# Graph Based Algorithms for Physical Access Control

**Odhran Sexton**

Final Year Project - BSc in Computer Science
Supervisor: Dr. Ken Brown

Department of Computer Science
University College Cork

April 2023

**Abstract**

Physical Access Control is a technique used to control a population's ability to access certain areas. In the context of this project, which relates to graphs, access control is modelled using the nodes and edges of a graph. If there is a path from any given node to another node in the graph, then the first node has access to the second. With this principle in mind, four algorithms were developed in this project, named as follows; *Greedy Last Edge Removal*, *Greedy Any Edge Removal*, *Any Random Edge Removal* and *Restricted Random Edge Removal*. The intention of these algorithms is to reconfigure the graph to break all directed paths between pairs of nodes designated as *Illegal*.

The two greedy algorithms are based on heuristics and involve the systematic removal of edges from the ends of *Illegal* paths in the graph. The difference between the two, is that *Greedy Any Edge Removal* aims to preserve the integrity of *Legal* pairs by only removing edges if they don't also break any *Legal* paths.

The two random algorithms implement a random probing approach towards removing edges from the graph. The difference between these is where *Any Random Edge Removal* can remove any edge from each *Illegal* path, *Restricted Random Edge Removal* only removes edges which are not also part of the minimum edge cut sets for the *Legal* paths.

All these algorithms were tested on a variety of random graphs of varying sizes and densities. This involved calculating the average run time of the algorithms over fifty iterations of the various graphs. After all of these tests had been run, *Greedy Last Edge Removal* proved to have the fastest overall run time, which came at the cost of breaking *Legal* paths which used edges in *Illegal* paths. The best algorithm in overall run time, while also achieving solutions which satisfied all restrictions was *Restricted Random Edge Removal*.

**Declaration of Originality**

In signing this declaration, you are conforming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Signed: Odhran Sexton
Date: 24 April 2023

**Acknowledgements**

In the undertaking of this project, I would like to thank my supervisor first and foremost. Thank you, Ken Brown, for your guidance throughout the project. Your feedback and outside eye were of great help when I found myself stuck in the ruts of code.

Thanks must also go to my classmates, who were all going through similar processes on their own projects. Knowing everyone else was also hard at work on their own projects kept me motivated to put my best foot forward and produce a set of algorithms which I am satisfied with.

My family deserve thanks, for keeping me working on this project with regular inquires into how my project was going. Thanks especially to my parents, Aidan and Orla, who although not directly involved were of great help in this project, particularly acting as a sounding board to bounce ideas off of.

Thanks also to the lecturers and faculty at UCC for providing an environment to learn the skills necessary for this project. This project literally would not have been possible without this knowledge.

Finally, special thanks to my friends, whose conversations and company helped keep the stress of working on this project at bay. It was a significant undertaking for all of us, and I'm proud of all of us for coming through our projects to the end.

# Contents

# 1 Foreword

## 1.1 Introduction

Physical Access Control is a technique used to control a population's ability to access certain areas. This prevents unauthorised access to areas where a general populace may not be allowed to enter, such as building sites to use one example, or are only allowed to enter at specific times, traffic lights being a prime example of this. There are numerous examples of physical access control measure which can be seen in our world including bouncers at events such as concerts, matches and clubs, passport control, ticket gates and much more. There is even a form of physical access control much closer to home in the form of card access to the labs in the Western Gateway Building (WGB) on campus.

In the context of this project, which relates to graphs, access control is modelled using the nodes and edges of a graph, where if there is a path from any given node to another node, then that first node has access to the second. Thus, for the purposes of this project, four algorithms are developed with this principle in mind, with the intention being to reconfigure graphs to break all access between source and target nodes in *Illegal* pairs while maintaining the integrity of paths between source and target nodes in *Legal* pairs.

For the purposes of these algorithms, graphs will be used to represent the topology of buildings or road networks, with nodes being used to represent points of intersections, while corridors, in the case of buildings, or roads, in the case of a road network, are represented as edges. This approach can and will result in large graphs due to the exponential nature of graph sizing. Even a relatively small corridor with five offices on each side will have twelve nodes and a maximum of one hundred and thirty two edges (*n*(n-1)* edges in a directed graph, excluding edges from a node to itself), given that this example is a dense graph.

The algorithms developed in this project were named as follows; *Greedy Last Edge Removal*, *Greedy Any Edge Removal*, *Any Random Edge Removal*, and *Restricted Random Edge Removal*. As the names hint, two of these algorithms are greedy algorithms, while two utilise random probing[4]. In developing these algorithms, smaller graphs are used to allow for ease of

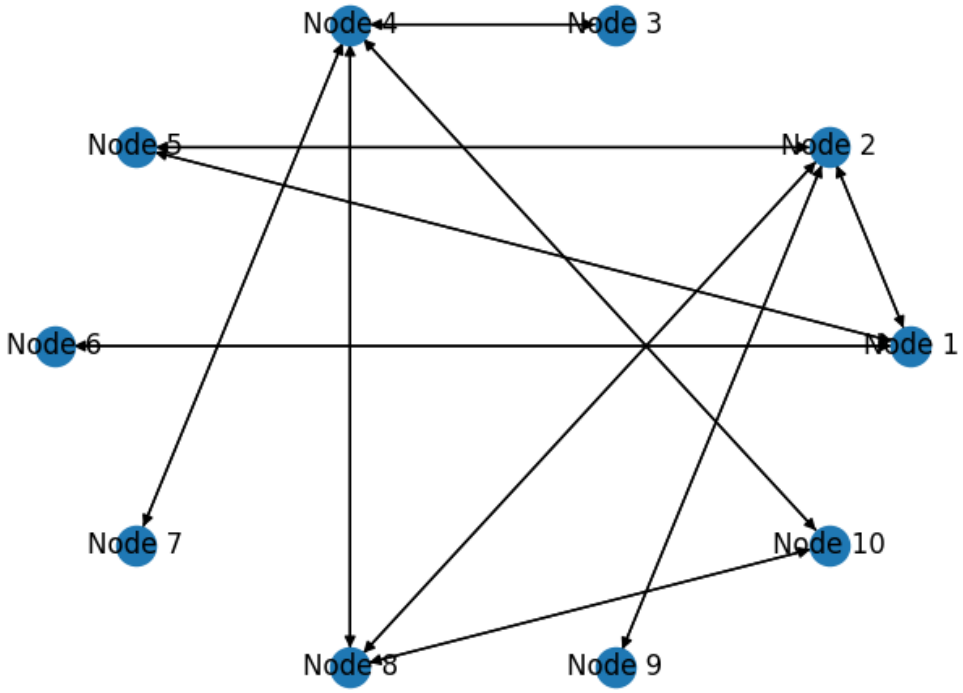


Figure 1: Development Graph

testing and development. Figure 1, above, shows the graph used in developing these algorithms.

