APPLYING UNINFORMED SEARCH ALGORITHMS FOR PATHFINDING IN VARIOUS ENVIRONMENTS"

[1]**Primbetov Abbaz Muratbay Uli**
orcid: 0009-0004-3120-1152
e-mail: abbaz0203@mail.ru
[2]**Jumaev Giyosjon Abdivahobovich**
orcid: 0009-0008-3069-4062
e-mail: giyosjonjumaev@utas.uz
[2]**Rahmonova Mahbuba**
e-mail: mahbubam217@gmail.com
[4]**Abdusalomov Shaxbozbek**
e-mail: skayoker2000@gmail.com
[1,2]Senior lecturer, Computer engenering, Tashkent University of Applied Sciences
[3,4]Student, Computer engenering, Tashkent University of Applied Sciences
https://doi.org/10.5281/zenodo.14860391

Annotation: The A* algorithm is a well-established search method that has been widely utilized in the pathfinding research community. Its strengths—efficiency, simplicity, and modularity—distinguish it from other pathfinding tools. Because of its proven effectiveness and adaptability, A* has become a popular choice among researchers looking to address complex pathfinding challenges. Its comprehensive approach allows for effective navigation in various applications, from robotics to video games, making it a cornerstone in the field of pathfinding research. This paper investigates the application of the A* algorithm in solving pathfinding problems, focusing on its effectiveness and efficiency in various environments

Keywords: A* algorithm, Pathfinding, optimazision.

1 Introduction.

Search algorithms have been a big topic in computer science for quite some time. Many applications of these algorithms need to work quickly and efficiently, often without a lot of processing power. When searches aren't fast and accurate, it can be a real problem. There are two main types of search algorithms: informed and uninformed. While both have their uses, informed searches are generally more popular because they use a heuristic function to estimate the distance to the goal, helping them make better choices. This paper will focus on the A* search algorithm, looking at how it compares to other pathfinding methods, its improvements, and what the future might hold for it.

Pathfinding generally refers to finding the shortest route between two endpoints. Examples of such problems include transit planning, telephone traffic routing, maze navigation, and robot path planning. As the need for efficient navigation solutions increases, pathfinding has become a popular and challenging problem across various fields. A common issue in pathfinding is how to avoid obstacles cleverly and seek out the most efficient path over different types of terrain. Early solutions to pathfinding problems, such as depth-first search, iterative deepening, breadth-first search, Dijkstra's algorithm, best-first search, A* algorithm, and iterative deepening A*, soon faced challenges due to the exponential growth in complexity of the environments being navigated. More efficient solutions are necessary to tackle pathfinding issues in increasingly complex settings with limited time and resources.

Due to the significant success of the A* algorithm in pathfinding, many researchers are focused on speeding it up to meet the evolving demands of various applications. Considerable effort has been made to optimize this algorithm over the past decades, leading to the introduction of numerous revised algorithms. Examples of these optimizations include improving heuristic methods, optimizing map representations, introducing new data structures, and reducing memory requirements. The next section provides an overview of A* techniques that are widely used in current pathfinding applications.

2 A* algorithm

A* is a versatile search algorithm that can tackle a variety of problems, with pathfinding being one of its key applications. When it comes to pathfinding, the A* algorithm consistently looks at the most promising unexplored spots. If it finds a location that matches the goal, it stops; if not, it keeps track of all the neighboring locations to explore next. A* is widely recognized as one of the most popular pathfinding algorithms in the field of artificial intelligence. Here's steps for the A* algorithm:

1. Add the starting node to the open list.
2. Repeat the following steps:
   a. Look for the node which has the lowest f on the open list. Refer to this node as the current node.
   b. Switch it to the closed list.
   c. For each reachable node from the current node
      i. If it is on the closed list, ignore it.
      ii. If it isn't on the open list, add it to the open list. Make the current node the parent of this node. Record the f, g, and h value of this node.
      iii. If it is on the open list already, check to see if this is a better path. If so, change its parent to the current node, and recalculate the f and g value.
   d. Stop when
      i. Add the target node to the closed list.
      ii. Fail to find the target node, and the open list is empty.
3. Tracing backwards from the target node to the starting node. That is your path.

