DOI: 10.32347/2412-9933.2023.55.92-97
UDC 004

Alpert Maksym
PhD Student, Department of Information Systems and Technologies, *https://orcid.org/0000-0002-8938-1473*
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
Onyshchenko Viktoriia
Professor of Department, Guarantor of the Doctor of Philosophy ESP Information Systems and Technologies, *https://orcid.org/0000-0002-3126-2260*
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine

FINDING THE BEST SHORTEST PATH ALGORITHM FOR SMART SUITCASE

Abstract. *Smart suitcases are a revolutionary new breed of travel accessory that utilize sophisticated technology for enhanced convenience and ease of journeying. These suitcases are equipped with a multitude of advanced features, such as internet connectivity, infrared sensors, inbuilt algorithms to bypass obstacles and an accompanying mobile app designed to track the belongings' owner. A key component lies in realizing this technology in the selection of an appropriate algorithm to calculate shortest paths through challenging environments. In general, there are four main classes of algorithm that may be considered as candidates: Dijkstra's algorithm, A-Star (A*), Bi-Directional A-Star (BiA*) and Rapidly-exploring random tree (RRT). Each offers its own advantages and limitations regarding performance, memory requirements and accuracy, which must be taken into account if it is to fulfill the purpose effectively. Moreover, these smart suitcases boast infrared sensors which allow them detect and avoid obstacles present in their paths via infrared sensors that reflect off nearby objects. Base information gathered by the sensors. Then filtered through an internal algorithm that distinguishes the best possible method for escape from indicated obstacle. Overall, smart suitcases signify a cutting-edge revolutionizing trend likely bound to captivate travelers across all types who seek effectiveness and efficiency during embarkment journeys.*

Keywords: smart suitcase; shortest path; Dijkstra's algorithm; A-Star; Bi-Directional A-Star; Rapidly-exploring random tree

Introduction

The development of smart suitcase technology has opened up many possibilities for travelers, allowing them to relish in the convenience and comfort that this new style of luggage provides. Of course, one important factor of a successful journey is arriving at your destination in the most efficient way possible. Finding the best shortest path algorithm for these types of suitcases is primary task. This paper seeks some existing algorithms used for route optimization and propose an ideal approach that could be implemented in the latest iterations of smart suitcase technology.

Finding the best shortest path algorithm for a smart suitcase has become an important consideration. The need to remain agile and efficient in travel is unlikely to dissipate any time soon, so having a reliable way of navigating from one place to another in the quickest possible time is important. There are several algorithms that can be used for this purpose. One such example is Dijkstra's algorithm. However, it may not necessarily be the most suitable approach for certain scenarios due to its complexity and applicability only on weighted graphs with non-negative weights. Therefore, other options should also be taken into account when conducting research on distinctive pathfinding techniques that could fit even more specific scenarios involving multiple trips between various locations.

Graph search algorithms are one possibility for route optimization such as Dijkstra's shortest path algorithm. This method consists of building a connected graph based on nodes/locations along with their distances apart from each other before traversing it to determine the most efficient route depending on user preferences or other factors like time or budget constraints. The advantage lies in its ability to maximize speed while minimizing cost. Despite its effectiveness, this particular method can be limited when considering more than two dimensions (i.e., factors outside just distance) when trying to calculate the best route(s).

Dijkstra's algorithm demonstrates the best characteristics overall when dealing with larger graphs containing non-overlapping regions. On the other hand, A-Star (A*) provides effective optimization techniques viable under a moderate computational resources' constraint. But it still requires additional heuristics incorporated beforehand, either manually or automatically via machine learning constructions, to generate trajectories accurately and efficiently. Otherwise, it will consume an excessive amount of space

and energy, potentially affecting battery life negatively and significantly, which is not desirable, particularly concerning the portability factor. There can be better alternatives in the context to avoid such issues arising given the possibility that the landscape is uncertain in advance and may not be properly handled by the situation arising.

