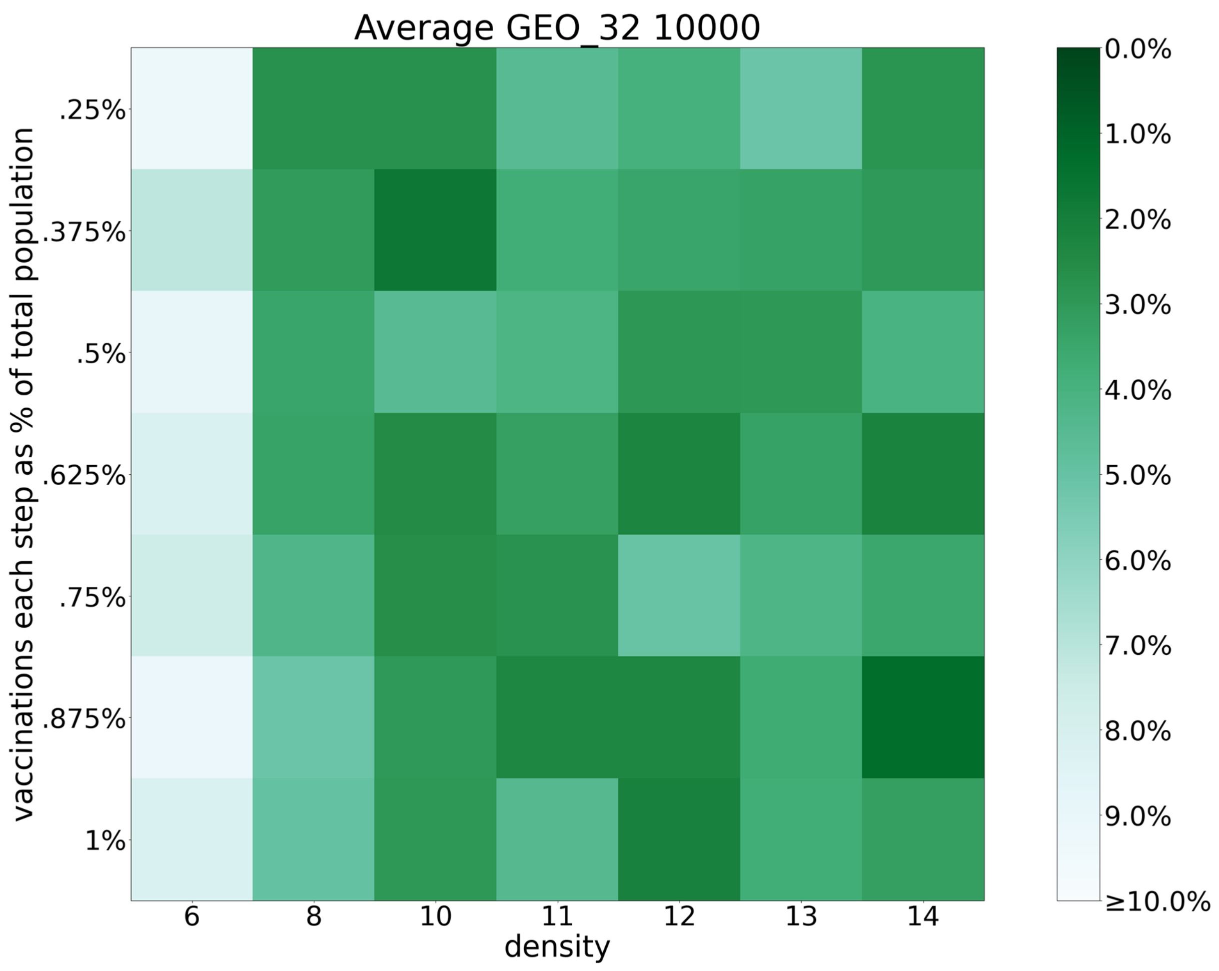

Figure 20 The average performance of the strategies on GEO_32

Figure 20 shows the average performance on GEO_32. One can observe that, unlike for RND and SWG, the average performance is consistent (except for 6) when the density increases.

