Shortest Path Finding in a Grid world

CS7IS2 Project (2019/2020)

Ashwin Sundareswaran R, Chaudhary Guroosh Gabriel Singh, Kavya Bhadre Gowda, Shubhanghi Kukreti

ramasuba@tcd.ie, csingh@tcd.ie, bhadregk@tcd.ie, kukretis@tcd.ie

Abstract. Path finding is a crucial problem in today's world which needs everyone's focus a nd attention. Various path finding algorithms are used to solve the problem of finding a route from the starting node to the target node. Hence in this research work various algorithms are evaluated and compared to find the best suitable path for traversal in a simulated grid world environment with a single agent.

Keywords: Shortest path finding, grid world, A-Star, Genetic Algorithm (GA), Reinforcement Learning

1 Introduction

The Shortest path finding problem has been solved by many researchers using many State-of-the-art algorithms like Breadth-First-Search, Depth-First-Search, A* etc. In this paper we have used GA, ACO, Q-RL and to search the destinations which are fast, optimal and credible and compared with the standard A* algorithm which is computationally expensive in the gaming world.

States of each node and the paths matters in solving various traversing problems like transportation, networks etc. Nodes and Paths vary throughout the states of traversal and the optimal solution for this shortest path solving problem involves paths which do not have loops, no irregular nodes, high speed and less time consuming with less distance of travel.

Implemented Gridworld Path Planning algorithms for dynamic grids using different techniques. The algorithms implemented consisted of both model based and model free. The motivation for implementing path planning algorithms was to compare them based on time, the final path and adaptiveness of the algorithm. This analysis will help to observe findings of path planning and can be extended to a real world scenario, for instance, a network of roads.

The environment is a grid world with a start position, end position and obstacles. The obstacles can be either pre-defined or randomly placed such that there isn't an obvious direct path between the start and end positions. In our implementation the agent has maximum four possible moves at any position, left, right, up and down. The agent will try to reach the destination with as few moves as possible.

For the experiment we have randomised the grid with 20% obstacles. Less obstacles gave straight answers and More obstacles also gave straight answers since the grid was too dense and there were only 1 or 2 possible paths from the start to the end. The start was at (0,0) and the destination node was at (size-1, size-1).

2 Related Work

Possible approaches to find optimal paths are Breadth First Search(BFS), Depth First Search(DFS), Dijkstra's algorithm, A-star. Local search techniques like hill climbing and genetic algorithms. The problem with these algorithms is that... .They are all model free algorithms and other ways to find short routes is by making a model either by Reinforcement learning or deep reinforcement learning. We will look into implementing genetic algorithms, reinforcement learning (and possibly ant colony). We have tried to improve the convergence rate in genetic by changing the crossover and mutation function stated in [2].

Chang Wook Ahn and Ramakrishna in their work on using GA for Shortest path problem and sizing of populations, have tried to solve the Shortest path problem with variable chromosomes length and genes by exchanging partial routes in cross over and selecting unique path every time in Mutation. This improved algorithm has solved the shortest path problem with a high convergence rate compared to state-of-the-art algorithms. In this the authors have specified the importance of Population size selection, cross-over technique to get the diversified route and mutation to improve the local and global convergence. This motivated us to choose Genetic algorithms to find the solution to the shortest path problem and evaluate the results with other chosen algorithms of Q learning, SARSA and A*.

In our work we have used static iterations with variable chromosomes size as after some iterations the shortest path will reach to local minimum. The General Genetic algorithm involves various stages in the solution finding problem such as Initialization of Population, Selection of best fit chromosomes, Crossover and Mutation. In our work we have tried to improve each stage to converge at a global minimal path.Yin-Hao Wang ,Tzuu-Hseng and Chih-Jui lin in their work compared the use of different reinforcement learning techniques like Q-learning, State-Action-Reward-State-Action (SARSA) in different applications and reported the comparison of the performance of these algorithms.It has been mentioned that the important part of the reinforcement learning is selecting the trade-off between exploration and exploitation techniques.SARSA is an on-policy learning technique . The agent estimates the next state and action value Q(s,a) by performing the action a in state s according to the formula as mentioned in

[12]:

$Q(s,a) \rightarrow Q(s,a) + learningrate[reward + discountfactor * (Q(s+1,a+1)) - Q(s,a)]$

Where Q(s,a) is the current state and action. Q(s+1,a+1) is the next state and action. Reward is the obtained reward from performing the current action on the current state.Thus it uses the previous state state and action to predict the next state and it's actions.

