How Pathfinding Algorithm is Used To Solve Linxicon

Josh Reinhart Zidik - 13524048
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: joshr1710@gmail.com , 13524048@std.stei.itb.ac.id

***Abstract*— Linxicon is a single-player word-association game in which players must connect two given words through a chain of intermediate words, where each consecutive pair shares a semantic similarity score above 41%. This study investigates the application of three pathfinding algorithms, Uniform Cost Search (UCS), Greedy Best-First Search (GBFS), and A*, to automatically solve Linxicon puzzles by modeling the game as a dynamically generated graph, in which words serve as vertices and similarity-based connections serve as weighted edges. Word similarity is computed using cosine similarity between embedding vectors produced by the pretrained all-MiniLM-L6-v2 sentence-transformer model, matching the semantic comparison method used by the game itself. Results show that GBFS produces the fastest solutions with the fewest intermediate words but does not guarantee the highest similarity path, UCS yields the highest average similarity score at the cost of significantly higher computation time and explored nodes, and A* achieves a balance between the two, producing the second-fastest results while maintaining a high average similarity score. These findings suggest that A* offers the most practical tradeoff between solution quality and computational cost for solving Linxicon-style semantic pathfinding problems.**

***Keywords*—Linxicon, Pathfinding Algorithm, Uniform Cost Search, Greedy Best-First Search, A* Algorithm, Semantic Similarity, Word Embedding, Graph Search, Natural Language Processing**

I. INTRODUCTION

Linxicon is a single-player game developed by Trainwreck Labs and playable through web browser. The goal of this game is to create a path that connects two given words using the least amount of intermediate words. A connection between two words can be created if the similarity rate between them is above 41%. In this context, similarity between words is measured semantically based on their meaning. As an example, *king* and *queen* have a relatively high similarity score because their definitions are closely related. This game also has an extra challenge to create a path which average connection is as high as possible. [1]

Fig. 1. Example of Linxicon's initial state

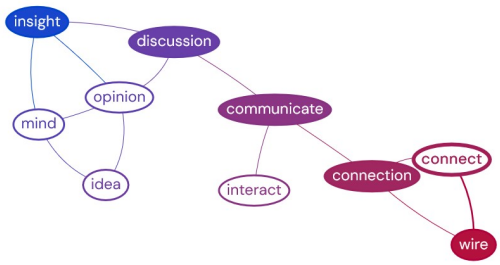


Fig. 2. Example of Linxicon's solved state

This paper explores the implementation of pathfinding algorithms to find the most optimal solution from any given initial state in the game *Linxicon*. The most optimal solution in this game is defined by the least amount of intermediate words required to form a valid path and the highest average similarity score between connected words. To address this problem, various pathfinding algorithms are implemented and evaluated based on their ability to efficiently navigate the search space and produce high-quality solutions.

II. THEORETICAL BASIS

A. Graph

A graph is a mathematical structure used to represent relationships between objects. Formally, a graph (G = (V, E)) consists of a set of vertices (or nodes) (V) and a set of edges (E) that connect pairs of vertices [2]. A path inside a graph is defined as a sequence of vertices where each consecutive pair is connected by an edge.

B. Route/Path Planning Algorithm

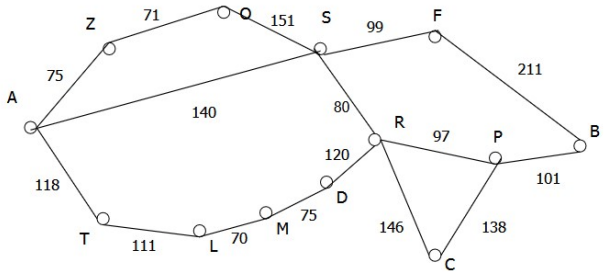
Path planning algorithms are methods used to find a path from an initial state to a goal state. These algorithms explore possible paths within a search space and determine a sequence of steps that can reach the desired destination. Path planning is widely used in fields such as artificial intelligence, robotics, navigation systems, and game solving [3].

Different path planning algorithms use different strategies to search for a solution. Some algorithms focus on finding a solution as quickly as possible, while others prioritize finding the most optimal path according to specific criteria, such as the shortest distance, lowest cost, or fewest number of steps [4].