Strategy	Rank	Strategy	Rank
4-INH	3.98	3-INH-VAC	-0.57
4-INH-IMM	2.70	4-NH	-0.65
2-INH-VAC	2.04	3-NH-iIMM	-0.72
2-INH-IMM	1.91	4-INH-VAC	-0.82
2-INH	1.69	1-INH	-0.83
2-NH	0.92	CC	-0.85
2-NH-iIMM	0.26	3-NH	-1.15
1-INH-VAC	0.21	1-NH	-1.38
3-INH	-0.06	R	-1.49
1-NH-iIMM	-0.24	BC	-1.79
3-INH-IMM	-0.26	4-NH-iIMM	-2.61
1-INH-IMM	-0.31		

Table 4 The $rank$ of the strategies for GEO_32

In Table 4 we see the $rank$ of the strategies on GEO_32. We can see that 2-INH-VAC, 2-INH-IMM, and 2-INH perform good, and close to each other. Similarly, 4-INH and 4-INH-IMM perform good, although it is unclear why 4-INH-VAC performs poorly as these strategies are similar. For the k-NH-iIMM we see bad performance, where 4-NH-iIMM has the lowest $rank$. We can also see that BC performs bad, together with the bad performance of R, it seems that BC and 4-NH-iIMM bad strategies.

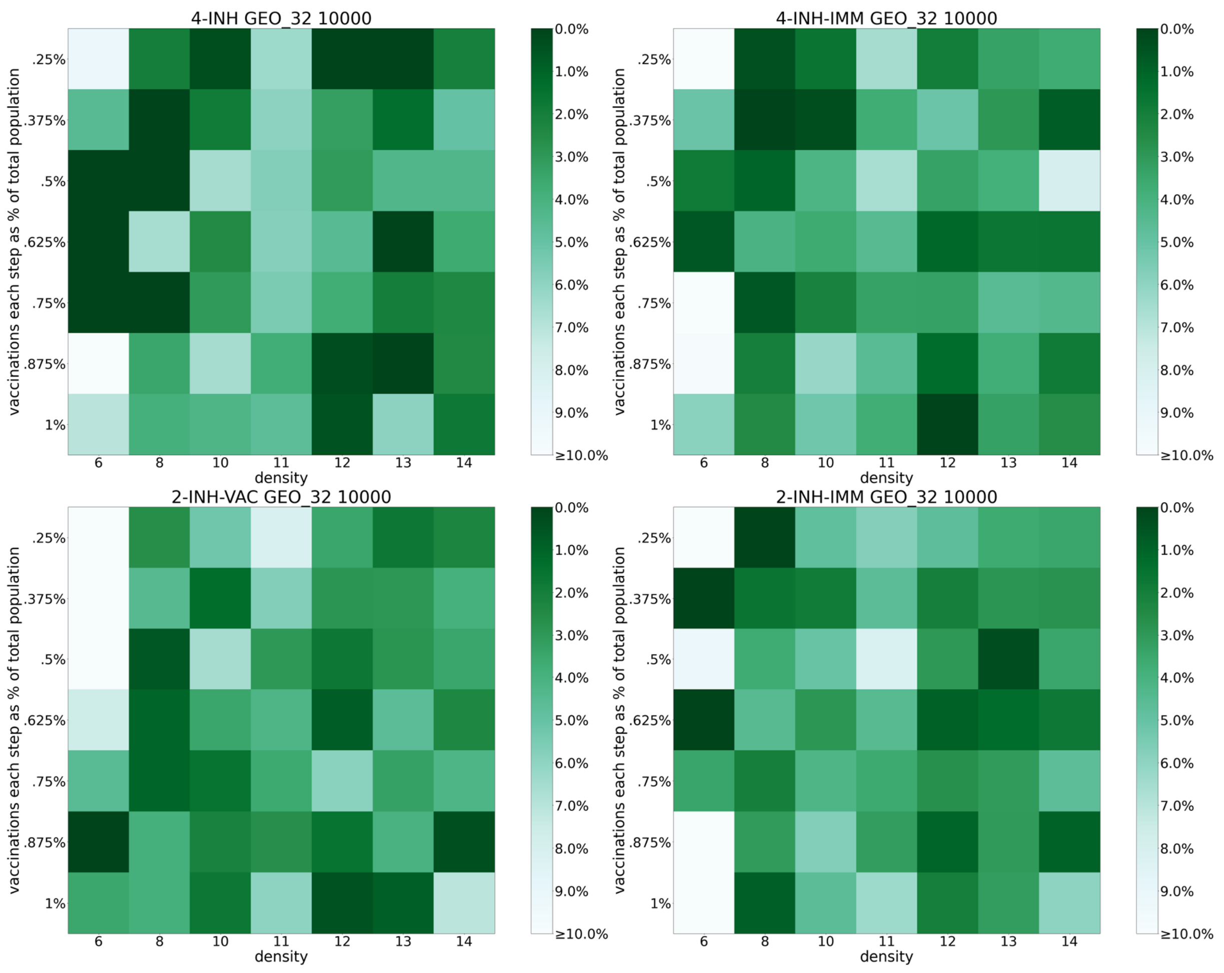

Figure 21 The 4 best strategies on GEO_32 according to $rank$, in order from left to right, and top to down.

