Comparing the speed and accuracy of procedurally generated Genetic Algorithms against conventional pathfinding algorithms in 2D maze-like grid environments

-

TOTAL WORD COUNT: 4000

Table of Contents

Introduction

Pathfinding, finding the shortest most optimal path from point A, to point B, is a problem long studied and researched by scientists in mathematics and engineering alike [Pan & Ching PUN-Cheng, 2010, Laparra, 2019]. Examples of such problems include Internet or telephone routing, GPS tracking, and Realistic AI game development [Laparra, 2019]. Specifically in the field of Computer Science, however, it has become a popular yet increasingly frustrating problem [Cui & Shi, 2011]. In recent years, there has been substantial development in the accuracy and efficiency of pathfinding algorithms. For example, considerable effort has been put in over the past decades to optimise and improve the A* pathfinding algorithm, held as a provable most optimal solution to the problem [Cui & Shi, 2011]. Furthermore, there are also different variations to the pathfinding problem, such as dynamic changes in the environment as the path is explored, or if the entire map is known or not [Algfoor et al, 2014]. In recent years, From the substantial size and growth of the technology sphere, more specifically in video games, it becomes easy to understand the importance of improving pathfinding algorithms. This paper proposes making use of metaheuristic procedures, more specifically in genetic algorithms, to optimise heuristic pathfinding algorithms. This investigation aims to further improve single-agent pathfinding through a randomised maze environment as a proof of concept to metaheuristics and its potential as an improvement to pathfinding as well as more broadly heuristic algorithms overall.

Pathfinding algorithms

Graph-search, more commonly known as pathfinding algorithms, traverse and carve paths through a graph between two endpoints. However, there is no explicit expectation that these paths would be the most optimal [Needham & Holder, 2019, Ch. 4]. These algorithms are made up of two different steps, graph generation, and pathfinding [Algfoor et al, 2014]. Graph generation refers to the presentation of the data available. For the sake of simplicity of this investigation, all representations of graphs and pathfinding in this essay will be in the form of a grid-like maze. Furthermore, how much of the data available is allocated to the pathfinding algorithm should also be considered. For example, in the case of evaluating the efficiency of different algorithms in maneuvering through traffic, it would be important to also consider the average traversal time (also known as weights) of the different paths between each node/junction.

Different pathfinding algorithms

Overall, Algorithms differ in the way they traverse the graph, more specifically the decision-making process for choosing which nodes or areas of the graph should be searched next. Depending on how it is programmed, different algorithms can diverge significantly in efficiency. Furthermore, when measuring the performance of different algorithms, two different properties need to be measured, the speed as well as the accuracy of the algorithm: speed within the context of this environment will refer to the number of nodes or paths that need to be searched before a solution can be found whereas accuracy will refer to the resemblance of the solution found to the most optimal. The speed of an algorithm also affects its efficiency. An algorithm that can find a solution by searching through fewer nodes will also use up less memory and processing, an important metric to any programmer or game developer.

Certain algorithms are more biased toward one aspect than the other. For example, the greedy best-first-search algorithm can find a solution in a much shorter time than the Dijkstra search algorithm can, however, it does not guarantee a most optimal solution like Dijkstra does (specifics of each algorithm explained further in the paper). Furthermore, each algorithm can also differ in whether they consider the weights of each path. This could lead to further complications regarding performance and efficiency.

Weighted vs Unweighted pathfinding

In pathfinding, the difference between the paths found through an algorithm that considers the weights of paths versus an algorithm that does not is in the assurance of optimal. In an unweighted path, breadth-first search guarantees that the path it finds is the shortest, however, with a weighted path, when we reach a node, we cannot always be sure that we have reached it through the shortest possible path [UCSD, n.d]. This is because the path found may have traversed through an area with a heavy weight (slow movement speed), and another solution may be able to reach the same node in a shorter time. The Dijkstra algorithm, however, makes use of the priority queue data structure to ensure that every path found *is* the shortest. This is done by always extending the *best* (quickest/shortest) path in terms of the sum of the weights of all paths from the source node to that node. One drawback, however, is that the quickest path extended may not always be in the direction of the endpoint or be part of the final solution, thus wasting time searching through unnecessary areas.

Cost-metrics (*heuristics*)

On the other hand, certain algorithms may also use estimates of the potential distance between the current node and the end node [Patel, 2022]. This heuristic value helps the algorithm find a solution in a smaller number of nodes by cutting out large parts of the search space early on. One example of such an algorithm is the greedy best-first-search algorithm. Which always tends towards the direction of the endpoint by using heuristic functions to determine its decisions. This allows the algorithm to find solutions in fewer nodes searched. However, where it gains in speed it loses in accuracy. A solution found by the greedy-best-first search algorithm is most likely suboptimal.

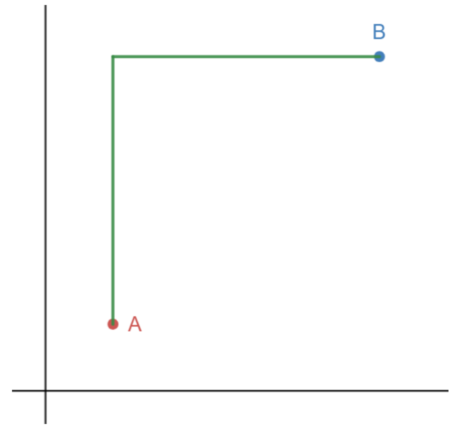

Figure 1: visual demonstration of Manhattan distance drawn in Desmos

Moreover, there are diverse ways in which the heuristic can be calculated, such as the Euclidean or the Manhattan distance. Whereas the Euclidean distance is calculated in a Pythagorean-like diagonal distance from the source node to the end node, the Manhattan distance takes the sum of the rise and run of the two points in coordinate space[Sharma 2020], a visual representation is shown in *figure 1 & figure 2*. For a square grid, such as the one used in this investigation, the Manhattan distance is the standard of the two as it assumes between two subsequent diagonal nodes an extra distance travelled horizontally or vertically, thus making it a better estimate of the shortest distance to the end node. [Patel, 2022]

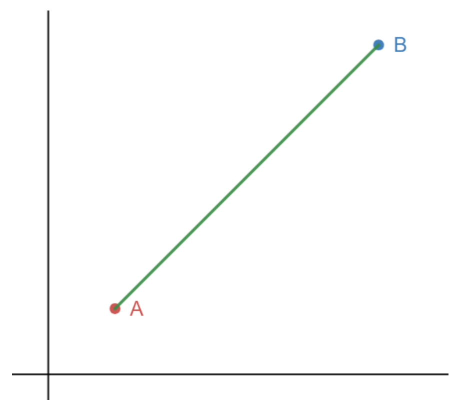

Figure 2: visual demonstration of Euclidean distance drawn in in Desmos

A* pathfinding

A* pathfinding is a popular choice for video game and software developers alike for its flexibility and adaptability [Patel, 2022]. Like Dijkstra, A* is a greedy pathfinding algorithm. At every iteration of the search, it keeps track of all the nodes that have been visited as well as their shortest known distance g(n) path to the source node. However, unlike Dijkstra, it also keeps track of a *heuristic* value h(n) of the estimated distance to the end node. These values are the elements that make up the cost metric of A*:

f(n) = h(n) + g(n) [Cui & Shi, 2011]. This cost function gives A* certain special properties over other search algorithms. First, that A* is provably optimal, that is it will find the optimal solution to a graph as long as the heuristic h(n) is always less than or equal to the cost of the actual most optimal path and it will in most cases find that path in fewer nodes searched [Hart, Nilson & Raphael, 1968, Pg. 103]. Furthermore, other factors can also be included in this function to aid in its flexibility to tailor to the problem at hand. At one extreme, if h(n) is always set to 0 and only g(n) makes up the cost metric, then A* becomes Dijkstra; the shortest path is guaranteed to be found, however, more nodes must be searched, wasting computer resources. On the other extreme, if g(n) is always set to 0, and only h(n) plays a role, A* becomes a Greedy-best-first search, being able to find a solution quickly, but sub-optimally [Patel, 2022]. The cost metric can therefore be adjusted, appending, or removing values per the type of environment and problem at hand to optimise either speed or accuracy or both. However, it still has its limitations. A* can be wasteful with resources. Due to the calculation of the cost metric, A* may consume too much CPU processing time for some practical uses.

Metaheuristics and Genetic algorithms

"A metaheuristic is formally defined as an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space" -[Osman, 1996]

Metaheuristics have been gaining popularity over the past decades as an optimisation tool for complex problems where heuristic methods (like the brute-force method) that always guarantee the most optimal solution are either impractical or would take too long [Blum & Roli, 2003]. Simply put, metaheuristics are methods of optimisation that guide the solution towards near-optimal solutions. They are made up of the elements of *diversification* and *intensification* [Blum & Roli, 2003]. Diversification, in short, means to explore the global scale or countably infinite set of discrete solutions, this is usually done through randomisation of variables to optimise the exploration. On the other hand, intensification refers to narrowing down the search scale around a satisfactory solution to find better solutions that ideally tend closer to the optimal [Blum & Roli, 2003]. Diversification on a global scale prevents the algorithm from being stuck within the same area as there is never a guarantee that a *good* solution would be near the *best* solution. Where metaheuristics outperform heuristics is in their adaptability. That is, they are merely methods to attain solutions, not the actual solutions themselves, and are usually developed in adherence to nature and evolution [Yang et al, 2013, Ch. 2.5.1]; Metaheuristics have a wide range of applications in different optimisation problems throughout distinct fields of studies, but they can never guarantee an optimal solution. Whereas heuristics are optimal solutions to problems that have been vigorously researched but do not apply universally.

Classification of Metaheuristics

Metaheuristics can be generally classified into two types of searches: Neighborhood search and population-base search [Bani Hani, 2020]. Certain algorithms such as the genetic algorithm, which makes use of a large population of different solutions generated randomly to explore the large space can be classified into the population-based search, whereas Neighborhood search-type algorithms, such as simulated annealing or Tabu search, tend to start with only one solution, which is updated and changed according to the fitness function as the algorithm tends towards an optimum. Population-based metaheuristics tend to have an advantage over Neighborhood search-based metaheuristics as multiple points can be assessed and changed at any point in time; more of the global search space can be assessed at a time.

Genetic Algorithms (GA)

The genetic algorithm, like other metaheuristics, is an approach to optimisation inspired by Darwin's theory of natural selection. Developed by (John Holland et al) in the 1970s, genetic algorithms sought to optimise solutions to problems through the abstraction of natural selection and were the first of their time to use crossover, recombination, mutation, and selection
[Yang, 2010, Ch. 11.1]. Ever since, many different implementations of genetic algorithms have been developed to solve a wide range of problems, from discrete to continuous systems across multiple different fields of study, highlighting the flexibility and practicality of genetic algorithms as metaheuristic algorithms. The implementation of a genetic algorithm typically involves first the initialisation of a large population of *chromosomes* (Strings of integers representing what is to be optimised, each integer is known as a *gene*). At initialisation, the researcher has a choice between decimal and binary genes. Binary genes are easier to manage as there exist only two states for each gene, 0 and 1 [Fawzy Gad, 2020]. However, making use of decimals allows for a wider range of

possible values in the population, meaning more divergent spaces within the global sphere can be explored at one point in time [J. Murray-Smith, 2012, Ch. 6.4.2]. This initial population is then compared to a pre-determined fitness function, with each set of parameters being ordered in ascending order from most fit to least fit. The operators *reproduction, crossover, and mutation* are then run; a pre-determined percentage of the population with the best fitness values will be retained for the next population, and the rest will be discarded and replaced with *offspring*. Genes from the current generation will be selected as *parent chromosomes*; genes from one parent will be replaced with genes from the other parent at random, creating *offspring* until is sufficient for the next generation. Finally, the investigator may allow for the *mutation* to occur, in which case on a random basis, genes in the population are selected and altered. This process is repeated until predetermined stopping criteria are met [Yang, 2010]

Methodology & Investigation

Firstly, a 2D grid environment was programmed using python and the pygame API, where each grid could be encoded with either the colour black, representing a barrier/wall, or white, representing free travel space. This environment was then used to measure the speed and accuracy of the pathfinding algorithms, measured in the number of nodes searched and the length of the solution found (Note: in this environment, there are no weights, and therefore the shorter the solution, the more optimal). To train the genetic algorithm, the PYGAD API was used to generate and sustain a population of fifty chromosomes over three thousand, and six thousand generations with a crossover rate of 50% and mutation chance of 10% (0.1). The fitness function made use of the maze-like grid environment, on initialisation, a random grid with a 20% random chance of barriers, or a pre-set grid was generated with a start and end node. The *chromosomes* were then assessed using each parameter as an index search of the "open nodes" list, if the parameter was larger than the number of items in the list, the algorithm will instead use *p mod n* (where p is the parameter index whereas n is the number of items in the list). The fitness score was then calculated using a sum operator on both the number of nodes searched and the length of the path found to ensure both values are minimised. For each solution, the investigation was conducted ten times (the genetic algorithm was run separately between the pre-set and random grid environments), and the fitness function was summed up to maximise the difference each change between generations made. The assumption was that this allowed for more targeted and optimal changes would be made between each generation of the genetic algorithm. The final most optimal solution found would then be run inside the environment, and finally, a graph of nodes searched vs solution length was plotted. The same methodology was used to investigate the A* and Dijkstra algorithms. The investigation aimed to compare the efficiency of the genetic algorithm against its heuristic counterparts as well as hopefully find an improvement in the genetic algorithm between generations. The hypothesis is that whilst the genetic algorithm may not be able to outperform the A* pathfinding algorithm (as it is provably optimal), it would still show some significant improvement over the Dijkstra pathfinding algorithm, notably in the nodes searched aspect. The full procedure of the investigation was as follows:

1. Creating a randomised environment and saving it as the pre-set environment
2. Initialise grid environment (either with a 20% chance of barrier or pre-set environment)
3. Initialise the Genetic algorithm, which includes:
    1. Creating the population of fifty *chromosomes*
    2. Establishing the number of generations in which the genetic algorithm will run for
    3. Setting the mutation and crossover rate at 0.1 and 0.5 respectively
    4. Establishing the fitness function
4. Run the program, which will output the final solution after x generations
5. Use the final solution in the test environment
6. Measure and plot the number of nodes searched vs path length

The investigation was conducted using a genetic algorithm generated after three thousand, as well as six thousand generations, as well as the Dijkstra and A* pathfinding algorithm (Manhattan distance) using both randomly generated environments and one pre-set environment. In both cases, the start and end nodes were randomised.