Path planning algorithms can generally be divided into two categories: uninformed search and informed search. Uninformed search algorithms explore the search space without additional knowledge about the goal, whereas informed search algorithms use heuristic information to guide the search toward more promising paths. As a result, informed search methods can often find solutions more efficiently than uninformed methods [3][5].

C. Uniform Cost Search

Uniform Cost Search (UCS) is an uninformed search strategy that expands nodes in order of their cumulative path cost from the start node [6]. Unlike Breadth-First Search, which treats every edge as having equal weight, UCS accounts for varying edge costs by maintaining a priority queue ordered by the cost function g(n), where g(n) represents the total cost accumulated from the start node to node n [6]. At each iteration, the algorithm selects and expands the node with the lowest cumulative cost, ensuring that when the goal node is finally popped from the queue, the path found is guaranteed to be the lowest-cost path available [6].



Simpul-E	Simpul Hidup
A	Z_{A-75}, T_{A-118}, S_{A-140}
Z_{A-75}	T_{A-118}, S_{A-140}, O_{AZ-146}
T_{A-118}	S_{A-140}, O_{AZ-146}, L_{AT-229}
S_{A-140}	O_{AZ-146}, R_{AS-220}, L_{AT-229}, F_{AS-239}, O_{AS-291}
O_{AZ-146}	R_{AS-220}, L_{AT-229}, F_{AS-239}, O_{AS-291}
R_{AS-220}	L_{AT-229}, F_{AS-239}, O_{AS-291}, $P_{ASR-317}$, $D_{ASR-340}$, $C_{ASR-366}$
L_{AT-229}	F_{AS-239}, O_{AS-291}, $M_{ATL-299}$, $P_{ASR-317}$, $D_{ASR-340}$, $C_{ASR-366}$
F_{AS-239}	O_{AS-291}, $M_{ATL-299}$, $P_{ASR-317}$, $D_{ASR-340}$, $C_{ASR-366}$, $B_{ASF-450}$
O_{AS-291}	$M_{ATL-299}$, $P_{ASR-317}$, $D_{ASR-340}$, $C_{ASR-366}$, $B_{ASF-450}$

Fig. 3. Example of Priority Queue used in UCS
Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/21-Route-Planning-(2026)-Bagian1.pdf, accessed on: 17 June 2026

This guarantee holds as long as all edge costs are non-negative, since a negative-cost edge could allow a cheaper path to be discovered after the goal has already been expanded. Because UCS does not use any estimate of distance to the goal, it must explore every node whose cumulative cost is lower than the goal's eventual cost, which can make it computationally expensive on graphs with many low-cost branches, even though it remains both complete and optimal under the non-negative cost assumption [6].

D. Greedy Best-First Search

Greedy Best-First Search is an informed search strategy that selects the next node to expand based solely on a heuristic evaluation function h(n), which estimates the cost from the current node to the goal [6]. Rather than considering the cost already incurred to reach a node, GBFS expands whichever node in the frontier appears closest to the goal according to this heuristic, making it a purely "greedy" approach to pathfinding [6]. A commonly used heuristic in route-planning contexts is the straight-line distance between a node and the goal, since this distance never requires traversing the actual graph and can be computed independently of the search process.

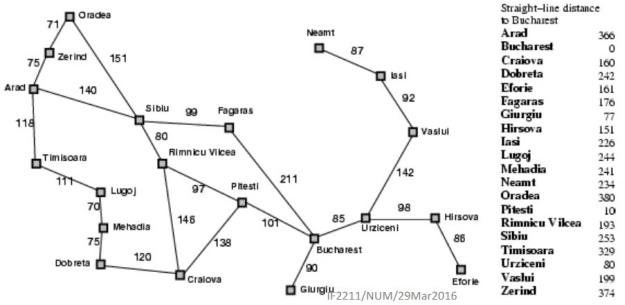


Fig. 4. Example of GBFS heuristic calculations
Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/21-Route-Planning-(2026)-Bagian1.pdf, accessed on: 17 June 2026

While this strategy often finds solutions quickly by aggressively pursuing the most promising-looking direction, it suffers from two significant drawbacks: it is not guaranteed to find the optimal path, since the heuristic only reflects estimated remaining distance rather than actual cost already spent, and it is not complete in certain graph configurations, where the algorithm may become trapped following a heuristic that consistently points toward a path away from the true goal [6]. As such, GBFS trades correctness guarantees for speed, performing well when the heuristic is reasonably accurate but potentially yielding suboptimal or even failing results otherwise [6].