Of the four algorithms, *Greedy Last Edge Removal* proved to be the fastest running algorithm, but did not always produce graphs which satisfied the conditions imposed upon them. From the other three, I believe *Restricted Random Edge Removal* to be the best algorithm, as outlined in section 4.3 of this report.

## 1.2 Literature Review

There is a body of literature relating to graphs, but few specifically relevant to this project. The first of these is *Ambient access control for smart spaces: dynamic guidance and zone configuration*[3]. This paper discusses *dynamic route guidance*, giving real time routes to users which avoid hazards which need to be avoided, or assets which need protecting. Similar principles are used in the algorithms developed in this project, although lacking the dynamic real time updates. Also discussed, is *dynamic zone classification*, that is to say assigning security classifications to user defined zones. The ability of users to access these zones is then decided by their security level. Finally, the paper discusses *dynamic configuration of zones*, wherein a facility is modelled as a directed graph, each node representing a location and each edge being either uncontrolled or controllable.

Another paper which is important to this project is *Eigen-Optimization on Large Graphs by Edge Manipulation*[2]. In this paper, *Chen et al.* discuss edge deletion in a proposed algorithm which outputs a set of edges whose deletion from a graph results in the largest decrease in the leading eigenvalue of the graph. The paper proposes that the problem of edge deletion can be transformed to one of node deletion. Subsequently, the node deletion problem is proven to be NP Complete by reduction from the independent node set problem. This proof is a primary reason for the development of the random probing algorithms in this project.

The third relevant work is a PhD thesis; *Nonsystematic Backtracking Search*[4]. In developing the second pair of algorithms in this report, I had the option to utilise random probing, or randomised backtracking. This thesis discusses backtracking in detail, comparing it to depth first search, iterative sampling and depth first search with restarts. Given these search algorithms relate more strongly to trees rather than graphs, I chose to instead implement the random probing technique. The two techniques are similar in effect but implemented differently. Random probing resets the search medium where a failure is encountered, while randomised backtracking unravels the previous $k$ iterations of the search. Resetting the graph was deemed a more effective solution since failure always occurs on the first previous edge removal.

# 2 Design

In this project there were four algorithms developed. Two of these are greedy algorithms, while two are random probing algorithms. The aim of each algorithm was to improve in some capacity from the previous one. In some cases this improvement meant increasing the chance of reaching a successful outcome, while in others it meant increasing the efficiency of the algorithms. In all the algorithms, a successful outcome is defined as achieving a configuration of the graph wherein all conditions imposed on the paths, either *Legal* or *Illegal*, are satisfied. Efficient edge deletion is a proven NP Complete[2] problem, meaning even the best available algorithms would take an impractically large amount of time to solve for large input graphs. As a result of this, the aim of the algorithms developed here is not to guarantee a solution regardless of time cost, but to attempt to find one in reasonable time.

## 2.1 Greedy Last Edge Removal

This algorithm was developed based on the pseudocode shown in Figure 2. The algorithm first tests each path designated as *Legal* in the input file. If any of these paths are unable to be traversed, i.e. there is no path between the given pair of nodes, then the algorithm outputs a failure message. This situation only ever occurs when the configuration of the graph on input is such that there are no possible paths between the source and target nodes in at least one of the paths designated as *Legal*.

Given that the algorithm successfully makes it past this initial check, it enters a while loop, controlled by a Boolean variable based on whether paths between *Illegal* pair exist in the current graph configuration. Then it starts checking the pairs designated as *Illegal* , using NetworkX's implementation of Dijkstra's Algorithm to determine if a path exists between the source and target nodes of the *Illegal* path. Where any such paths exist, they are added to a list of illegalPaths. Otherwise, no path is available between the source and target node of this particular *Illegal* pair, and they are removed from consideration. Finally, for each path in the list illegalPaths, the algorithm removes the last edge from that path. This breaks one possible path between the source and target nodes of *Illegal* paths, so the algorithm loops until there are no more *Illegal* pairs to remove from consideration. At this point, the algorithm then checks each *Legal* pair again to check if the paths

```
def greedyLastEdgeRemoval():
    invalidPaths = True

    while invalidPaths:
        validPaths = []
        invalidPaths = []

        for validPath:
            if validPath is traversable:
                Append validPath to validPaths
            else:
                output failure message
        for invalidPath:
            if invalidPath is traversable:
                Append invalidPath to invalidPaths
            else:
                Remove invalid pair from list

        if ∃path in invalidPaths which is traversable:
            remove the last edge in the path
            print the removed edge
        else:
            output success message
            invalidPaths = False
```

Figure 2: Pseudocode used for developing Greedy Last Edge Removal

between these have remained intact. If any one of the *Legal* pairs no longer have a path between them, then a failure message is outputted. Otherwise a success message is outputted, along with the new graph configuration.

Consider a dense graph of ten nodes. This graph will have ninety directed edges. Assume there is only one *Legal* pair of nodes and one *Illegal* pair of nodes, which are distinct from each other. This algorithm will isolate the target node of the *Illegal* pair by removing all edges into that node, thus breaking all *Illegal* paths. Given that the target of the *Legal* path was specified to not be the target of the *Illegal* path, this is a successful configuration of the graph, since there is still a path from the source to the target of the *Legal* pair.

Now consider the same dense graph, with the same restrictions of two paths. One *Legal* path from a source node to a target node, and one *Illegal* path

from a different source node to the same target node. It is quickly apparent that this will result in a failing outcome using this algorithm given that the algorithm removes all edges into the target node of the *Illegal* path. This highlights the shortcomings of this algorithm and is a huge weakness in the algorithm. In saying that however, not all graphs are dense, and there may be configurations of graphs which allow for both *Legal* and *Illegal* paths to have the same target nodes where the paths may be part of distinct sub-graphs which are only linked by the target node, which would result in a successful outcome for this algorithm.