In Figure 21 we see the 4 best performing strategies for GEO_32. We can see that 4-INH has good performance when the density is low (an area where the average is bad), this leads to the significantly higher $rank$ value than the next three. Overall, we see a good performance over all the densities and vaccination fractions for all the 4 best performing strategies.

5.5 Results for GEO_128

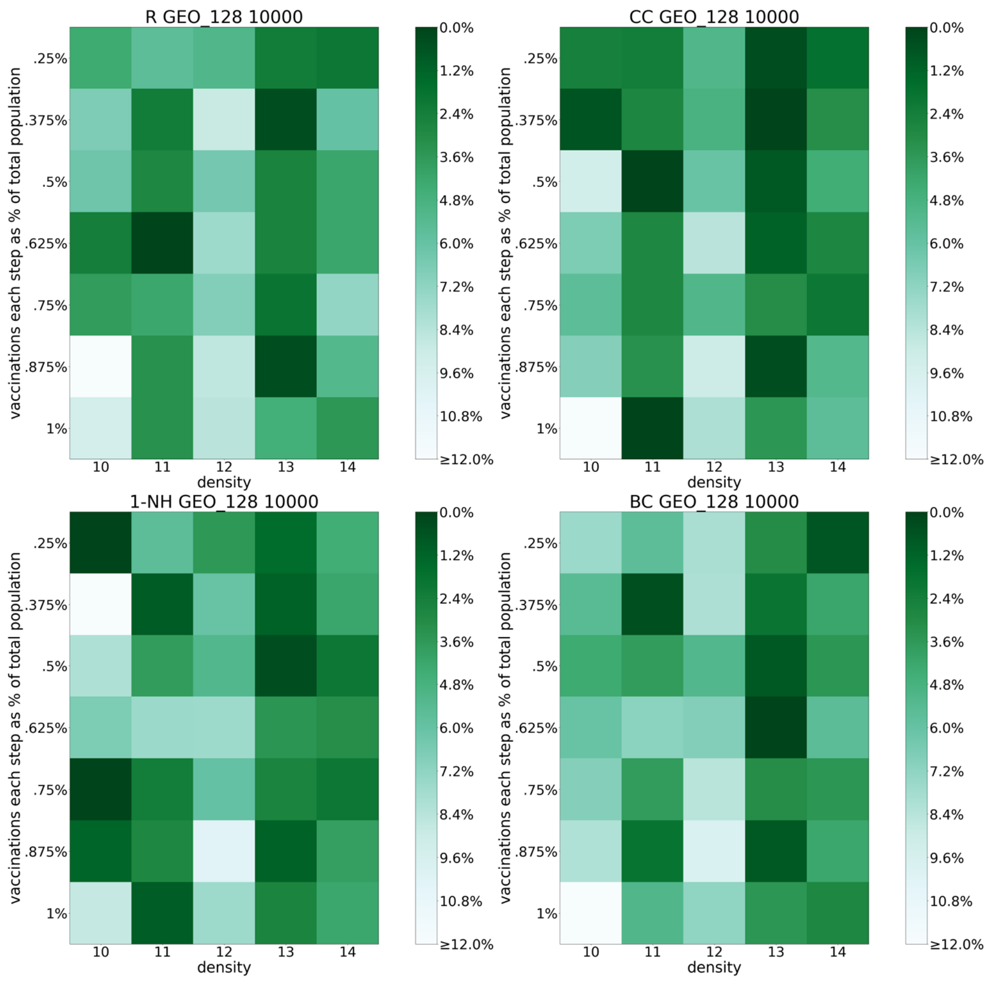

Figure 22 An overview of 4 different strategies (Random (R), closeness (CC), 1-neighbourhood (1-NH), and betweenness (BC)) on GEO_128 with 10000 *vertices.*

In Figure 22 we see the effectiveness the 4 overview strategies. Of these, CC has the highest $rank$ value (followed close by 1-NH), and BC the lowest. Unlike for the other graph classes we do not see bad performance on the low densities for these strategies, but there are strategies that perform bad on this density.