Fig.1 steps for the A* algorithm:

In the standard terminology used when discussing A*, $g(n)$ represents the exact cost from the starting point to any point n while $h(n)$ represents the estimated cost from point n to the destination. The function $(n)=g(n)+h(n)$ combines these two values. A* has several useful properties that were established by Hart, Nilsson, and Raphael in 1968. First, A* is guaranteed to find a path from the start to the goal if a path exists. It is also optimal if $h(n)$ is an admissible heuristic, meaning $h(n)$ is always less than or equal to the actual cheapest path cost from n to the goal. Additionally, A* makes the most efficient use of the heuristic, ensuring that no search method using the same heuristic function examines fewer nodes than A*. Although A* is a popular choice for pathfinding, its application depends on the nature of the problem and the internal representation of the environment. For instance, in a rectangular grid of 1000×1000 squares, there are 1 million possible squares to search, which can make finding a path quite labor-intensive.

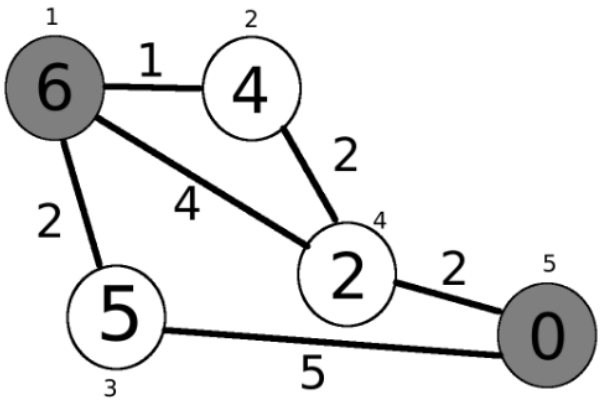

Fig.2 Example for A* search algorithm

In this diagram, each node is numbered from 1 to 5 for easy identification, with node 1 as the starting point and node 5 as the goal. Inside each node, there's a number showing the estimated distance to the goal, which represents $h(n)$ The lines connecting the nodes have lengths indicating the actual distances. At first glance, it's clear that the best path is from nodes 1 to 2, then to 4, and finally to 5, totaling (1 + 2 + 2) = 5 units of distance.

At the beginning, the algorithm adds node 1 (the starting node) to the open set. Since node 1 is the only one there, it gets searched right away. After that, it's moved to the closed set. Next, the algorithm looks at its neighbors—nodes 2, 3, and 4. The scores for these nodes are shown in the following table:

Table 1 Calculating the distance from node 1 to the goal

Node	g(n)	h(n)	f(n) = g(n) + h(n)
2	1	4	5
3	2	5	7
4	4	2	6

Since node 2 has the lowest f score (5), it gets searched next. The neighbors of node 2 are nodes 1 and 4, but since node 1 is already in the closed set, we only look at node 4. Initially, node 4 has a score of 6 if we start from node 1. However, if we go through node 2 to reach node 4, the f score becomes (1 + 2) + 2 = 5, which is better! This means we update the path from 1 -> 4 to 1 -> 2 -> 4. After that, node 2 is added to the closed set.

Next, node 4 has the lowest f score of 5, so it gets searched. Then, the goal node is checked, and the program finishes. The optimal path found by this algorithm is from node 1 to node 2 to node 4 to node 5.

3 Experimental results and their discussion

In the application, each node that isn't on the edges or corners has eight neighbors. For any vertical or horizontal move, the algorithm records the distance as 1. If it's a diagonal move, the distance is noted as the square root of 2. This way, the algorithm accurately reflects the different distances depending on the direction of movement.

