Type of paper: Original scientific paper
Received: 10.06.2023.
Accepted: 04.07.2023.
DOI: https://doi.org/10.18485/edtech.2023.3.1.5
UDC: 004.421:795

Pathfinding Algorithms in Games

Marko Novaković*
*Information Technology School – ITS, Belgrade, Serbia; marko45521@its.edu.rs

Abstract: This article describes the A*, Dijkstra's, and genetic pathfinding algorithms used in games, providing a comparison and information about them. While these are not the only algorithms used in game pathfinding, they are currently the most commonly used ones. As games increasingly demand the presentation of more data (higher-quality graphics, complex environmental communication systems, better sound effects, advanced character movement sets, smarter AI, etc.) in shorter time frames, algorithms must be developed to become more optimal. In the near future, they will be replaced by improved versions or entirely new algorithms.

Keywords: ant colony optimization, AI, A* algorithm, breadth-first search, Dijkstra's algorithm, game character, graph, genetic algorithm, level, pathfinding, ant colony optimization.

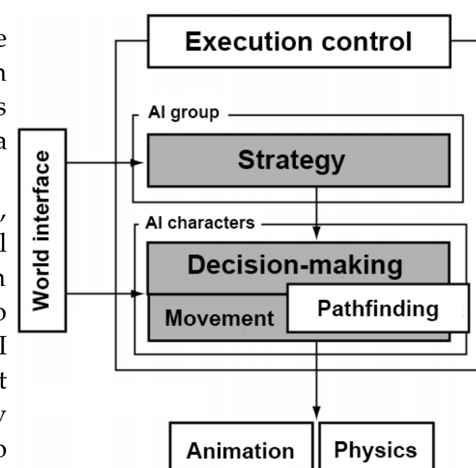
I. Introduction

Games characters often need to move around at certain levels. Sometimes these movements are predetermined by developers, such as a guard patrolling a specific path or a small enclosed area where a dog can move randomly. Fixed paths are easy to implement, but errors can easily occur if an object ends up on the path. Characters that move randomly may appear as having no aim and can easily get stuck.

More complex characters do not know in advance where they will move. Units in real-time strategy games may receive orders from players to go to a specific point on the map at any given time. In games where stealth is important, a patrolling guard may need to go to the nearest alarm location and call for backup. Enemies in platform games may need to chase the player across gaps using available platforms.

For each of these characters, Artificial Intelligence (AI) must be able to calculate a suitable path through the game level to reach the goal from their current location. We want the path to be reasonable, as short as possible, and for the character to move fast (it wouldn't look smart if a character walked from the kitchen to the living room through the attic).

This is called pathfinding, sometimes referred to as path planning, and it is essential in AI of games. In the example of the AI game model shown in Figure 1, pathfinding lies at the boundary between decision-making and movement. It is often used only to determine how to move towards the goal, while the goal itself is determined by other AI components, and the pathfinder only calculates the path. To achieve this, it can be integrated into the movement control system so that it is called only when it is necessary to plan a path. However, the pathfinding AI can also be used to determine both the goal and the path.



The majority of games use a pathfinding solution called the A* algorithm (A-star). While efficient and easy to implement, A* cannot directly operate with game-level data. It requires the game level to be represented in a specific data structure: a directed, weighted graph. [1] *Figure 1: AI game model [1]*

II. Dijkstra's Algorithm

Given a graph and a starting node, determine the shortest path from the starting node to all other nodes in the graph.

We generate a shortest path tree, taking the starting node as the root. We have two sets of data: the first set contains the nodes included in the shortest path tree, and the second set contains the nodes that have not yet been included. In each iteration of the algorithm, we find the node in the set that has not been included in the shortest path and has the smallest distance from the source.

Detailed steps of the algorithm:

1. Create a set called sptSet (shortest path tree set) to keep track of the nodes included in the shortest path – those whose shortest distance from the source has been calculated and confirmed. Initially, this set is empty.

2. Assign distance values to all nodes in the graph. Initialize all distances to infinity. Assign a distance of 0 to the starting node so that it is selected first.