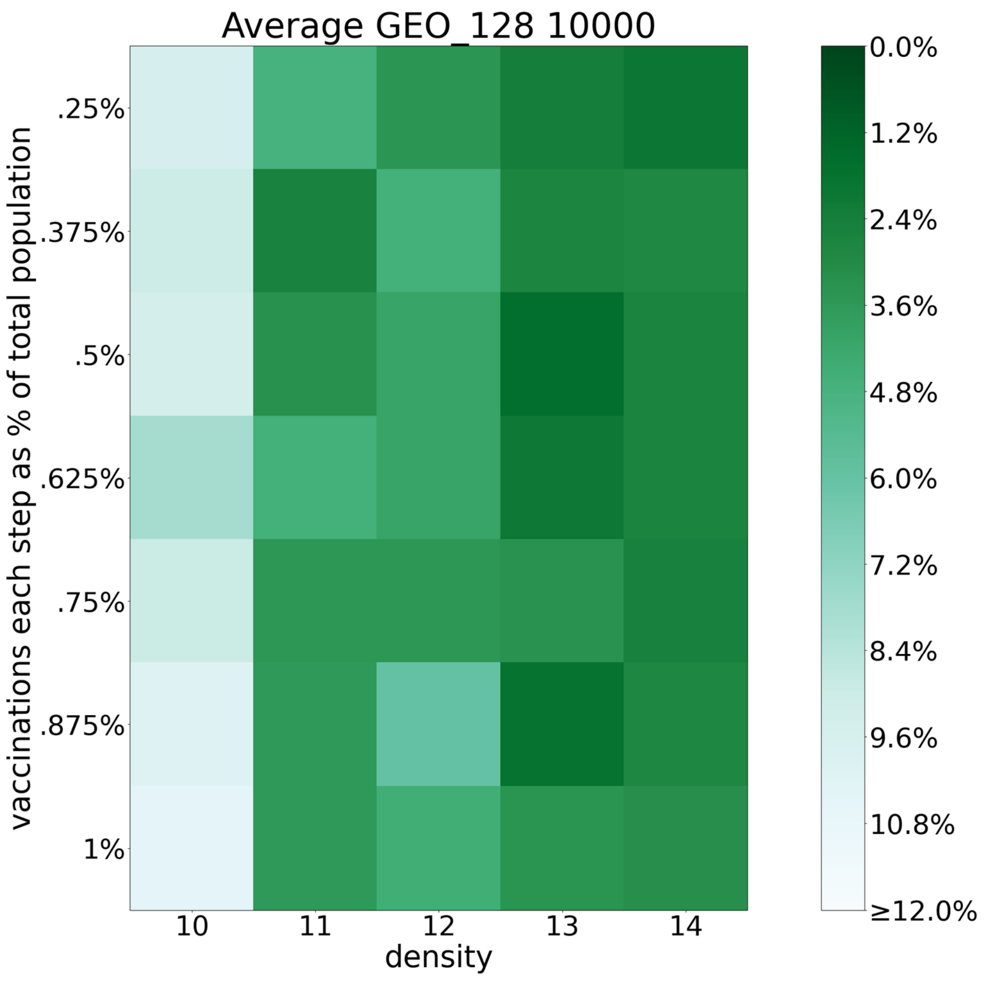

Figure 23 The average performance of the strategies on GEO_128

In Figure 23 we see the average performance of the strategies on GEO_128. For density 10 we see that the average performance is bad, while for the other densities the performance is consistently good, with some increase for the higher densities.

Strategy	Rank	Strategy	Rank
3-NH-iIMM	1.81	2-INH-VAC	-0.33
CC	1.61	1-INH-VAC	-0.35
1-NH	1.16	2-INH	-0.37
4-NH-iIMM	0.72	3-INH-IMM	-0.53
1-INH	0.42	3-INH	-0.53
3-NH	0.41	4-INH-VAC	-0.54
2-INH-IMM	0.40	4-INH-IMM	-0.58
1-INH-IMM	0.40	R	-0.65
4-NH	0.34	BC	-0.93
2-NH-iIMM	-0.01	4-INH	-1.04
1-NH-iIMM	-0.16	3-INH-VAC	-1.05
2-NH	-0.21		

Table 5 The $rank$ of the strategies for GEO_128

In Table 5 we see the $rank$ of the strategies for the different strategies on GEO_128. We can see that, unlike for the other graph classes, both CC and 1-NH performs good. Similarly, to SWG and GEO_32, we see that R gets a low $rank$, indicating that most strategies are better than average. We can also see that 3,4-NH-iIMM does good.

