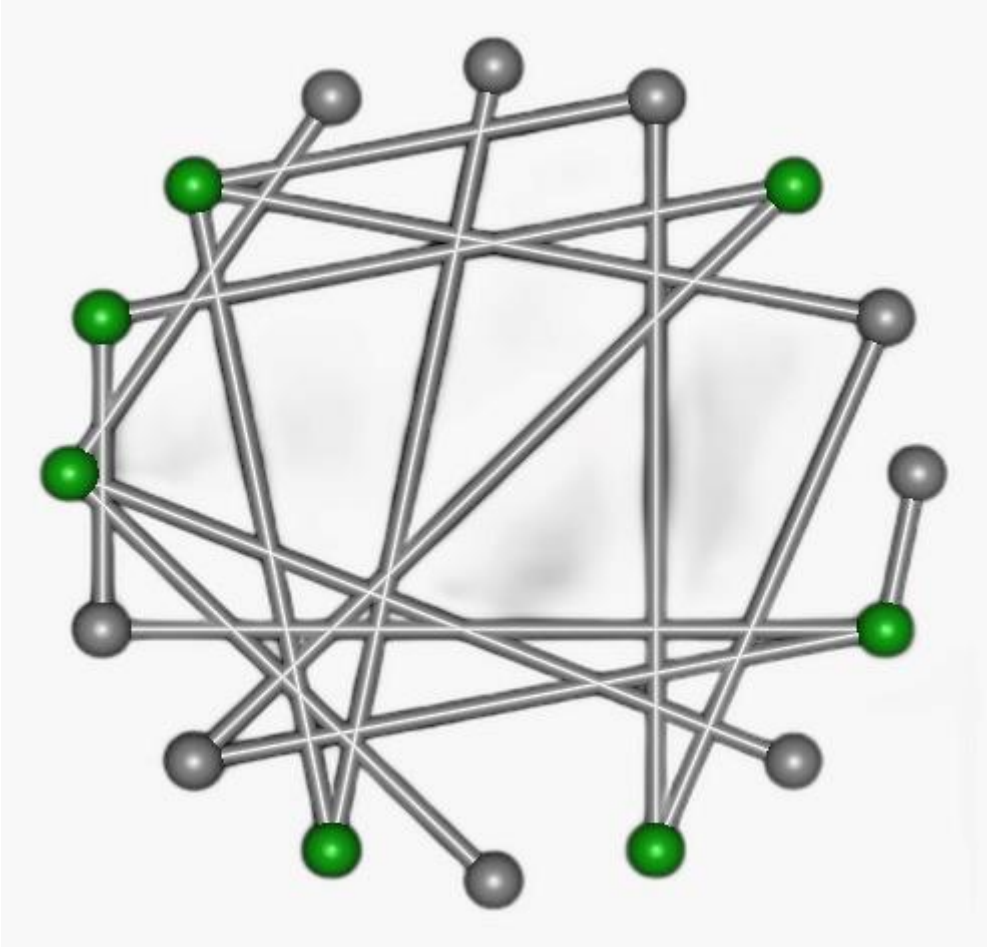
Final Project Report

Min Vertex Cover

Final project report for the course "Introduction to Artificial Intelligence" (67842)



Tal Joseph, Michael Joseph, Rotem Brillant, Israel Ben-Attar

Hebrew University of Jerusalem

Table of Contents

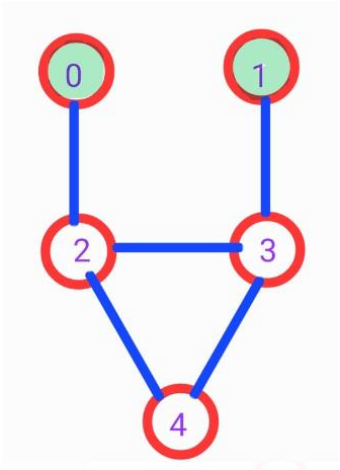
Introduction

In our project we decided to deal with the subject of Minimum Vertex Cover Problem. This is an NP-complete problem (one of Karp's 21 NP-complete problems), and therefore can be solved in polynomial-time if and only if P=NP. There is a well-known 2-approximate polynomial time algorithm which is the best approximation algorithm found so far. It has also been proven that it is NP-Hard to approximate within any factor smaller than 1.3606. [1]

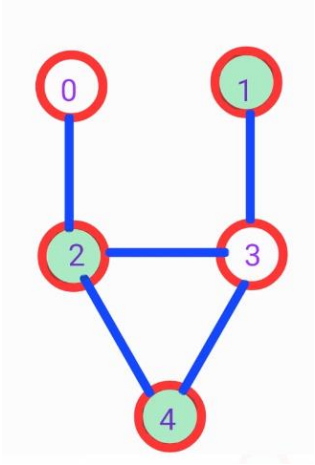
The Minimum Vertex Cover problem is defined as follows:

A vertex cover V' of an undirected graph $G = (V, E)$ is a subset of V such that $\forall (u, v) \in E \rightarrow u \in V' \; or \; v \in V'$. The Minimum Vertex Cover problem is to find the minimum sized vertex cover in $G = (V, E)$. I.e., if V' is a minimum vertex cover in G then $\forall V'' \; vertex \; cover \; in \; G, |V''| \geq |V'|$.

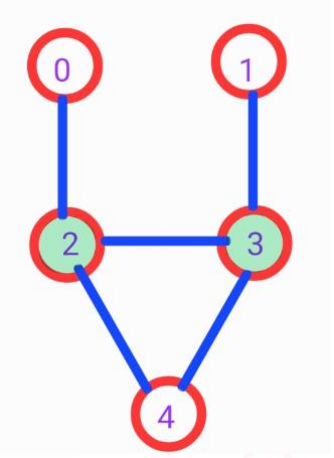
In the following example we see a graph where the green vertices represent the chosen cover:



In this case vertices 0 and 1 are chosen and do not represent a valid vertex cover as the edge (2,4) for example is not covered



Here vertices 1, 2 and 4 are chosen which represent a valid vertex cover, though not minimal