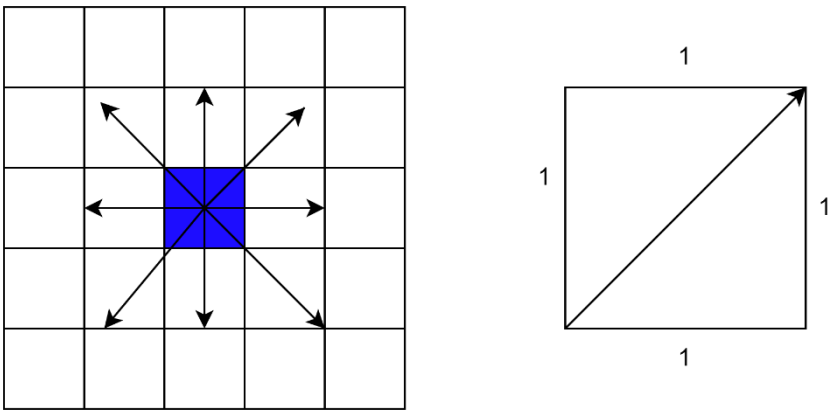

Fig.3 Application for pathfinding problem using A* algoritm

For this application, the heuristic function $h(n)$ calculates the shortest possible distance from the current node to the goal, considering horizontal, vertical, and diagonal steps. This means that the actual shortest distance (n) is always greater than or equal to $h(n)$(i.e., $h(n) \leq d(n)$).

Now, let's prove that this algorithm always finds the shortest path. When the algorithm reaches the goal node, it must have the lowest f score. Since the distance from the goal node to itself is zero, we have $h(n)=0$, which means $f(n)=g(n)$. This means the f score reflects the exact distance from the start node to the goal node along that path. We can call this distance d.

Now, let's look at every node in the open set. We know their f scores must be greater than dd. Again, let (n) be the actual shortest distance from the current node to the goal. This gives us the inequality:$d \leq f(n)=g(n)+h(n) \leq g(n)+d(n)$.So, we can say $d \leq g(n)+d(n)$. This means that taking any other path will result in a distance greater than dd. Therefore, the path with distance dd is indeed the shortest one!. In the image below, you can see the Python implementation of the A* algorithm