E. A Algorithm*

A* Search combines the strengths of both Uniform Cost Search and Greedy Best-First Search by evaluating nodes using the function f(n) = g(n) + h(n), where g(n) is the cumulative cost from the start node and h(n) is the heuristic estimate of the remaining cost to the goal [7]. By incorporating both the cost already incurred and an estimate of the cost still to come, A* directs its search toward the goal more efficiently than UCS while avoiding the shortsightedness of pure GBFS [7]. The correctness of A* depends critically on the admissibility of its heuristic: a heuristic is considered admissible if it never overestimates the true cost required to reach the goal from any given node [7]. When this condition holds, A* is guaranteed to find the optimal path, since the algorithm will never prematurely commit to a node whose true cost turns out to be higher than an alternative path that was undervalued by an inadmissible heuristic [7]. In route-finding problems specifically, the straight-line distance between two points serves as a natural admissible heuristic, as it can never exceed the actual distance along any real path through the graph, which may include detours, turns, or obstacles [7]. This balance of cost-awareness and goal-directed estimation makes A* both complete and optimal under admissible heuristic conditions, while typically expanding far fewer nodes than UCS alone.

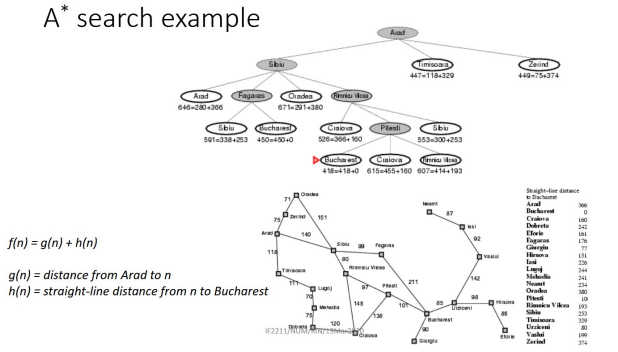


Fig. 5. Example of A* calculations
Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/22-Route-Planning-(2026)-Bagian2.pdf, accessed on: 17 June 2026

III. IMPLEMENTATION

A. Solution Design

To implement pathfinding algorithms in Linxicon, the search space is first represented as a graph. Each word corresponds to a vertex, and each valid connection between two words corresponds to an edge. The similarity score between the connected words is used as the edge weight, this score must be inverted because the higher the score the lower the weight between vertices.

However, the English language contains an enormous amount of distinct words, making it impractical to construct the entire graph beforehand. Since every word would need to be represented as a vertex and every valid similarity relationship as an edge, generating the complete graph would require a massive amount of memory and computation time. Furthermore, many of these vertices and edges would never be explored during the search process.

To address this issue, the graph is generated dynamically rather than being constructed in its entirety. Instead of modeling all possible vertices and edges at the beginning, only the vertices that are required during the search are expanded. When a pathfinding algorithm visits a word, its neighboring words are generated on demand based on the game's similarity criteria. This approach significantly reduces memory usage and avoids unnecessary computations, allowing the algorithms to focus only on the portion of the search space that is relevant to finding a solution.

This graph representation enables pathfinding algorithms to search for optimal paths between the starting and target words. The 3 pathfinding algorithms used in this implementation are Uniform Cost Search (UCS), Greedy Best-First Search (GBFS), and A* algorithm. These algorithms will be tested for its result and computing efiiciency.

B. Algorithm Implementation

The solution design above is implemented using the Python programming language. Python was chosen because of its extensive natural language processing libraries and models, which accommodate the computation of semantic similarity between words. These tools provide access to pre-trained language models capable of representing words as numerical vectors, allowing similarity scores to be calculated efficiently.

1) Word Embedding and Cost Calculation

In order to calculate the semantic similarity between words, the algorithm first represents words as embedding vectors. These embedding vectors are generated by a pretrained *all-MiniLM-L6-v2* sentence-transformer model. This model is chosen to match with how Linxicon calculate its similarity rates [1]. The similarity between two words are determined by the angle between their two vector representation, which can be calculated using cosine product of the two vectors, which later is named *cosine similarity*.