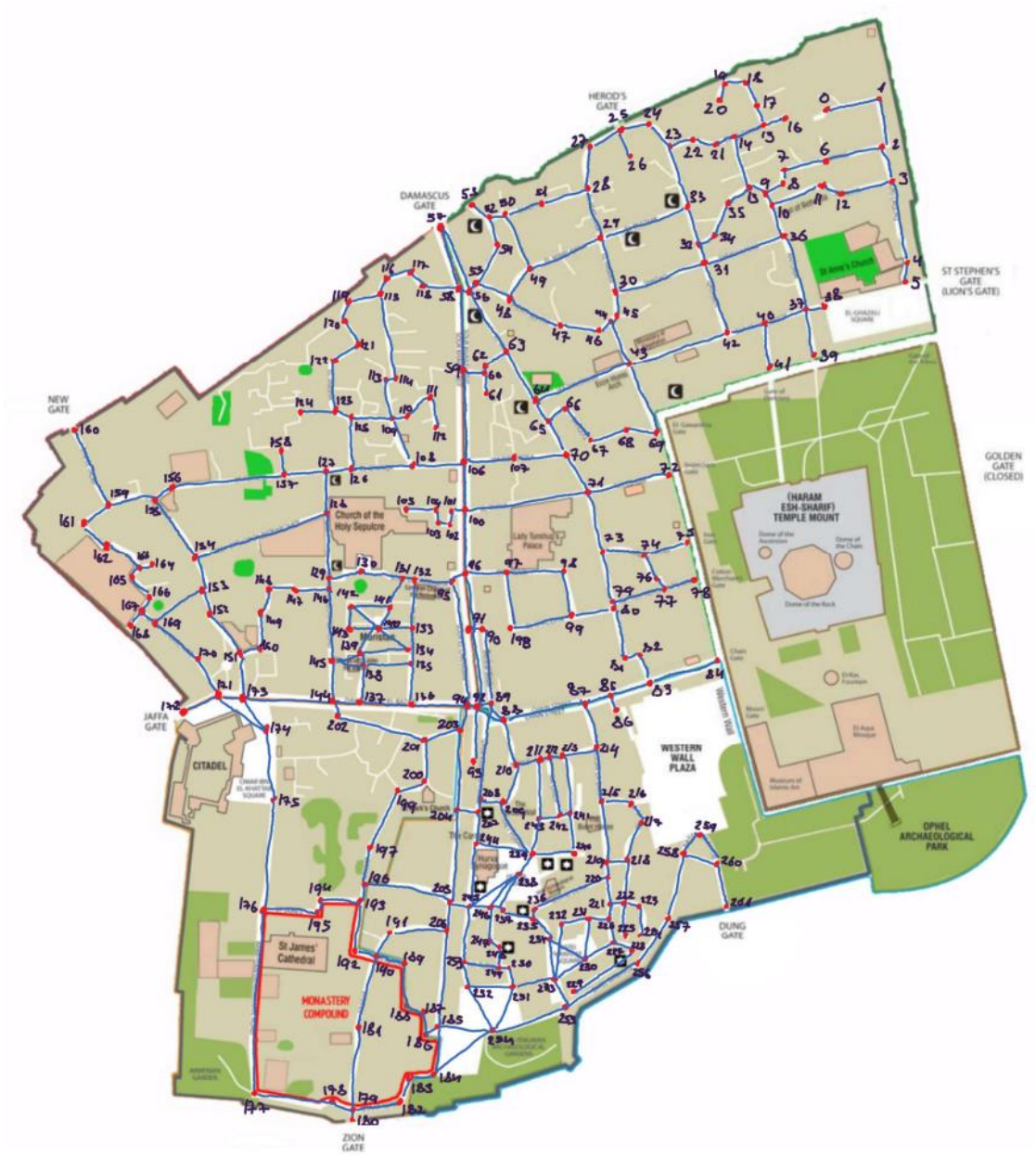
Vertices 2 and 3 are chosen which represent a minimal vertex cover, as clearly no cover with less vertices exists and all edges are covered.

Real Life Usages

As previously discussed, the MVC problem is NP-Complete. Therefore, there is theoretical importance to solving this problem, given that we could then solve any NP-Complete problem efficiently. However, there are many usages for this problem in real life too:

- Computer network security – Researchers used MVC algorithms to simulate propagation of stealth worms on large computer networks and design optimal strategies for protecting the network against such virus attacks in real-time. The servers are represented by the vertices while connections between servers are represented by the edges between the vertices.
- Computational Biology – Resolving conflicts between sequences by removing some sequences. A conflict graph is defined by each sequence being represented by a vertex and each conflict between 2 sequences is represented by an edge between the corresponding vertices. In this problem we'd like to remove as little as possible sequences so there will be no more conflicts between sequences. The way to do so is to find a MVC and remove the cover found, and then the remaining graph contains no edges, therefore no conflicts. [2]
  We can perform this operation on any problem which can be represented by a conflict graph where we'd like to get rid a minimal number of vertices such that no conflict will remain.
- Security Cameras – Suppose there is an art gallery, and we'd like to place security cameras to cover all hallways (against theft). We'd like to use a minimal number of cameras for financial reasons. We can represent each junction of hallways as a vertex, each hallway between 2 junctions will be an edge. Once we find a VC, we can place a $360°$ camera at each junction of the VC to cover all hallways. [3]

An example where finding a MVC to minimize the number of security cameras used is in the Old City of Jerusalem, where security is of great importance in almost every nook.



The illustration above shows a map of the Old City of Jerusalem marked with small red dots representing optional camera positions, i.e., vertices, to cover all the streets in the Old City. We marked edges between 2 camera positions if they can recognize each other. There is a total of 262 optional camera positions and 338 edges that cover the entire Old City.

We will discuss later in the report how our algorithms performed on this important task.

Approaches and Overview

This is an optimization problem where the path to the goal is irrelevant. The only thing which matters is the goal state itself, the solution. Local Search algorithms are useful for these types of problems and therefore, we chose to investigate what results can be accomplished using the local search methods learnt in the course. Those include:

1. Hill-Climbing
2. Simulated annealing
3. Local Beam Search
4. Genetic Algorithms

Later in the report we will elaborate on the different local search methods and our implementations. We attempt to reach a decent approximation within a reasonable time.