Randomly Generated grid environment

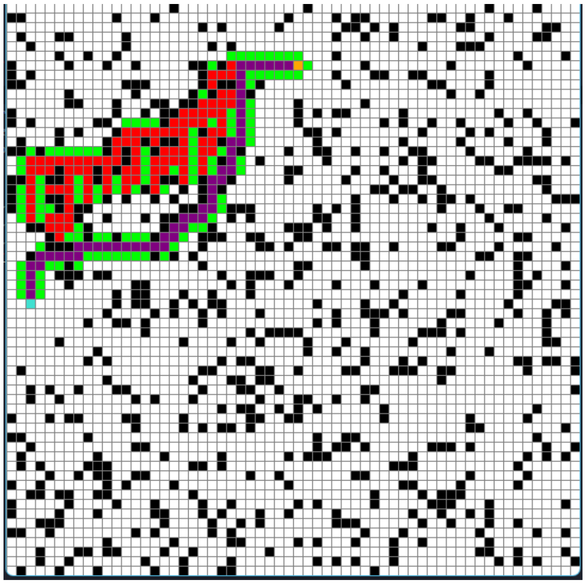

Figure 3: Demonstration of running of A pathfinding in a randomised environment*

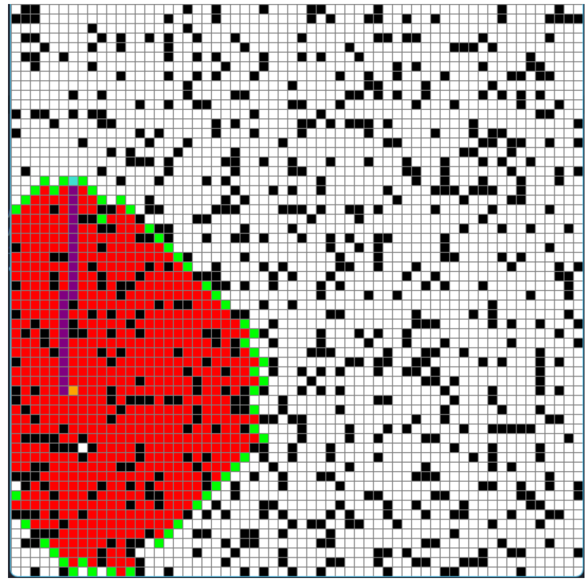

Figure 4: Demonstration of running of Dijkstra pathfinding in a randomised environment

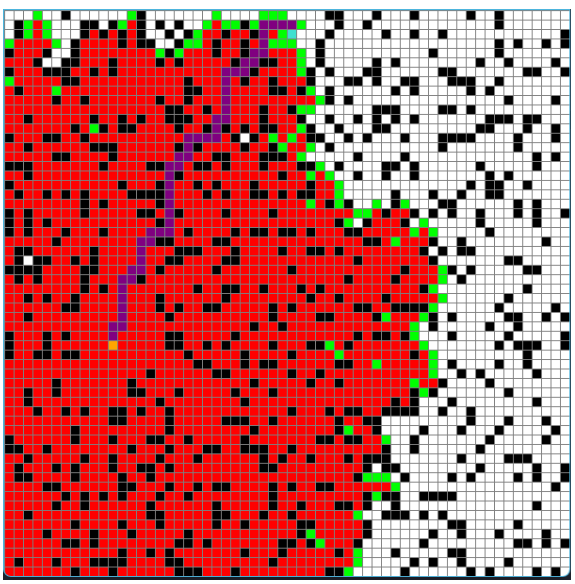

Figure 5: Demonstration of running of genetic algorithm after 3000 generations in a randomised environment

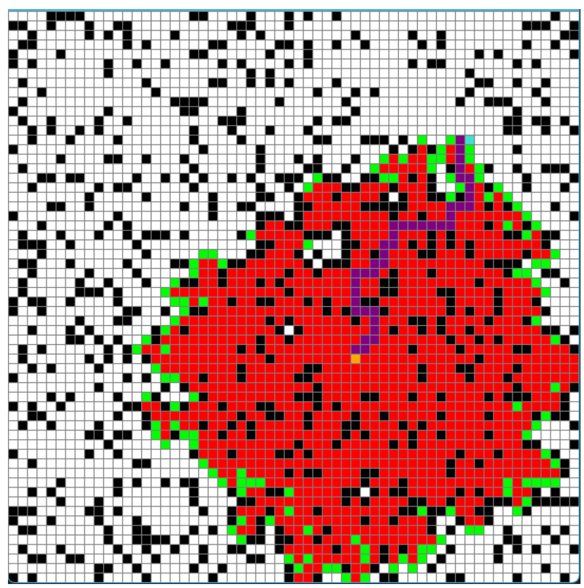

Figure 6: Demonstration of running of genetic algorithm after 6000 generations in a randomised environment

As shown by the diagrams above, there is a qualitative improvement in the efficiency of the genetic algorithm between three thousand and six thousand generations, however, the performance still tails significantly behind that of the A* algorithm. Further, it is also difficult to access the efficiency purely qualitatively as the starting position and ending position is randomised.

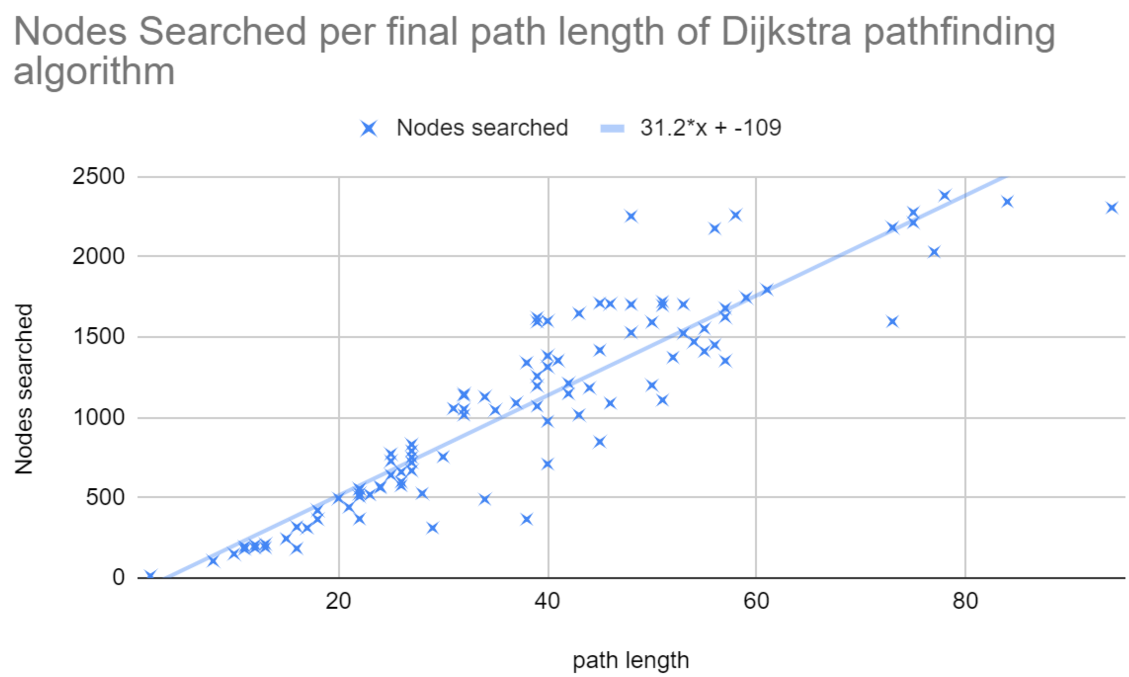

Figure 7: Plot of Nodes searched against Path length for 100 paths found by the Dijkstra pathfinding algorithm in the randomised grid environment

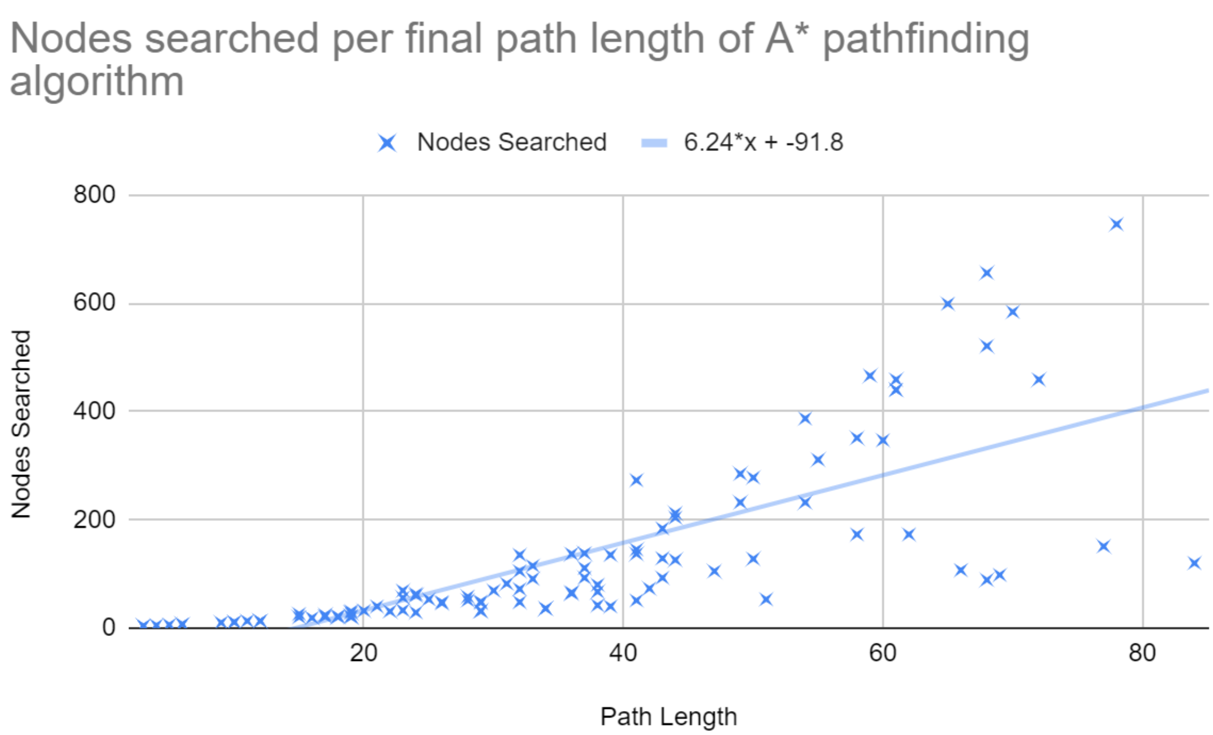

Figure 8: Plot of Nodes searched against Path length for 100 paths found by the Dijkstra pathfinding algorithm in the randomised grid environment

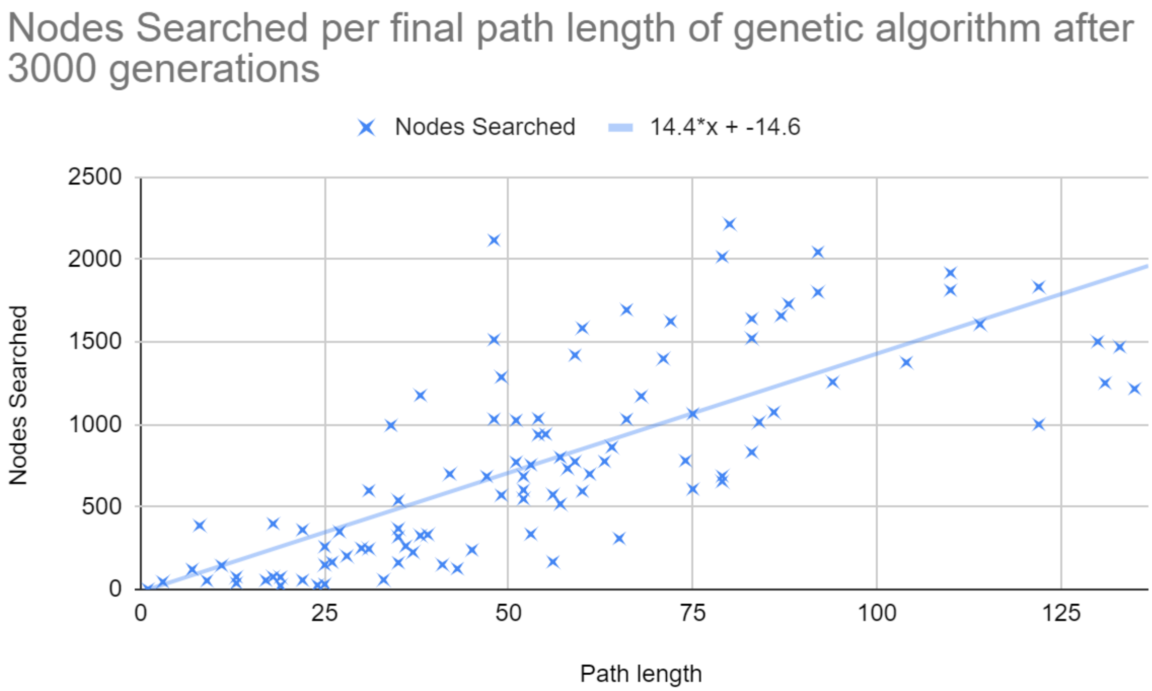

Figure 9: Plot of Nodes searched against Path length for 100 paths found by the Genetic algorithm heuristic after three thousand generations in the randomised grid environment