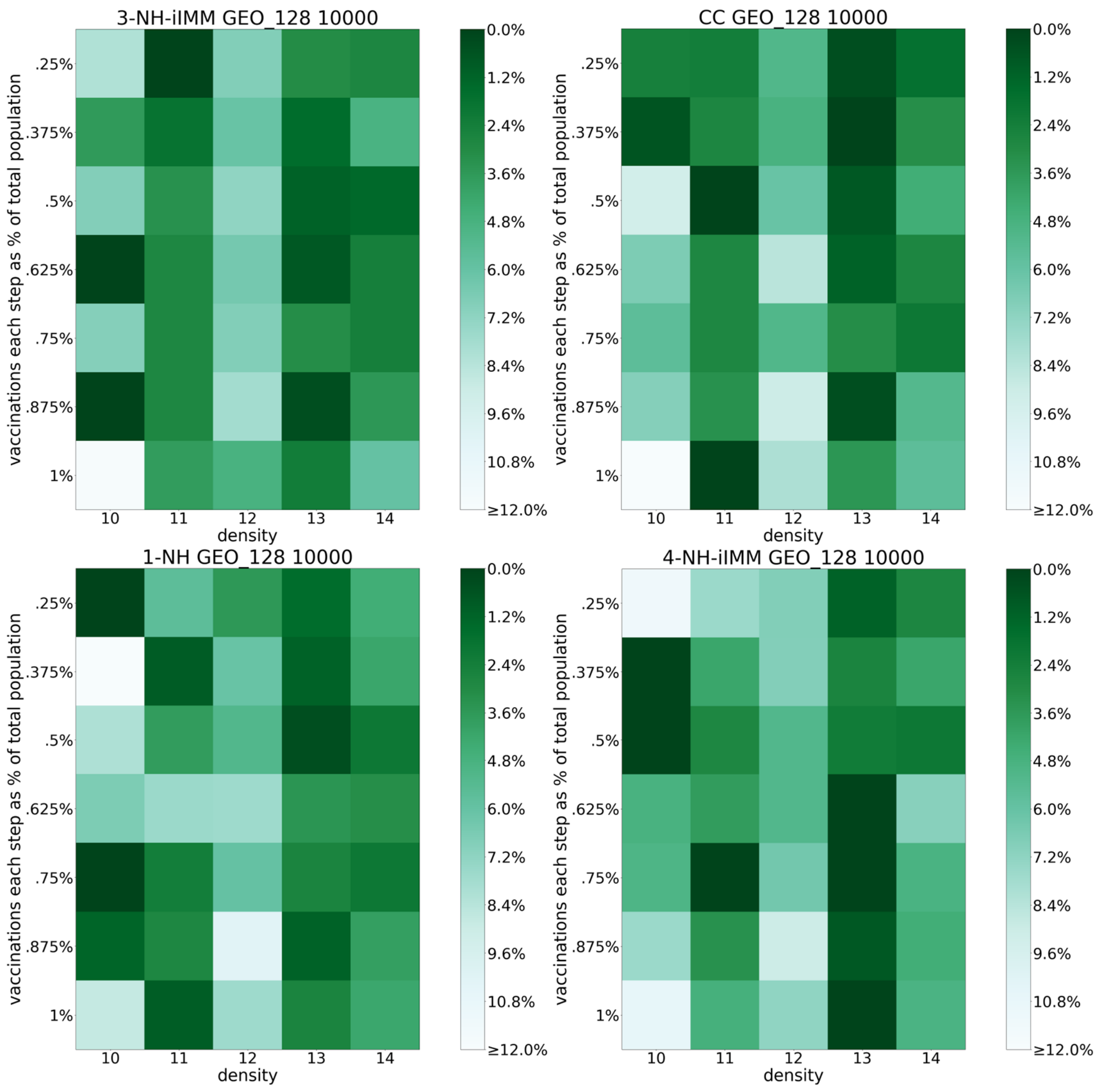

Figure 24 The 4 best strategies on GEO_128 according to $rank$, in order from left to right, and top to down.

In Figure 24 we see the 4 best performing strategies for GEO_128. We can see that all the strategies do good, this is also supported by their similar $rank$.

Note: In Figure 22 and Figure 24 we weak performance for density 12. This can also be seen in the average performance (Figure 23). This is because 1-INH-IMM and 2-INH-IMM (seen in Figure 25) performed good for this density. The reason they have bad $rank$ value is from the bad performance on density 10. The bad performance on 10 for these has a large effect on the average and is the main reason why the average looks bad for density 10.