3. While sptSet does not contain all nodes:

3.1. Select the node that is not in sptSet and has the smallest distance from the last node.

3.2. Include it in sptSet.

3.3. Update the distances of all nodes that are adjacent to node **u** (the last node). To update the distances, go through all the neighboring nodes of node **u**. For each neighboring node **v**, if the sum of the distance from **u** and the weight (cost factor – the distance in this case) from **u** to **v** is less than the distance of **v**, then update the distance of **v**.

Example of a graph "Figure 2" to illustrate the algorithm's operation:

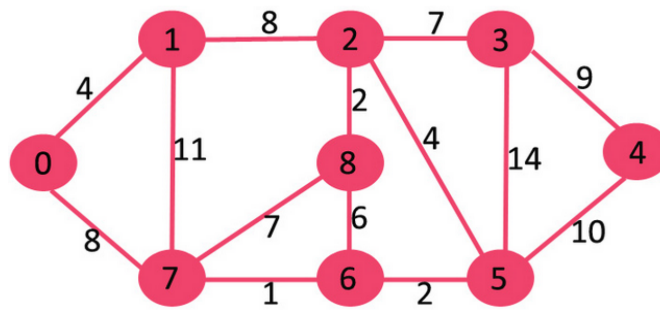


Figure 2: Example of a weighted graph [2]

The set sptSet is initially empty, and the distances assigned to the nodes are {0, INF, INF, INF, INF, INF, INF, INF}, where INF represents infinity. Now, a node with the minimum distance value is selected. Node 0 is chosen and included in sptSet. Now, sptSet is {0}. After adding 0 to sptSet, the distances of its neighboring nodes are updated. The neighboring nodes of 0 are 1 and 7. The distance values of 1 and 7 are updated to 4 and 8, respectively. The following subgraph shows the nodes with their distance values, only displaying nodes with finite distance values. The nodes included in the Shortest Path Tree (SPT) are marked in green "Figure 3".

Select the node with the smallest distance that is not already included in the SPT (not in sptSet). Node 1 is selected and added to sptSet. Now, sptSet looks like: {0, 1}. Update the distance values of the neighboring nodes of node 1. The distance of node 2 becomes 12 "Figure 4".

Select the node with the smallest distance that is not already included in the SPT (not in sptSet). Node 7 is chosen. The distance values of nodes 6 and 8 become finite (15 and 9, respectively) "Figure 5".

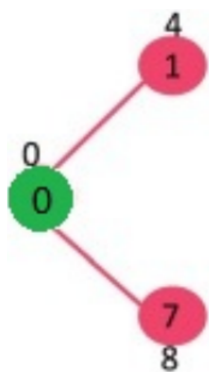


Figure 3: Subgraph 1 [2]

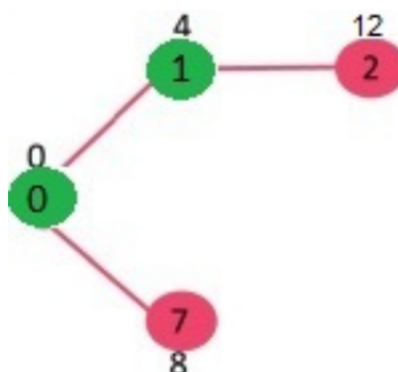


Figure 4: Subgraph 2 [2]

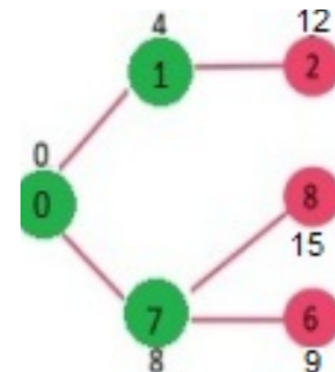


Figure 5: Subgraph 3 [2]

Choose the node with the smallest distance that is not already included in the SPT (not in sptSet). Node 6 is selected. Now, sptSet looks like: {0, 1, 7, 6}. Update the distance values of the neighboring nodes of node 6. The distances of nodes 5 and 8 are updated "Figure 6".