The entire vocabulary used in this algorithm is sourced from the NLTK English word corpus and filtered to words between 3 and 15 characters, again, to match the vocabulary

used in the game. The vocabulary is encoded once into a 384-dimensional embedding matrix (into the vocab_embeddings variable) and cached to disk (vocab_embeddings.npy and vocab_words.npy stored in the same directory) alongside a word_to_index dictionary mapping each word to its row position. This precomputation step eliminates the need to repeatedly call the language model during pathfinding, since every subsequent similarity lookup is reduced to a direct array index and a cosine-similarity computation rather than a fresh model call.

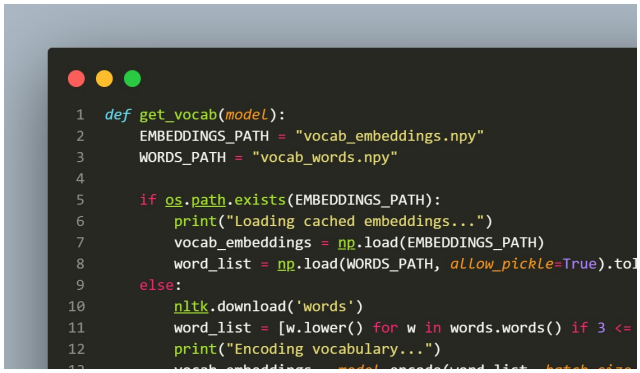


Fig. 6. Function to preprocess the natural language model

Since the graph is constructed dynamically, it's necessary to make a function which generates all word within the similarity threshold from a single word, each generated word from this function is connected to the main word. This function, named get_linked_words, computes the cosine similarity between a single word and the entire vocabulary matrix in one vectorized operation then filters the result above a fixed similarity threshold of 0,41. This constant is chosen to match the threshold used by the Linxicon game itself. The result of this function is a list of word sorted by similarity in descending order.

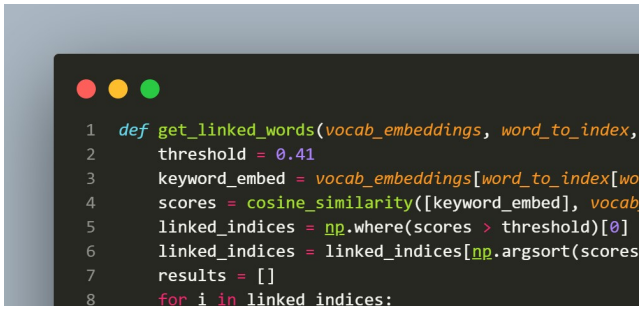


Fig. 7. Function to generate connected words

Another function worth mentioning is get_similarity, this function calculates the cosine similarity between two input words. The return of this function is a float that is the representation of edge cost between the two words.

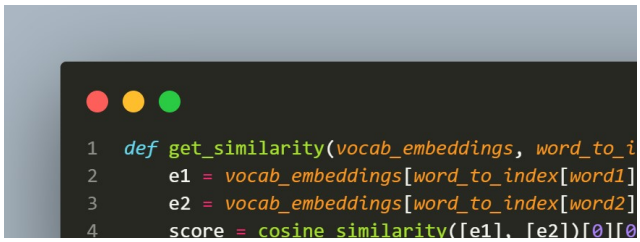


Fig. 8. Function to calculate word similarity

Because edge cost in this problem is inversely related to similarity, every algorithm defines the cost of traversing from one word to a linked word as 1 - similarity_score, so that more similar word pairs correspond to cheaper edges.

2) *Uniform Cost Search Implementation*

The UCS implementation (ucs.py) maintains a min-heap priority queue of (cumulative_cost, word) tuples and a *visited* dictionary mapping each discovered word to its best known (cost, parent) pair. On each iteration, the node with the lowest cumulative cost is popped from the heap, if the popped word matches the goal, the path is reconstructed using make_path function. This function works by walking backward through parent references stored in *visited* dictionary. Otherwise, the algorithm retrieves up to the top 20 linked words using the function get_linked_words, computes each neighbor's new cumulative cost, and pushes any word whose cost improves on its previously recorded value back onto the heap. Limiting expansion to the top 20 neighbors per node serves to bound the branching factor of the search, avoiding potential computation overload.

Fig. 9. UCS implementation and helper function

The use of heapq in this implementation is to effectively pop and pushes entry into a list while keeping it sorted. This is the main reason why priority queue tuple has cost as its first element, heapq orders the list based on the first element in a tuple. This sorted pop and push helps reducing the complexity, avoiding repeatedly sorting the list for each iteration.