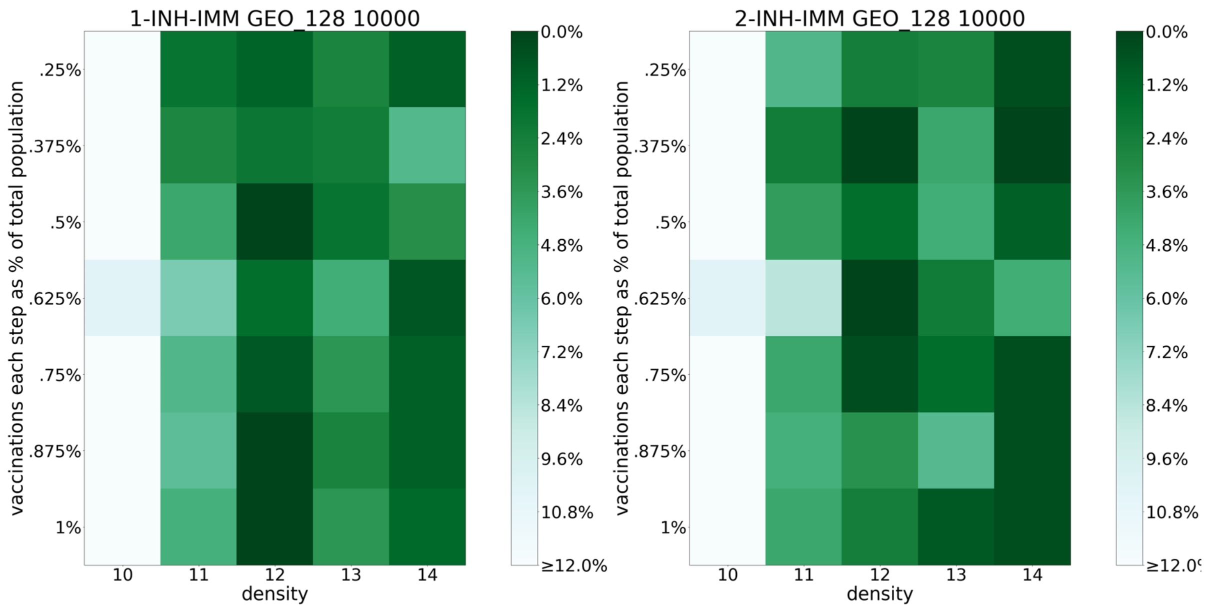

Figure 25 The strategies 1-INH-IMM and 2-INH-IMM on GEO_128.

5.6 Comparing results between graph classes

In Section 5.1.3 we introduced a cut-off value c for each of the graph classes, with c set to 2%, 3%, 10%, and 12%, for RND, SWG, GEO_32, and GEO_128 respectively. This indicates that the spread in the performance of the strategies is different depending on the graph classes. Moreover, as c increases when the diameter of the graphs increases, as $diam(\text{RND}) < diam(\text{SWG}) < diam(\text{GEO\_32}) < diam(\text{GEO\_128})$, we can see that as the diameter increases some strategies become stronger. As can be seen with CC, where on GEO_128 it has the 2nd highest $rank$, while for the other graph classes CC has a low $rank$.

For BC we see that it consistently scores low. This can be explained by that betweenness centrality is unstable, in that the centrality of a vertex can change significantly when some other vertex is removed. As we do not recalculate the betweenness, this can become a problem.

For R (random) we can see that, except for on RND, the strategy is among the worst performing. This is to be expected as most of the strategies end up prioritizing vertices that have high degree.

Overall, we see that the k-INH-group (k-INH, k-INH-IMM, and k-INH-VAC) does good. As all these prioritize nodes that are close to infected nodes and have a large neighbourhood. These strategies end up vaccinating nodes that have a high possibility of being infected in the next few steps. For k-INH-IMM and k-INH-VAC we also prioritize nodes that can reach the most infectable nodes in few steps, unlike k-INH, which can deprioritize nodes that are effectively isolated.

For k-NH-iIMM, we see that for RND and GEO_32 it scores among the average. For SWG and GEO_128 k-NH-iIMM scores good, having the 1st and 2nd best $rank$ on SWG with $k = 1, 2$, and the 1st and 4th best $rank$ on GEO_128. As this strategy uses the number of nodes distance at most k away, when we "remove" immune nodes, we get a similar effect to that of k-INH-IMM where we deprioritize nodes that are likely isolated (at least for higher values for k).

For k-NH we see that it performs good on SWG and GEO_128, while for RND and GEO_32 we see much worse performance. This strategy is like k-NH-iIMM, but does not consider infected nodes, or vaccinated/immune nodes. This gives a similar disadvantage to that of BC and CC.

6 Conclusion

6.1 Strategies

Using the simulation and vaccination strategies described in Chapter 4, we found that when the diameter for the graphs increases, there is more spread in the performance of the different strategies. We saw that the strategies that does not use the state of the simulation (BC, CC, k-NH) performs less consistently good. Both k-NH-iIMM and the k-INH-group perform good on all the graph classes for some k's, indicating that it is important to have strategies that use the state of the simulation for their prioritization of which nodes to vaccinate.