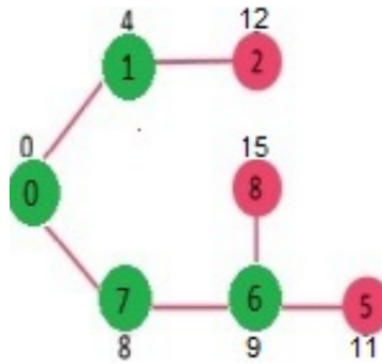


Figure 6: Subgraph 4 [2]

We repeat these steps until all nodes are included in the sptSet. In the end, we obtain the following Shortest Path Tree (SPT) "Figure 7" [2].

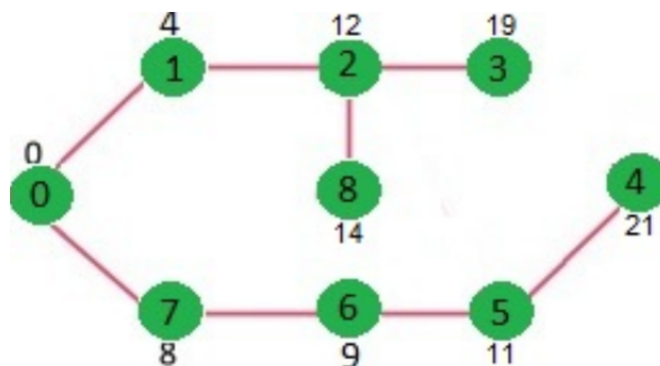


Figure 7: Subgraph 5 [2]

III. A* ALGORITHM

A* (pronounced "A star") is an algorithm commonly used for pathfinding and graph traversal. The algorithm efficiently finds the path of movement between graph nodes.

On a map with multiple obstacles, finding paths between points A and B can be challenging. For example, a robot without additional instructions about the direction of movement would continue moving until it encounters an obstacle ("Figure 8").

However, the A* algorithm introduces heuristics into standard graph search algorithms, essentially planning ahead at each step to make a more optimal decision. With A*, the robot would search for a path as shown in "Figure 9".

A* is an extension of Dijkstra's algorithm with some characteristics of breadth-first search (BFS) [3].

Similar to Dijkstra's algorithm, A* constructs the shortest path tree from the initial node to the goal node. What makes A* different and more effective for many searches is its use of a function $f(n)$ for each node, which provides an estimate of the total cost (length) of the path if that node is used. Therefore, A* is a heuristic function, which means that the heuristic is more of an estimation rather than a provably accurate value.

A* expands paths that are shorter (cheaper) by using the function:

$$f(n) = g(n) + h(n)$$

where:

$f(n)$ = total estimated cost of the path through node **n**,

$g(n)$ = accumulated cost to reach node **n**,

$h(n)$ = estimated cost from node **n** to the goal. This is the heuristic part of the function, making an assumption.

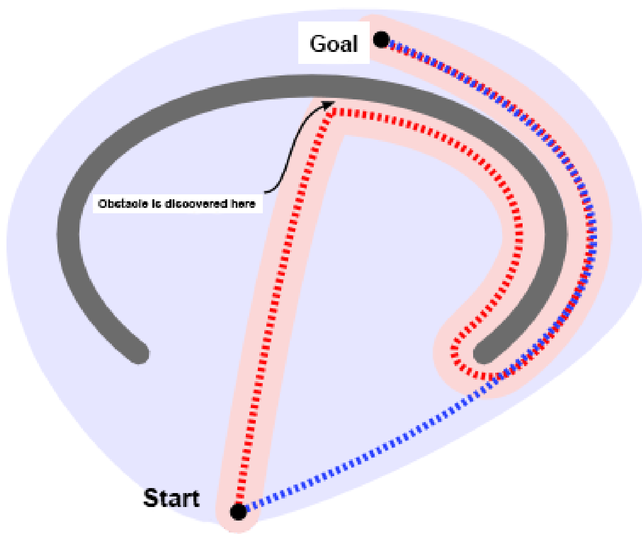


Figure 8: Inefficient way of finding a path [5]