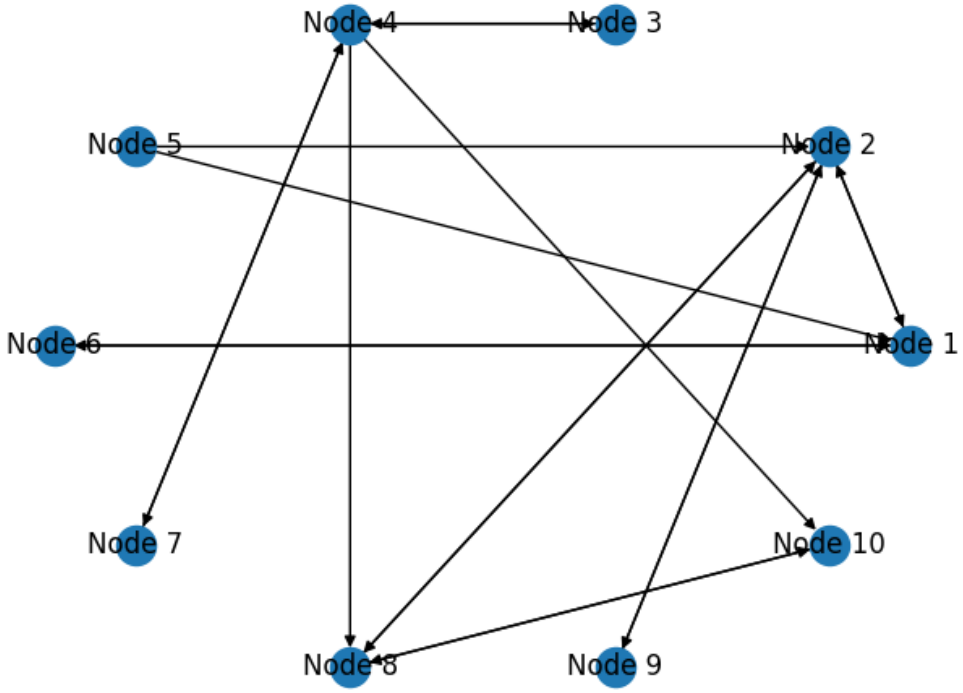


Figure 3: Shows the output graph of **Greedy Last Edge Removal**, with the paths designated as Legal and Illegal in Figure 11.

The time complexity for this algorithm stems from the for loop checking the validity of *Legal* paths, and each iteration of the while loop calling search-Graph() which is a call to the NetworkX implementation of Dijkstra's Algo-

rithm. This implementation of Dijkstra's Algorithm uses a priority queue, which has a time complexity of *O((E+V) log V)*, which may be called up to M times. All of this put together results in an overall time complexity of *O(N*M*(E+V) log V)* for the *Greedy Last Edge Removal* algorithm, where $N$ is the number of *Legal* paths, $M$ is the number of *Illegal* paths, $V$ and $E$ are the number of nodes and edges in the graph.

## 2.2 Greedy Any Edge Removal

The second algorithm was developed based on *Greedy Last Edge Removal*, with the aim being to improve upon its failings. Specifically, I am referring to the fact that removing edges from the end of *Illegal* paths may result in *Legal* paths being broken, with no scope to rectify this. Thus, the aim of this algorithm is to prevent this occurrence by checking each the graph as each edge is removed to see if any *Legal* paths are now broken. If they are, the edge is restored and the next edge in the *Illegal* path is removed, with the check repeated as needed.

```
def greedyAnyEdgeRemoval():
    invalidTraversable = True

    while invalidTraversable:
        validPaths = []
        invalidPaths = []

        for validPath:
            if validPath is traversable:
                Append validPath to validPaths
            else:
                output failure message
        for invalidPath:
            if invalidPath is traversable:
                Append invalidPath to invalidPaths
            else:
                Remove invalid pair from list

        if ∃path in invalidPaths which is traversable:
            remove the last edge in the path
            if ∃path in validPaths now broken:
                remove the next edge and restore the previously removed one
            print the removed edge
        else:
            output success message
            invalidPaths = False
```

Figure 4: Pseudocode used for developing Greedy Any Edge Removal

As with the *Greedy Last Edge Removal*, the first step of this one is to ensure there are paths between all node pairs designated as *Legal* before starting

the work of removing edges from the graph. Once this check succeeds, the algorithm then starts generating the *Illegal* paths, using NetworkX's implementation of Dijkstra's Algorithm. It's at this point that the two algorithms diverge, since this one removes the last edge from an *Illegal* path. Then, the algorithm immediately checks if the removal of this edge has broken any *Legal* paths. If so, the algorithm removes the next edge in the sequence of edges in the *Illegal* path and restores the previously removed edge to the graph. Again, the *Legal* path is checked to see if it is broken or not. If the path remains broken, the algorithm loops removing edges from the *Illegal* path and restoring the edge from the previous iteration until either finding an edge which doesn't break the *Legal* path on removal, or testing all edges in the *Illegal* path results in failure to break the *Illegal* path without breaking the *Legal* path.

Provided that the algorithm finds an edge in the *Illegal* path to remove without breaking the *Legal* path, this loop repeats for each *Legal* path in the list of restrictions. Once every *Illegal* path has been broken while retaining connectivity for *Legal* paths, the algorithm then checks each *Illegal* pair to see if there are other alternate paths which the process needs to be repeated for. If there are, the algorithm loops. Otherwise it terminates, and outputs the new configuration for the graph.

This algorithm fixes the shortcomings of the first by allowing removed edges to be added back into their graph if their removal breaks all possible paths between *Legal* pairs of nodes. This prevents the scenario discussed in section 2.1, where the algorithm might isolate a node in the graph, breaking *Legal* paths. In the case of the dense, ten node graph discussed in the same section, where the *Legal* and *Illegal* paths both targeted the same node from different source nodes, the algorithm prevents the case where the *Legal* path is broken by isolating the target node of the *Illegal* path. Instead, it may remove all edges into the target node, except one. This last edge may come from the source node of the *Legal* path, or another node which is part of the shortest path from the *Legal* source node. All edges into any node in the *Legal* path would also be removed, thus breaking the *Illegal* path while maintaining the integrity of the *Legal* path.