6.2 Algorithms for centralities

In Chapter 3 we introduced 3 algorithms, *k-neighbourhoods, distance-layers, betweenness-centrality*, and showed that our algorithms are fast for graphs with density less than 30, and with $n \leq 10000$.

6.3 Future work

6.3.1 Simulation

There are several examples of centralities that were not tested here, two such are k-shell centrality, and percolation centrality, which both can perform good. Especially k-shell centrality, which can be updated during the simulation as it is fast to calculate (like for k-NH-iIMM).

We used the SIR model to build our simulation. One could use other compartmental models, like SIS (susceptible – infected – susceptible) or SIRS (susceptible – infected – resistant – susceptible). Where nodes can be infected multiple times. Together with temporary immunity, both from moving from moving back into the susceptible group, and from vaccines. This would give a more realistic model.

6.3.2 Algorithms

In Chapter 3.2 we introduced algorithms *k-neighbourhoods* and *distance-layers*. We only tested for $n \leq 10000$, density ≤ 30, and diameter ≤ 128, but from the figures the algorithms look to have potential for larger graphs. In Chapter 3.3.1 we mentioned preliminary results using vectorized instruction, this should give better performance. In addition, both *k-*

neighbourhoods and distance-layers should have fast parallelized algorithms for both CPU and GPU.

Bibliography

Bader, D. A., Kintali, S., Madduri , K., & Mihail , M. (2007). Approximating Betweenness Centrality. *Bonato, A., Chung, F.R.K. (eds) Algorithms and Models for the Web-Graph*, pp. 124-137. Retrieved from https://doi.org/10.1007/978-3-540-77004-6_10

Brandes, U. (2000, December 15). A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, pp. 163-177.

Chakrabarti, D., Wang, Y., Wang, C., Leskovec, J., & Faloutsos, C. (2008, January 1). Epidemic Thresholds in Real Networks . *ACM transactions on information and system security*, pp. 1-26.

Chen, C., Tong, H., Prakash, B. A., Tsourakakis, C. E., Eliassi-Rad, T., Faloutsos, C., & Chau, D. H. (2016, January). Node Immunization on Large Graphs: Theory and Algorithms. *IEEE transactions on knowledge and data engineering*, pp. 113-126.

Chung, F., & Lu, L. (2001). The Diameter of Sparse Random Graphs. *Advances in Applied Mathematics*, pp. 257-279.

Croccolo, F., & Roman, H. E. (2020, October). Spreading of infections on random graphs: A percolation-type model for COVID-19. *Chaos, Solitons & Fractals*.

Das, K., Samanta, S., & Pal, M. (2018, Febuary 28). Study on centrality measures in social networks: a survey. *Soc. Netw. Anal. Min.*(8), p. 13. doi:https://doi.org/10.1007/s13278-018-0493-2

Erdős, P., & Rényi, A. (1959, December 28). ON THE EVOLUTION OF RANDOM GRAPHS. *Mathematical Institute of the Hungarian Academy of Sciences*.

FHI. (2021, January 06). *Koronavaksinasjon - statistikk*. Retrieved November 01, 2022, from Folkehelseinstituttet: https://www.fhi.no/sv/vaksine/koronavaksinasjonsprogrammet/koronavaksinasjonsstatistikk/

Kermack, W. O., & McKendrick, A. G. (1927, August 01). A contribution to the mathematical theory of epidemics. *Proc. R. Soc. Lond.*, pp. 700-721.

Krause, E. F. (1996, October). Maximizing the Product of Summands; Minimizing the Sum of Factors. *Mathematics Magazine - Vol. 69*, pp. 270-278. Retrieved from https://doi.org/10.2307/2690532

Leskovec, J., & Sosic, R. (2016). SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, p. 1. Retrieved from Standford Snap.

Menezes, M. B., Kim, S., & Huang, R. (2017, June 12). Constructing a Watts-Strogatz network from a small-world network with symmetric degree distribution. *PLoS ONE 12(6)*. Retrieved from https://doi.org/10.1371/journal.pone.0179120

Riondato, M., & Kornaropoulos, E. M. (2014, Feburary 24). Fast approximation of betweenness centrality through sampling. *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, pp. 413-422. Retrieved from https://doi.org/10.1145/2556195.2556224

Tarjan, R. E., & Van Leeuwen, J. (1984, April). Worst-case Analysis of Set Union Algorithms. *Journal of the ACM 31*, pp. 245-281. Retrieved from https://doi.org/10.1145/62.2160

Watts, D. J., & Strogatz, S. H. (1998, June 04). Collective dynamics of ‘small-world’ networks. *Nature 393*, pp. 440-442. Retrieved from https://doi.org/10.1038/30918

Appendix 1 Upper bound for number of shortest paths

We show that $3^{\frac{n}{3}}$ is an upper bound for the number of shortest paths between two vertices v_1 and v_2 in a graph $G = (V, E)$ with $|V| = n + 2$. We can do this, as if $3^{\frac{n}{3}}$ is an upper bound, then $3^{\frac{n+2}{3}} = 3^{\frac{2}{3}} \cdot 3^{\frac{n}{3}} \in O\left(3^{\frac{n}{3}}\right)$ is also an upper bound.