2-approximatation Algorithm

The following is a well-known 2-approximation algorithm for the Minimum Vertex Cover (MVC) problem. In other words, this algorithm returns a vertex cover that is at most twice as big as the minimum vertex cover in the graph.

```
APPROXIMATION-VERTEX-COVER(G)
C = ∅
E'= G.E

while E' ≠ ∅:
    let (u, v) be an arbitrary edge of E'
    C = C ∪ {u, v}
    remove from E' every edge incident on either u or v

return C
```

Run time complexity – $O(|E|)$

MVC Heuristics

To traverse through the solution space, local search algorithms use cost functions to advance to a new state. We define a state in the MVC problem to be the current vertices in the cover. A neighbouring state is defined by removing one vertex from the current state or adding a vertex to the current state. We used several different cost functions in our implementations:

1) $(2+\epsilon) * |E'| - \frac{\sum_{v \in V'} |V| - \deg(v)}{|V|}$ where V' is the current state (vertices in the current cover) and E' are the edges covered by V'.
   This cost function guarantees that while we haven't reached a vertex cover, the state defined by adding the vertex with the most uncovered edges has the highest score. Also, the cost maintains the property – adding a vertex that covers a new edge is better than not adding at all. Once reaching a vertex cover, the state defined by subtracting the vertex with the lowest degree that can be removed, while maintaining a valid vertex cover, has the highest score.
2) $\Sigma_{e \in E'} w(e)$ where $w: E \to \mathbb{R}_{\geq 0}$. The weighting function will be explained in the relevant algorithms.
3) $\Sigma_{v \in V'} w(v)$ where $w: V \to \mathbb{R}_{\geq 0}$
4) $2 * |E'| - |V'|$. This cost function performs similarly to the first cost function, except once reaching a vertex cover, all states in which we subtract a vertex, while staying with a valid vertex cover, have the same score.
5) $|E'| - |V'|$. This cost is bounded by $|E| - |MVC|$. Therefore, it is always in our interest to find a state with maximum cost. If we find a state with cost C, then we can find a valid vertex cover of size at most $|E| - C$ simply by adding a vertex for each edge not in the cover.
6) $|E'| - |V'| - punishment$ where $punishment > 0$ if V' does not represent a valid vertex cover.

Hill-Climbing

In hill-climbing we move iteratively to a neighbouring state with a higher score until there exists no neighbouring state with a better score.

In this approach, we tested different hill-climbing variants and different cost functions.

The Greedy Approach:

In every iteration, pick the neighbour with the highest score. If there is more than one, pick randomly between them.

We implemented Greedy Hill Climbing (GHC) in the following ways:

1) greedy_hill_climbing: Implements the 1st cost function and performs regular GHC, starting from a given initial state.
   Run time complexity - $O(\min(|V|,|E|) * |V|^2)$
2) ghc_weighted_edges: Implements the 2nd cost function with weight function $w: E \to \mathbb{R}_{\geq 0}$ initialized to 1 $\forall e \in E$ and with the initial empty state. The weight function is updated every time we advance to a new state - $\forall e \in (E \backslash E')$, $w(e) += 1$, until we reach a vertex cover. We then attempt to remove vertices according to the first cost function approach. We rerun this algorithm as the number of iterations decided upon, while continuing with the weight function from the previous iteration.
   The motivation behind the weight function is to prioritize edges which were covered later in the previous run of the algorithm, and by doing so, exploring different paths towards a vertex cover.
   Run time complexity - $O(\min(|E|,|V|) * (|V|^2 + |E|) * k)$ where k is the number of iterations to run the algorithm.
3) ghc_weighted_vertices: Implements the 3rd cost function.
   First, we define the following graph: $G' = (V, E \backslash E')$. We also define $N(v) = Neighbours(v)\ in\ the\ graph\ G'$. Now we define a weight function $w: V \to \mathbb{R}_{\geq 0}$ over G' s.t $\forall v \in V$:
   $w(v) = 0\ if\ \deg(v) = 0$. Otherwise, the weight is defined:

$$w(v) = \sum_{u \in N(v)} \begin{cases} \min\limits_{u' \in N(u)} (\deg(u')) & if\ \deg(u) \geq \deg(u')\ \forall u' \in N(u) \\ \max\limits_{u' \in N(u)} (\deg(u')) & else \end{cases}$$

   In other words, the weight of each vertex is defined by the sum of scores of all its neighbours in G'. For each neighbour, that score is the minimum degree over all its neighbours if it has a greater or equal degree than all its neighbours. Otherwise, the score is the maximum degree over all its neighbours.
   The motivation behind this weight function is that if a vertex degree is high (higher than all its neighbours' degrees) we would probably rather take that vertex. Therefore, we prefer to return a minimal score to its neighbours in the calculations. This way, the neighbours get a lower weight in the score sum calculations and would likely be chosen after this vertex. On the other hand, if a vertex has a low degree (lower than at least one of its neighbours), we

probably wouldn't want to take that vertex, so we return a maximal score. The neighbours will receive a higher weight in the score sum calculations and will likely be chosen before this vertex.
We perform this weight function untill a VC is discovered. Then we attempt to decrease the number of vertices according to a similar weight function $w: V' \rightarrow \mathbb{R}_{\geq 0}$ over the original graph $G = (V, E)$ and choosing the vertex with the highest weight to remove. The weight function is updated to be:

$$w(v) = \sum_{u \in N(v)} \begin{cases} \deg(u) & if\ \deg(u) \geq \deg(u')\ \forall u' \in N(u) \\ \min_{u' \in N(u) \cup \{u\}} (\deg(u')) & else \end{cases}$$

The motivation behind this weight function is analog to the first.
Run time complexity - $O(\min(|V|, |E|) * |V|^2)$

Stochastic Approach:

In every iteration, select a state randomly among all neighbouring states with a higher score.
We've implemented this approach in the following way:
stochastic_hill_climbing: implements the 4th cost function.
The first cost function will return the same results (any neighbour with a higher score using the first cost function is also higher using the fourth cost function and vice versa). Using cost functions 5 and 6 would yield non-valid VC results as once we have a vertex with one remaining edge uncovered, the cost of adding it is equivalent to the cost without it. Also using the weighted cost functions (3, 4) is pointless, as both would simply pick a random vertex from all the vertices of edges not covered.
Run time complexity - $O(\min(|V|, |E|) * |V|^2)$

First Choice Approach:

In this approach, in every iteration select the first neighbouring state with a better score.
We've implemented this approach in the following way:
first_choice_hill_climbing: implements the 4th cost function.
Similarly to Stochastic Hill Climbing, this is the only relevant cost function.
Run time complexity - $O(\min(|V|, |E|) * |V|^2)$ though in practice works much faster.

Random-Restart Approach:

To avoid local maxima, conducts a series of greedy hill-climbing runs on random initial states.
To ensure randomness of the initial states, we chose the number of vertices in the state uniformly $k \leftarrow [0, |V|]$ and then chose uniformly k vertices from V.
Run time complexity - $O(\min(|V|, |E|) * |V|^2 * k)$ where k is the number of iterations to run GHC.

Simulated Annealing

In Simulated Annealing we attempt to escape local maxima by allowing "bad" moves which probabilistically decrease over time.
To achieve this, we have a temperature function $temp: \mathbb{N} \to \mathbb{R}_{\geq 0}$ s.t $\lim_{t\to\infty} temp(t) = 0$.