The A-Star Search Algorithm has been a prevalent solution among many developers as it combines characteristics of both breadth-first search and depth-first search while dedicating effort towards achieving optimal paths by keeping track of cost associated with each node up until its destination point. Furthermore, Bi-Directional A-Star Algorithm (BiA*) leverages bidirectional searches along with heuristic estimates alongside A-Star's same principles; eliminating redundant routes and staying optimally focused on only those paths deemed necessary. Thus, further improving upon the performance and accuracy of typical A-Star implementations out there that don't always guarantee results due to lack diligence or inadvertent overlooks during execution cycles. Otherwise, there are mistaken assumptions regarding effective navigation sequences.

Objective of the paper

Four different algorithms (A*, BiA*, Djikstra's and RRT) are investigated and compared for finding the shortest path in the context of a smart suitcase. The paper aims to evaluate the effectiveness and performance of these algorithms in enabling the smart suitcase to navigate efficiently through various environments while avoiding obstacles. By conducting a comprehensive analysis and comparison of the algorithms, the objective is to determine the most suitable algorithm that can be employed in the smart suitcase to ensure optimal path planning and efficient movement.

Approach for choosing basic technology for creation of smart suitcase

Let's conduct analysis and make comparison with other solutions for smart suitcases.

The article The Design of Smart Suitcase [1] suggests using image recognition to determine the distance between a suitcase and a person. If the suitcase with a camera detects that the captured image of a person is too large, it will perform a slow-down action. Moreover, when the target pixel is too large and the relative position of the target to the environment remains the same, the suitcase will automatically stop to avoid collision. Also, a software application is additionally used, which can notify about the loss of communication.

In the article Smart Luggage Carrier system with Theft Prevention and Real Time Tracking Using Nano Arduino structure [2], it is proposed to use ultrasonic sensors that send sound waves, and they will calculate the distance between the bag and the person by collecting reflected waves when it hits an obstacle.

The Smart Luggage Carrier article [2] suggests using infrared sensors to avoid obstacles. The suitcase determines the value of the distance to the object using IR sensors.

The article Smart Airline Baggage Tracking and Theft Prevention with Blockchain Technology offers a solution for tracking baggage using RFID tags and blockchain technology. Intelligent data management provides an integrated deployment and monitoring service. Collaboration technology is implemented to support end-to-end tracking and alerts.

The article Luggage Theft Identification And Smart Lock Using Face Recognition [3] uses facial recognition technology to avoid theft.

In the article [4] unmanned ground vehicle is used to detect obstacles such as potholes but the decision of avoiding is on operator.

All these solutions are not automated and they are controlled by human. In my future realization I will use not only camera to recognize but also one of proposed algorithm to avoid obstacles.

Careful comparison of A*, BiA*, Dijkstra's and RRT algorithms

Let's examine the selected algorithms in more detail. We will delve deeper into their characteristics and functionalities.

Completeness: All four algorithms are complete, meaning they will always find a path if one exists.

Optimality: A*, BiA*, and Dijkstra's algorithms are all optimal. It means that they will always find the shortest path if one exists. RRT is not optimal, as the path it finds may not be the shortest.

Time Complexity: A*, BiA*, and Dijkstra's algorithms have the same worst-case time complexity of $O(b^d)$, where b is the branching factor and d is the depth of the solution. BiA* can be faster than A* and Dijkstra's algorithm in some cases, as it searches from both the start and target nodes simultaneously, which can reduce the number of nodes expanded.

Space Complexity: A*, BiA*, and Dijkstra's algorithms have the same worst-case space complexity of $O(b^d)$, where b is the branching factor and d is the depth of the solution. BiA* requires more space than A* and Dijkstra's algorithm because it needs to keep track of nodes from both the start and target nodes.

Use Case: A*, BiA*, and Dijkstra's algorithms are best suited for finding paths in static environments, where the obstacles do not move frequently. Dijkstra's algorithm is especially useful when all edge weights are non-negative. RRT is designed for dynamic environments, where the obstacles move frequently and the path needs to be recalculated often.

The use case of A* is finding paths in static environments. The purpose of BiA* is finding paths in static environments, faster than A* in some cases.

Djikstra's algorithm use case is finding paths in static environments, when all edge weights are non-negative.

RRT is used to sampling-based planning in dynamic environments, suitable for high-dimensional spaces.