3 Problem Definition and Algorithm

The problem is to come up with approaches/algorithms to find the smallest paths from start to destination in a grid world environment in as little time as possible. A path p1 is better than another path p^2, if the total length of p1 (from start to end) is less than the total length of p^2 (from start to end). The length of the path will also define the number of moves by the agent, i.e. "number of moves = length of path - 1".

3.1 A-star Algorithm

A-star (A*) algorithm was implemented to find the best path in any environment. The algorithm used is defined in [1] which traverses a graph, starting from the start node, until it finds the destination node. At any node, the algorithm looks at the unvisited child nodes and marks them as open (EXPLAIN ALGORITHM). The algorithm uses an evaluation function which helps to traverse the graph optimally. The evaluation function at node n is f(n) = g(n) + h(n), where g(n) is the distance of node n from the start node h(n) is the heuristics between node n and the destination node. The Admissibility and Optimality of an A* algorithm depends on the heuristics function. If the heuristic is good (or what else do we call it? Link), the algorithm always guarantees an optimal path i.e. having the least steps from the start to destination node. The final path generated by A* algorithm will set a benchmark for other algorithms and the paths can be compared on the basis of their length.

For A* to work correctly and be efficient, the heuristics must provide as correct information as possible compared to the actual distance between the current node and the destination node. In general the heuristics should satisfy the following properties:

- The heuristic should not overestimate the actual distance.
- Compared to the actual distance, the heuristic should be as accurate as possible.

For the algorithm to be admissible and optimal, the 1st point should be satisfied. In that case the answer may not always be optimal and may give longer paths. As long as the the heuristic does not overestimate the actual distance, A* will give an optimal answer but the efficiency will still depend on the heuristics. For the algorithm to be efficient, the 2^{nd} point should be satisfied along with the 1st point. The most optimal scenario is when the heuristic function returns the actual distance between the current node and the destination node but this is not possible to evaluate since there might be obstacles in the environment, also if it would be possible to evaluate the exact distance to the destination we wouldn't need an algorithm. If the heuristic function always returns a 0 (or any other fixed constant), the algorithm proceeds forward based on the current nodes traversed from the start. This is also called Dijkstra's algorithm, which becomes a special case of A* algorithm.

In our implementation there are 4 possible moves from any node (no diagonal moves are allowed) and therefore the shortest distance between the start and destination, without any obstacles, can be calculated using the Manhattan distance.

3.2 Genetic Algorithm

Genetic algorithm (GA) is a self-adaptive search algorithm which is based on the idea of natural selection of population and genes. This algorithm is mainly focused on selecting the fittest survival which is necessary for the evolution in natural systems. In this application we have used GA to find the shortest path of travel from source node to destination node which is consistently accurate even though this itself is not the perfect solution to our path finding problem. Based on the references from the previous work we have tried to improve our GA by using Static Population Selection. The Population selection in our problem is affecting the best solution path. Random dynamic population selection provides inaccurate results hence fixing the population selection iteration count for a particular problem scenario helps the genetic algorithm to adapt quickly.

The genetic algorithm consists of population creating followed by iterations of crossover, mutation and population reduction.

- **Population Creation** The initial population creation is done by generating a list of k paths. Each path is a randomly generated path from from the start node to the destination node. This is done using a random depth first search and returning the path as soon as the destination node is reached for the first time. After this the population goes through iterations of crossover, mutation and evaluation until the best path in the population converges to a minima (short length path). The minima may be the global minima (can be verified by the A-star result) or can be a local minima. To decrease the chances of converging on a local minima, mutation is used.