If we run BFS from a vertex v_1 in G, we get all the distances from v_1 to every other vertex in the graph. If we want to get all the shortest paths between v_1 and some other vertex v_2, we can go from v_2 to any neighbour of v_2 that is closer to v_1 than v_2 is. Then again for each of these vertices. If we have a graph where all the edges between the vertices with distance k from v_1, and all the vertices with distance $k+1$ from v_1, exist, that is, $\forall u, v \in V, dist(v_1, u) + 1 = dist(v_1, v) \rightarrow (u, v) \in E$, and where we have v_2 being the vertex strictly furthest away from v_1, that is $\forall u \in V\backslash\{v_2\}, dist(v_1, u) < dist(v_1, v_2)$. If we let c_k be the number of vertices with distance k from v_1, and $dist(v_1, v_2) = l + 1$, then the number of shortest paths between v_1 and v_2 is given by $\prod_{i=1}^{l} c_i$, as we can choose to go from any vertex k away (c_k) to any vertex $k+1$ (c_{k+1}) away. We also have that $\sum_{i=1}^{l} c_i = n$. Since we want an upper bound, we want to maximize $\prod_{i=1}^{l} c_i$, either by changing the sizes of the c_i's or by changing l. This gives the following problem:

$$1 \leq l \leq n \text{ and } \forall 1 \leq i \leq l : 1 \leq c_i \leq n$$

$$Maximize\ C = \prod_{i=1}^{l} c_i \text{, while } \sum_{i=1}^{l} c_i = n$$

This is shown (Krause, 1996) to have solutions:

Let C^* be the maximal value for C, then

$$n \equiv 0\ (mod\ 3) \rightarrow C^* = 3^{\frac{n}{3}}$$

$$n \equiv 1\ (mod\ 3) \rightarrow C^* = 2^2 \cdot 3^{\frac{n-4}{3}}$$

$$n \equiv 2\ (mod\ 3) \rightarrow C^* = 2 \cdot 3^{\frac{n-2}{3}}$$

In general, this gives an upper bound for $C^* \leq 3^{\frac{n}{3}}$.

Appendix 2 Number to edge function

We will construct a function M that generates all the edges for a graph G with n vertices.

$M\colon \left[0, \frac{n(n-1)}{2} - 1\right] \to (a, b)$, with $a < b$.

$$M(x) = \left(x - \frac{\delta(x)^2 - \delta(x)}{2}, \delta(x)\right), \delta(x) = \left\lfloor \frac{1 + \sqrt{1 + 8x}}{2} \right\rfloor$$

We know that for an undirected graph with n vertices there exists $\frac{n(n-1)}{2}$ edges. We let the edges be on the form $(a, b), a < b$. We want the function M to work with all values for n, and for any value $x \in \left[0, \frac{n(n-1)}{2} - 1\right]$.

We want the edges in the order $(0,1), (0,2), (1,2), (0,3), (1,3), (2,3) \cdots$, that is, in the order such that we first get all edges on the form $(t, 1)$, then $(t, 2)$, etc. We let $\delta(x)$ be the second value of an edge. Clearly there is k edges with k as the second value, as there are k numbers less than k. Because of this $\delta(x)$ only increases when the inverse of $\frac{y(y-1)}{2}$ reaches a new integer value. From this we get $\delta(x) = \left\lfloor \frac{1+\sqrt{1+8x}}{2} \right\rfloor$, the floor of the inverse of $\frac{y(y-1)}{2}$.

When $\delta(x)$ increases, the first value in the edge is 0. We know that when $\delta(x)$ changes,

As $\delta(x)$ is smaller than or equal to the inverse of $\frac{y(y-1)}{2}$, hence $x = \frac{\delta(x)(\delta(x)-1)}{2} = \frac{\delta(x)^2-\delta(x)}{2}$. As $\delta(x)$ does not increase before x has increased by $\delta(x)$, we can set the first value of the edge as $x - \frac{\delta(x)^2-\delta(x)}{2}$. This gives the desired function.