Pseudo Code:

```
function SIMULATED-ANNEALING(problem,schedule) returns a state
current ← problem.initial-state
for t = 1 to ∞ do
    T ←schedule(t)
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← next.value – current.value
    if ΔE > 0 then current ← next
            else current ← next only with probability e^(ΔE/T)
```

The way the implementation above works is in every iteration we pick a neighbour randomly. If it has a better value, we advance to that neighbour. However, if it has a worse value: $\Delta E < 0$, we advance to that state only with a probability which is directly affected by the temperature: $e^{\frac{\Delta E}{T}}$ where $T \leftarrow temp(t)$. As $\Delta E < 0$, for large values of T we get $\frac{\Delta E}{T} \approx 1$ while for lower values of T, $\frac{\Delta E}{T} \approx 0$. Therefore, we would like a slowly decreasing temperature function to explore a lot of states first, but later would rather exploit.
The temperature function we used will be discussed later.
We used 3 different cost functions: 4, 5, and 6.
Cost 4 is a straight forward solution, always attempt to add edges/remove useless vertices.
As for cost 5 the motivation is to allow removal of insignificant vertices (vertices that cover only 1 edge) as the cost stays the same, since $\Delta E = 0 \to e^{\frac{\Delta E}{T}} = 1$. This allows to explore more solutions. However, due to the larger exploration while almost at a VC, many times we do not achieve a valid VC. To solve this problem, we can simply run the returned solution through GHC as a starting state.
Cost function 6 is also a solution for this problem, enforces preference for a vertex cover.
Run time complexity - $O\big(|E| * (temp(t) = 0)\big)$ where $temp(t) = 0$ is the number of iterations till $temp(t)$ is equal to 0 (the stopping condition in Simulated Annealing).

Local Beam Search

In Local Beam Search we start from k random states. In each iteration we generate all the neighbours of those k states and keep the k best for the next iteration. The main difference between Local Beam Search and Random Restart is that information is shared between the different states.
In general, the algorithm stops once it reaches a goal state, however, our goal state is to attain a MVC. Since we don't know the size of the MVC, we do not know what state represents a goal state. Therefore, we adjusted the algorithm and return the best solution encountered. Similarly to greedy Hill-Climbing, the 1st cost function was best suited for this algorithm.
Run time complexity – $O\big(|V| * (k * |V| * \log(k * |V|) + k * |V| * |E|)\big) = O\big(k * |V|^2 * (\log(k * |V|) * |E|)\big)$

Genetic Algorithms

A genetic algorithm is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.
Every Genetic Algorithm (GA) has the following 5 properties:

1) Initial population – k individuals, where each one is a state of the problem we want to solve.
2) Fitness function – determines how fit an individual is.
3) Selection – the probability to be chosen for reproduction, usually would be evaluated according to the fitness function.
4) Crossover – point of merge between 2 vectors, which may or may not be chosen randomly.
5) Mutation – to reduce locality, each point is subject to random mutation with some small probability.

Pseudo Code:

```
function GENETIC_ALGORITHM( population, FITNESS-FN) return an
  individual
  input: population, a set of individuals
       FITNESS-FN, a function which determines the quality of the
  individual
  repeat
       new_population ← empty set
       loop for i from 1 to SIZE(population) do
              x ← RANDOM_SELECTION(population, FITNESS_FN)
              y ← RANDOM_SELECTION(population, FITNESS_FN)
              child ← REPRODUCE(x,y)
              if (small random probability) then child ← MUTATE(child )
              add child to new_population
       population ← new_population
  until some individual is fit enough or enough time has elapsed
  return the best individual
```

We've implemented and tested quite a few Genetic Algorithms with the following traits:

- Initial population was selected randomly among all possible states.
- Selection was calculated using Softmax over the fitness of the individuals.
- Fitness: used cost functions 1, 4, 5, and 6
- Reproduce:
  1. select each node with probability p from parent 1, and with probability $1-p$ from parent 2, where $p = fitness1/(fitness1 + fitness2)$.
  2. Select $k \leftarrow [n]$ uniformly as a crossover point between the parents.
- Mutation:
  1. With probability $\frac{1}{n}$, each node is flipped.
  2. With probability $\frac{2}{3}$, same as previous mutation. With probability $\frac{1}{3}$, add a random node of an edge which isn't covered by the state.

Run time complexity in general -

$$O\left(numGenerations * \left(populationSize * (mutation * reproduce + fitness)\right)\right)$$

In our implementations mutation1 runs in $O(|V|)$, mutation2 runs in $O(|V| + |E|)$, reproduce1 and reproduce2 run in $O(|V|)$, and the fitness functions run in $O(|V| + |E|)$

We get run time of:

$$O\left(numGenerations * \left(populationSize * (|V| * |V| + |V| + |E|)\right)\right)$$

Or when using mutation2,

$$O\left(numGenerations * \left(populationSize * \left((|V| + |E|) * |V| + |V| + |E|\right)\right)\right)$$
$$= O\left(numGenerations * (populationSize * (|V| + |E|) * |V|)\right)$$

Results and Conclusions

Below are the results of running our algorithms, as well as the 2-approximation algorithm on 5 different Graphs: a couple of benchmark graphs, a couple of random graphs, and The Old City of Jerusalem graph.

Notice – The genetic algorithms we present in the results use the following fitness, reproduce and mutation functions respectively:

- Regular VC GA – cost4, reproduce1, mutation1
- Regular VC GA2 – cost5, reproduce1, mutation1
- VC Punish GA – cost6, reproduce1, mutation2

p_hat300-3.mis - |V| = 300, |E| = 11460, |OPT VC*| = 264; 10 Iterations

Algorithm Name	Avg VC Size	Min VC Size	Avg Run Time (sec)
2-approximate	294	294	0.016691
Greedy Hill-Climbing	268.5	267	0.09508
Stochastic HC	278.5	273	0.1236
Random Restart HC	265.5	264	4.3398
Simulated Annealing	267.5	266	19.40348
Local Beam Search	278.7	275	38.10352
GHC Weighted Edges	265.3	264	74.76031
GHC Weighted Vertices	266.1	265	0.35721
Regular VC GA – population=10	267.6	266	134.5328
Regular VC GA2 – population=10	266.9	266	142.2055
VC Punish GA – population=10	268.7	266	227.7184