Table 1 – ***Comparison of A*, BiA*, Dijkstra's and RRT algorithms***

Algorithm	A*	BiA*	Dijkstra's	RRT
Completeness	Yes	Yes	Yes	Yes
Optimality	Yes	Yes	Yes	No
Time Complexity	$O(b^d)$	$O(b^{(d/2)})$	$O(b^d)$	Varies
Space Complexity	$O(b^d)$	$O(b^d)$	$O(b^d)$	Varies

Branching factor is marked as "b". It's the average number of child nodes for each node in the search tree. Depth of the solution is marked as "d", which is the number of steps required to reach the target node from the start node. The depth of the solution represents the length of the shortest path between the start and target nodes in pathfinding problems. "b^d" represents the worst-case time and space complexity for the algorithms. The worst-case time complexity refers to the maximum amount of time the algorithm may take to find the solution, while the worst-case space complexity refers to the maximum amount of memory the algorithm may use while searching for the solution.

The time and space complexities can vary for RRT algorithm depending on the specific implementation and the characteristics of the environment.

The summarized results are presented in the table 1, providing a concise overview of the performance metrics of each algorithm. This table allows for easy comparison and identification of the algorithm that best meets the criteria for optimal pathfinding in a static obstacle environment.

Detailed comparison of A*, BiA*, Dijkstra's and RRT algorithms

A* is a popular pathfinding algorithm that uses a heuristic function to guide its search towards the target node. The heuristic function estimates the distance between a given node and the target node, and the algorithm uses this information to prioritize nodes that are closer to the target. A* expands the node with the lowest estimated total cost, which is the sum of the actual cost from the start node to the current node (known as the g-value) and the estimated cost from the current node to the target node (known as the h-value).

A* algorithm operates by exploring the search space using a combination of heuristic estimates and cost values, aiming to find the most efficient path from the start node to the goal node.

The A* algorithm begins by initializing the open list and closed list. The starting node is added to the open list. As long as the open list is not empty, the algorithm continues to iterate. It selects the node with the lowest f cost from the open list, checks if it is the end node, and returns the path if so. The current node is then moved from the open list to the closed list. For each neighbor of the current node, if the neighbor is not traversable or already in the closed list, it is skipped. If the neighbor is not in the open list, it is added to the open list with its parent set to the current node. Otherwise, the algorithm checks if the path to the neighbor from the current node is shorter than the previous path. If it is, the neighbor's parent and f score are updated. If there is no path from the start node to the end node, the algorithm returns failure.

Strengths:

– A* is complete and optimal if the heuristic function is admissible and consistent;

– it can be very efficient in practice, especially if the heuristic function is well-designed and the search space is not too large;

– it can be easily modified to handle different types of search spaces, such as grids or graphs.

Weaknesses:

– A* can be slow if the heuristic function is not well-designed or if the search space is too large;

– it can be memory-intensive if the search space is too large, as it stores all of the nodes it expands in memory;

– it is not well-suited for dynamic environments where the obstacles move frequently.

BiA* is a bidirectional version of A* that simultaneously searches from both the start node and the goal node. The algorithm terminates when the two search trees meet in the middle, which means that a path has been found. BiA* can be faster than A* in some cases because it expands nodes from both ends of the search space, which can reduce the number of nodes that need to be expanded overall.

The BiA* algorithm begins by initializing two open lists, one for the start node and one for the end node. The start node is added to the start open list, and the end node is added to the end open list. While both open lists are not empty, the algorithm proceeds. It selects the node with the lowest f cost from the start open list and checks if it is in the end closed list, returning the path if so. The current node is then moved from the start open list to the start closed list. For each neighbor of the current node, if the neighbor is not traversable or already in the start closed list, it is skipped. If the neighbor is not in the start open list, it is added to the start open list with its parent set to the current node. Otherwise, the algorithm checks if the path to the neighbor from the current node is shorter than the previous path and updates the neighbor's parent and f score accordingly. These steps are repeated for the

end open list. If there is no path from the start node to the end node, the algorithm returns failure.

Strengths:

– BiA* is complete and optimal if the heuristic function is admissible and consistent;

– it can be faster than A* in some cases, especially if the search space is relatively small or the path is easy to find;

– it can be memory-intensive, but it requires less memory than A* because it only needs to store the nodes from one search tree at a time.