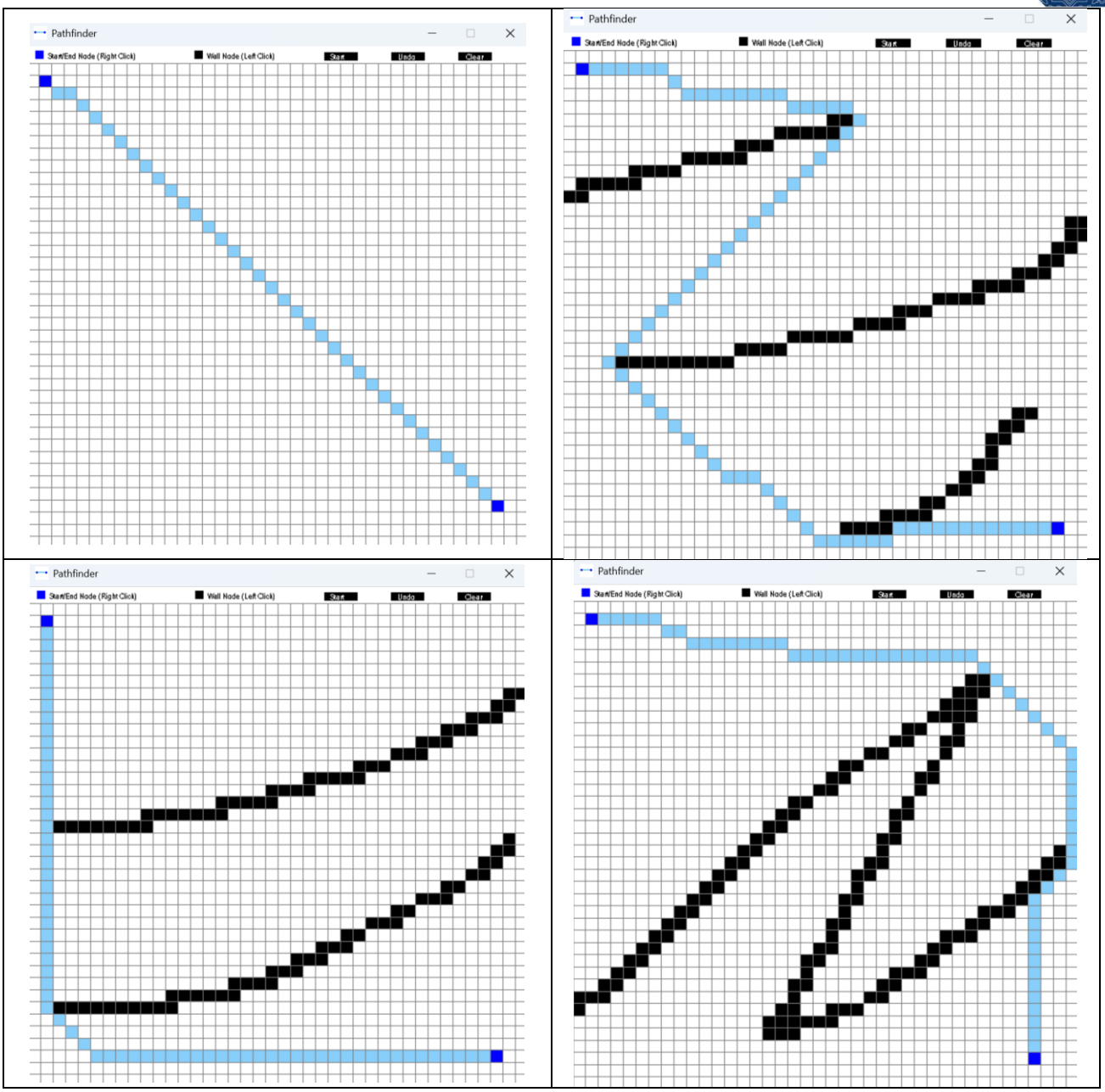

Fig.4 Python implementation of the A* algorithm (results 50.5, 89.74, 69.66, 68.8)

In the abouve images pathfinding algorithm written in Python using the Pygame engine. It implements the A* search algorithm to find the shortest distance between two points while avoiding obstacles. In the diagram above, the two endpoints are shown in dark blue, the wall nodes are in black, and the shortest path is highlighted in light blue, with a total distance of about 50 to 68 This algorithm is guaranteed to find the shortest path.

4 Conclusions

In conclusion, the A* algorithm stands out as a powerful and efficient method for finding the shortest path between two nodes in various applications, from robotics to gaming. By combining the actual cost to reach a node with an estimated cost to get to the goal, A* effectively balances exploration and optimization. Its use of heuristic functions allows for adaptable and intelligent pathfinding, making it suitable for both static and dynamic environments.

References:

1. Primbetov, A. (2024). Automatic Red Eye Remover using OpenCV. Modern Science and Research, 3(1), 1-3.

2. Primbetov, A., Saidova, F., Yembergenova, U., & Primbetov, A. (2024). REAL TIME LOGO RECOGNITION USING YOLO ON ANDROID. Modern Science and Research, 3(1), 1-5.
3. Abbaz, P. (2024). REAL TIME LOGO RECOGNITION USING YOLO ON ANDROID. Eurasian Journal of Academic Research, 4(7S), 1094-1099.
4. Giyosjon, J., Anvarjon, N., & Abbaz, P. (2023). SAFEGUARDING THE DIGITAL FRONTIER: EXPLORING MODERN CYBERSECURITY METHODS. JOURNAL OF MULTIDISCIPLINARY BULLETIN, 6(4), 77-85.
5. ОБУЧИТЕ YOLOV8 НА ПОЛЬЗОВАТЕЛЬСКОМ НАБОРЕ ДАННЫХ Примбетов Аббаз,Саидова Фазилат,Нормуминов Анваржон, Примбетов Азиз, https://doi.org/10.5281/zenodo.8023689
6. DEVELOP AN APPLICATION THAT CONVERTS VIDEOS INTO SLIDES UTILIZING BACKGROUND ESTIMATION AND FRAME DIFFERENCING IN OPENCV Primbetov Abbaz, Abdusalomov Shaxbozbek, To'lqinov Oybek, Rahmonova Mahbuba, Pardaboyeva Mohichexra. https://doi.org/10.5281/zenodo.13364871
7. Cui, X., & Shi, H. (2011). A*-based pathfinding in modern computer games. International Journal of Computer Science and Network Security, 11(1), 125-130.
8. Verma, M. G., & Kumar, M. A. Algorithmic Pathfinding: Comparing Dijkstra's and A* Algorithms in Complex Grid Environment.
9. Foead, D., Ghifari, A., Kusuma, M. B., Hanafiah, N., & Gunawan, E. (2021). A systematic literature review of A* pathfinding. Procedia Computer Science, 179, 507-514.
10. Lester, P. (2005). A* pathfinding for beginners. online]. GameDev WebSite. http://www. gamedev. net/reference/articles/article2003. asp (Acesso em 08/02/2009).