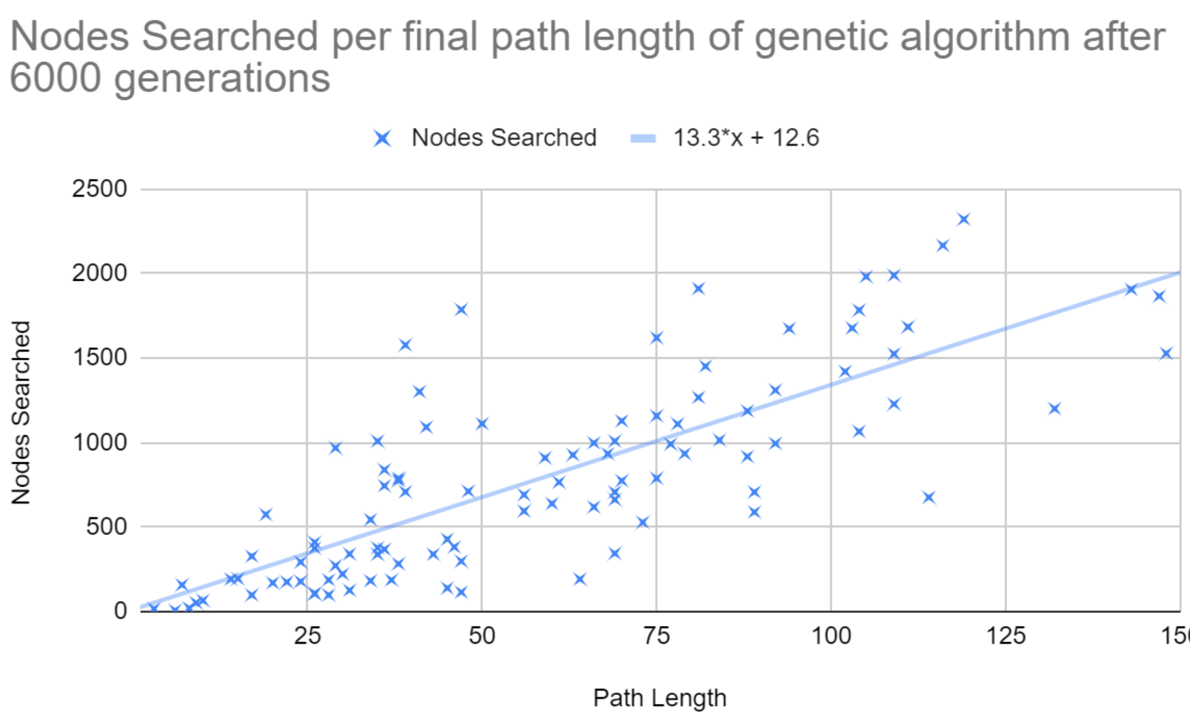

Figure 10: Plot of Nodes searched against Path length for 100 paths found by the Genetic algorithm heuristic after six thousand generations in the randomised grid environment

Pre-set grid environment

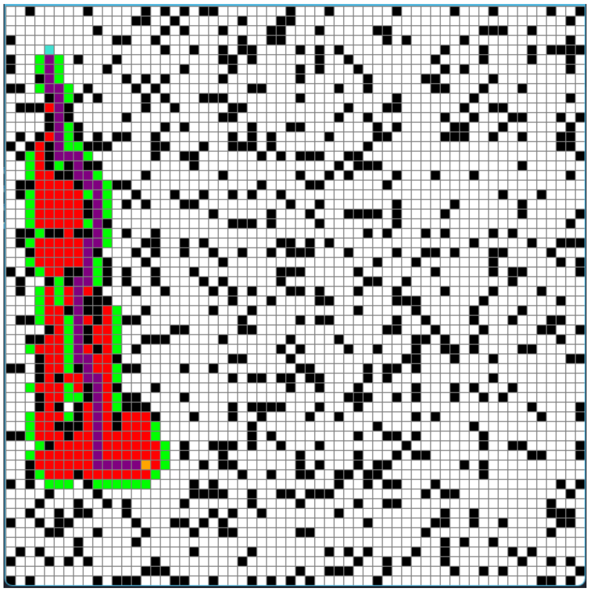

Figure 11: Demonstration of running of A pathfinding in a pre-set environment*

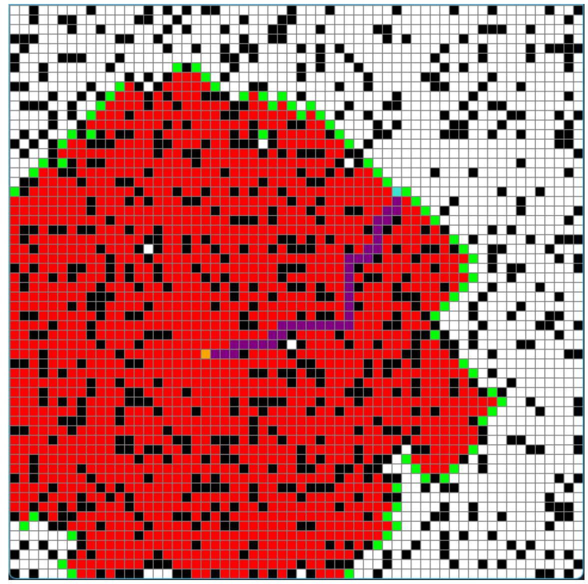

Figure 12: Demonstration of running of Dijkstra pathfinding in a pre-set environment

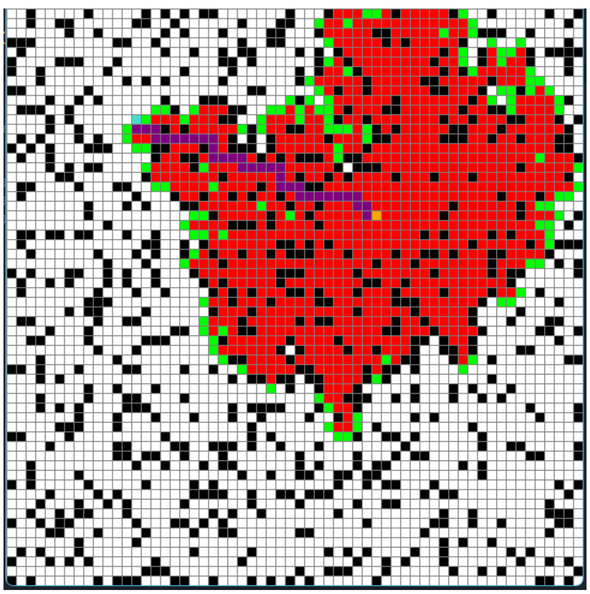

Figure 13: Demonstration of running of Genetic algorithm heuristic after three thousand generations in a pre-set environment

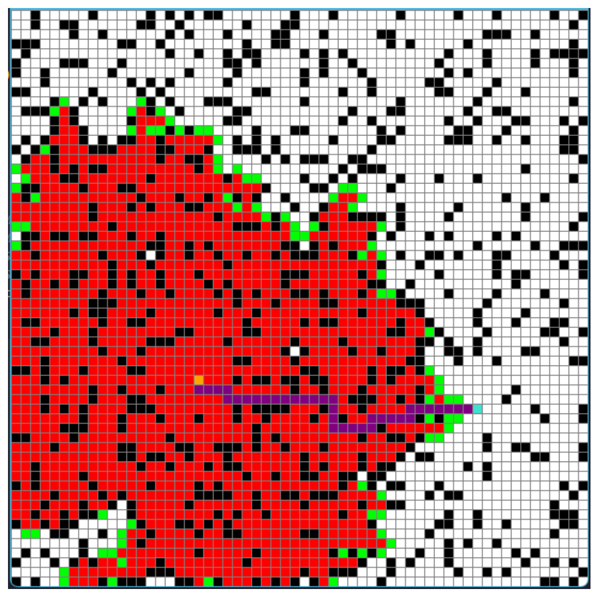

Figure 14: Demonstration of running of Genetic algorithm heuristic after six thousand generations in a pre-set environment

One interesting observation was that within the context of the pre-set grid environment, even though purely qualitative lenses, the genetic algorithms (at both generation three thousand and six thousand) seemed to be performing *faster* than the Dijkstra algorithm.

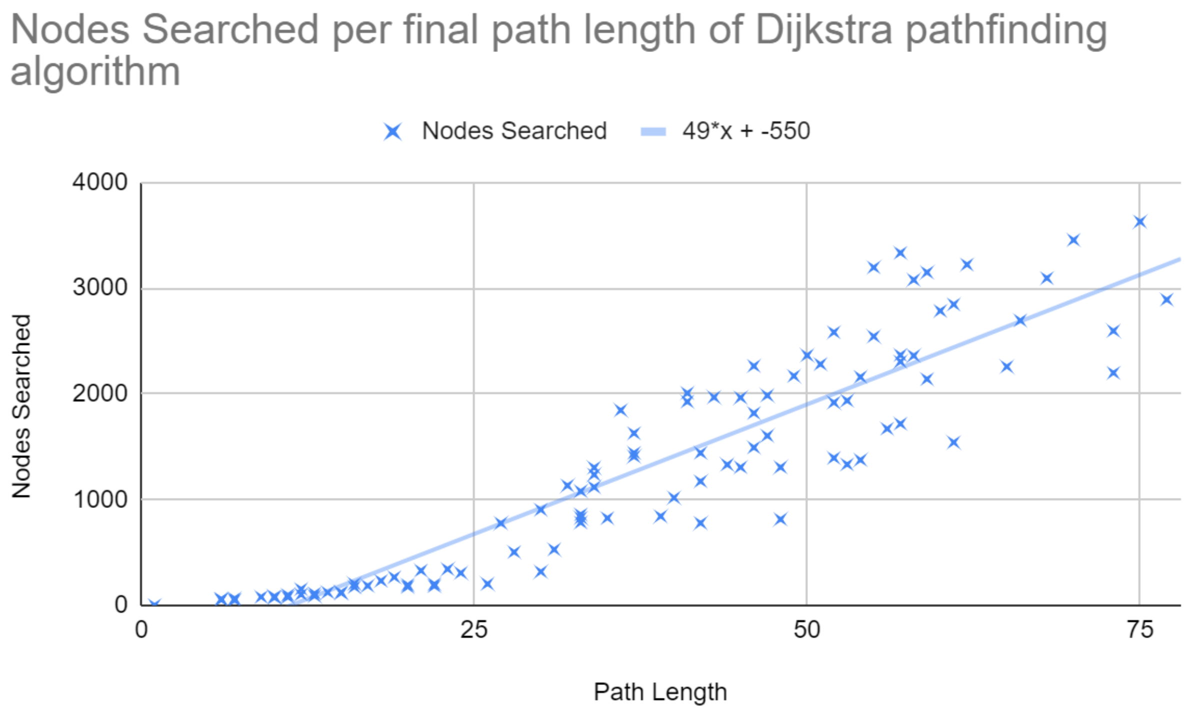

Figure 15: Plot of Nodes searched against Path length for 100 paths found by the Dijkstra pathfinding algorithm in the pre-set environment

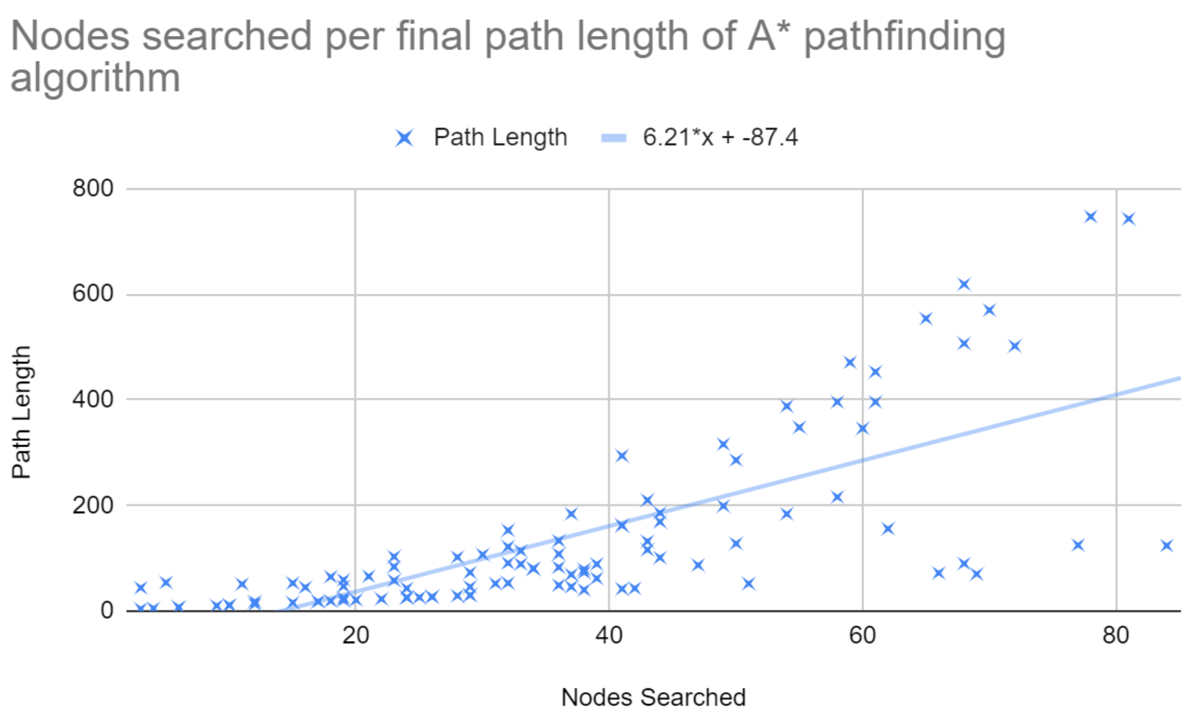

Figure 16: Plot of Nodes searched against Path length for 100 paths found by the A pathfinding algorithm in the randomised grid environment*