This algorithm does have it's own limitations however. Due to it's nature as a greedy algorithm and the fact that it can end up iterating through every

edge in a legal path, it quickly becomes inefficient on graphs as they grow larger. Although this algorithm does fix the failing scenario from *Greedy Last Edge Removal* by allowing for edges to be restored to the graph, this extra computational cost becomes expensive quickly.

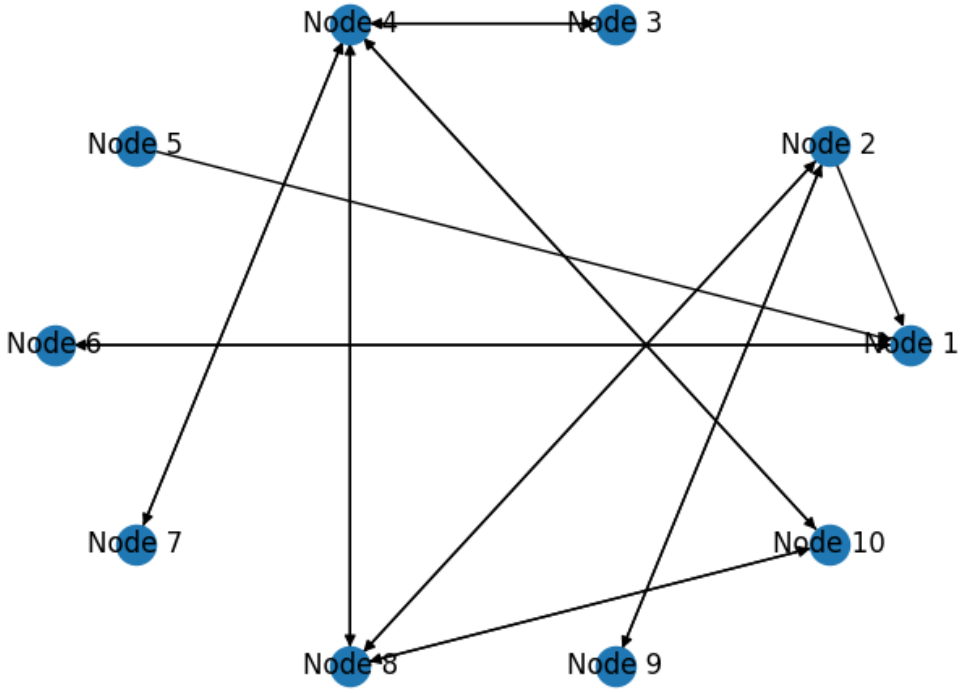


Figure 5: Shows the output graph of **Greedy Any Edge Removal**, with the paths designated as Legal and Illegal in Figure 11.

This algorithm has the same worst case time complexity as the first, due to the fact that the worst case is assumed to mean every edge in the graph must be removed. However, on average, this algorithm will always take longer to run to completion than the previous one due to the increased number of edge removals attempted, combined with the possible restoration of edges back into the graph. This may result in the same edges being tested multiple times if they are used in multiple *Illegal* paths. This algorithm has additional nested for loops in comparison to the first, resulting in increased time complexity, which is the cost of attempting to ensure that connections

between *Legal* pairs of nodes remain intact, as opposed to simply failing and not restoring edges as *Greedy Last Edge Removal* does.

This algorithm was the most frustrating of the four to develop and had to go through several iterations before eventually finding one which worked. Due was due to the tendency of the algorithm to get stuck in infinite loops or removing and restoring the same edge on repeat. I believe the final implementation has solved these issues but it is entirely possible some edge case has slipped through the cracks.

## 2.3 Any Random Edge Removal

The third algorithm was the first of two using a Random Probing[4] approach towards this problem. This change in approach was made because the first two algorithms are based on heuristics. It can be the case that there is a complex set of edges to remove from the graph in order to satisfy both the *Legal* and *Illegal* constraints. Greedy algorithms don't allow for this. There is the option to run a full exhaustive search for all sets of edges to remove from the graph, but given that this is an NP Complete problem[2], this will result in large run times. Random Probing allows for control over the amount of run time an algorithm will take, allowing users to set a time limit for the algorithm to find a solution. This leads to the Random Probing method, of randomly selecting an edge from the *Illegal* paths to remove. This algorithm was developed based on the pseudocode shown in Figure 6, below. This algorithm differs from the previous two given that it takes a user specified time limit, in seconds, as input.

```
def anyRandomEdgeRemoval():
    Legal = []
    Illegal = []

    for each pair in Illegal:
        if ∃path between pair:
            randomly removeEdge
    if ∃pair in Legal with no path:
        (1) Restore initial state and repeat from start <- Random Probing
    if ∃pair in Illegal with path and not exceeds time limit:
        loop from start
    else if time limit exceeded:
        report timeout
    else:
        report success
```

Figure 6: Pseudocode used to develop Any Random Edge Removal

The lists *Legal*, and *Illegal* contain the pairs of nodes which are designated as *Legal* and *Illegal* paths. Initially the algorithm, like the first two before it, checks the *Legal* pairs to ensure there are connections available between the nodes in *Legal* paths. Otherwise the algorithm terminates since a path has been designated as *Legal* when none exists. Once this check has succeeded,

the algorithm moves into a while loop, which iterates until a successful configuration of the graph is found, or the time limit is reached.

Inside the while loop, the first task for the algorithm is to determine if there are available paths for the *Illegal* pairs. If there are any, an edge is randomly chosen from the sequence in the path, and removed from the graph. This is done for every *Illegal* pair in turn. After removing an edge from every *Illegal* pair's path, the algorithm then iterates through all of the *Legal* pairs. The graph is searched to check if any *Legal* paths have been broken, and if so, the graph is reset to it's initial state and the algorithm starts searching from the start. This is known as Random Probing.

Once the algorithm has reached a configuration of the graph where the initially found *Illegal* paths are broken while the integrity of the *Legal* paths is maintained, it then checks the *Illegal* pairs again to see if there are other possible paths to connect the source and target nodes. If there are, a counter is increased. If this counter is greater than zero, and the time limit has not been exceeded, the algorithm loops through the while loop. Otherwise, if the counter is zero, and all *Legal* paths are intact, the algorithm reports a successful outcome and terminates.