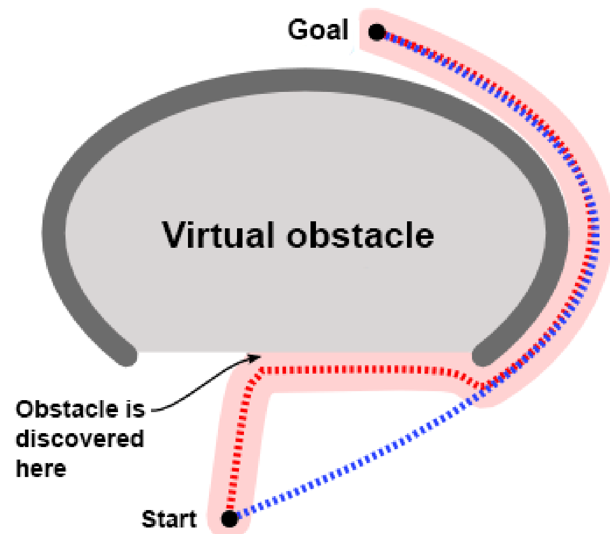


Figure 9: Example of using A* for pathfinding [5]

In the grid "Figure 10", the A* algorithm starts from the beginning (red node) and considers all neighboring nodes. Once the list of neighboring nodes is filled, those that are inaccessible (walls, obstacles, out of bounds) are filtered out. Then, the node with the lowest cost, determined by $f(n)$, is chosen. This process is recursively repeated until the shortest path to the goal (blue node) is found. The calculation of $f(n)$ is done heuristically, typically yielding good results.

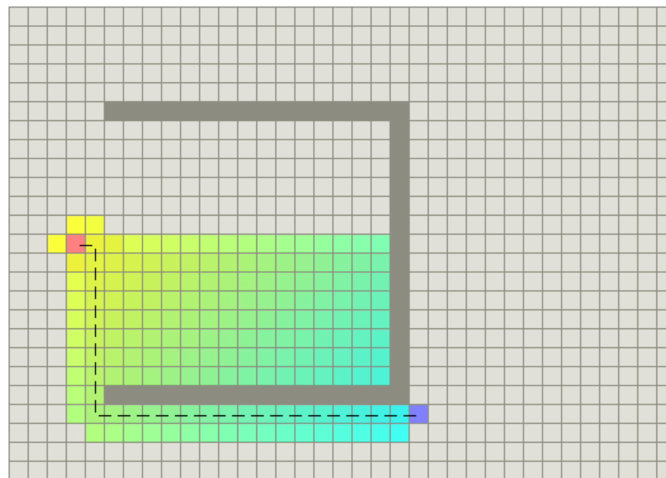


Figure 10: Using the A* algorithm [5]

Calculation of h(n) can be done in several ways:

The most common approach is to use the Manhattan distance [4] from node n to the goal. This is a standard heuristic for grid-based problems.

If h(n) = 0, A* becomes Dijkstra's algorithm, which guarantees to find the shortest path.

The heuristic function must be admissible, meaning it should never overestimate the cost required to reach the goal. Both the Manhattan distance and h(n) = 0 are admissible.

Using a good heuristic is important for determining the performance of the A* algorithm. The ideal value of h(n) would be the exact cost of reaching the goal.

However, this is not possible since the path is unknown. But a method can be chosen that gives reasonably accurate values, such as when traveling in a straight line without obstacles. This would result in optimal A* performance.

It is desirable to choose an h(n) function that costs less than actually reaching the goal. This allows h(n) to work accurately. If a higher value is chosen, it would lead to faster but less accurate performance. Therefore, it is often the case that h(n) is chosen to be less than the actual cost.