Weaknesses:

– BiA* can be slower than A* in some cases, especially if the path is difficult to find or the search space is large;

– it requires more bookkeeping than A* because it needs to maintain two separate search trees;

– like A*, it is not well-suited for dynamic environments where the obstacles move frequently.

Dijkstra's algorithm is a classic pathfinding algorithm that finds the shortest path between the start node and all other nodes in the search space. It works by starting at the start node and iteratively expanding the node with the lowest cost. The algorithm stops when it reaches the goal node or when all nodes in the search space have been explored.

To find the shortest path using Dijkstra's algorithm, the starting node is initialized with a distance of 0 and added to the unvisited set. While the unvisited set is not empty, the algorithm continues. It selects the node with the lowest distance from the unvisited set and checks if it is the end node, returning the path if so. The current node is then moved from the unvisited set to the visited set. For each neighbor of the current node, if the neighbor is not traversable or already in the visited set, it is skipped. The algorithm calculates the distance from the starting node to the neighbor and updates the neighbor's distance if it is lower than the current distance. The neighbor's parent is set to the current node. If there is no path from the start node to the end node, the algorithm returns failure.

Strengths:

– Dijkstra's algorithm is complete and optimal if all edge weights are non-negative;

– it can be faster than A* in some cases, especially if the heuristic function is not well-designed or the search space is relatively small;

– it is memory-efficient, as it only needs to store the nodes that have been expanded.

Weaknesses:

– Dijkstra's algorithm can be slow if the search space is too large or the heuristic function is well-designed;

– it is not well-suited for dynamic environments where the obstacles move frequently.

RRT is a popular algorithm for motion planning in dynamic environments, but it sacrifices completeness and optimality for the ability to handle complex and changing environments. It is commonly used in robotics and autonomous systems for generating feasible paths considering obstacles and dynamic constraints.

To execute the Rapidly-exploring Random Trees (RRT) algorithm, an empty tree is initialized with the start node as the root. While the algorithm is running, it randomly samples a point in the search space and identifies the nearest node in the tree to that point. The algorithm extends the tree by creating a new node from the nearest node towards the sampled point, adhering to the maximum distance or step size. If the new node is collision-free and not within an obstacle, it is added to the tree and connected to the nearest node. This process of sampling, finding the nearest node, extending the tree, and adding nodes continues until the maximum number of iterations or a termination condition is met. If the termination condition is reached without the goal node being reached, the algorithm returns failure. However, if the goal node is reached, the path from the goal node to the start node is traced by following the connections in the tree. Ultimately, the algorithm returns the path from the start node to the goal node.

Strengths:

– RRT is particularly effective in high-dimensional and complex search spaces;

– it can handle dynamic environments where obstacles move or change over time;

– RRT is capable of exploring and adapting to the changing environment by constantly expanding the tree;

– it is well-suited for scenarios where the exact goal is not known, as it can explore the search space to find potential solutions.

Weaknesses:

– RRT is not goal-directed and does not guarantee finding the optimal path;

– it can be computationally expensive in terms of time and memory, especially in high-dimensional spaces;

– the quality of the generated paths highly depends on the sampling strategy and the exploration bias;

– RRT may struggle to find feasible paths in cluttered environments with narrow passages or tight spaces.

Future Realization

1. The autonomous suitcase will consist of a Raspberry Pi microcomputer, 2 motors for the wheels, a camera, 2 servo motors for turning the wheels and 2 servo motors for, as well as additional sensors to determine the distance between the suitcase and the user's phone.

2. You can use infrared or ultrasonic sensors to avoid common obstacles - other people, suitcases, etc.

3. The decision algorithm uses two sensors as a reference displacement. The essence of obstacle avoidance is how quickly the sensor recognizes that there

is an obstacle in front of the suitcase. The decision-making algorithm method starts with the detection of an obstacle by an ultrasonic sensor and calculates the distance in front of the suitcase, and we set the minimum distance of the drone near the obstacle to be 50 cm.