In terms of the complexity of this algorithm, it is dependent on the number of *Legal* paths, the number of *Illegal* paths, the amount of times the while loop iterates and the size of the graph in terms of nodes and edges. Given the initial search for *Legal* paths, this has a time complexity of $O(L*(V+E) \log V)$, where $L$ is the number of *Legal* paths. This stems from each call to Dijkstra's Algorithm for every *Legal* path. In a similar vein, Dijkstra's Algorithm is called $I$ times in each iteration of the while loop, where $I$ is the number of *Illegal* paths, resulting in a time complexity of $O(MI*(V+E) \log V)$ for the while loop, where $M$ is the number of iterations. Put together, this results in a time complexity of $O((L+MI)(V+E) \log V)$ for the *Any Random Edge Removal* algorithm, where $L$ and $I$ are the numbers of *Legal* and *Illegal* paths respectively, $M$ is the number of iterations through the while loop, and $E$ and $V$ are the count of edges and nodes in the graph. This is similar to the two greedy algorithms, given the worst case is all pairs of nodes in the graph are designated as Illegal.

Despite there not being a major difference in the worst case time complexity

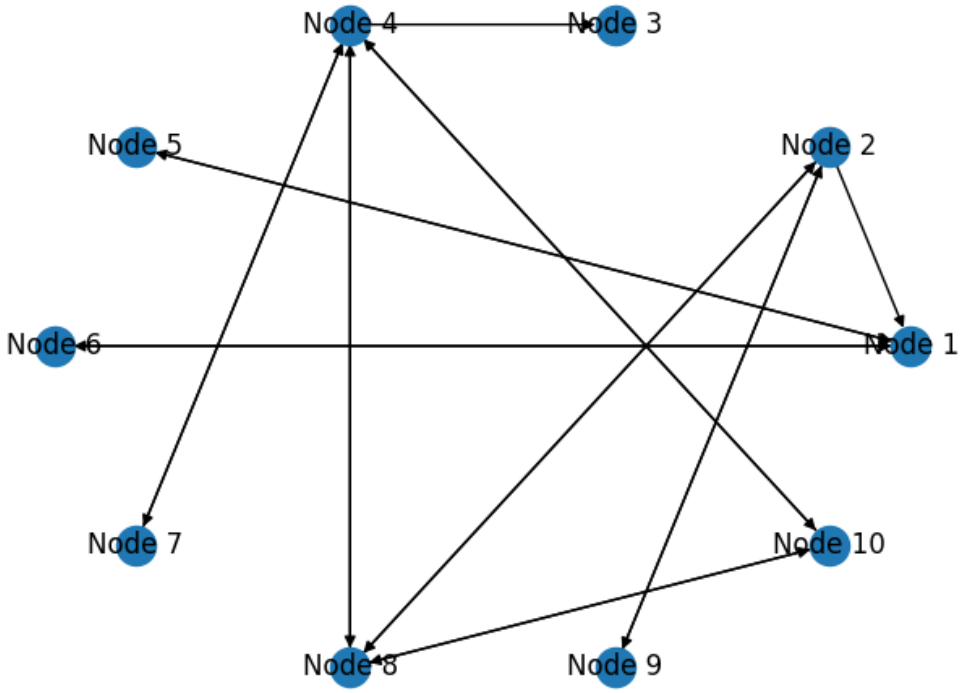


Figure 7: Shows the output graph of **Any Random Edge Removal**, with the paths designated as Legal and Illegal in Figure 11.

for this algorithm compared to the previous two, the average case tends to be much faster than *Greedy Any Edge Removal*, particularly as graph size increases. This time difference comes as a result of random choosing of edges compared against the systematic testing of edges. It may be the case the the first edge out of the *Illegal* source node is the only one which can be removed without breaking a *Legal* path. *Any Random Edge Removal* has the possibility of choosing this edge on the first iteration of removing edges, while *Greedy Any Edge Removal* has to iterate through every other edge in the path first. This highlights the benefits of using the Random Probing approach, especially in large graphs where there may be large numbers of edges in paths between *Illegal* pairs.

## 2.4 Restricted Random Edge Removal

The fourth and final algorithm developed over the course of this project is also a Random Probing algorithm, with the aim being to improve on the previous algorithm. Due to the random nature of the algorithm in choosing edges to remove, it can happen that an edge is removed from an *Illegal* path, which is an essential edge in a *Legal* path. Therefore, the aim of this algorithm is to reduce the likelihood of this occurring using minimum cardinality edge cut sets.

```
def restrictedRandomEdgeRemoval():
    Legal = []
    Illegal = []
    doNotRemove = []

    for each pair in Legal:
        compute the minimum edge cut set
        append these to doNotRemove
    for each pair in Illegal:
        if ∃path between pair:
            randomly select an edge in path
            if edge in doNotRemove:
                ignore
            else:
                remove the edge
    if ∃pair in Legal with no path:
        (1) Restore initial state and repeat from start <- Random Probing
    if ∃pair in Illegal with path and not exceeds time limit:
        loop from start
    else if time limit exceeded:
        report timeout
    else:
        report success
```

Figure 8: Pseudocode used for developing Restricted Random Edge Removal

As with the previous algorithms, initially the algorithm does a check to ensure all the *Legal* pairs have paths between them. Once this check is successful, the algorithm iterates over the *Legal* pairs and computes the minimum cardinality edge cut sets. These are the smallest sets of edges that cannot be

removed from the graph without breaking the *Legal* paths. Every edge in these sets is designated as *do not remove*, an identification which is used when deciding which edges to remove from the graph.

The algorithm then proceeds into the while loop, as with the previous one, and tests the *Illegal* pairs to see if there are paths between them. Where a path exists between source and target nodes of an *Illegal* path, an edge is chosen at random from that path. The algorithm then checks if the chosen edge is designated *do not remove*. If so, the removal operation is ignored. Otherwise the edge is removed. The algorithm then tests the *Legal* pairs, to ensure they are still connected. If the connection between *Legal* source and target nodes is broken, the graph is reset to it's original state and the algorithm begins anew. If the connections remain intact, the algorithm tests the *Illegal* pairs again to check if there are any alternate paths available, counting up for each pair that still has a path.

If the counter for *Illegal* paths remains at zero, and all *Legal* pairs are intact, the algorithm terminates with a success message and outputs the new graph configuration. Otherwise, the algorithm loops until such a time, or until the time limit has been reached.