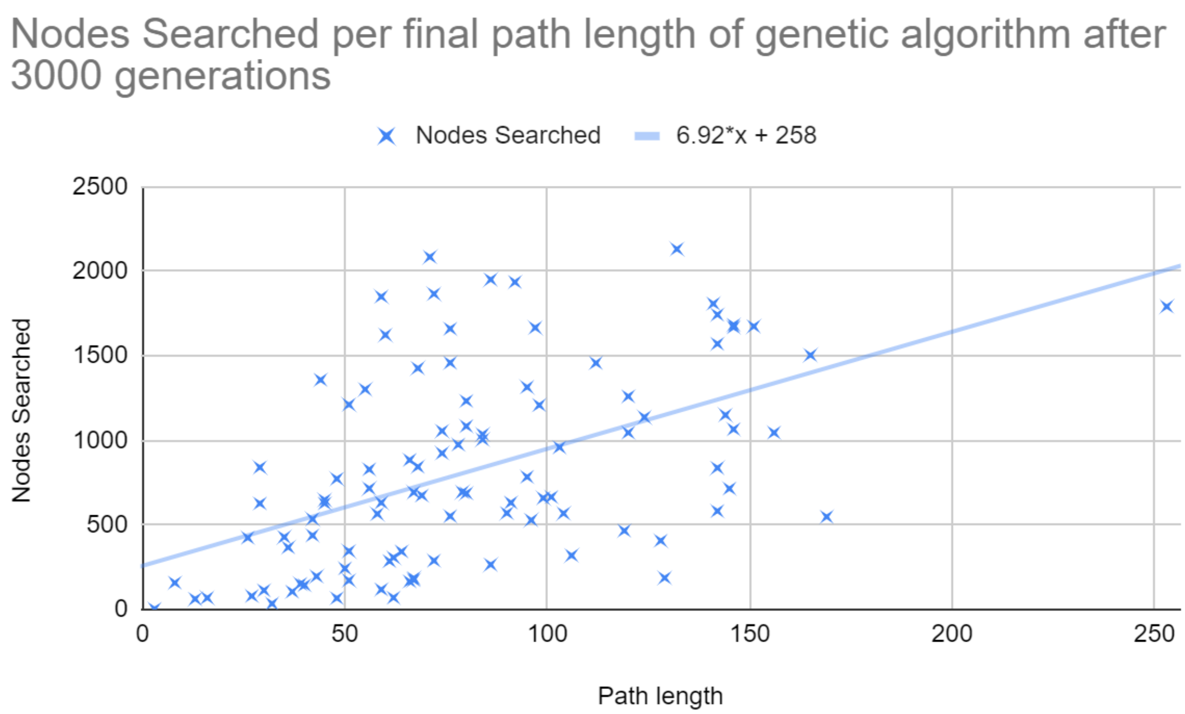

Figure 17: Plot of Nodes searched against Path length for 100 paths found by the Genetic algorithm heuristic after three thousand generations in the pre-set environment

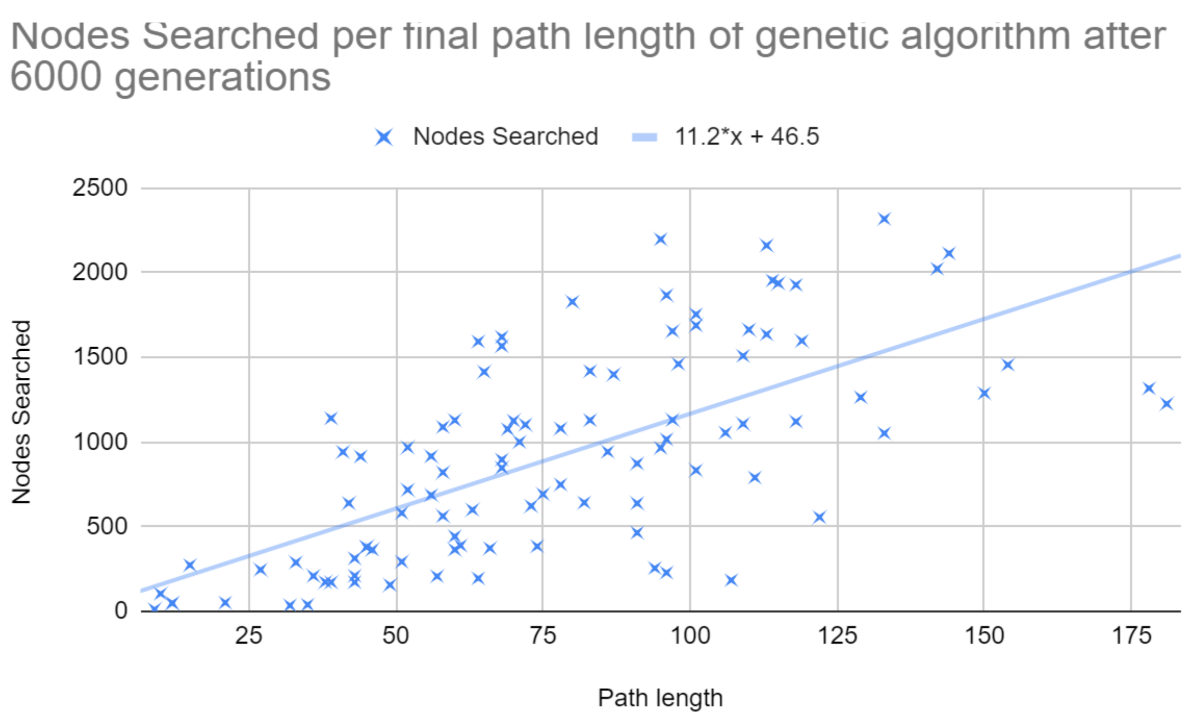

Figure 18: Plot of Nodes searched against Path length for 100 paths found by the Genetic algorithm heuristic after six thousand generations in the randomised grid environment

Data Analysis

Furthermore, within the context of this experiment, a lower gradient on the graph (in other words a low ratio between the number of nodes searched and the final path length) represents a better-performing algorithm. Finding a solution (Not explicitly the optimal solution) in a lower number of nodes searched meant that the algorithm was faster whilst finding a solution with a shorter path length proved a more accurate algorithm. As a combination of these two metrics, the ratio between the two measurements represents the overall efficiency of the algorithms, however, together with this value, the average Number of Nodes searched path length as well as the range of the Nodes searched and pathlength was considered to create a more vivid picture of each of the algorithm's weaknesses as well as strengths.

Environment	Algorithm	Average path length	Path length range	Average Nodes searched	Nodes searched range	Nodes searched per path length
Randomised environment	Dijkstra	38	94	1078	2382	30.5
	A*	36	81	134	742	6.61
	Three thousand generations	55.4	134	785	2213	14.7
	Six thousand generations	59	145	794	2315	13.3
Pre-set environment	Dijkstra	38	76	1226	2889	49.5
	A*	39	71	532	1657	19.7
	Three thousand generations	81	250	820	2128	6.92
	Six thousand generations	77	172	910	2306	11.4

Figure 19: Table of processed data of investigation

Implications of Data

Whilst the data collected shows that there is a possibility of genetic algorithms surpassing normal heuristic solutions in certain respects, it is not conclusive. Within the context of a randomised environment, the evidence collected does not suggest that the genetic algorithm performs better than the A* pathfinding algorithm; all measurements including the average path length as well as the nodes searched per path length were much lower in the case of the A* pathfinding algorithm. The average path length solved by the Genetic algorithm was even ~1.45~ times longer than that of the A* pathfinding algorithm. Furthermore, the genetic algorithm searched through more than double the number of nodes A* needed to conclude a path between the start and end nodes. All this points towards overall inferior performance. However, in comparison to the Dijkstra pathfinding algorithm, both genetic algorithms managed to find paths to the final solution within fewer average nodes searched (785 & 794 versus 1078). However, such an algorithm cannot compete with the Dijkstra pathfinding algorithm in accuracy, with many solutions being much longer and therefore much less optimal than solutions found by the Dijkstra algorithm. Moreover, both the genetic algorithm's path length and node searched range is larger than the Dijkstra algorithm; whilst on average the genetic algorithms may be able to find a solution in a shorter number of searches than the Dijkstra algorithm, this comes at a sacrifice to both accuracy and stability. The solutions found by the genetic algorithms are much less likely to be optimal, along with the algorithm's heightened probability to perform poorly depending on the environment. Overall, it is difficult to conclude that the Genetic algorithm serves a practical purpose within the context of this investigation. A randomized environment along with randomized start and end nodes may be currently out of reach for this implementation of a Genetic algorithm. The search space was simply too large to be explored within 3000 to 6000 generations, and therefore between the two algorithms, not much significant improvement can even be pointed out. Furthermore, within a randomized grid environment with no stability or repetition, there is not much basis for the genetic algorithm to improve upon; the problem constantly changes, so a requirement for the heuristic solution that comes about from a search space that does not stay constant between generations is improbable. It is a similar case for the pre-set environment, even here the A* algorithm proves its unparalleled efficiency. However, the genetic algorithm seems to have diverged between generation three thousand and generation six thousand. Whilst the algorithm at six thousand generation averaged a shorter solution, it must on an average search through a larger number of nodes to find it. And even then, the solution found is still double the length of the solutions found by the Dijkstra search algorithm. the evidence found does not point towards a conclusive potential for genetic algorithms to converge around a pathfinding solution that could outperform the A* or even the Dijkstra pathfinding algorithm, especially in terms of accuracy. This may have been due to poor or inadequate implementation of the genetic algorithm for the context of the problem, with the search space being too big for such an implementation to handle, furthermore, it may have been too ambitious to hope that the genetic algorithm would be able to converge at a solution for a randomized search space with no repetition or anchor for the algorithm to converge to. However, worse performance does not necessarily signify the genetic algorithms serve no purpose. Due to the unstable nature of the algorithm (the range of the path lengths found by the genetic algorithm after three thousand generations in the preset environment is 3.5 times larger than its A* counterpart, for example), there is a possibility of genetic algorithms being used in video game design to create realistic or organic path lengths for entities; Because of the extensive range of path lengths, an element of induced randomisation could be created, which may serve as an interesting addition to adventure or horror game development. Furthermore, within the case of the randomised environment, there was (however insignificant) an improvement across all statistics measured between generation three thousand and generation six thousand, indicating slight potential for genetic algorithms to converge given more thoroughly researched implementations with a more compact search space.

Limitations of the methodology

The methodology of this investigation contained the following shortcomings which impacted the investigation and the data gathered:

1. The grid environment created did not include weights of paths which is where the Dijkstra and A* pathfinding algorithms truly proved their worth. Within this implementation, every node was considered to have a weight of one, however, in such a case, the Dijkstra algorithm ultimately reduces to not much more than the breadth-first search algorithm whilst the A* pathfinding acts very similar to the Greedy-best-first search. An implementation of weights to the environment may have extended the gap in performance between the heuristic algorithms and the genetic algorithms
2. The fitness function included both measurements of nodes searched as well as path length. This was in hopes that both measurements would be optimised in tandem. However, it may have been too ambitious, and implementation of the fitness function only focussing on minimising either metric may have wielded more conclusive results.
3. The mutation probability of 10% chosen was too small for the large, randomised search space. With fifty *chromosomes* in each generation, each with one hundred *genes*, the randomisation was too small to search the global search space in a wide enough range to find an optimal solution

Further research

Due to the impractical use of Genetic algorithms to solve pathfinding as a whole, shown in this research paper, due to the algorithm's inability to search such a large global space in a manner that is both accurate and at a speed that serves a practical use, other metaheuristic algorithms, such as a local search algorithm, targeted on building on and improving the already efficient and provably optimal A* pathfinding algorithm, perhaps with a goal of improving the memory or data usage whilst the algorithm is running. Furthermore, the global scale can also be decreased for further research, investigating on a smaller grid or a grid/maze environment created using wave function collapse, this way, the genetic algorithm may be able to develop a link and understanding between the different modules. On the other hand, instead of metaheuristics, a neural network may be implemented, having the final solution be able to make actual logical decisions whilst the program is running, allowing the solution to adapt to even changing environments.

Conclusion

In conclusion, this investigation has highlighted the versatility of metaheuristics and AI as well as their role in the future of the web, especially in creating realistic AIs which can be implemented into video games, as well as their limitations and shortcomings in terms of efficiency when compared to heuristic solutions built for a specific purpose. Although the data gathered is inconclusive, this investigation leaves many open doors for the future development and of metaheuristic algorithms as a whole.

References

Xiao Cui, Hao Shi. (Nov 2010). A*-based Pathfinding in Modern Computer Games. Retrieved from

https://www.researchgate.net/profile/Xiao-Cui-12/publication/267809499_A-based_Pathfinding_in_Modern_Computer_Games/links/54fd73740cf270426d125adc/A-based-Pathfinding-in-Modern-Computer-Games.pdf

Daniel Monzonís Laparra. (15 Janruary 2019). PATHFINDING ALGORITHMS IN GRAPHS AND APPLICATIONS. Retrieved from

http://diposit.ub.edu/dspace/bitstream/2445/140466/1/memoria.pdf

Tong Pan, Shuk Ching PUN-Cheng. (28 August 2020). A Discussion on the Evolution of the Pathfinding

Algorithms. Retrieved from

https://www.preprints.org/manuscript/202008.0627/v1

Zeyad Abd Algfoor et al. (18 March 2015). A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games. Retrieved from

https://dl.acm.org/doi/epdf/10.1155/2015/736138

Pulkit Sharma (22 June 2022). 4 Types of Distance Metrics in Machine Learning. Retrieved from

https://www.analyticsvidhya.com/blog/2020/02/4-types-of-distance-metrics-in-machine-learning/

Amit Patel. (18 August 2022). Introduction to A*. Retrieved from

http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html#:~:text=Dijkstra%27s%20Algorithm%20works%20harder%20but,it%27s%20not%20the%20right%20path.

Mark Needham and Amy E. Hodler. (2019). Graph Algorithms Practical Examples in Apache Spark and Neo4j. O'Reilly Media, Inc

University of San Diego. (n.d.). Weighted vs. unweighted shortest path algorithms. Retrieved from

https://cseweb.ucsd.edu//~kube/cls/100/Lectures/lec12/lec12-28.html

Amit Patel. (18 August 2022). A*'s Use of the Heuristic. Retrieved from

http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

David J. Murray-Smith. (2012). Modelling and Simulation of Integrated Systems in Engineering. Woodhead Publishing

Xin-She Yang. (2010). Engineering Optimization: An Introduction with Metaheuristic Applications. John Wiley & Sons Inc.