4. In addition to obstacle avoidance, the suitcase can implement algorithms for finding the shortest path to efficiently navigate around obstacles. Some possible algorithms include Dijkstra's algorithm, A-Star, Bi-Directional A-Star and Rapidly-exploring Random Tree (RRT).

5. The minimum distance to the owner and bypassing all obstacles when moving in different directions.

Experiment

To evaluate the performance of different pathfinding algorithms in a static environment, we conducted a comprehensive experiment using four popular algorithms: A*, BiA*, Dijkstra's algorithm, and RRT. The goal of the experiment was to compare the algorithms' effectiveness in finding optimal paths while navigating through a static obstacle environment.

For this experiment, we implemented the A*, BiA*, Dijkstra's, and RRT algorithms using a custom software framework developed in Python. The framework provided us with the necessary tools and functionality to execute the algorithms and analyze their performance.
In the experiment, each algorithm was provided with the same start and goal positions. The task assigned to the algorithms was to find the optimal path from the start position to the goal position while avoiding static obstacles present in the environment. The start and the end positions of maze can be found in the next figure.

```
maze = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 1, 0, 0, 1, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

start = (0, 0)
end = (9, 8)
```

Figure – The start and the end positions of maze

To create a realistic scenario, we set up a simulated environment using a grid-based map. The map was represented as a matrix, where obstacles were marked with the value "1" and non-obstacle areas with the value "0". This map served as the basis for the algorithms to navigate through and find the optimal paths. The experiment aimed to provide valuable insights into the strengths and weaknesses of each algorithm in a static environment. By comparing their performance metrics, such as path length, computation time, and efficiency, we could determine which algorithm was most suitable for real-world applications involving path planning in static obstacle environments.

It's important to note that the focus of the experiment was on static obstacles rather than dynamic ones. The algorithms were designed to navigate through the predefined static obstacles efficiently.

Additionally, we considered the starting and ending points of movement in order to decide on the most appropriate algorithm. We recognized that different algorithms may excel in different scenarios, and the choice of algorithm could depend on factors such as computational efficiency, accuracy, and adaptability to the environment.

Results of the experiment

The experiment provided valuable insights into the performance of A*, BiA*, Dijkstra's, and RRT algorithms in a static obstacle environment. The findings can be used to guide the selection of the most suitable algorithm for real-world applications involving path planning in similar environments.

Table 2 – ***Result of the experiment***

Algorithm	Obstacle type	Estimated time (sec)
A*	Static	20.26
BiA*	Static	30.46
Djiksta’s	Static	49.96
RRT	Static	130.34

The results of the experiment can be seen in table 2. According to the results of the experiment, the best algorithm is A* because it has the least estimated time. Also, this algorithm will be used in future realization of a smart suitcase.

Conclusion

In summary, our research focused on finding the best shortest path algorithm for smart suitcases in a static obstacle environment. We conducted a comprehensive experiment comparing the performance of four popular algorithms: A*, BiA*, Dijkstra's, and RRT. The objective was to identify the most effective algorithm for guiding smart suitcases through obstacle-rich scenarios.

In this paper, we provided an overview of the existing solutions and realizations of smart suitcases. Also, each algorithm was described as they were implemented as a Python program.

Furthermore, our research contributes to the broader field of pathfinding algorithms by providing empirical evidence and comparative analysis of the four algorithms in a static obstacle environment. The results offer insights into the strengths and limitations of each algorithm, which can be valuable for researchers and practitioners seeking to optimize path planning in various contexts.

It is important to note that our study focused specifically on a static obstacle environment. Future research could explore the performance of these algorithms in dynamic and unpredictable obstacle scenarios to further enhance the applicability of smart suitcases in real-world settings.

Overall, the outcomes of this study pave the way for advancements in smart suitcase technology and contribute to the ongoing pursuit of efficient and effective path planning algorithms for various applications.

References

1. Yang, C.-S., Zhang, B.-H., Wei, H.-W. and Lee ,W.-T. (2019). The Design of Smart Suitcase. Procc. 2019 IEEE International Conference on Consumer Electronics – Taiwan (ICCE-TW), Yilan, Taiwan, pp. 1–2, doi: 10.1109/ICCE-TW46550.2019.8991728.