This algorithm improves on the previous one by never allowing edges which are essential to the connectivity of *Legal* pairs to be removed. This means there would have to be a minimum of two edges in sequence from the *Legal* path removed to break the path. Due to the random nature of the removal of edges from *Illegal* paths, this is unlikely to happen frequently, resulting in fewer resets to the original state of the graph. Given that fewer resets occur, this should reduce the average time taken to achieve a desirable configuration of the graph.

This algorithm has a similar worst case complexity to the previous one given how similar they are in implementation. As before, the complexity is determined by the number of *Legal* and *Illegal* paths and the size of the graph. Given the initial check that all *Legal* pairs are valid, each pair resulting in a call to Dijkstra's Algorithm, this gives an initial complexity of *O(L*(V+E) log V)*. Then, for each of these pairs, the minimum edge cut sets are generated, also with complexity *O(L*(V+E) log V)*. The while loop executes up to M times, where M is the number of *Illegal* paths, calling Dijkstra's Algo-

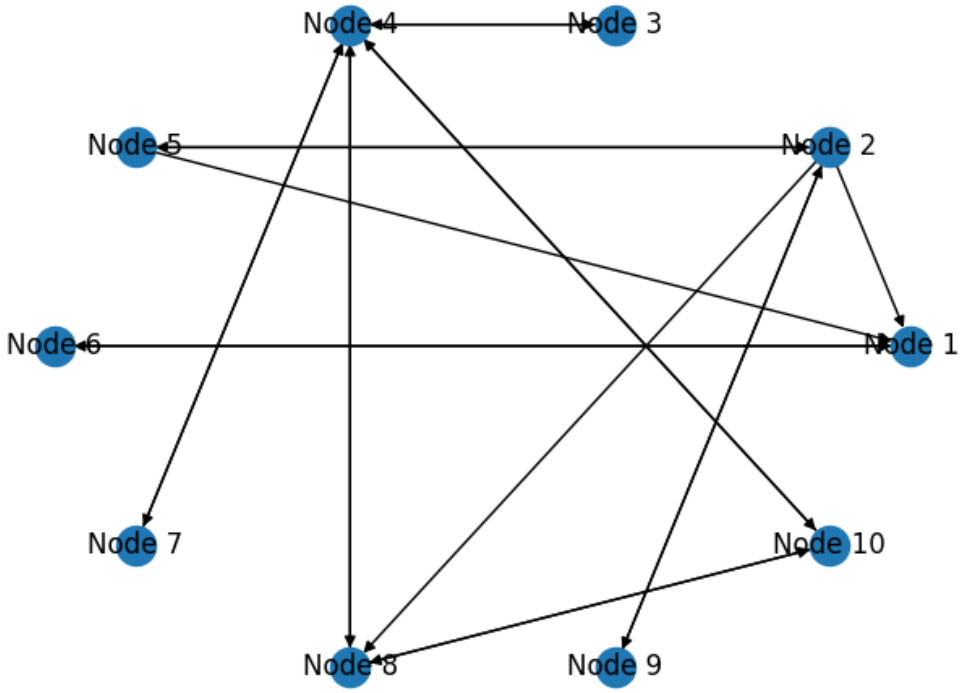


Figure 9: Shows the output of **Restricted Random Edge Removal**, with the paths designated as Legal and Illegal in Figure 11.

rithm each time. This has a complexity of *O(M*(V+E) log V)*. In the worst case, every path is Illegal, and there are I *Illegal* paths. Therefore, the worst case time complexity of the *Restricted Random Edge Removal* algorithm is *O((L+I)*(V+E) log V)*, where L and I are the counts of *Legal* and *Illegal* paths respectively, and V and E are the number of nodes and edges in the graph.

# 3 Implementation

The project is implemented using Python. Python was chosen due to familiarity and comfort in using the language, as well as the large volume of useful libraries which were available. There are numerous libraries available, with NetworkX being the most prominent, but others were available such as graph-tool and rustworkx to name but two. All of these libraries provide tools for creating and manipulating graphs, which are primary requirements for this project, however, NetworkX was chosen due to the fact it supports all common platforms and therefore can be used on said platforms.

NetworkX is a self professed *Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks*[1]. NetworkX also comes preloaded with implementations of many standard graphing algorithms, including Dijkstra's Algorithm, the Bellman-Ford Algorithm and Johnson's Algorithm. Dijkstra's Algorithm was used extensively in determining paths in the algorithms developed in this project. This being implemented in NetworkX already was a point in favour of using the library as it allowed for immediate starting into the development of the algorithms in this project.

Another major technology used in this project is Graph Modelling Language (GML), which was used to represent the graphs being fed into the algorithms. GML format largely resembles JSON, with the key difference for this project being that NetworkX can directly read graphs from GML files as opposed to requiring intermediary steps when reading from JSON files.

```
node [
    id 1
    label
    "Node 1"
]
```

Figure 10: Sample of a node in GML format

Each of the four algorithms developed in this project read graphs in from a file in GML format, as well as a set of conditions from a text file. These

conditions follow a format of identifying whether the path is deemed as *Legal* or *Illegal*, followed by the ordered end points of the path identified by the node names. This format is demonstrated in Figure 11, below.

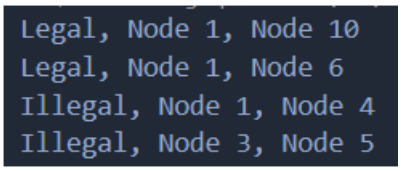


Figure 11: Sample format for Legal and Illegal Paths

Based on the figure shown here, and the development graph shown in the Introduction section, paths from Node 1 to Node 10, and Node 1 to Node 6 are *Legal*, and the algorithms wish to maintain the connections between these nodes, while simultaneously breaking any paths from Node 1 to Node 4, and Node 3 to Node 5, given that these paths are designated as *Illegal*.

Each algorithm seen in this report was implemented using calls to the NetworkX library for searching graphs for paths utilising Dijkstra's Algorithm and adding and removing edges as needed. The logic behind deciding which edges to remove was developed during this project and differs for each algorithm, as discussed in each algorithm's design section.

# 4 Experiments

## 4.1 Expectations

Testing the algorithms consists of comparing their run times against each other, on varying sizes of graphs. Given this method of comparison, it is expected that there will be differences in the capabilities of each algorithm, particularly as the sizes of the graphs being tested increases. Each algorithm will be tested on randomly generated graphs of varying sizes, with varying levels of density inside each size bracket. I have my own expectations about where each algorithm developed ranks against the rest in terms of run time, as highlighted in each dedicated subsection to follow.