3) *Greedy Best-First Search Implementation*

The GBFS implementation (gbfs.py) is similar to UCS implementation with one of the key difference, that is the ordering of its priority queue. GBFS implementation orders the priority queue purely by heuristic value rather than accumulated cost. Each queue entry stores (heuristic_value, word, path), where the heuristic is computed as 1 – get_similarity(word, goal). The full path to each node is carried directly inside the queue entry, so the return doesn't need a path-reconstruction helper function. A *visited* set prevents re-expansion of already-explored words. Because the algorithm never accounts for the cost already incurred to reach a node, it greedily pursues each neighbor that appears to be the closest one to the goal at each step, which can result in fast computation. However, the result offers no guarantee of finding the lowest-cost path.

Fig. 10. GBFS implementation

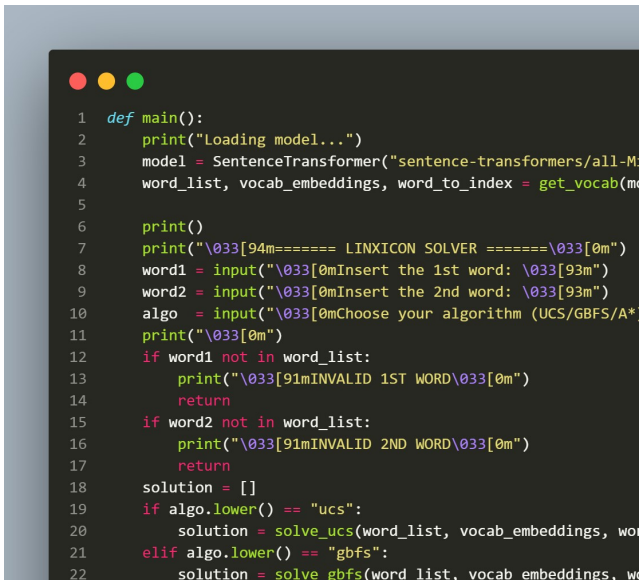
4) *A* Algorithm Implementation*

The A* implementation (astar.py) reuses the UCS implementation but changes the priority queue ordering by adding heuristic component. Priority queue is ordered by equation f = g + h where g is the immediate cost and h is the heuristic component. A separate h_cost dictionary caches each word's heuristic value, this dictionary is computed once using the get_similarity function, so that repeated heuristic lookups for the same word avoid redundant computation. Since the heap stores combined f values rather than raw cumulative cost, the true cost g for a popped node is recovered by subtracting its cached heuristic (cur_cost = cur_f - h_cost[cur_node]), which is then used both for the staleness check against visited and for computing neighbor costs. Similar to the UCS implementation, neighbor expansion is capped at the top 20 linked words per node, and path reconstruction reuses the same make_path function. By combining accumulated cost with a goal-directed heuristic, this implementation is designed to explore fewer nodes than UCS while still aiming to preserve solution optimality, contingent on the admissibility of the chosen similarity-based heuristic.

Fig. 11. A* implementation

5) *Program Flow*

The entry point (linxicon_solver.py) loads the all-MiniLM-L6-v2 model in CPU-only, offline mode to avoid unnecessary network calls, builds or loads the cached vocabulary embeddings via get_vocab, and prompts the user for a start word, goal word, and algorithm choice (UCS, GBFS, or A*). Both input words are validated against the vocabulary before dispatching to the corresponding solver function. Once a solution path is returned, print_solution displays the sequence of words along with the pairwise similarity score between each consecutive pair in the resulting path.



```
def main():
    print("Loading model...")
    model = SentenceTransformer("sentence-transformers/all-M
    word_list, vocab_embeddings, word_to_index = get_vocab(m

    print()
    print("\033[94m======= LINXICON SOLVER =======\033[0m")
    word1 = input("\033[0mInsert the 1st word: \033[93m")
    word2 = input("\033[0mInsert the 2nd word: \033[93m")
    algo  = input("\033[0mChoose your algorithm (UCS/GBFS/A*
    print("\033[0m")
    if word1 not in word_list:
        print("\033[91mINVALID 1ST WORD\033[0m")
        return
    if word2 not in word_list:
        print("\033[91mINVALID 2ND WORD\033[0m")
        return
    solution = []
    if algo.lower() == "ucs":
        solution = solve_ucs(word_list, vocab_embeddings, wo
    elif algo.lower() == "gbfs":
        solution = solve_gbfs(word_list, vocab_embeddings, w
```