C250.9.ims - |V| = 250, |E| = 3141, |OPT VC*| = 206; 10 Iterations

Algorithm Name	Avg VC Size	Min VC Size	Avg Run Time (sec)
2-approximate	246	246	0.004686
Greedy Hill-Climbing	211.9	209	0.043508
Stochastic HC	217.1	215	0.045313
Random Restart HC	210	209	1.004598
Simulated Annealing	210.1	208	5.129145
Local Beam Search	218.6	217	11.3374
GHC Weighted Edges	207	207	20.9666
GHC Weighted Vertices	209.8	209	0.106251
Regular VC GA – population=9	210.5	208	39.84058
Regular VC GA2 – population=9	209	206	39.2607
VC Punish GA – population=9	211.1	208	61.10549

p-random graph, p=0.006 - |V| = 500, |E| = 724, |OPT VC*| = ?; 10 Iterations

Algorithm Name	Avg VC Size	Min VC Size	Avg Run Time (sec)
2-approximate	362	362	0.003128
Greedy Hill-Climbing	245.4	243	0.035364
Stochastic HC	266.9	260	0.048665
Random Restart HC	243	242	1.794430
Simulated Annealing	240.9	238	2.423276
Local Beam Search	268.2	262	30.58385
GHC Weighted Edges	239.1	238	10.87041
GHC Weighted Vertices	235.3	235	0.037525
Regular VC GA – population=14	250.4	246	29.85816
Regular VC GA2 – population=14	237.8	236	27.30596
VC Punish GA – population=14	254.5	249	40.84746

Random graph - |V| = 500, |E| = 5,000, |OPT VC*| = ?; 10 Iterations

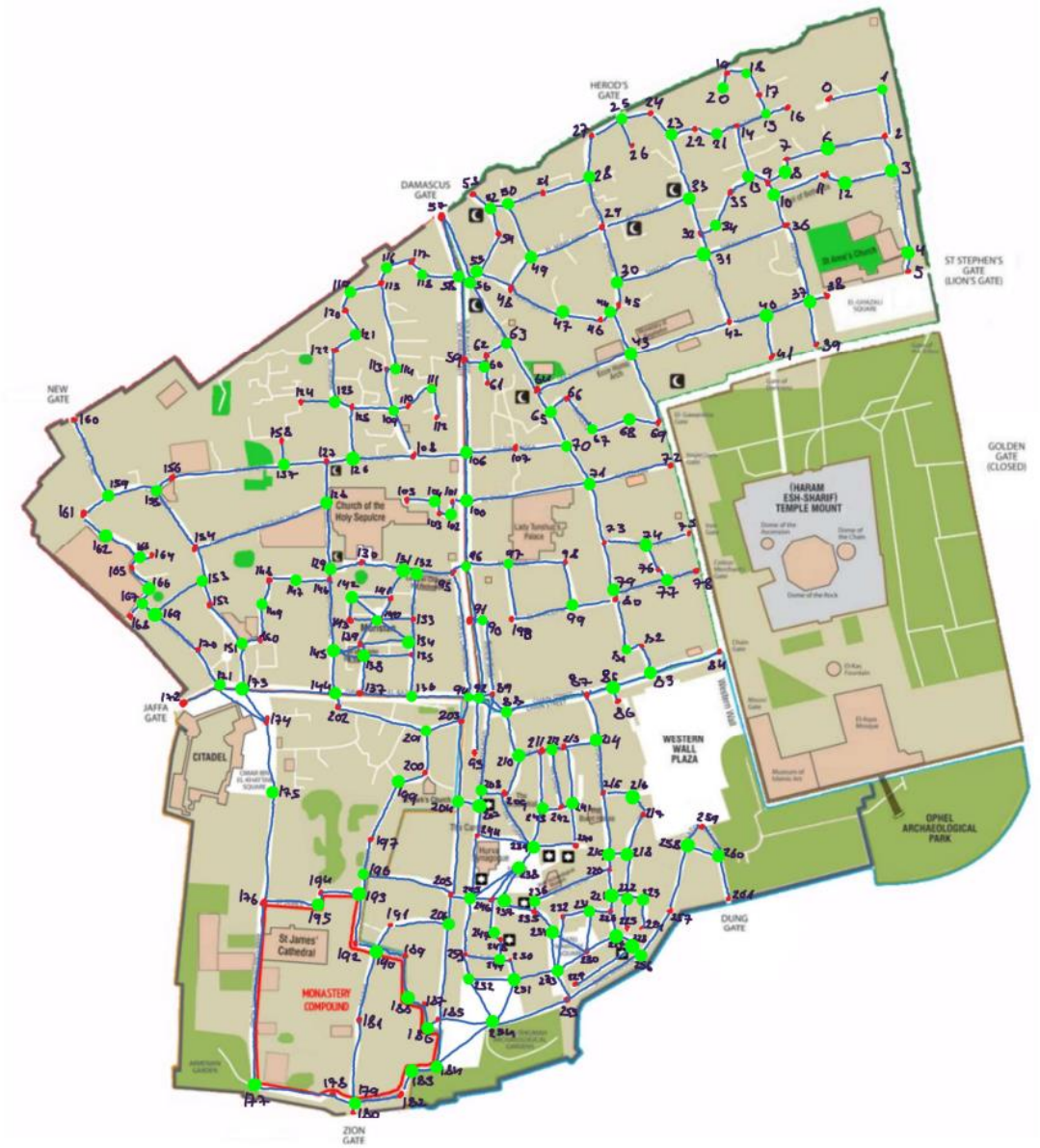
Algorithm Name	Avg VC Size	Min VC Size	Avg Run Time (sec)
2-approximate	476	476	0.012499
Greedy Hill-Climbing	408.8	403	0.136902
Stochastic HC	425	416	0.172918
Random Restart HC	405.6	403	4.958932
Simulated Annealing	408.2	405	16.12664
Local Beam Search	421.9	415	70.15138
GHC Weighted Edges	405.8	404	107.0366
GHC Weighted Vertices	404.9	402	0.576152
Regular VC GA – population=14	411	406	135.724
Regular VC GA2 – population=14	405.6	401	127.3823
VC Punish GA – population=14	418	413	202.0307

Old City graph - |V| = 262, |E| = 338, |OPT VC*| = ?; 10 Iterations

Algorithm Name	Avg VC Size	Min VC Size	Avg Run Time (sec)
2-approximate	218	218	0.001995
Greedy Hill-Climbing	138.4	137	0.009875
Stochastic HC	149.8	147	0.012769
Random Restart HC	138.1	138	0.248908
Simulated Annealing	137.4	136	1.24657
Local Beam Search	147.4	143	8.640009
GHC Weighted Edges	135.8	135	3.056195
GHC Weighted Vertices	134.4	134	0.012172
Regular VC GA – population=10	136.7	135	13.58261
Regular VC GA2 – population=10	135.6	134	11.60129
VC Punish GA – population=10	139	136	15.66034

The rest of the results can be found in the results directory.