### 4.1.1 Greedy Last Edge Removal

This is the algorithm I would expect to have the fastest average run time out of the four developed. The algorithm simply removes the last edges from any available paths between source and target nodes of *Illegal* pairs, without regard to any consequences for the connectivity of *Legal* pairs. As a result of this, even in large, dense graphs, *Greedy Last Edge Removal* should be one of the fastest, if not the fastest algorithm in reaching completion. This speed is attained at the cost of risking *Legal* paths getting broken in the pursuit of breaking all *Illegal* paths.

### 4.1.2 Greedy Any Edge Removal

*Greedy Any Edge Removal* is the algorithm I expect to have the largest average run time. This algorithm takes the approach of ensuring *Legal* paths remain intact when removing edges from *Illegal* paths by only removing an edge if it doesn't also break a *Legal* path. This results in extra computational cost which will be reflected in the average run times. Therefore, it can be quickly seen that the average run time for the algorithm will be much higher for dense graphs, where there are numerous possible paths from source to target nodes in *Illegal* paths. These types of graphs will therefore have more edges to remove, resulting in more checks to see if any *Legal* paths have been broken, increasing the average run time.

### 4.1.3 Any Random Edge Removal

I would place this algorithm somewhere in the middle ground out of the four. It is unlikely that it will beat *Greedy Last Edge Removal* in average run time given that the graph may get reset to its original state if a *Legal* path gets broken. However, *Any Random Edge Removal* should prove a quicker algorithm than *Greedy Any Edge Removal* since it is not a heuristic algorithm, allowing for more complex sets of edges to be removed from the graph. This should hold especially true on larger, more dense graphs, which the performance of *Greedy Any Edge Removal* is expected to fall down upon.

### 4.1.4 Restricted Random Edge Removal

*Restricted Random Edge Removal* is in a similar boat to *Any Random Edge Removal*. I would expect it to be quicker on average due to the inclusion of the minimum cut sets of edges being designated *do not remove*. This results in a slight increase in computational cost, but given that these edges can never be removed from the graph, the algorithm should reset the graph to its original state fewer times on average since it is now harder to completely break *Legal* paths. Given this information, it would be expected that *Restricted Random Edge Removal* will be the fastest algorithm to find a solution which satisfies all *Legal* and *Illegal* path restrictions, while *Greedy Last Edge Removal* will be faster at running to completion.

## 4.2 Results

Each algorithm was tested on fifty random configurations of graphs, of various sizes and densities,. The paths designated as *Legal* and *Illegal* were also randomly generated, with the total number of paths accounting for forty percent of the number of nodes, i.e. a graph with ten nodes has four path restrictions, split into two *Legal* and two *Illegal* paths. The run times were then averaged, the results of which are tabulated in this section, all measured in seconds. The names of each algorithm are shortened in the table for the sake of space. In each case, these tests were run with sixty second time limits on the two random probing algorithms. Note that in the *Graph Size* column in the table, an entry of *xN, yE* means the graph contains $x$ Nodes and $y$ Edges.

Infinity entries were used in places where no times were measured. This occurred in *Greedy Any Edge Removal*, where the algorithm quickly becomes inefficient as graphs increase in size. In the instances where infinity entries are placed, *Greedy Any Edge Removal* started to take significant amounts of time, with the tests stopped after the timer had gone greater than five minutes for a single iteration of testing.

### 4.2.1 Results on Smaller Scale Graphs

The first tests run were on graphs of ten nodes and thirty edges, with the graphs steadily growing larger. The aim of this was to model the different run times on graphs which were less dense compared to graphs which were more dense. Each group of graphs tested here has different levels of density, showcasing the performances of the algorithms across a variety of graphs.

| Graph Size | Greedy Last | Greedy Any | Any Random | Restricted Random |
|---|---|---|---|---|
| 10N, 30E | 0.32 | 9.81 | 5.65 | 5.24 |
| 10N, 60E | 0.37 | 17.96 | 9.78 | 10.91 |
| 20N, 100E | 0.56 | $\infty$ | 13.69 | 13.31 |
| 20N, 250E | 1.1 | $\infty$ | 20.58 | 38.01 |
| 50N, 400E | 1.87 | $\infty$ | 48.43 | 33.64 |

Table 1: Average run time in seconds over fifty iterations

These results match with the expectations held prior to testing the algorithms. *Greedy Last Edge Removal* was the fastest across all the graphs, without regard for size or density. This came at the cost of failing outcomes for most if not every test. *Any Random Edge Removal* and *Restricted Random Edge Removal* also had behaviours much as expected, with very similar run times. It was somewhat surprising that *Any Random Edge Removal* had the faster run times on more dense graphs, but this could be explained by not having enough data points. Both random probing algorithms found graph configurations which satisfied all restrictions. The difference in run time of the *Restricted Random Edge Removal* compared to *Any Random Edge Removal* on the graphs containing fifty nodes and four hundred edges is interesting and suggests that *Restricted Random Edge Removal* is the best algorithm to use on larger graphs.

### 4.2.2 Results on Larger Scale Graphs

The table below shows the average run times of the random probing algorithms on much larger graphs. There is one caveat with the times displayed, in that the experiments were stopped as soon as a successful completion was reached due to the large amount of time required to run all fifty tests with five minute time limits on the algorithms in these experiments. This means the results are not accurate, but they do display the ever increasing time required to find solutions given the algorithms were given five minute time limits for these tests.

| Graph Size | Any Random | Restricted Random |
|---|---|---|
| 100N, 4500E | 263.14 | 291.29 |

Table 2: Average run time in seconds to the point of stoppage

In the case of *Restricted Random Edge Removal*, I did notice the successful graph configuration resulted in the removal of 1,699 edges from the graph, which was approximately a third of the graph. Whether every one of these edges needed to be removed, or was just the result of the random nature of the algorithm is unknown, but it highlights the fact the algorithm is well capable of executing a large volume of work in relatively short time. Another significant finding I noticed was that *Any Random Edge Removal* resulted in a larger number of graph resets than *Restricted Random Edge Removal*, which was likely due to the minimum edge cut sets being preserved in the latter algorithm.