- **Crossover** The crossover function is used to create new paths by taking paths from the existing population, known as the parents and creating new paths which is created by mixing sub-paths from 2 or more parents. This helps in diversification of the population. We have implemented crossover by selecting different pairs of 2 parents and finding a random node R, occurring in both the parents. If the start and end nodes of the chosen paths are S1, S2 and E1, E2 respectively then the paths in the parents can be defined as $S1 \to R \to E1$ and $S2 \to R \to E2$, since both consist of a random node R. Therefore the generated paths after crossover will be $S1 \to R \to E2$ and $S2 \to R \to E1$. Since we are keeping the parents in the population, the total size of the population increases.

- **Mutation** Mutation is used to avoid a local minima by introducing changes to a certain percentage of the population such that it deviates from being optimal. This introduces the concept of exploration in the algorithm. We are implementing mutation by selecting a certain number of paths, finding a random node R, and finding a random path from either the start node to R or from R to the destination node. So if the initial path is represented as $S \to R \to E$. After mutation it could either be $S_{new} \to R \to E$ or $S \to R \to E_{new}$ with a 50% probability, where the new random path either starts from $S_{new} toR$ or from $RtoE_{new}$. This helps in preventing the algorithm from converging to a local minima.

- **Population Reduction** Since the population size increases after crossover, the total population count has to be restricted to a limit which may or may not be equal to the initial population count. Population reduction is done at the end of each iteration by using a fitness criteria. In our implementation the fitness criteria is the length of a path, lower length means a better path. Based on this, the population is sorted and top N paths are stored and the remaining are removed.

  The iterations can be stopped either when there is no significant improvement in the previous iterations or when a certain number of iterations is reached.

3.3 Ant Colony Optimization Algorithm

Ant colony optimization algorithm (ACO) consists of multiple iterations where each iteration consists of sending a group of ants (agents) which traverse the environment (GridWorld), updating the pheromone value and evaporation of the already existing pheromone values. After each iteration the best path is stored and updated.

- **Ants Traversal** The ants traverse to find the destination node. When ants move from one node to another, they move according to the formula;

$$((\tau)^{\alpha} * (\eta)^{\beta} / \sum((\tau)^{\alpha} * (\eta)^{\beta}))$$

which returns the probability of moving to the next nodes. Tau is the pheromone level at the next nodes to which an ant can move and eta is the inverse of the heuristics of moving to the next node. At each node an ant calculates the probability of movement to the adjacent four nodes (or less than four nodes if all moves are not possible). Based on this probability an ant decides and moves forward.

- **Pheromone Update** The pheromone value at a node represents the traffic at that node by the previous ants. If more ants have traversed through a particular node in the previous iterations the τ value will be high. This concept is based on how ants locate food in real scenarios where a short route will be traversed more by ants and therefore it will have higher pheromone. To emulate this behaviour in our algorithm, after a group of ants reach the destination, the pheromone is updated on the path travelled by them. Further, to give importance to smaller paths the pheromone update is inversely proportional to the length of the path the ant travelled on.

- **Pheromone Evaporation** At the end of each iteration, the pheromone level of all nodes is reduced by some percentage, irrespective of the value or which path it belongs to.

  In the first iteration, the pheromone level at each node is a constant so at first the ants move randomly. Since shorter paths are updated by a higher pheromone value, over time the ants start taking the shorter paths. Due to evaporation new paths are given importance and the initial random paths are given less importance. Over time the algorithm converges to a minima.

3.4 Q - Learning Algorithm

Reinforcement Learning is a form of Machine Learning in which the agent learns through observation of the environment. Based on its observations of the environment, the agent takes an action and receives positive or negative rewards. The main task of the agent is to maximize its rewards and find the optimal solution to reach a goal. Q-learning is one of the simplest reinforcement learning algorithms which has been used to solve maze navigation problems. The Q-learning algorithm involves the interaction between an agent and an environment. The agent chooses an action A for a given state S which results in a reward R. When Q-learning is employed a Q-table or matrix is created which consist of values for State-Action pairs. At the beginning of the algorithm, the values in the Q table, called Q-values, are initialised as zero. But as the algorithm reiterates, the Q-values are updated after every episode. The agent uses the Q-table as a reference to decide the best action for a particular Q-value. For any given state stS, the agent must choose an action At from a set of available actions and change its state to S_{t+1} after receiving a reward rt from the environment. The agent also learns the mapping between the states S and action A defined as policy . The agent has to follow the path that maximizes its total reward. If the agent hits an obstacle or does not follow the optimal path, it is penalised by getting a negative reward. The Q-values in the Q-table are updated using the following equation:

$$Q(s_t, a_t) = Q(s_t, a_t)(1 - \propto_t (s_t, a_t)) + \propto_t (s_t, a_t) * [r_t + \mathit{\Gamma}_t max_a Q(s_{t+1}, a)]$$

where, $\propto_t$ is known as the learning rate which represents how much the algorithm prefers the old value vs the new value. Learning rate value of 0 portrays that the agent does not learn anything whereas the value of 1 indicates that only the recently acquired knowledge will be considered by the agent and the knowledge gained in the past will be lost[13].

$\mathit{\Gamma}_t$ is known as the discount factor determines the preference of the agent in choosing between the long-term high reward or choose the greedy reward; its value is between 0 and 1. Discount rates of one or more than one have proven to diverge the algorithm. As the value of $\mathit{\Gamma}$ approaches 1, the agent prefers to choose long-term reward [13].

$\mathit{\Gamma}_t$ refers to the reward at time t.

$max_a Q(s_{t+1}, a)]$ represents that the agent takes maximum of the future rewards and adds that value to the current reward.

These agents use exploration and exploitation strategies to find the next possible state.Exploitation is the way to find the next action derived on the obtained Q-value.This is an off-policy learning where the agent exploits the environment to find the path yielding the maximum reward.Exploration is the way of taking random action in the environment such that the agent tries to explore new paths to reach the destination.This helps the agent not to converge with local minima and gives way to explore new paths.This an on-policy learning method.It is always important for the agent to have trade-off between the exploitation and exploration values in order to learn the optimal solution. Epsilon-greedy algorithm is normally used to do the trade-off between these two selections.A value of epsilon is defined and a random number is generated.If the random number is greater than the epsilon value then the agent exploits and find the states based on the obtained Q values.If the number is less than epsilon, then the agent explores with random actions.

3.5 State-Action-Reward-State-Action(SARSA) Algorithm

SARSA is an on-policy reinforcement learning algorithm.The agent learns from the action performed on the environment.It predicts the next action to take based on the action performed on the current state and the reward obtained.It is unlike Q-learning where the agent looks for the maximum rewarded next-state.The agent interacts with the grid-world environment with grid size 10 *

10 and having random obstacles present between the starting point and the destination point.The agent receives a reward of -100 if it hits an obstacle in the way and a reward of +100 if it reaches the destination.The agent is trained for 1000 epochs and with the increase in the training time period,the agent learns the optimal path and follows the same to reach the destination.As the formula for SARSA has number of hyper parameters which can be modified to tune the agent to learn accordingly.The following parameter value are selected.

Table 1. Hyperparameters selection for SARSA agent.

Hyperparameters	Values
Epsilon	0.1
Learning Rate	0.1
Discount Rate	0.9

4 Experimental Results

4.1 A-Star (A*) Algorithm

Methodology: To see the effects of the heuristics on the efficiency and the optimality of the algorithm, we ran the algorithm across a fixed environment using different heuristic functions.

Table 2. Results of A* on a fixed 40x40 grid with random obstacles.

Heuristic function	Length of the final path	No. of nodes visited	No. of nodes traversed through
0 (a fixed constant)	81	1164	1164
Euclidean distance $\sqrt{(X^2+Y^2)}$	81	1086	1065
Manhattan distance (X + Y)	81	625	559
X^2+Y^2	85	170	106
2 * (X + Y)	85	210	141

X and Y are the distance between x-coordinates and y-coordinates respectively.

Result: Since diagonal movements are not allowed in our implementation, Manhattan distance between 2 nodes is an optimal and efficient heuristic function, since it is the closest we can get to the actual distance. Using underestimated heuristic functions like Euclidean distance or a constant value also give an optimal result but the efficiency is reduced and the algorithm takes longer to complete. On the other hand, using heuristic functions which overestimate the actual distance give non-optimal paths, although they may be designed to complete very fast. The results of different heuristic functions can be seen in Table I. Since we are using A* as a benchmark to compare optimality for other algorithms, we used Manhattan distance in our final implementation.