Xin-She Yang et al. (2013). Metaheuristic Applications in Structures and Infrastructures. Elsevier Inc.

Christian Blum, Andrea Roli (Janruary 2001). Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. Retrieved from

https://www.researchgate.net/publication/221900771_Metaheuristics_in_Combinatorial_Optimization_Overview_and_Conceptual_Comparison

Ibrahim H Osman. (October 1996). Metaheuristics: A Bibliography. Retrieved from

https://www.researchgate.net/publication/234014434_Metaheuristics_A_Bibliography

Dana Bani-Hani. (14 June 2020). Genetic Algorithm (GA): A Simple and Intuitive Guide. Retrieved from

https://towardsdatascience.com/genetic-algorithm-a-simple-and-intuitive-guide-51c04cc1f9ed

Ahmed Fawzy Gad. (2020). Working with Different Genetic Algorithm Representations in Python. Retrieved from

https://blog.paperspace.com/working-with-different-genetic-algorithm-representations-python/

P. E. Hart, N. J. Nilsson & B. Raphael. (July 1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*.

Tech with Tim YouTube Channel. (17 July 2020). A* Pathfinding Visualization Tutorial - Python A* Path Finding Tutorial. Retrieved from

https://youtu.be/JtiK0DOeI4A

Appendix

Code for the Genetic algorithm using PYGAD API. Adapted from Official PYGAD documentation: https://pygad.readthedocs.io/en/latest/README_pygad_ReadTheDocs.html

(*Note: the main() function loads up the environment and runs the designated pathfinding algorithm)*

```
def fitness_function(population, solution_idx):
  count = 0
  for x in range(10):
    count += main(600,population, savedGrid)
  return -count

def generation(ga_instance):
  print("Generation: ", ga_instance.generations_completed)

def on_start(ga_instance):
  print("Generation: 0")
ga_instance = pygad.GA(num_generations=6000,
             num_parents_mating=25,
             sol_per_pop=50,
             num_genes=100,
             fitness_func=fitness_function,

             init_range_low=0,
             init_range_high=1000,
             gene_type=int,

             mutation_type="random",
             mutation_probability=0.1,
             random_mutation_min_val= 0,
             random_mutation_max_val= 1000,
             mutation_by_replacement=True,

             crossover_type="single_point",

             save_best_solutions=True,
             on_generation=generation,
             on_start=on_start
             )
```

```
ga_instance.run()
print()
ga_instance.plot_fitness()

solution, solution_fitness, solution_idx = ga_instance.best_solution()
print("Parameters of the best solution : {solution}".format(solution=solution))
print("Fitness value of the best solution =
{solution_fitness}".format(solution_fitness=solution_fitness))
print("Index of the best solution : {solution_idx}".format(solution_idx=solution_idx))
```

Code for the main() function which loads up the environment and runs a designated pathfinding algorithm. Adapted from Tech With Tim YouTube Channel (2020) (*Note: the SavedGrid functionality has already been implemented within this code)*

```
from ast import Lambda
from calendar import c
from lib2to3.refactor import MultiprocessingUnsupported
from operator import le
from pickle import REDUCE
from re import L
import pygame
import math
from queue import PriorityQueue
import random
import re
height = 800
width = 600
win = pygame.display.set_mode((width,width))
pygame.display.set_caption("pathfinding :DDD")
savedGrid = [[5, 5, 11, 11, 10, 10, 6, 11, 7, 11, 3, 9, 11, 3, 11, 8, 4, 11, 11, 8, 6, 4, 11, 11, 7, 5, 5,
11, 11, 2, 10, 11, 8, 2, 9, 2, 11, 8, 2, 4, 0, 6, 7, 11, 10, 8, 1, 6, 1, 5, 0, 4, 10, 3, 2, 11, 9, 8, 2, 10],
[11, 8, 11, 0, 10, 3, 2, 9, 7, 11, 1, 5, 1, 11, 7, 2, 6, 11, 3, 4, 4, 9, 11, 6, 11, 10, 11, 9, 5, 0, 1, 11, 5,
11, 7, 8, 8, 9, 8, 0, 1, 4, 2, 2, 5, 6, 11, 11, 11, 0, 8, 11, 10, 6, 10, 2, 7, 7, 11, 11], [5, 2, 8, 0, 8, 11,
11, 3, 2, 11, 5, 3, 11, 6, 9, 1, 10, 3, 9, 8, 10, 9, 6, 5, 7, 5, 1, 1, 1, 0, 6, 2, 6, 11, 8, 11, 11, 11, 4, 11,
11, 2, 0, 7, 1, 9, 5, 7, 6, 6, 11, 4, 11, 2, 1, 11, 10, 9, 3, 1], [8, 8, 10, 6, 8, 0, 9, 10, 11, 0, 11, 11, 10,
11, 11, 11, 4, 8, 0, 2, 8, 7, 11, 9, 6, 6, 10, 11, 6, 9, 10, 6, 11, 7, 3, 5, 9, 5, 8, 8, 6, 4, 7, 7, 11, 2, 3,
8, 0, 11, 2, 0, 0, 11, 5, 5, 11, 1, 2, 7], [11, 2, 1, 7, 11, 4, 1, 0, 2, 0, 1, 2, 6, 7, 4, 7, 11, 3, 4, 9, 10, 5,
11, 10, 2, 10, 10, 10, 5, 9, 10, 7, 7, 3, 9, 7, 10, 11, 7, 0, 11, 10, 11, 2, 7, 0, 8, 9, 10, 4, 2, 11, 11,
11, 1, 7, 1, 7, 10, 2], [11, 11, 2, 10, 11, 8, 0, 1, 9, 11, 11, 5, 1, 5, 1, 11, 11, 8, 2, 5, 3, 10, 2, 6, 1, 8,
11, 11, 10, 10, 11, 3, 4, 10, 9, 10, 7, 2, 9, 1, 10, 6, 11, 11, 1, 8, 8, 7, 6, 9, 5, 6, 11, 10, 10, 6, 11, 0,
10, 10], [9, 1, 9, 10, 11, 6, 0, 6, 11, 7, 7, 11, 11, 9, 5, 6, 9, 11, 7, 1, 0, 10, 11, 1, 6, 3, 11, 10, 10, 5,
9, 1, 1, 1, 6, 4, 11, 0, 6, 3, 11, 11, 11, 7, 9, 5, 4, 11, 0, 8, 11, 11, 7, 3, 11, 11, 5, 0, 2, 6], [0, 7, 5, 6,
4, 4, 2, 11, 6, 2, 2, 7, 10, 11, 5, 11, 7, 3, 1, 11, 11, 4, 11, 0, 8, 0, 3, 8, 8, 11, 11, 9, 11, 11, 9, 7, 4,
10, 11, 11, 3, 7, 6, 2, 5, 6, 0, 9, 11, 7, 6, 11, 10, 6, 7, 6, 11, 1, 0, 11], [0, 11, 7, 10, 0, 1, 1, 6, 11, 5,
2, 4, 11, 11, 10, 11, 0, 7, 7, 8, 7, 1, 7, 3, 11, 7, 3, 4, 11, 11, 11, 11, 3, 5, 11, 8, 8, 0, 6, 6, 0, 6, 2, 3,
7, 0, 5, 2, 2, 11, 2, 10, 9, 11, 3, 5, 5, 0, 11, 9], [9, 1, 4, 5, 6, 11, 4, 8, 1, 8, 8, 7, 4, 11, 3, 5, 10, 0, 6,
4, 3, 11, 6, 1, 5, 11, 11, 11, 1, 11, 2, 2, 9, 0, 5, 9, 7, 2, 1, 8, 9, 8, 2, 3, 9, 0, 7, 8, 6, 6, 11, 8, 4, 4, 4,
8, 7, 4, 8, 2], [5, 7, 11, 1, 11, 3, 1, 10, 8, 8, 5, 2, 11, 0, 6, 0, 0, 11, 2, 8, 4, 4, 8, 1, 2, 1, 3, 7, 0, 4, 1,
7, 2, 10, 5, 11, 3, 0, 11, 10, 7, 11, 11, 7, 11, 11, 9, 6, 8, 11, 5, 10, 1, 2, 7, 1, 3, 2, 11, 4], [4, 9, 11,
4, 6, 9, 11, 2, 2, 5, 11, 10, 11, 7, 3, 8, 1, 11, 1, 11, 8, 3, 11, 1, 3, 6, 11, 11, 2, 10, 8, 11, 0, 3, 6, 5,
11, 4, 1, 11, 11, 5, 4, 9, 11, 11, 7, 5, 8, 10, 11, 11, 0, 10, 9, 5, 3, 0, 11, 6], [11, 8, 10, 9, 6, 7, 1, 11,
7, 0, 10, 2, 7, 1, 6, 7, 0, 4, 11, 3, 8, 2, 4, 10, 11, 9, 10, 0, 11, 3, 6, 11, 1, 8, 11, 5, 11, 4, 4, 11, 11,
4, 0, 3, 2, 8, 11, 8, 7, 3, 2, 1, 11, 0, 11, 4, 3, 9, 11, 1], [11, 5, 8, 8, 8, 7, 11, 1, 11, 11, 9, 2, 7, 10, 1,
0, 11, 10, 0, 11, 3, 4, 2, 11, 2, 11, 7, 3, 9, 7, 11, 7, 10, 11, 6, 2, 4, 1, 4, 10, 11, 10, 6, 7, 5, 11, 1, 4,
11, 3, 0, 2, 2, 11, 4, 9, 5, 11, 10, 11], [8, 3, 11, 1, 6, 8, 3, 11, 9, 9, 8, 10, 11, 11, 11, 6, 2, 0, 6, 1, 6,
```