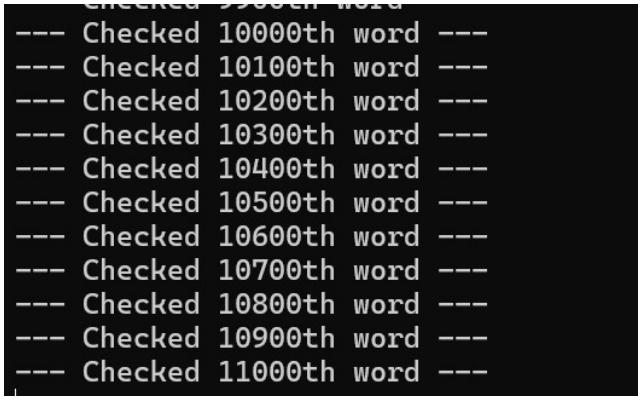
Fig. 12. The main program

IV. Results and Discussion

A. Uniform Cost Search Result

1) Testcase Double - Tail

This testcase uses the combination of word *double* and *tail*, taken from the official Linxicon game #857. Figure below shows the result of pathfinding using UCS implementation



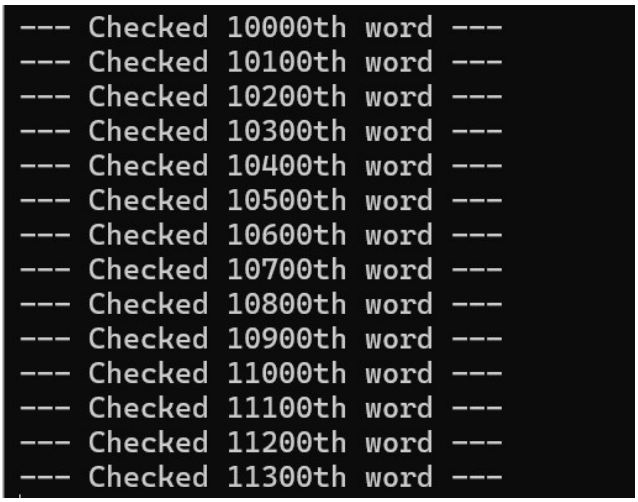
```
--- Checked 10000th word ---
--- Checked 10100th word ---
--- Checked 10200th word ---
--- Checked 10300th word ---
--- Checked 10400th word ---
--- Checked 10500th word ---
--- Checked 10600th word ---
--- Checked 10700th word ---
--- Checked 10800th word ---
--- Checked 10900th word ---
--- Checked 11000th word ---
```

Fig. 13. UCS Testcase Double - Tail

The first test case was not completed by UCS because the search exceeded 10,000 word inspections, resulting in an impractically long computation time.

2) Testcase Hair – Join

This testcase uses the combination of word *hair* and *join* taken from the official Linxicon game #858. Figure below shows the result of pathfinding using UCS implementation.



```
--- Checked 10000th word ---
--- Checked 10100th word ---
--- Checked 10200th word ---
--- Checked 10300th word ---
--- Checked 10400th word ---
--- Checked 10500th word ---
--- Checked 10600th word ---
--- Checked 10700th word ---
--- Checked 10800th word ---
--- Checked 10900th word ---
--- Checked 11000th word ---
--- Checked 11100th word ---
--- Checked 11200th word ---
--- Checked 11300th word ---
```

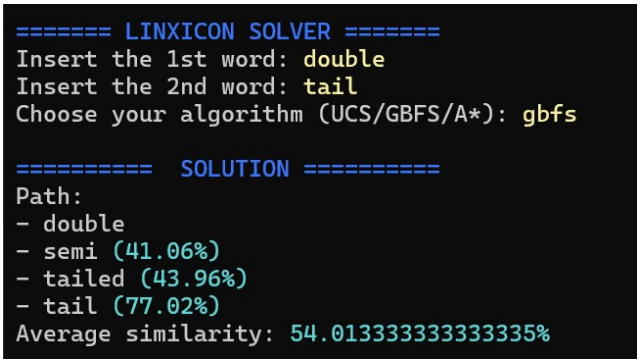
Fig. 14. UCS Testcase Hair - Join

The second test case was not completed by UCS because the search exceeded 10,000 word inspections, resulting in an impractically long computation time.