2. Krishnan, P. L. S., Valli, R., Priya, R. and Pravinkumar, V. (2020). Smart Luggage Carrier system with Theft Prevention and Real Time Tracking Using Nano Arduino structure. Procc. International Conference on System, Computation, Automation and Networking (ICSCAN), Pondicherry, India, pp. 1–5, doi: 10.1109/ICSCAN49426.2020.9262445.

3. Jagadheeswaran, R., Arjunan, R., Balamurugan, N., Barath kumar, D. and Ramya, E. (2020). Luggage Theft Identification And Smart Lock Using Face Recognition. Procc. 6th International Conference on Advanced Computing and Communication Systems (ICACCS), Coimbatore, India, pp. 302–305, doi: 10.1109/ICACCS48705.2020.9074452.

4. Alpert, M., Onyshchenko, V. (2022). Recognition of Potholes with Neural Network Using Unmanned Ground Vehicles. In: Hu, Z., Dychka, I., Petoukhov, S., He, M. (eds) Advances in Computer Science for Engineering and Education. ICCSEEA 2022. Lecture Notes on Data Engineering and Communications Technologies, vol 134. Springer, Cham. https://doi.org/10.1007/978-3-031-04812-8_18

Стаття надійшла до редколегії 20.07.2023

Альперт Максим Іоганович
Аспірант кафедри інформаційних систем та технологій, факультет інформатики та обчислювальної техніки,
https://orcid.org/0000-0002-8938-1473
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», Київ
Онищенко Вікторія Валеріївна
Доктор технічних наук, професор, факультет інформатики та обчислювальної техніки,
https://orcid.org/0000-0002-3126-2260
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», Київ

ПОШУК НАЙКРАЩОГО АЛГОРИТМУ НАЙКОРОТШОГО ШЛЯХУ ДЛЯ РОЗУМНОЇ ВАЛІЗИ

Анотація. *Розумні валізи – це новий революційний вид аксесуарів для подорожей, які використовують сучасні технології для підвищення зручності та легкості подорожей. Ці валізи оснащені багатьма сучаснимих функціями, такими як підключення до інтернету, інфрачервоні датчики, вбудовані алгоритми для оминання перешкод і супровідний мобільний додаток, призначений для відстеження власника речей. Ключовим компонентом у реалізації цієї технології є вибір відповідного алгоритму для розрахунку найкоротших шляхів у складних умовах. Отже, вирізняють чотири основні класи алгоритмів, які можуть розглядатися як кандидати: алгоритм Дейкстри, алгоритм пошуку A*, двонаправлений алгоритм пошуку BiA* та алгоритм швидкозростаючого випадкового дерева. Кожен з них має свої переваги та обмеження щодо продуктивності, вимог до пам'яті та точності, які необхідно враховувати для ефективного виконання поставленого завдання. Крім того, ці розумні валізи оснащені інфрачервоними сенсорами, які дають змогу їм виявляти й обходити перешкоди на своєму шляху за допомогою інфрачервоних датчиків, що відбивають промені від сусідніх об'єктів. Базова інформація, зібрана датчиками, потім фільтрується за допомогою внутрішнього алгоритму, який визначає найкращий спосіб обходу перешкоди, що є безцінним, коли мова йде про тривалі подорожі. Отже, «розумні» валізи є передовою революційною тенденцією, яка, ймовірно, приверне увагу мандрівників усіх типів, які прагнуть до ефективності та зручності під час подорожей.*

Ключові слова: розумна валіза; найкоротший шлях; алгоритм Дейкстри; алгоритм пошуку A*; двонаправлений алгоритм пошуку BiA*; алгоритм швидкозростаючого випадкового дерева

Link to publication

APA Alpert, Maksym & Onyshchenko, Viktoriia. (2023). Finding the best shortest path algorithm for smart suitcase. *Management of Development of Complex Systems*, 55, 92–97, dx.doi.org\10.32347/2412-9933.2023.55.92-97.

ДСТУ Альперт М. І., Онищенко В. В. Пошук найкращого алгоритму найкоротшого шляху для розумної валізи. *Управління розвитком складних систем.* Київ, 2023. № 55. С. 92 – 97, dx.doi.org\10.32347/2412-9933.2023.55.92-97.