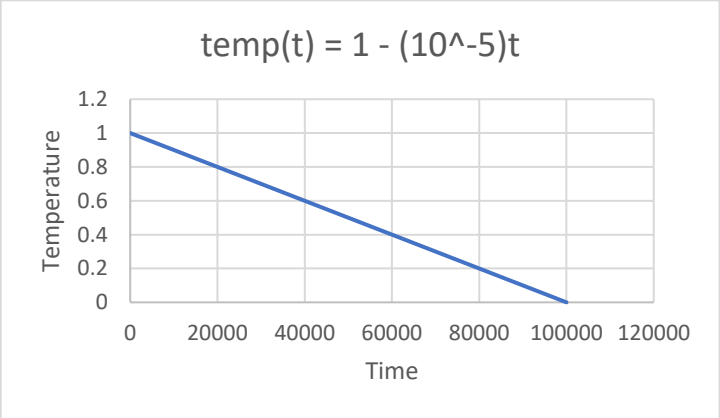
In the image below we see the map of the Old City of Jerusalem after representing the results of the smallest vertex cover found by our algorithms (Regular VC GA2\GHC Weighted Vertices – 134).
The vertices of the cover are marked in green:

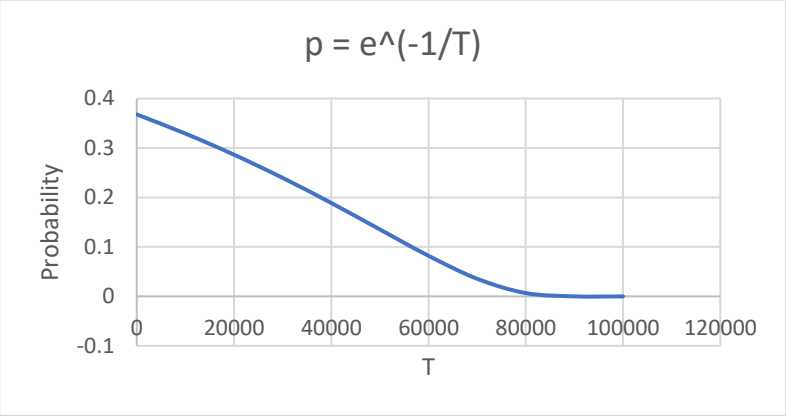


As we can see, every street is covered while using only 134 cameras out of 262 possible locations. This is a significant difference comparing to the 218 positions the 2-approximate algorithm achieved.

Before analyzing our results, we will first discuss why we chose certain parameters for certain algorithms:

1. k – for Local Beam Search, after several tests on different graphs, we observed that $25 \leq k \leq 30$ gave the best results. We decided to use k=25 for better run time.
2. Simulated Annealing temperature - $temp(t) \rightarrow 1 - 10^{-5}t$.
   After trying a few temperature functions, we have concluded that the best temperature function for this problem is a linear reduction function: $temp(t) = k - \alpha t$, where $k > 0$ is some constant and $\alpha > 0$ is the slope.
   As discussed earlier, we would like a slowly decreasing temperature function to explore a lot of states first but later would rather exploit.
   Specifically, we chose the temperature function above which answers this criterion and gave the best results overall.

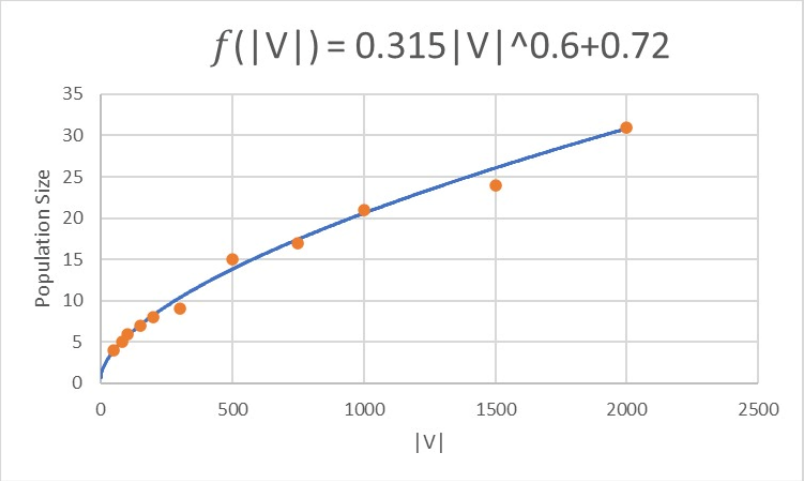




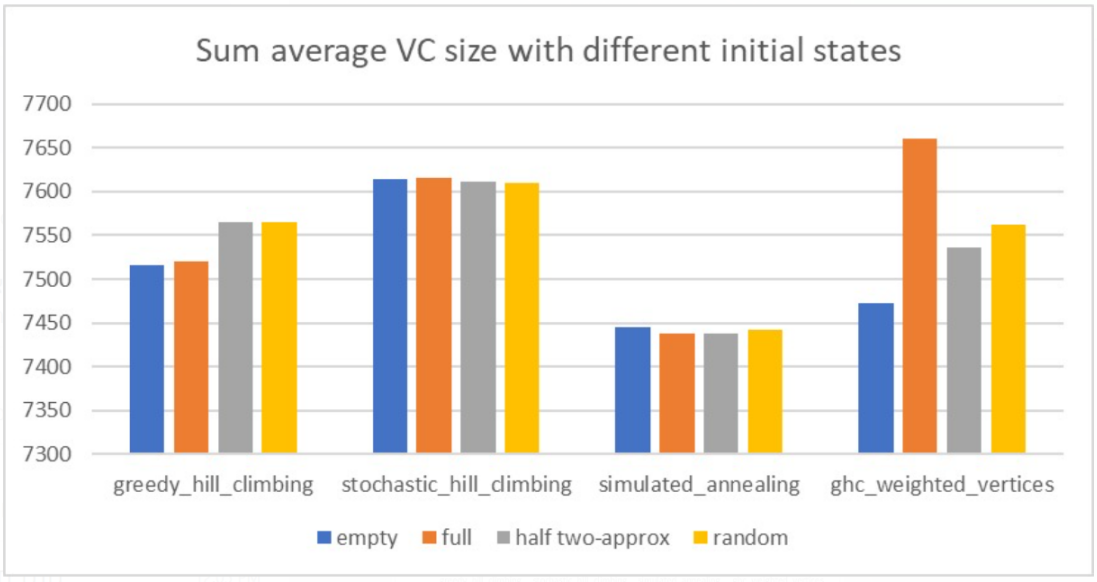
   We see that for $\Delta E = -1$ we have a decreasing probability function which is almost linear until 80,000 iterations, and afterwards we mainly exploit. As the cost difference between neighbouring states isn't large, especially once the state is almost a VC, this graph illustrates well the probabilities of choosing a "bad" neighbour and escaping a local maximum.
3. GHC weighted edges number of iterations – 100.
   Since this algorithm has a weight function which updates and affects the algorithm's decisions, we want to run this for as many iterations as possible to explore more states and achieve better results while maintaining a decent runtime.
4. Random Restart iterations – 100.
   Starting at different states gives different results for GHC. For the same reasons as GHC weighted edges' number of iterations, we chose this number here.
5. GAs – We checked quite a few of GAs with the different fitness, reproduce, and mutation functions discussed previously, though we noticed these 3 give us the best results. Some other GAs gave similar results where the change was not significant.
   a. Regular GA 1 – usually does not give a valid VC, though once running the result through GHC, we achieve a good valid VC size.
   b. Regular GA 2 – Has more of a bias towards a valid VC, as the fitness function pushes towards adding as many edges as possible.