B. Greedy Best-First Search Result

1) Testcase Double - Tail

This testcase uses the combination of word *double* and *tail*, taken from the official Linxicon game #857. Figure below shows the result of pathfinding using GBFS implementation.



```
======= LINXICON SOLVER =======
Insert the 1st word: double
Insert the 2nd word: tail
Choose your algorithm (UCS/GBFS/A*): gbfs

==========  SOLUTION ==========
Path:
- double
- semi (41.06%)
- tailed (43.96%)
- tail (77.02%)
Average similarity: 54.013333333333335%
```

Fig. 15. GBFS Testcase Double - Tail

2) Testcase Hair - Join

This testcase uses the combination of word *hair* and *join* taken from the official Linxicon game #858. Figure below shows the result of pathfinding using GBFS implementation.

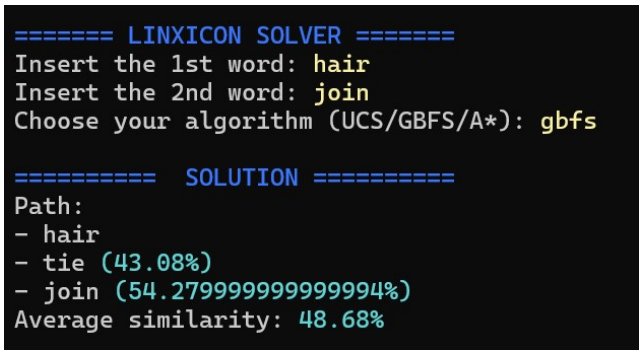


Fig. 16. GBFS Testcase Hair – Join

3) Testcase Medicine – Reporter

This testcase uses the combination of word *medicine* and *reporter*. Figure below shows the result of pathfinding using GBFS implementation.

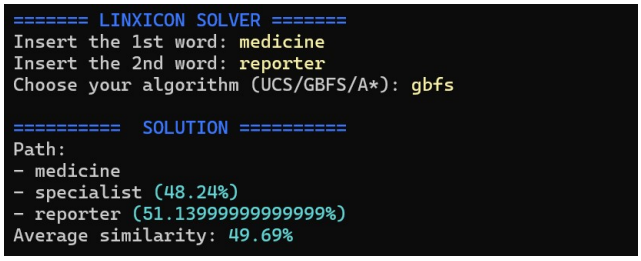


Fig. 17. GBFS Testcase Medicine - Reporter

C. A Result*

1) Testcase Double - Tail

This testcase uses the combination of word *double* and *tail*, taken from the official Linxicon game #857. Figure below shows the result of pathfinding using A* implementation

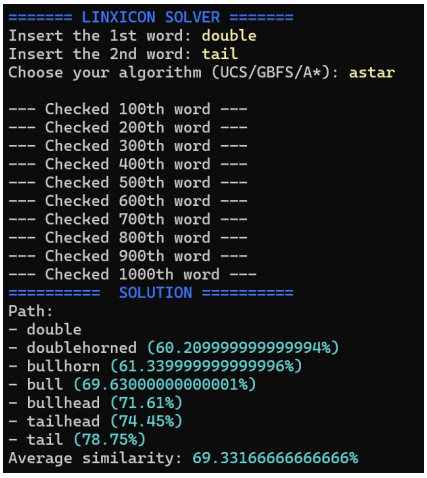


Fig. 18. A* Testcase Double - Tail

2) Testcase Hair - Join

This testcase uses the combination of word *hair* and *join* taken from the official Linxicon game #858. Figure below shows the result of pathfinding using A* implementation.

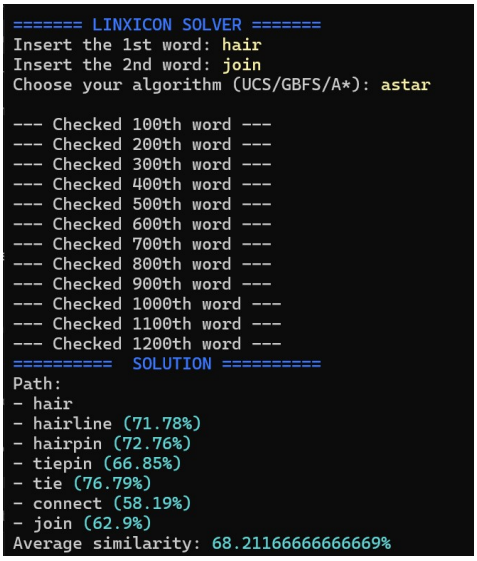


Fig. 19. A* Testcase Hair – Join

3) Testcase Medicine – Reporter

This testcase uses the combination of word *medicine* and *reporter*. Figure below shows the result of pathfinding using A* implementation.