2, 11, 11, 11, 0, 11, 11, 3, 4, 8, 1, 9, 11, 10, 0, 4, 1, 11, 3, 4, 11, 4, 5, 2, 11, 2, 9, 5, 4, 7, 10, 0, 5,
5, 8, 11, 11, 4, 0], [8, 11, 1, 11, 10, 2, 11, 7, 11, 6, 10, 11, 1, 11, 10, 2, 8, 4, 2, 5, 4, 4, 2, 4, 1, 7, 2,
1, 11, 3, 3, 10, 6, 11, 1, 8, 10, 6, 1, 0, 8, 3, 6, 11, 0, 1, 11, 6, 3, 0, 6, 2, 2, 1, 10, 1, 7, 11, 8, 11],
[11, 5, 5, 9, 4, 0, 2, 5, 1, 11, 2, 2, 3, 0, 10, 1, 3, 1, 11, 6, 10, 1, 10, 1, 2, 0, 8, 4, 9, 7, 3, 6, 11, 5,
10, 8, 5, 10, 0, 7, 4, 4, 2, 0, 11, 6, 9, 4, 6, 8, 6, 9, 11, 5, 7, 4, 0, 2, 11, 0], [2, 11, 1, 4, 1, 11, 6, 5, 3,
6, 5, 11, 4, 5, 11, 4, 4, 10, 5, 11, 2, 8, 5, 11, 11, 5, 0, 7, 4, 2, 10, 5, 11, 4, 5, 10, 11, 3, 4, 11, 2, 7,
1, 6, 11, 11, 7, 5, 10, 2, 11, 0, 11, 0, 5, 4, 4, 0, 11, 11], [10, 10, 4, 11, 2, 10, 3, 7, 7, 1, 10, 0, 1, 10,
11, 11, 10, 4, 5, 11, 7, 0, 0, 3, 4, 9, 11, 3, 6, 10, 6, 5, 5, 8, 1, 9, 1, 1, 0, 4, 9, 11, 9, 0, 4, 6, 1, 2, 11,
11, 2, 6, 10, 11, 7, 11, 4, 6, 6, 3], [3, 9, 6, 5, 5, 6, 10, 1, 11, 5, 5, 10, 2, 11, 9, 0, 5, 6, 11, 8, 1, 7, 8,
11, 2, 6, 10, 11, 5, 2, 6, 5, 10, 4, 2, 0, 8, 7, 0, 1, 4, 5, 2, 2, 0, 6, 11, 5, 3, 11, 6, 6, 11, 10, 6, 9, 2, 7,
3, 11], [1, 4, 10, 4, 3, 9, 6, 0, 11, 3, 0, 6, 1, 1, 10, 2, 7, 7, 10, 3, 11, 1, 10, 10, 11, 0, 2, 8, 11, 3, 11,
7, 9, 9, 10, 3, 11, 2, 3, 7, 7, 2, 8, 9, 11, 9, 4, 10, 11, 11, 6, 11, 4, 3, 2, 6, 0, 8, 11, 11], [4, 11, 3, 11,
11, 0, 11, 3, 11, 8, 11, 1, 10, 1, 6, 6, 11, 10, 1, 3, 1, 3, 4, 3, 9, 11, 4, 11, 6, 0, 4, 5, 5, 11, 9, 8, 1, 4,
9, 3, 6, 9, 0, 4, 11, 11, 11, 10, 7, 11, 8, 4, 11, 11, 6, 6, 8, 5, 7, 0], [7, 8, 3, 3, 11, 11, 6, 8, 10, 11,
11, 6, 11, 11, 9, 0, 4, 7, 11, 0, 9, 11, 1, 10, 8, 10, 3, 10, 11, 11, 6, 2, 3, 7, 1, 11, 2, 11, 3, 6, 11, 11,
9, 1, 11, 2, 11, 7, 4, 9, 8, 11, 4, 11, 0, 5, 10, 2, 7, 2], [11, 2, 11, 11, 4, 10, 9, 10, 7, 11, 9, 0, 7, 11,
3, 9, 9, 7, 8, 2, 1, 11, 10, 5, 11, 5, 1, 10, 2, 6, 10, 11, 11, 0, 7, 11, 5, 5, 4, 5, 9, 7, 2, 10, 6, 3, 10,
11, 0, 9, 3, 6, 1, 2, 10, 7, 11, 3, 1, 0], [5, 2, 10, 11, 11, 2, 10, 11, 6, 4, 8, 11, 11, 11, 4, 10, 10, 10,
11, 3, 4, 11, 5, 7, 7, 3, 8, 0, 11, 7, 11, 2, 11, 6, 0, 10, 0, 11, 1, 6, 11, 5, 11, 11, 11, 2, 1, 0, 8, 11,
11, 10, 1, 0, 1, 11, 6, 1, 11, 9], [10, 1, 3, 7, 1, 0, 4, 11, 4, 4, 10, 10, 3, 8, 11, 1, 5, 11, 9, 2, 6, 7, 1,
5, 5, 6, 6, 6, 10, 8, 0, 3, 9, 2, 7, 10, 10, 11, 2, 3, 11, 11, 10, 4, 7, 2, 5, 3, 10, 5, 10, 11, 8, 7, 0, 10,
6, 4, 5, 7], [0, 11, 11, 10, 10, 3, 2, 4, 9, 1, 7, 1, 11, 11, 5, 9, 8, 0, 0, 7, 11, 11, 2, 8, 11, 5, 3, 3, 1,
11, 4, 2, 7, 8, 10, 8, 2, 5, 9, 5, 11, 3, 3, 11, 3, 4, 1, 11, 1, 10, 4, 0, 7, 3, 3, 6, 7, 1, 11, 3], [11, 11,
11, 0, 1, 10, 3, 5, 8, 8, 1, 2, 9, 4, 11, 3, 9, 3, 11, 6, 5, 8, 9, 11, 10, 5, 11, 11, 3, 6, 4, 6, 6, 5, 11, 5,
8, 11, 9, 3, 11, 8, 7, 10, 11, 7, 0, 1, 7, 11, 6, 6, 0, 6, 10, 10, 7, 6, 0, 5], [11, 5, 4, 6, 5, 1, 2, 4, 8, 7,
0, 11, 0, 10, 1, 5, 5, 5, 6, 11, 3, 3, 7, 11, 11, 4, 11, 6, 11, 3, 9, 8, 2, 11, 5, 11, 4, 11, 11, 4, 5, 8, 5,
5, 10, 6, 0, 5, 0, 11, 4, 6, 4, 7, 5, 0, 6, 0, 11, 11], [3, 9, 0, 11, 2, 11, 11, 0, 11, 1, 8, 11, 8, 6, 11, 6,
11, 4, 1, 6, 2, 8, 5, 6, 11, 4, 11, 9, 11, 6, 8, 11, 8, 4, 11, 0, 11, 3, 8, 9, 2, 5, 0, 2, 0, 11, 11, 11, 0, 3,
11, 6, 1, 11, 1, 5, 2, 11, 1, 6], [8, 8, 8, 2, 3, 8, 5, 7, 9, 6, 0, 3, 3, 6, 11, 5, 7, 11, 10, 7, 11, 7, 9, 11,
11, 1, 2, 3, 9, 0, 0, 0, 7, 5, 3, 6, 11, 11, 1, 0, 7, 11, 4, 1, 7, 1, 2, 11, 9, 11, 10, 8, 10, 11, 3, 2, 10, 3,
11, 3], [8, 11, 1, 11, 11, 11, 8, 7, 8, 7, 2, 6, 9, 0, 11, 10, 10, 11, 7, 11, 10, 11, 3, 8, 3, 3, 7, 3, 11,
11, 2, 11, 10, 8, 10, 1, 0, 11, 11, 1, 11, 7, 10, 7, 11, 7, 11, 0, 9, 11, 2, 2, 9, 2, 1, 1, 5, 4, 8, 5], [10,
7, 5, 0, 10, 9, 1, 9, 5, 4, 0, 6, 11, 5, 8, 2, 11, 2, 6, 5, 1, 9, 0, 11, 0, 0, 1, 0, 9, 11, 1, 11, 2, 0, 5, 2, 7,
7, 5, 4, 7, 8, 10, 6, 4, 5, 0, 1, 0, 11, 10, 10, 1, 1, 7, 11, 9, 11, 2, 11], [6, 6, 9, 11, 8, 8, 1, 11, 8, 6,
11, 9, 10, 7, 5, 11, 0, 10, 9, 7, 0, 6, 5, 3, 10, 3, 8, 1, 3, 1, 6, 8, 9, 11, 1, 0, 3, 5, 4, 10, 3, 11, 9, 2, 0,
7, 9, 11, 11, 3, 8, 11, 4, 5, 10, 1, 0, 11, 1, 7], [7, 0, 7, 3, 11, 3, 6, 6, 5, 11, 9, 5, 4, 4, 11, 0, 11, 7, 2,
3, 11, 4, 9, 1, 11, 7, 6, 2, 6, 3, 5, 0, 6, 3, 11, 5, 1, 2, 3, 11, 8, 3, 1, 4, 7, 7, 3, 11, 9, 7, 7, 2, 0, 6, 8,
11, 5, 11, 4, 10], [8, 0, 5, 1, 1, 11, 2, 0, 8, 6, 0, 3, 9, 3, 11, 11, 6, 2, 8, 6, 11, 4, 8, 9, 10, 4, 11, 1,
11, 4, 11, 8, 2, 4, 7, 9, 8, 2, 11, 11, 11, 0, 9, 11, 2, 11, 11, 8, 0, 10, 11, 5, 3, 9, 3, 8, 11, 1, 1, 10],
[2, 8, 6, 0, 8, 2, 11, 11, 9, 9, 11, 2, 6, 11, 2, 11, 8, 1, 0, 4, 11, 4, 11, 1, 5, 9, 4, 11, 0, 3, 9, 4, 8, 6,
6, 1, 11, 11, 5, 11, 8, 5, 2, 9, 6, 3, 7, 11, 11, 0, 3, 8, 11, 5, 5, 11, 10, 6, 1, 0], [5, 11, 2, 3, 1, 4, 8, 5,
10, 6, 4, 11, 11, 3, 2, 11, 4, 9, 0, 7, 11, 2, 0, 2, 7, 11, 10, 11, 6, 2, 9, 9, 5, 10, 11, 7, 9, 4, 4, 0, 9, 5,
11, 7, 5, 6, 11, 11, 0, 7, 2, 8, 1, 0, 10, 8, 9, 2, 11, 0], [4, 11, 11, 5, 6, 7, 2, 2, 3, 11, 2, 9, 11, 2, 9, 7,
7, 11, 9, 8, 6, 1, 11, 8, 2, 7, 9, 6, 4, 8, 2, 6, 11, 1, 8, 8, 11, 11, 0, 9, 4, 0, 10, 11, 6, 2, 11, 9, 11, 7,
11, 4, 6, 6, 6, 6, 11, 11, 2, 2], [8, 1, 6, 10, 5, 3, 8, 8, 11, 11, 0, 11, 3, 8, 7, 9, 11, 10, 6, 11, 11, 11,
0, 1, 11, 4, 9, 10, 11, 11, 5, 11, 11, 4, 10, 8, 11, 8, 2, 11, 3, 7, 4, 11, 0, 11, 3, 1, 11, 2, 11, 1, 1, 5,

```
0, 3, 2, 8, 3, 11], [7, 2, 11, 8, 2, 3, 10, 2, 6, 5, 6, 6, 10, 10, 7, 5, 9, 9, 5, 4, 2, 0, 4, 4, 4, 10, 9, 11, 8,
11, 10, 5, 5, 4, 1, 11, 9, 9, 3, 0, 6, 8, 8, 2, 11, 6, 10, 9, 1, 8, 1, 7, 3, 6, 1, 0, 11, 4, 3, 9], [7, 8, 0, 0,
3, 9, 0, 2, 0, 0, 7, 10, 10, 6, 0, 3, 8, 10, 10, 10, 6, 8, 5, 5, 1, 11, 0, 3, 9, 11, 11, 8, 1, 11, 11, 11, 11,
1, 11, 11, 0, 11, 7, 11, 6, 11, 7, 6, 3, 3, 10, 2, 6, 9, 10, 11, 1, 4, 7, 5], [5, 4, 1, 3, 3, 8, 11, 9, 6, 2, 4,
6, 6, 2, 7, 4, 0, 2, 9, 8, 8, 8, 11, 1, 11, 6, 3, 3, 1, 5, 9, 11, 2, 11, 1, 6, 7, 7, 5, 5, 3, 1, 0, 4, 4, 11, 10,
10, 5, 11, 8, 9, 5, 11, 2, 4, 7, 8, 3, 2], [0, 2, 10, 4, 11, 2, 11, 11, 7, 1, 0, 0, 4, 1, 5, 10, 11, 4, 8, 5, 9,
1, 4, 4, 11, 2, 11, 8, 1, 5, 8, 5, 11, 1, 7, 8, 1, 7, 1, 9, 7, 11, 1, 10, 8, 9, 1, 0, 11, 4, 6, 5, 11, 3, 3, 1,
1, 2, 6, 10], [6, 6, 3, 11, 9, 11, 7, 11, 10, 9, 11, 2, 11, 9, 2, 7, 7, 10, 6, 3, 11, 4, 11, 3, 8, 10, 4, 1,
11, 10, 6, 10, 9, 11, 11, 7, 2, 4, 0, 9, 1, 9, 1, 7, 5, 7, 7, 5, 8, 11, 10, 11, 11, 6, 11, 11, 11, 5, 10, 5],
[10, 4, 6, 1, 5, 3, 3, 9, 10, 5, 7, 11, 11, 10, 9, 5, 8, 7, 6, 11, 6, 1, 4, 4, 11, 4, 10, 6, 7, 0, 1, 5, 4, 6,
11, 11, 1, 1, 11, 11, 6, 6, 6, 1, 6, 10, 0, 6, 10, 11, 7, 0, 0, 9, 5, 2, 0, 11, 5, 11], [1, 8, 11, 6, 7, 11, 8,
0, 0, 11, 10, 11, 11, 7, 8, 4, 1, 6, 3, 1, 0, 3, 9, 8, 9, 0, 2, 3, 7, 1, 1, 2, 0, 10, 8, 9, 4, 4, 4, 3, 6, 8, 5,
1, 5, 5, 11, 0, 10, 1, 7, 6, 8, 0, 11, 0, 10, 10, 5, 1], [9, 5, 4, 2, 8, 6, 9, 3, 11, 6, 11, 0, 8, 6, 9, 4, 11,
8, 6, 4, 8, 1, 7, 11, 10, 4, 7, 3, 1, 0, 11, 11, 10, 11, 11, 9, 9, 3, 11, 3, 8, 4, 11, 8, 5, 7, 9, 3, 10, 0, 0,
11, 11, 2, 2, 5, 2, 9, 6, 3], [7, 11, 5, 11, 11, 10, 4, 11, 10, 2, 7, 1, 7, 1, 1, 0, 2, 2, 4, 0, 7, 4, 5, 6, 0,
0, 2, 7, 2, 11, 3, 0, 11, 9, 4, 10, 10, 1, 7, 11, 3, 9, 7, 11, 11, 10, 11, 11, 2, 0, 3, 9, 6, 7, 10, 2, 6, 11,
6, 10], [8, 11, 5, 5, 8, 9, 11, 2, 9, 11, 6, 1, 11, 10, 10, 10, 5, 0, 3, 2, 7, 7, 11, 8, 11, 11, 11, 2, 11, 5,
7, 4, 6, 0, 6, 11, 6, 11, 10, 8, 2, 6, 1, 5, 8, 1, 8, 10, 5, 8, 10, 8, 4, 3, 11, 9, 1, 2, 1, 11], [1, 11, 4, 0,
8, 1, 4, 7, 7, 11, 0, 11, 1, 5, 3, 6, 9, 9, 11, 5, 5, 4, 1, 3, 11, 9, 3, 5, 8, 4, 0, 4, 0, 10, 9, 1, 2, 3, 9, 11,
2, 1, 1, 3, 0, 0, 6, 2, 9, 2, 1, 0, 11, 1, 6, 0, 4, 11, 10, 3], [5, 8, 4, 4, 8, 10, 0, 5, 7, 0, 11, 4, 1, 9, 4, 8,
1, 9, 6, 8, 6, 2, 11, 9, 4, 9, 11, 11, 4, 4, 2, 11, 1, 8, 5, 2, 2, 5, 11, 7, 1, 0, 4, 0, 11, 4, 0, 0, 5, 10, 3,
9, 4, 7, 3, 11, 9, 1, 3, 4], [2, 2, 8, 7, 5, 10, 9, 11, 5, 3, 11, 9, 11, 9, 8, 11, 7, 5, 8, 11, 11, 3, 4, 3, 1,
2, 11, 7, 10, 10, 11, 9, 9, 7, 11, 7, 9, 5, 7, 7, 6, 10, 0, 3, 11, 9, 5, 0, 4, 3, 0, 4, 6, 7, 7, 1, 11, 8, 2,
9], [0, 11, 3, 11, 11, 9, 5, 10, 1, 3, 8, 1, 4, 10, 1, 1, 5, 6, 2, 4, 7, 10, 6, 11, 8, 7, 1, 3, 2, 9, 2, 7, 0, 4,
11, 2, 3, 3, 10, 10, 11, 7, 3, 4, 8, 11, 5, 6, 4, 9, 8, 0, 3, 0, 6, 11, 1, 6, 5, 10], [10, 8, 9, 5, 9, 7, 1, 1,
0, 3, 8, 2, 10, 10, 10, 9, 10, 7, 11, 2, 3, 2, 6, 10, 1, 1, 8, 5, 11, 4, 1, 4, 11, 10, 0, 4, 8, 5, 5, 10, 7,
11, 4, 7, 1, 11, 6, 3, 6, 4, 6, 9, 2, 3, 7, 7, 8, 10, 11, 6], [5, 3, 4, 11, 10, 7, 5, 5, 1, 2, 7, 10, 7, 9, 1, 7,
7, 4, 6, 10, 2, 0, 5, 9, 11, 2, 0, 3, 4, 10, 8, 11, 11, 2, 1, 4, 9, 1, 0, 8, 1, 5, 5, 9, 6, 2, 9, 9, 3, 2, 11,
11, 10, 11, 5, 9, 11, 6, 11, 11], [10, 2, 6, 11, 6, 9, 2, 4, 5, 2, 11, 8, 11, 11, 1, 9, 4, 7, 2, 10, 8, 8, 5,
11, 1, 11, 9, 3, 5, 11, 2, 11, 7, 11, 0, 8, 3, 4, 5, 10, 6, 3, 10, 8, 9, 5, 2, 5, 11, 4, 8, 11, 11, 6, 3, 10,
2, 10, 2, 9], [11, 0, 11, 11, 11, 11, 6, 1, 11, 1, 1, 11, 11, 11, 5, 6, 11, 3, 9, 5, 4, 11, 9, 11, 10, 9, 1,
5, 7, 4, 2, 2, 8, 5, 10, 11, 0, 9, 3, 9, 3, 0, 0, 1, 5, 7, 9, 1, 5, 11, 9, 11, 11, 0, 4, 10, 11, 0, 11, 7], [0,
4, 6, 11, 4, 9, 5, 5, 9, 3, 11, 3, 6, 8, 11, 4, 1, 3, 2, 2, 11, 1, 10, 11, 0, 11, 11, 6, 1, 1, 1, 1, 11, 0, 4,
11, 5, 6, 8, 5, 11, 11, 9, 9, 11, 11, 6, 4, 11, 2, 1, 6, 3, 10, 0, 0, 5, 11, 1, 11], [7, 5, 11, 7, 11, 11, 9,
11, 9, 10, 8, 2, 4, 11, 1, 2, 10, 9, 8, 3, 9, 5, 7, 7, 0, 10, 11, 2, 4, 4, 4, 9, 5, 10, 8, 6, 11, 6, 6, 10, 8,
1, 8, 11, 1, 8, 4, 7, 11, 6, 4, 4, 11, 7, 6, 7, 11, 4, 11, 9]]

red = (255,0,0)
green = (0,255,0)
blue = (0,255,0)
yellow = (255,255,0)
white = (255,255,255)
black = (0,0,0)
purple = (128,0,128)
orange = (255,165,0)
grey = (128,128,128)
```

```
turquoise = (64,224,208)
color1 = (252, 252, 247)
color2 = (252, 252, 235)
color3 = (250, 250, 222)
color4 = (247, 247, 215)
color5 = (250, 250, 205)
color6 = (250, 250, 195)
color7 = (247, 247, 188)
color8 = (250, 250, 177)
color9 = (252, 252, 169)
color10 = (252, 252, 159)
class Node:
    def __init__(self, row, col ,width, total_rows):
        self.row = row
        self.col = col
        self.x = row*width
        self.y = col*width
        self.color = white
        self.neighbors = []
        self.width = width
        self.total_rows = total_rows

    def get_pos(self):
        return self.row, self.col

    def is_closed(self):
        return self.color == red

    def is_open(self):
        return self.color == green

    def is_barrier(self):
        return self.color == black

    def is_start(self):
        return self.color == orange

    def is_end(self):
        return self.color == turquoise

    def is_speed1(self):
        if not self.is_barrier():
            return self.color == color1
    def is_speed2(self):
        if not self.is_barrier():
            return self.color == color2
```

```
def is_speed3(self):
    if not self.is_barrier():
        return self.color == color3
def is_speed4(self):
    if not self.is_barrier():
        return self.color == color4
def is_speed5(self):
    if not self.is_barrier():
        return self.color == color5
def is_speed6(self):
    if not self.is_barrier():
        return self.color == color6
def is_speed7(self):
    if not self.is_barrier():
        return self.color == color7
def is_speed8(self):
    if not self.is_barrier():
        return self.color == color8
def is_speed9(self):
    if not self.is_barrier():
        return self.color == color9
def is_speed10(self):
    if not self.is_barrier():
        return self.color == color10

def reset(self):
    self.color = white

def make_closed(self):
    self.color = red

def make_open(self):
    self.color = green

def make_barrier(self):
    self.color = black

def make_start(self):
    self.color = orange

def make_end(self):
    self.color = turquoise

def make_path(self):
    self.color = purple
```

```
def make_speed1(self):
    if not self.is_barrier():
        self.color = color1

def make_speed2(self):
    if not self.is_barrier():
        self.color = color2

def make_speed3(self):
    if not self.is_barrier():
        self.color = color3

def make_speed4(self):
    if not self.is_barrier():
        self.color = color4

def make_speed5(self):
    if not self.is_barrier():
        self.color = color5

def make_speed6(self):
    if not self.is_barrier():
        self.color = color6

def make_speed7(self):
    if not self.is_barrier():
        self.color = color7

def make_speed8(self):
    if not self.is_barrier():
        self.color = color8

def make_speed9(self):
    if not self.is_barrier():
        self.color = color9

def make_speed10(self):
    if not self.is_barrier():
        self.color = color10


def draw(self,win):
    pygame.draw.rect(win,self.color,(self.x,self.y,self.width,self.width))
```

```
    def update_neighbors(self,grid):
        self.neighbors = []
        if self.row < self.total_rows - 1 and not grid[self.row + 1][self.col].is_barrier():  # DOWN
            self.neighbors.append(grid[self.row + 1][self.col])

        if self.row > 0 and not grid[self.row - 1][self.col].is_barrier():  # UP
            self.neighbors.append(grid[self.row - 1][self.col])

        if self.col < self.total_rows - 1 and not grid[self.row][self.col + 1].is_barrier() : # RIGHT