Figure 11 illustrates the pseudocode of the A* algorithm written in Python-like syntax [5].

```
napravi otvorenu listu koja se sastoji samo iz početnog čvora
napravi praznu zatvorenu listu

while (nije se stiglo do ciljnog čvora):

    uvrsti čvor sa najmanjom f vrednosti u otvorenu listu

    if(ovaj čvor je naš odredišni čvor):

        završili smo

    if not:

        uvrsti trenutni čvor u zatvorenu listu i pregledaj sve njegove susedne čvorove

        for(svaki susedni čvor trenutnog čvora):

            if(ako susedni čvor ima manju g vrednost od trenutnog čvora i u zatvorenoj je listi):

                zameni suseda sa novim koji ima manju g vrednost
                trenutni čvor je sada roditelj susednom čvoru

            else if(ako je g vrednost trenutnog čvora manja od susednog i ovaj sused je u otvorenoj listi):

                zameni suseda sa novim kojim ima manju g vrednost
                ažuriraj roditelja suseda na trenutni čvor

            else if(ovaj sused nije ni u jednoj listi):

                dodaj ga u otvorenu listu i dodeli mu g vrednost
```

Figure 11: A Pseudocode [5]*

For more details on the A* algorithm, refer to the article [6].

IV. Heuristic Techniques

Heuristic techniques are used to solve problems in a faster and more efficient way by optimizing solution quality, accuracy, and precision [7]. Heuristic algorithms aim to find a good solution to a specific problem, such as pathfinding, within a reasonable computation time, but without guaranteed efficiency. "Heuristics" means "to find" in Greek [8]. Heuristic algorithms include Dijkstra's algorithm and A* algorithm, which were described in the previous text, as well as the breadth-first search (BFS) algorithm [3].

V. Metaheuristic Techniques

Metaheuristics are essentially high-level strategies that combine lower-level techniques to describe and exploit the search space. Metaheuristics are a higher level of heuristics and usually exhibit better performance than heuristics. Metaheuristics can reduce search time and provide satisfactory solutions for complex pathfinding problems in video games. Based on studies, metaheuristic algorithms such as genetic algorithms and ant colony optimization have been used in games to solve pathfinding problems. Metaheuristics are based on certain natural phenomena, and the most successful metaheuristic algorithms are inspired by natural systems. For example, ant colony optimization [9] and bee algorithm were developed based on animal behaviors [10].

VI. Genetic Algorithm

Genetic algorithms are among the most popular evolutionary algorithms in terms of the diversity of their applications. A wide range of well-known optimization problems have been attempted to be solved using genetic algorithms. Furthermore, genetic algorithms are population-based, and many modern evolutionary algorithms are either based on genetic algorithms or share significant similarities.

The essence of genetic algorithms is encoding the optimization function as a sequence of bits or characters representing chromosomes, manipulating strings using genetic operators, and selecting suitable individuals with the aim of finding a good (even optimal) solution to the problem. In the following text, fitness and fitness function will be used. Fitness refers to the desired characteristics to be obtained through algorithm iterations.

This is usually done through the following procedure:

1. Encoding goals or cost functions.
2. Defining a fitness function or selection criteria.
3. Creating a population of individuals.
4. Performing an evolutionary cycle or iterations by evaluating the fitness of all individuals in the population, creating a new population through crossover and mutation, suitable reproduction, etc., ultimately modifying the old population and iterating using the new population.
5. Decoding the results obtained by the solution.

These steps can be represented schematically as the pseudocode of genetic algorithms ("Figure 12").

One iteration of creating a new population is called a generation. In most genetic algorithms, fixed-length strings are commonly used during each generation, although there is substantial research on variable-length strings and code structures. In adaptive genetic algorithms, encoding the fitness function often takes the form of binary strings or arrays with real values. For simplicity, binary strings were used in the discussion. Genetic operators include crossover, mutation, and selection from the population.

Crossover, denoted as $\mathbf{P_c}$, is the main operator with a high probability, and it is performed by replacing a segment of one chromosome at a randomly chosen position with the corresponding segment of another chromosome ("Figure 13").

The mutation operator is obtained by randomly changing the value (0 → 1 or 1 → 0) at a randomly selected bit ("Figure 14"). The probability of mutation is denoted as $\mathbf{P_m}$ and is often small. Additionally, mutations can occur at multiple locations, which can be advantageous in practice and application.