Side Observation: Adding a constant value to the heuristic function does not change the behaviour. For example a function (X + Y + 4) will give the same results as (X + Y).

4.2 Genetic Algorithm

Genetic algorithms can be evaluated using 2 criteria:

- How accurate is the result to the global optima (minima in our case).
- How fast the algorithm converges.

Test for algorithm correctness: To test the correctness, the algorithm was run multiple times using different mutation rates. Mutation rate is the percentage of the population which undergoes mutation in each iteration. The code was run for various grid sizes and the local minima was compared to the global minima derived using A* on the same grid. The values of the minima obtained after 400 iterations are plotted (smoothed) for various grid size and mutation rate in figure I. The algorithm was run 3 times for each grid size and the average was taken, therefore, for a 15x15 grid the algorithm was run on 3 different randomized grids of size 15.

Result: In “Fig 1“, it can be seen that for small environments (grid size ¡= 25), the best result is obtained when mutation rate is high (70%, 100%). As the environment size increases lower mutation rates (30% and 50%) give better results than higher mutation rates. Also, when the mutation rate is 0 i.e. no mutation, then the local minima is way off from the global minima, therefore some mutation is always required.

Discussion: It is clearly visible that mutation is necessary for the algorithm to explore and eventually find the global minima but finding the correct mutation amount can be tricky, specially for randomized environments. Even after finding an optimal mutation rate, the algorithm may not always converge to the global minima, especially for complex environments.

Test for Convergence:

genetic.png

Fig. 1. Local minima for different mutation rates across different grid sizes.

Methodology: To test the convergence, we ran the algorithm through different sized environments and for each grid size the number of iterations were calculated in which the algorithm converges. In the experiment we say that the algorithm converges if there is no improvement, in the best path, in the previous 50 iterations. The algorithm was run 10 times for each grid size and the average was taken, therefore, for a 15x15 grid the algorithm was run on 10 different randomized grids of size 15. The mutation rate was fixed to 30% and the population size after every iteration was fixed to 50.

Results: In "Fig 2", the Red coloured plot shows the actual values while the Blue coloured plot is the smoothed plot of the original values, which was implemented using a gaussian filter.

Discussion: As the size of the grid increases, the number of iterations required to reach convergence also increases. The increment in the graph is less than a linear increase, its because if the grid size increases the global minima will also increase and therefore the length of each individual in the population also increases. The algorithm works to find a longer path and therefore there is more scope for improvement after each iteration.

Genetic algorithms are also affected by other parameters like initial population count, population count after each iteration, amount of crossover, etc. Based

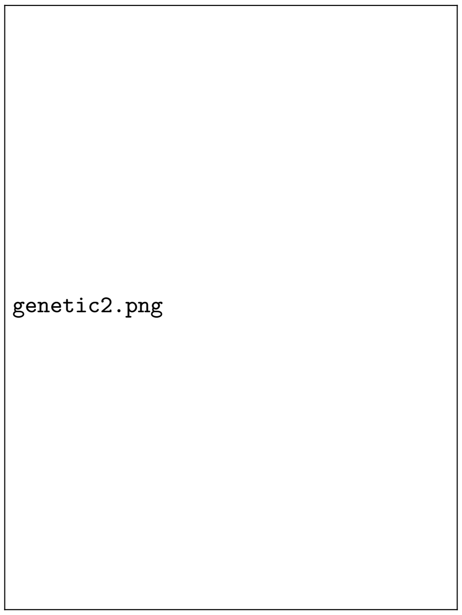

Fig. 2. Convergence rate of genetic algorithm across different grid sizes.

on the requirement of the algorithm, the parameters can be fixed to improve the quality of the answer or be more efficient. Usually there is a tradeoff between the two, increasing the population size may help in the quality of the answer but decreases the efficiency.