## 4.3 Conclusions

On a whole, buildings and road networks, which these algorithms were developed in mind of, will tend towards being less dense than towards being fully connected. Based on this, I believe the best algorithm is *Restricted Random Edge Removal*. This finding comes based on the average run times from the smaller scale graphs since the less dense of these closer model the networks of buildings and roads than the more dense graphs.

Another factor in forming my opinion however is the inclusion of minimum edge cut sets being protected. This leads to the belief the algorithm will result in fewer resets than *Any Random Edge Removal*, which was reflected in the outcomes of the tests on the graphs containing one hundred nodes and four and a half thousand edges. Fewer resets should result in faster run times on average to find configurations which satisfy all restrictions.

The smaller graphs also better reflect the amount of restrictions a network may have placed on it in general use, but in the interest of fair testing, the number of restrictions was based on the number of nodes in the graphs, as mentioned in the *Results* section. It is my firm belief that if the large graphs in section 4.2.2 had a comparable number of *Legal* and *Illegal* paths to those in section 4.2.1, the average times on those large graphs would have been much shorter, and not reached time out so frequently.

Where it can be guaranteed that *Legal* paths do not require access through an end node of an *Illegal* path, then *Greedy Last Edge Removal* would be the best to use. However, this is difficult to guarantee unless specifically manufactured. Thus, given all these factors, ***Restricted Random Edge Removal*** is the best of the four algorithms in my opinion.

# 5 Practical Applications

Graphs can be used to model a variety of different situations. As an example, a road network can be modelled as a graph, with the roads themselves being edges in the graphs, and any junctions or intersections being nodes. This representation of a road network allows for the algorithms developed here to be used. If a road needs to be closed for any reason, if a tree has fallen on one side for example, these algorithms could be used to determine if the road can be completely closed to traffic until the road is cleared, or if a manual traffic light, one way system needs to be put in place.

Another example of where these algorithms can be used is buildings. A building floor plan can be modelled as a graph, with corridors being represented as edges and doorways being nodes in the graph. In this case, if a hazard breaks out, these algorithms could be used to isolate parts of a building where the hazard has reached, while simultaneously allowing access to emergency exits.

Pipe networks can also be represented as graphs, with the algorithms developed in this project able to be used to determine if certain pipes are vital to the integrity of a network or not. This can result in maintenance being completed quicker or slower if the pipe can be shut down temporarily or not, similar to the case with road networks.

A somewhat less obvious application for these algorithms would be computer chip manufacture. These algorithms could be used with theoretical new connection locations on a chip to determine if the chosen location is suitable or not, given the manufacturer does not wish for current to interact with certain parts of the chip while allowing it to flow to others.

These algorithms can be used in any scenario where something can reasonably be represented as a graph, with the examples named here being but a few such instances.

# 6 Future Work

To any who decide to take up the matter of *Graph Based Algorithms for Physical Access Control* in future, I have a few recommendations. The first of these is to utilise an interactive GUI display of the graphs to allow users to dynamically choose nodes and decide if these are *Legal* or *Illegal* paths rather than having these read from a file. This would allow for greater control of the graphs, as well as allowing for the paths to be updated in real time if circumstances change drastically.

In a similar vein, the algorithms could be improved to be dynamic. In the case of a building where a hazard breaks out, the hazard may travel between areas in said building. The algorithms developed here only prevent access to the starting point of the hazard if specified as an *Illegal* path, which is a limitation which could be improved upon.

In terms of expanding upon the algorithms developed in this project, I believe an algorithm which builds upon the strengths of *Restricted Random Edge Removal* would be a better algorithm. Given the random nature of the edge removals, it may be the case that a particular edge being removed results in the graph being reset. However, subsequent iterations of the loop may also result in attempts to remove that same edge, causing further resets. I believe if this occurs, the edge should be designated as a *do no remove* edge, similar to the minimum edge cut sets. This should result in fewer resets because the same edges won't be causing multiple resets of the graph, making for a more efficint algorithm.

# 7 Conclusions

Developing the algorithms in this project was a more complicated process than I had expected at the commencement of the project. There were many places to get caught, or where code would fall into infinite loops which needed preventing. While there is always scope for improvement, I believe the algorithms as they are now, satisfy the goals laid out from the beginning of the project. *Greedy Any Edge Removal* proved the most complex algorithm to implement. At one iteration of development, the algorithm was getting caught in an infinite loop of removing and restoring the same edge to the graph. This problem was solved by removing the next edge before restoring the previous one as opposed to the reverse which had been the first approach towards implementing the algorithm.

Of the four algorithms I would use one of the two random probing options. *Any Random Edge Removal* or *Restricted Random Edge Removal* but configure graphs in similar times and are more efficient than *Greedy Any Edge Removal*. These two algorithms are also far more likely to find a solution, which was defined in Section 2 as a configuration of the graph for which all *Legal* and *Illegal* pairs were satisfied. Although *Greedy Last Edge Removal* had the fastest average run times by far, it very rarely found solutions for the graphs, making it a poor choice of algorithm to use unless it can be guaranteed that the last edges in any *Illegal* paths are not used in any *Legal* paths. All these factors taken into account, I believe the best algorithm to use is *Restricted Random Edge Removal* given that this algorithm should reset the graph to its original state fewer times on average than its *Any Random Edge Removal* counterpart due to the inclusion of the minimum edge cut sets.

# References


[1] Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart. "Exploring network structure, dynamics, and function using NetworkX". In: *Proceedings of the 7th Python in Science Conference (SciPy2008)* (2008), pp. 11–15.

[2] Chen Chen, Hanghang Tong, B. Aditya Prakash, Tina Eliassi-Rad, Michalis Faloutsos, and Christos Faloutsos. "Eigen-Optimization on Large Graphs by Edge Manipulation". In: *ACM Transactions on Knowledge Discovery from Data (TKDD)* 10.4 (2016), pp. 1–30.

[3] Seán Óg Murphy, Liam O'Toole, Luis Quesada, Kenneth N. Brown, and Cormac J. Sreenan. "Ambient access control for smart spaces: dynamic guidance and zone configuration". In: *Procedia Computer Science* 184 (2021). The 12th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 4th International Conference on Emerging Data and Industry 4.0 (EDI40) / Affiliated Workshops, pp. 330–337. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2021.03.042. URL: https://www.sciencedirect.com/science/article/pii/S1877050921006736.

[4] W. Harvey. "Nonsystematic Backtracking Search". PhD thesis. Stanford University, 1995.

# List of Figures



# List of Tables