c. VC Punish GA – Has an even higher bias towards a valid VC, as the fitness function pushes towards a VC. Furthermore, the mutation pushes towards adding vertices of edges which are not covered.

6. Generations – 10,000.
   In Genetic Algorithms we noticed that in the first generations, the best solution updates rapidly, while as we advance the best solution updates seldomly. Specifically, between generations 2,000 and 10,000 there are only several updates. We want the algorithm to run for a reasonable number of generations to achieve the best results while maintaining a decent runtime.
7. Population size - $round(0.315 * |V|^{0.6} + 0.72)$
   We noticed that there is a direct relationship between the number of vertices in a graph and the effectiveness of the population size, while the number of edges has no effect. Therefore, we created several graphs of different sizes and checked when the effect of enlarging the population size is negligible. We noticed the results resemble a root function and were able to extract from the results the function above. [4]
   Here are the samples we achieved while testing (orange dots) and the function:



8. Initial State – Empty state.
   In some algorithms – GHC, Stochastic HC, Simulated Annealing, and weighted vertices HC, it is possible to start from different initial states. In the graph presented below, we checked how different initial states – empty, random, full, and a random state of size half the 2-approximate VC size (the min VC is at least that size) affected the results of those algorithms:



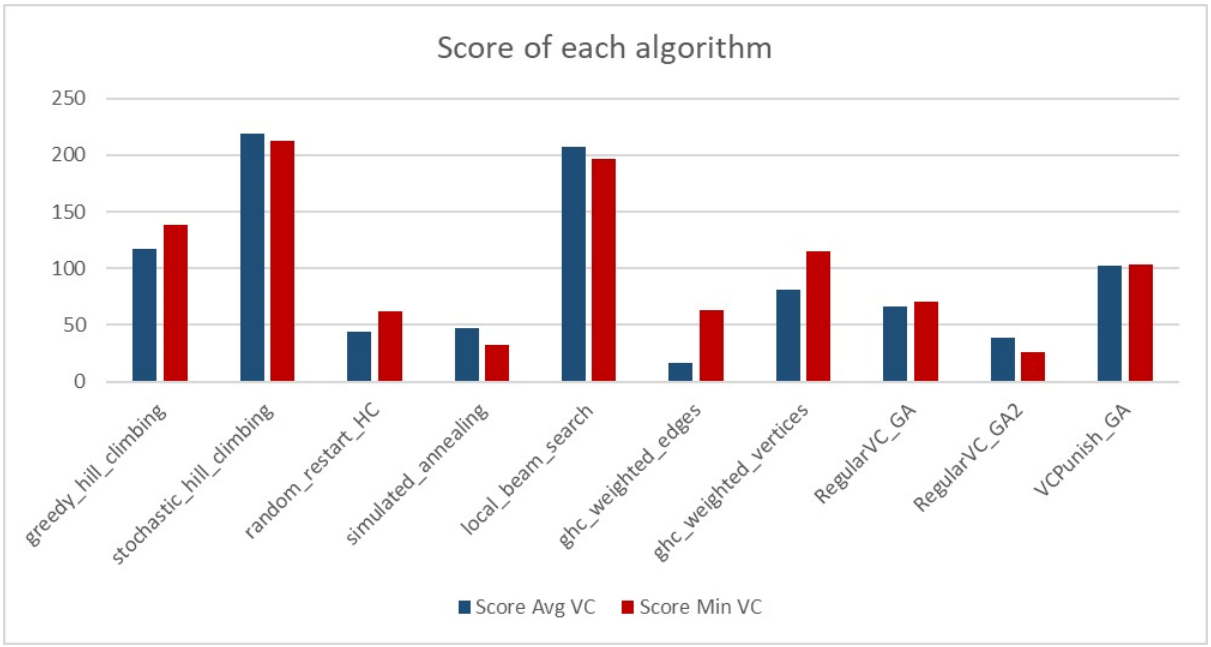
***Note – lower is better.**

As we can see, the effect is negligible in Simulated Annealing and Stochastic HC. As for GHC it is better to start with the empty state, though starting with the full state is also fine. For weighted vertices HC, the difference is quite significant, it is better to start with the empty state.

We provide below a graph which represents the scores each algorithm achieved on average in relation to all other algorithms. **The lower the score, the better the algorithm performed**. The scores were calculated both over the average VC size and the minimum VC size in the following way:

$$\forall alg \in algorithms, \qquad avgVCScore = \sum_{g \in Graphs} \left(avgVC(alg) - \min_{a \in algorithms} \{avgVC(a)\} \right)$$

The min VC score was calculated similarly.
In other words, for each graph, the algorithm with the lowest VC (average or minimum according to the score calculated) achieves a score of 0, and all other algorithms achieve the VC size difference.



The 2-approximate algorithm achieved Avg VC score of 730.2 and Min VC score of 784.