4.3 RL Using Q-learning algorithm

Methodology: To test the Q-learning algorithm, random obstacles were introduced in the grid. The task of the agent was to avoid the obstacles and find an optimal path from starting point to the ending point. For each grid size, 1000 episodes were run with the parameters values being learning rate as 0.1 , discount factor as 0.9 and epsilon as 0.1. If the agent hits an obstacle, it is penalised by receiving a -100 reward. But if the agent reaches the target, it receives a reward of +100. After exploring the environment for a few episodes, the agent finally learns to navigate the grid without hitting any obstacle. Soon after that, the agent finds the optimal path and the algorithm converges and the final reward stays at 100. In "Fig 3", it can be observed that for a grid size of 10*10, the rewards are negative at first indicating the agent hitting an obstacle or straying away from the end point. But as the number of episodes increases, the agent ends with the same reward indicating the convergence of the algorithm.

Additionally, the grid size was varied and the performance of the Q-learning agent was measured against the performance of A* algorithm by comparing the number of steps it takes for each algorithm to reach the target. Moreover, the

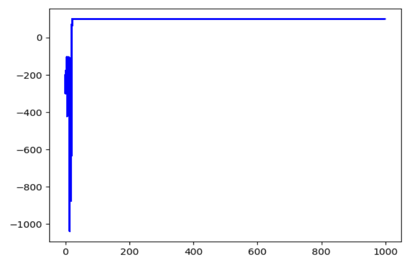

Fig. 3. Number of Episodes Vs Reward for Q-learning.

time taken by the algorithm to converge was also calculated with the increase in the grid size.

Results and Discussion: The comparison in the performance of Q-learning agent and A* , our baseline algorithm, is shown in the following table. The performance is measured based on the length of the path undertaken by an algorithm to reach the target point from the starting point. It can be observed that A* algorithm consistently provides shorter path length to reach the target as compared to Q-learning algorithm. Also, as the size of the grid increases the difference in the path lengths increases.

Table 3. Performance Comparison of A* and Q-learning algorithm.

Grid Size (M*M)	A* Algorithm (Path Length)	Q-Learning Algorithm (Path Length)
5	7	8
7	11	12
9	15	16
10	17	20
11	19	24

Q-learning has been widely used in Reinforcement Learning applications but the algorithm suffers from a few drawbacks. It required more time to reach the optimal solution. The time taken for the algorithms to reach the optimal solution is shown in Table 4.

4.4 SARSA algorithm

The SARSA agent learns the environment and converges to the output after few iterations of training.The following graph shows the comparison between the number of training iteration and reward.The agent gets morte reward as it

Table 4. Time to Converge comparison for Q-learning and A* algorithm.

Grid Size (M*M)	A* Algorithm ((Time to Converge in sec))	Q-Learning Algorithm ((Time to Converge in sec))
5	0.43	1.75
7	0.99	2.29
9	1.07	2.58
11	1.89	3.03

is getting trained.This implies that the agent learns the obstacle paths and then avoid hitting it once it knows the destination is known.It can be seen after 20 iterations the agent learns and it reaches almost the maximum rewards state by not hitting the obstacles.

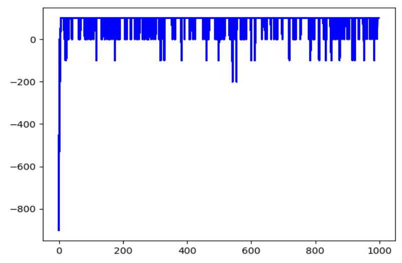

Fig. 4. Number of Episodes Vs Reward for SARSA Agent.

5 Conclusions

Provide a final discussion of the main results and conclusions of the report. Comment on the lesson learnt and possible improvements.

A standard and well formatted bibliography of papers cited in the report. For example:

References

1. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. J. Mol. Biol. 147, 195?197 (1981). doi:10.1016/0022-2836(81)90087-5
2. May, P., Ehrlich, H.-C., Steinke, T.: ZIB structure prediction pipeline: composing a complex biological workflow through web services. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 1148?1158. Springer, Heidelberg (2006). doi:10.1007/11823285_121

3. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco (1999)
4. Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C.: Grid information services for distributed resource sharing. In: 10th IEEE International Symposium on High Performance Distributed Computing, pp. 181?184. IEEE Press, New York (2001). doi:10.1109/HPDC.2001.945188
5. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The physiology of the grid: an open grid services architecture for distributed systems integration. Technical report, Global Grid Forum (2002)
6. National Center for Biotechnology Information. http://www.ncbi.nlm.nih.gov