            self.neighbors.append(grid[self.row][self.col + 1])

        if self.col > 0 and not grid[self.row][self.col - 1].is_barrier():  # LEFT
            self.neighbors.append(grid[self.row][self.col - 1])

        if len(self.neighbors) == 0:
            return False



    def __lt__(self,other):
        return False


def h(p1,p2):
    x1,y1 = p1
    x2,y2 = p2
    return 1.01*(abs(x1-x2) +abs(y1-y2))

def reconstruct_path(came_from, current, draw):
    while current in came_from:
        current = came_from[current]
        current.make_path()
        draw()




def RandomWalk(draw,grid,start,end):
    start.reset()
    visited = [start]
    queue = [start]
    came_from={}
    while len(queue) != 0:
        if len(queue) == 1:
            current = queue.pop(len(queue)-1)
        else:
```

```
            current = queue.pop(random.randint(0,len(queue)-1))
        current.reset()
        if current == end:
            reconstruct_path(came_from,end,draw)
            end.make_end()
            start.make_start()
            return True
        Temp_Neighbors = current.neighbors
        for x in Temp_Neighbors:
            if x not in queue and x not in visited:
                came_from[x] = current
                visited.append(x)
                queue.append(x)
                x.make_open()
        draw()
        if current != start:
            current.make_closed()




def Astar(draw,grid,start,end):
    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    came_from = {}
    g_score = {spot: float("inf") for row in grid for spot in row}
    g_score[start] = 0
    f_score = {spot: float("inf") for row in grid for spot in row}
    f_score[start] = h(start.get_pos(),end.get_pos())
    isRun = True
    open_set_hash = {start}

    while not open_set.empty() and isRun == True:
        for event in pygame.event.get():
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_t:
                    isRun = False
        current = open_set.get()[2]
        open_set_hash.remove(current)

        if current == end:
            reconstruct_path(came_from,end,draw)
            end.make_end()
            start.make_start()
```

```
            return True

        for neighbor in current.neighbors:
            if neighbor.is_speed1():
                temp_g_score = g_score[current] + 1
            elif neighbor.is_speed2():
                temp_g_score = g_score[current] + 2
            elif neighbor.is_speed3():
                temp_g_score = g_score[current] + 3
            elif neighbor.is_speed4():
                temp_g_score = g_score[current] + 4
            elif neighbor.is_speed5():
                temp_g_score = g_score[current] + 5
            elif neighbor.is_speed6():
                temp_g_score = g_score[current] + 6
            elif neighbor.is_speed7():
                temp_g_score = g_score[current] + 7
            elif neighbor.is_speed8():
                temp_g_score = g_score[current] + 8
            elif neighbor.is_speed9():
                temp_g_score = g_score[current] + 9
            elif neighbor.is_speed10():
                temp_g_score = g_score[current] + 10
            else:
                temp_g_score = g_score[current] + 1
            if temp_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = temp_g_score
                f_score[neighbor] =  temp_g_score +h(neighbor.get_pos(),end.get_pos())

                if neighbor not in open_set_hash:
                    count += 1
                    open_set.put((f_score[neighbor],count,neighbor))
                    open_set_hash.add(neighbor)
                    neighbor.make_open()
        draw()
        if current != start:
            current.make_closed()
    return None

def DFS(draw,grid,start,end):
    queue = [start]
    visited = [start]
    came_from = {}
    isRun = True
    while len(queue) != 0 and isRun == True:
```

```
        for event in pygame.event.get():
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_t:
                    isRun = False
        current = queue.pop(len(queue)-1)
        if current == end:
            reconstruct_path(came_from,current,draw)
            end.make_end()
            return True
        Temp_Neighbors = current.neighbors
        for neighbor in Temp_Neighbors:
            end.make_end()
            if neighbor == end:
                came_from[neighbor] = current
                reconstruct_path(came_from,neighbor,draw)
                end.make_end()
                start.make_start()
                return True
            if neighbor not in queue and neighbor not in visited:
                visited.append(neighbor)
                queue.append(neighbor)
                came_from[neighbor] = current
                neighbor.make_open()
                continue

        if current != start:
            visited.append(current)
            current.make_closed()
        draw()

def genAlg3000Map(draw,grid,start,end):
    population = "353 609 592  25  21  97  55  64 369  20 848 637 966 594  32 327  63 809 833
46 316  46 769  85 792 448  94 992  12 858  99  33 792  84  66 323 66  93 592 839  24 921  99
600 978 954  31 522  49 585  81 826 456  44 626 642   1  30  69 690 298 890  16 540  99 120
670 767 477 173 814  27 41   3 109 416  53 906 370  25 161 130 603 475 988 192 513  96 454
47 691  66 612 601 413  48  30  20 341 566"
    population = re.sub('\\s+', ' ', population)
    population = list(population.split(" "))
    for i in range(0, len(population)):
        population[i] = int(population[i])
    populationCopy = population.copy()
    queue = [start]
    visited = [start]
    came_from = {}
    isRun = True
    while len(queue) != 0 and isRun == True:
```

```
        for event in pygame.event.get():
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_t:
                    isRun = False

        if len(population) == 0:
            population = populationCopy.copy()
        index = population.pop()
        if index >= len(queue):
            current = queue.pop((index % len(queue))-1)
        else:
            current = queue.pop(index)
        if current == end:
            reconstruct_path(came_from,current,draw)
            end.make_end()
            return True
        Temp_Neighbors = current.neighbors
        for neighbor in Temp_Neighbors:
            end.make_end()
            if neighbor == end:
                came_from[neighbor] = current
                reconstruct_path(came_from,neighbor,draw)
                end.make_end()
                start.make_start()
                return True
            if neighbor not in queue and neighbor not in visited:
                visited.append(neighbor)
                queue.append(neighbor)
                came_from[neighbor] = current
                neighbor.make_open()
                continue

        if current != start:
            visited.append(current)
            current.make_closed()
        draw()

def genAlg3000SameMap(draw,grid,start,end):
    population = "100 237 751 388 915  48 596 878 947 937 937 267 956 612 813 321 591 358
495 696 491  89 258 267 530 597 364 126 761 897 330 590 906 457 689 884 159  16 383 506
722 631 290  13 411  58 356 769 794 397 575 241 422 527 769 950 912 482 407 184 341  26
661 773 592 900 686 770 954 872 625 918 936 741  71 913 886  28 650  95 240 200 933 136
677 851 305 306 905 955 61 906  81 819 259 394 632 179 855 478"
    population = re.sub('\\s+', ' ', population)
    population = list(population.split(" "))
```

```
for i in range(0, len(population)):
    population[i] = int(population[i])
populationCopy = population.copy()
queue = [start]
visited = [start]
came_from = {}
isRun = True
while len(queue) != 0 and isRun == True:

    for event in pygame.event.get():
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_t:
                isRun = False

    if len(population) == 0:
        population = populationCopy.copy()
    index = population.pop()
    if index >= len(queue):
        current = queue.pop((index % len(queue))-1)
    else:
        current = queue.pop(index)
    if current == end:
        reconstruct_path(came_from,current,draw)
        end.make_end()
        return True
    Temp_Neighbors = current.neighbors
    for neighbor in Temp_Neighbors:
        end.make_end()
        if neighbor == end:
            came_from[neighbor] = current
            reconstruct_path(came_from,neighbor,draw)
            end.make_end()
            start.make_start()
            return True
        if neighbor not in queue and neighbor not in visited:
            visited.append(neighbor)
            queue.append(neighbor)
            came_from[neighbor] = current
            neighbor.make_open()
            continue

    if current != start:
        visited.append(current)
        current.make_closed()
    draw()
```

```
def genAlg6000SameMap(draw,grid,start,end):
    population = "375 476 280 636 666 428  65 625 436  28 782 629 586 956 130  99 340   2 931
675 572 151 614 958 904 247 842 299 486 222 106  39 965 555 933 439 250 386 812 656  86
116 107 966 596 467 646  17 911 734 613 489 114 266 369  17 777 914 450 529 303 156 188
921 432  88 583 937 885  66 413 769 512 384 555  31 338 431 221 563 475 380 326 560 474
139 381 816 741 406 480 347 486 560 125   3 373 338  86 923"
    population = re.sub('\\s+', ' ', population)
    population = list(population.split(" "))
    for i in range(0, len(population)):
        population[i] = int(population[i])
    populationCopy = population.copy()
    queue = [start]
    visited = [start]
    came_from = {}
    isRun = True
    while len(queue) != 0 and isRun == True:

        for event in pygame.event.get():
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_t:
                    isRun = False

        if len(population) == 0:
            population = populationCopy.copy()
        index = population.pop()
        if index >= len(queue):
            current = queue.pop((index % len(queue))-1)
        else:
            current = queue.pop(index)
        if current == end:
            reconstruct_path(came_from,current,draw)
            end.make_end()
            return True
        Temp_Neighbors = current.neighbors
        for neighbor in Temp_Neighbors:
            end.make_end()
            if neighbor == end:
                came_from[neighbor] = current
                reconstruct_path(came_from,neighbor,draw)
                end.make_end()
                start.make_start()
                return True
            if neighbor not in queue and neighbor not in visited:
                visited.append(neighbor)
                queue.append(neighbor)
                came_from[neighbor] = current
```

```
            neighbor.make_open()
            continue

        if current != start:
            visited.append(current)
            current.make_closed()
        draw()

def genAlg6000Map(draw,grid,start,end):
    population = "33   2 356  70  48 677 426 745  14 478 413 423 443 655 122 586  73 996 650
69 530   4 750 319 382 488 324 321 423  79  59 961 182 657 272  80 62  89  26 718   7 620 619
66  45 179 215 248 363  26 776 364   8  54 3 939 806 558  20 674 172 876  81  46  64 182 546
81 191  36  34  57 663 186 306  83 640 590 515  82 168 312 782 617  68 668   0 156 836 470
513 155 296 332 567  64  78   3 204 541"
    population = re.sub('\\s+', ' ', population)
    population = list(population.split(" "))
    for i in range(0, len(population)):
        population[i] = int(population[i])
    populationCopy = population.copy()
    queue = [start]
    visited = [start]
    came_from = {}
    isRun = True
    while len(queue) != 0 and isRun == True:

        for event in pygame.event.get():
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_t:
                    isRun = False

        if len(population) == 0:
            population = populationCopy.copy()
        index = population.pop()
        if index >= len(queue):
            current = queue.pop((index % len(queue))-1)
        else:
            current = queue.pop(index)
        if current == end:
            reconstruct_path(came_from,current,draw)
            end.make_end()
            return True
        Temp_Neighbors = current.neighbors
        for neighbor in Temp_Neighbors:
            end.make_end()
            if neighbor == end:
                came_from[neighbor] = current
```

```
            reconstruct_path(came_from,neighbor,draw)
            end.make_end()
            start.make_start()
            return True
        if neighbor not in queue and neighbor not in visited:
            visited.append(neighbor)
            queue.append(neighbor)
            came_from[neighbor] = current
            neighbor.make_open()
            continue

    if current != start:
        visited.append(current)