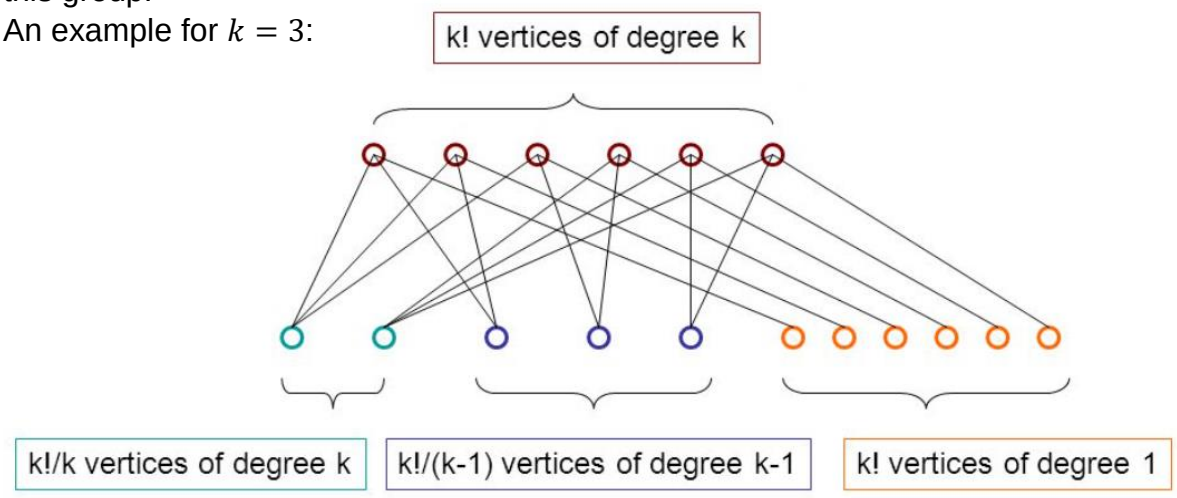
As we can see in the graph, Stochastic Hill Climbing and Local Beam Search performed the worst overall on both Avg VC and Min VC scores. We also see that Regular VC GA2 gave the best results out of the Genetic Algorithms, though it is important to remember that this algorithm usually does not achieve a VC and must be run through Greedy Hill Climbing. Which brings us to the next point - GHC did not perform that well either (although it gave decent results for a simple algorithm). However, it is a very important algorithm and can be useful when starting from a "smart" state (i.e., the result from Regular VC GA2).

Overall, Regular VC GA2, simulated annealing and weighted edges performed the best out of the bunch. We notice that while GHC Weighted Edges has the best Avg Score, the Min Score is quite high. This result can be explained by the results that we got on graph gen400_p0.9_75. GHC weighted edges and the other 2 algorithms on average got similar results, though GHC weighted edges achieved Min VC of 357 while the other 2 achieved a Min VC of 325, which may be a result of a lucky run.

While some of these results are quite impressive, it is important to consider the difference in runtime of certain algorithms. The runtime of all algorithms is polynomial in the graph size as stated above, though in practice that might not be enough. Regular GA2 gave the best results overall, though it also has the highest run time (of the better algorithms), followed by GHC weighted edges, which isn't too dissimilar (both in results and runtime), while Simulated Annealing returns similar results and has quite a good runtime too. On the other hand, Random Restart HC performs slightly worse than all the above though has a great runtime.

Even though we got good results over all the graphs we checked (all algorithms performed better than the 2-approximate), it is important to remember that local search algorithms do not promise achieving them. An example for this is a bipartite graph with the following build:

Choose some $k \in \mathbb{N}$. The upper side has $k!$ vertices, each of degree k.
The lower side is split into groups: $\frac{k!}{k}$ vertices of degree k, $\frac{k!}{k-1}$ vertices of degree $k-1$, $\frac{k!}{k-2}$ vertices of degree $k-2$,…, $k!$ vertices of degree 1.
For each group in the lower side, each vertex v from left to right connects to the first $\deg(v)$ vertices on the upper side (from left to right) which were not yet connected by this group.
An example for $k = 3$:



In this case, the greedy algorithm can choose all the lower vertices from left to right. The MVC in this graph is of size $k!$ (bipartite graph – select the upper side). The 2-approximate algorithm will achieve a $2k!$ VC, while the greedy achieves:

$$k!\left(\frac{1}{k}+\frac{1}{k-1}+\frac{1}{k-2}+\cdots+1\right)\approx k!\log(k)$$

For large enough k ($k \geq 5$) we get an even worse result than the 2-approximate algorithm. This isn't too surprising as the greedy HC is a known approximation algorithm for the Set Cover problem with $\ln(n)$ approximation, while MVC is a subproblem of Set Cover with the covered set being the edges and each subset is the pair of vertices connecting each edge. [5]

Summary

In our project, we attempted to find a decent approximation to the Minimum Vertex Cover problem using Local Search algorithms, such as Hill Climbing, Simulated Annealing and Genetic Algorithms while integrating different heuristics, and we show how much we can improve over the 2-approximation algorithm.

In our results, we concluded that Weighted Edges Hill Climbing, Simulated Annealing and Regular GA2 achieve the best results in general, although some algorithms perform better for different graphs. While our results are quite decent for the graphs we've checked, we were not able to reach the minimum vertex cover many times on the benchmark graphs.

This brings us to possible directions we could continue with our research:

- Understanding on what types of graphs certain algorithms are more likely to find a minimal vertex cover.
- Finding better weight functions over the edges, given that the weighted edges Hill Climbing algorithm seemed to have produced very stable and decent results. We could for example try to decrease the weights as well, instead of only increasing.
- Searching for states that might be better to begin with, especially with Genetic Algorithms.

Code Running Instructions

Install all requirements in requirements.txt.
Run main with the following arguments –

Algorithm:

- two_approximate_vertex_cover
- greedy_hill_climbing
- random_restart_hill_climbing
- ghc_weighted_edges
- ghc_weighted_vertices
- simulated_annealing
- local_beam_search
- stochastic_hill_climbing
- RegularVC_GA
- RegularVC_GA2
- VCPunish_GA
- all

Where "all" runs all the algorithms above.

Graph Type:

- prandom_numVertices_p
- random_numVertices_numEdges
- provide the path of one of the graph files from the directory "graph_files".

Where numVertices and numEdges should be an integer, and p a float between 0 and 1.

Iterations:

Provide an integer greater than 0.

Results Path:

Provide a path to a csv file where the results will be saved.

For example:

```
python3 main.py all prandom_100_0.31 5 results.csv
```

The above will run all algorithms on a p-random graph with 100 vertices and p=0.31 5 times, and save the results to results.csv in the project directory.

The result is a csv file with 4 columns:

1) Algorithm name – the name of the algorithm.
2) VC sizes – a list containing the size of the VC found in each iteration.
3) Average time – the average run time of the algorithm in seconds.
4) Min VC – a list of vertices of the minimum VC found in all iterations.

Sources

[1] https://www.wisdom.weizmann.ac.il/~dinuri/mypapers/vc.pdf

[2] https://www.dharwadker.org/pirzada/applications/

[3] https://www.wikiwand.com/en/Vertex_cover

[4] https://www.dcode.fr/function-equation-finder

[5] https://slideplayer.com/slide/4826675/