Selection of individuals in the population is done by evaluating fitness, and an individual can be included in the next generation if a certain fitness threshold is reached. Furthermore, selection can be fitness-based, so that the reproduction of the population is proportional to fitness. This means that individuals with higher fitness have a greater chance of reproducing [11].

```
Objective function f(x), x = (x1,..., xd)T
Encode solutions into chromosomes (strings)
Define fitness function F (e.g., F (ovde ide
znak sa slike) = f(x) for maximization)
Generate initial population
Define crossover probability (Pc) and mutation
probability (Pm)
while (t < maximum number of generations)
Generate new solutions through crossover
and mutation
Perform crossover with probability Pc
Perform mutation with probability Pm
Accept new solutions if their fitness increases
Select the current best for the next generation
Update t = t + 1
end while

Decode the results and visualize them
```

Figure 12: Pseudocode of genetic algorithms [11]

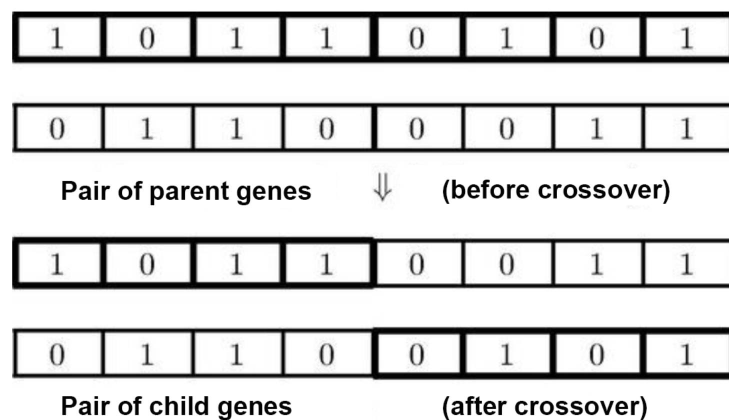


Figure 13: Diagram of crossover of a random segment in genetic algorithms [11]

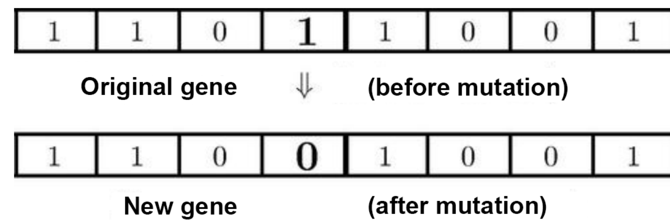


Figure 14: Diagram of mutation of a random bit [11]

Acknowledgement

This work was done as part of the course "Fundamentals of Applied Research," under the guidance of Prof. Dr. Slavko Pokorni.

References

1. Millington, I. AI for Games, 3rd edition, 2019, pp. 195–196.
2. "Dijkstra's shortest path algorithm | Greedy Algo-7." GeeksForGeeks. Available at: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/. Accessed: April 4, 2022.
3. "Breadth First Search (BFS) Algorithm." algotree.org. Available at: https://algotree.org/algorithms/tree_graph_traversal/breadth_first_search. Accessed: April 4, 2022
4. Black, P. E. "Manhattan distance." In Dictionary of Algorithms and Data Structures [online]. Paul E. Black, ed. 11 February 2019. Available at: https://www.nist.gov/dads/HTML/manhattanDistance.html. Accessed: April 4, 2022.
5. "A* Search." Brilliant.org. Retrieved 10:24, April 4, 2022, from https://brilliant.org/wiki/a-star-search/.

6. Patrick, L. "A* pathfinding for beginners." GameDev WebSite. Available at: https://www.gamedev.net/reference/articles/article2003.asp. Accessed: April 4, 2022.
7. Wolsey, L. A. "Heuristic Algorithms." Integer Program., no. January, p. 17, 1998.
8. Rafiq, A. et al. "2020 IOP Conf. Ser.: Mater. Sci. Eng. 769 012021 'accepted for publication'."
9. Yang, X. S. Nature-Inspired Optimization Algorithms, 2014, pp. 305–308.
10. Yang, X. S. Nature-Inspired Optimization Algorithms, 2014, pp. 308–312.
11. Yang, X. S. Nature-Inspired Optimization Algorithms, 2014, pp. 116–130

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.