        current.make_closed()
    draw()

def BFS(draw,grid,start,end):
    queue = [start]
    visited = [start]
    came_from = {}
    isRun = True
    while len(queue) != 0 and isRun == True:
        for event in pygame.event.get():
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_t:
                    isRun = False
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit
        current = queue.pop(0)
        if current == end:
            reconstruct_path(came_from,current,draw)
            end.make_end()
            start.make_start()
            return True
        Temp_Neighbors = current.neighbors
        for neighbor in Temp_Neighbors:
            end.make_end()
            if neighbor == end:
                came_from[neighbor] = current
                reconstruct_path(came_from,neighbor,draw)
                end.make_end()
                start.make_start()
                return True
            if neighbor not in queue and neighbor not in visited:
                visited.append(neighbor)
```

```
            queue.append(neighbor)
            came_from[neighbor] = current
            neighbor.make_open()
            continue

        if current != start:
            visited.append(current)
            current.make_closed()
        draw()

def Djisktra(draw,grid,start,end):
    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    came_from = {}
    g_score = {spot: float("inf") for row in grid for spot in row}
    g_score[start] = 0
    isRun = True

    open_set_hash = {start}

    while not open_set.empty() and isRun == True:
        for event in pygame.event.get():
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_t:
                    isRun = False

        current = open_set.get()[2]
        open_set_hash.remove(current)

        if current == end:
            reconstruct_path(came_from,end,draw)
            end.make_end()
            start.make_start()
            return True

        for neighbor in current.neighbors:
            if neighbor.is_speed1():
                temp_g_score = g_score[current] + 1
            elif neighbor.is_speed2():
                temp_g_score = g_score[current] + 2
            elif neighbor.is_speed3():
                temp_g_score = g_score[current] + 3
            elif neighbor.is_speed4():
                temp_g_score = g_score[current] + 4
            elif neighbor.is_speed5():
```

```
            temp_g_score = g_score[current] + 5
        elif neighbor.is_speed6():
            temp_g_score = g_score[current] + 6
        elif neighbor.is_speed7():
            temp_g_score = g_score[current] + 7
        elif neighbor.is_speed8():
            temp_g_score = g_score[current] + 8
        elif neighbor.is_speed9():
            temp_g_score = g_score[current] + 9
        elif neighbor.is_speed10():
            temp_g_score = g_score[current] + 10
        else:
            temp_g_score = g_score[current] + 1

        if temp_g_score < g_score[neighbor]:
            came_from[neighbor] = current
            g_score[neighbor] = temp_g_score


            if neighbor not in open_set_hash:
                count += 1
                open_set.put((g_score[neighbor],count,neighbor))
                open_set_hash.add(neighbor)
                neighbor.make_open()
    draw()
    if current != start:
        current.make_closed()

return None

def make_grid(rows,width):
    grid = []
    gap = width//rows
    for i in range(rows):
        grid.append([])
        for j in range(rows):
            spot= Node(i,j,gap,rows)
            grid[i].append(spot)
    return grid

def draw_grid(win,rows,width):
    gap = width//rows
    for i in range(rows):
        pygame.draw.line(win,grey,(0,i*gap),(width,i*gap))
        for j in range(rows):
            pygame.draw.line(win,grey,(j*gap,0),(j*gap,width))
```

```
def draw(win,grid,rows,width):
    win.fill(white)

    for row in grid:
        for spot in row:
            spot.draw(win)

    draw_grid(win,rows,width)
    pygame.display.update()

def get_clicked_pos(pos,ROWS,width):
    gap = width//ROWS
    y,x = pos
    row = y//gap
    col = x//gap
    return row,col


def main(win,width, choice, savedGrid):
    WEIGHT = (10)
    ROWS = 60
    grid = make_grid(ROWS,width)

    gap = width//ROWS
    start = grid[random.randint(1,49)][random.randint(1,49)]
    end = grid[random.randint(1,49)][random.randint(1,49)]
    while start == end:
        end = grid[random.randint(1,49)][random.randint(1,49)]
        start = grid[random.randint(1,49)][random.randint(1,49)]
    start.make_start()
    end.make_end()
    run = True
    started = False

    while run:
        draw(win,grid,ROWS,width)
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False
            if started:
                continue
            if pygame.mouse.get_pressed()[0]:  #Left mouse button
                pos = pygame.mouse.get_pos()
                row,col = get_clicked_pos(pos,ROWS,width)
                if row >= 50 or col >= 50:
```

```
            continue
        spot = grid[row][col]
        if spot != end and spot != start:
            spot.make_barrier()

    elif pygame.mouse.get_pressed()[2]:  # Right mouse button
        pos = pygame.mouse.get_pos()
        row,col = get_clicked_pos(pos,ROWS,width)
        if row >= 50 or col >= 50:
            continue
        spot = grid[row][col]
        spot.reset()
        if spot == start:
            start = None
        elif spot == end:
            end = None

    if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_SPACE and not started:
            for row in grid:
                for spot in row:
                    spot.update_neighbors(grid)
            if choice == 0:
                Astar(lambda: draw(win,grid,ROWS,width), grid, start, end)
            elif choice == 1:
                Djisktra(lambda:  draw(win,grid,ROWS,width), grid, start, end)
            elif choice == 2:
                RandomWalk(lambda: draw(win,grid,ROWS,width), grid, start, end)
            elif choice == 3:
                BFS(lambda: draw(win,grid,ROWS,width), grid, start, end)
            elif choice ==4:
                DFS(lambda: draw(win,grid,ROWS,width), grid, start, end)
            elif choice == 5:
                genAlg3000Map(lambda: draw(win,grid,ROWS,width), grid, start, end)
            elif choice == 6:
                genAlg6000Map(lambda: draw(win,grid,ROWS,width), grid, start, end)
            elif choice == 7:
                genAlg3000SameMap(lambda:  draw(win,grid,ROWS,width), grid, start, end)
            elif choice == 8:
                genAlg6000SameMap(lambda:  draw(win,grid,ROWS,width), grid, start, end)

        elif event.key == pygame.K_g:
            grid = make_grid(ROWS,width)
            start = grid[random.randint(1,49)][random.randint(1,49)]
            end = grid[random.randint(1,49)][random.randint(1,49)]
            while start == end:
```

```
            end = grid[random.randint(1,49)][random.randint(1,49)]
            start = grid[random.randint(1,49)][random.randint(1,49)]
        for x in range (ROWS):
            for y in range (ROWS):
                spot = grid[x][y]
                if random.random() <0.2 and spot != start and spot != end:
                    spot.make_barrier()
        row = start.row
        col = start.col
        for x in range(row-3,row+4):
            for y in range(col-3,col+4):
                if x<0 or x >ROWS-1 or y <0 or y>ROWS-1:
                    continue
                spot = grid[x][y]
                spot.reset()
        row = end.row
        col = end.col
        for x in range(row-3,row+3):
            for y in range(col-3,col+3):
                if x<0 or x >ROWS-1 or y <0 or y>ROWS-1:
                    continue
                spot = grid[x][y]
                spot.reset()
        start.make_start()
        end.make_end()

    elif event.key == pygame.K_c:
        grid = make_grid(ROWS,width)
        start = grid[random.randint(1,49)][random.randint(1,49)]
        end = grid[random.randint(1,49)][random.randint(1,49)]
        while start == end:
            end = grid[random.randint(1,49)][random.randint(1,49)]
            start = grid[random.randint(1,49)][random.randint(1,49)]
        for x in range (ROWS):
            for y in range (ROWS):
                spot = grid[x][y]
                if random.random() <0.2 and spot != start and spot != end:
                    spot.make_barrier()
                Temp_Random = random.randint(0,10)
                if not spot.is_barrier():
                    if Temp_Random == 0:
                        spot.make_speed1()
                    if Temp_Random == 1:
                        spot.make_speed2()
                    if Temp_Random == 2:
                        spot.make_speed3()
```

```
            if Temp_Random == 3:
                spot.make_speed4()
            if Temp_Random == 4:
                spot.make_speed5()
            if Temp_Random == 5:
                spot.make_speed6()
            if Temp_Random == 6:
                spot.make_speed7()
            if Temp_Random == 7:
                spot.make_speed8()
            if Temp_Random == 8:
                spot.make_speed9()
            if Temp_Random == 9:
                spot.make_speed10()
saveGrid = []
for x in range(ROWS):
    saveGrid.append([])
    for y in range(ROWS):
        spot = grid[x][y]
        if spot.is_barrier():
            saveGrid[x].append(11)
        elif spot.is_speed1():
            saveGrid[x].append(1)
        elif spot.is_speed2():
            saveGrid[x].append(2)
        elif spot.is_speed3():
            saveGrid[x].append(3)
        elif spot.is_speed4():
            saveGrid[x].append(4)
        elif spot.is_speed5():
            saveGrid[x].append(5)
        elif spot.is_speed6():
            saveGrid[x].append(6)
        elif spot.is_speed7():
            saveGrid[x].append(7)
        elif spot.is_speed8():
            saveGrid[x].append(8)
        elif spot.is_speed9():
            saveGrid[x].append(9)
        elif spot.is_speed10():
            saveGrid[x].append(10)
        else:
            saveGrid[x].append(0)
print(saveGrid)
row = start.row
col = start.col
```

```
            for x in range(row-3,row+4):
                for y in range(col-3,col+4):
                    if x<0 or x >ROWS-1 or y <0 or y>ROWS-1:
                        continue
                    spot = grid[x][y]
                    spot.reset()
            row = end.row
            col = end.col
            for x in range(row-3,row+3):
                for y in range(col-3,col+3):
                    if x<0 or x >ROWS-1 or y <0 or y>ROWS-1:
                        continue
                    spot = grid[x][y]
                    spot.reset()
            start.make_start()
            end.make_end()

        elif event.key == pygame.K_v:
            grid = make_grid(ROWS,width)
            end = grid[random.randint(1,49)][random.randint(1,49)]
            start = grid[random.randint(1,49)][random.randint(1,49)]
            while start == end:
                end = grid[random.randint(1,49)][random.randint(1,49)]
                start = grid[random.randint(1,49)][random.randint(1,49)]
            for x in range(ROWS):
                for y in range(ROWS):
                    spot = grid[x][y]
                    if savedGrid[x-1][y-1] == 11:
                        spot.make_barrier()
            row = start.row
            col = start.col
            for x in range(row-3,row+4):
                for y in range(col-3,col+4):
                    if x<0 or x >ROWS-1 or y <0 or y>ROWS-1:
                        continue
                    spot = grid[x][y]
                    spot.reset()
            row = end.row
            col = end.col
            for x in range(row-3,row+3):
                for y in range(col-3,col+3):
                    if x<0 or x >ROWS-1 or y <0 or y>ROWS-1:
                        continue
                    spot = grid[x][y]
                    spot.reset()
            start.make_start()
```

```
                end.make_end()
            elif event.key == pygame.K_f:
                grid = make_grid(ROWS,width)
                start = grid[random.randint(1,49)][random.randint(1,49)]
                end = grid[random.randint(1,49)][random.randint(1,49)]
                while start == end:
                    end = grid[random.randint(1,49)][random.randint(1,49)]
                    start = grid[random.randint(1,49)][random.randint(1,49)]
                start.make_start()
                end.make_end()



    pygame.quit()

choice = 0
# 0 - Astar
# 1 - Djisktra
# 2 - RandomWalk
# 3 - BFS
# 4 - DFS
# 5 - genAlg3000Map
# 6 - genAlg6000Map
# 7 - genAlg3000SameMap
# 8 - genAlg6000SameMap
main(win,width, choice, savedGrid)
```