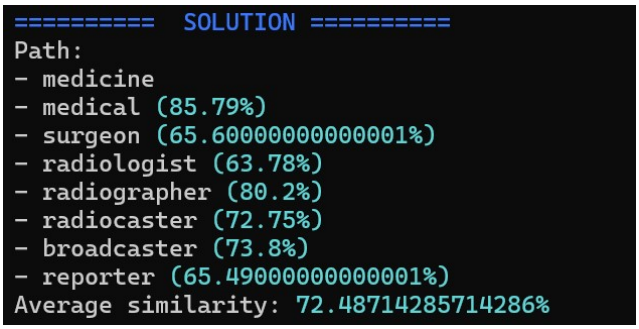


Fig. 20. A* Testcase Medicine - Reporter

D. Discussion

From the test cases used in the three algorithm, GBFS produced the fastest result and also the result with the least amount of words. This is consistent with the behavior of GBFS which searches for the word with the smallest weight, or in another word, the closest similarity rate to the goal node, instead of the starting node.

Different from GBFS, A* algorithm produced the second fastest result but with higher average similarity rate between words. Again, this is consistent with the behavior of A* which not only accounts similarity rate from current node, but also heuristic calculation, which is the similarity rate to the goal

node. The tradeoff to this algorithm is the significant increase in explored node.

The worst algorithm in this study turns out to be UCS due to its ineffectiveness in exploring the search space, this algorithm produced the slowest result even though it gives the highest average similarity rate. This is due to the behavior of UCS algorithm which doesn't expand towards the goal node immediately but instead expands BFS-like to maintain the lowest-cost path. This algorithm becomes extremely ineffective when met with a huge and complex graph, reaching to thousands of explored nodes before getting to the goal node.

V. Conclusion

Linxicon is a game about connecting two unrelated words with a sequence of semantically similar words. This game challenges its player to find the fewest amount of word and to find the path with highest average similarity rate. The solution to said challenges can be computed using the pathfinding algorithm by representing the game as a graph. The game is first converted into a graph, by turning each word into a vertex, each connection into an edge, and each similarity rate into an edge weight/cost. The pathfinding used in this study is Uniform Cost Search (UCS), Greedy Best-First Search (GBFS), and A* Algorithm, each algorithm makes a path from the starting word to the goal word. The test cases shows that UCS solves the game the slowest, while GBFS solves the game the fastest. The result that UCS gives tend to be much longer because keeping the average similarity rate high, while the result that GBFS gives is much shorter because of its goal to find the least amount of words used. On the other hand, A* algorithm being the combination of both other algorithms becomes the second place in both solving the game and giving the highest average similarity path. A* in this case can be a replacement to UCS to avoid high computation price while keeping the result's average similarity score high.

VI. Appendix

Github Repository

https://github.com/Achideon/Linxicon-Solver

VII. Acknowledgment

The author of this paper would like to express his gratitude to God for His blessings that help the writing of this paper. The author would also like to express his gratitude to his parents and family who have been supportive during author's study in the college and also during the writing of this paper. Additionally, the author would like to thank the lecturer Dr. Ir. Rinaldi Munir, M.T.. for all the subject he has taught the author during the third semester. The author would also thank the help of generative AI during the writing of this paper for its grammatical fixes, code debugs, and proofreading. Last but not least, the author would like to express his deepest gratitude to his friends who's always around him during the brightest or even the darkest of times.

VIII. References

[1] https://linxicon.com/faq [Accessed: 17 Jun 2026]

[2] Bondy, J. A., & Murty, U. S. R. (2008). *Graph theory*. Springer. https://doi.org/10.1007/978-1-84628-970-5

[3] Russell, S. J., & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.

[4] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.). MIT Press.

[5] LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press. https://lavalle.pl/planning/

[6] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/21-Route-Planning-(2026)-Bagian1.pdf [Accessed: 17 Jun. 2026]

[7] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/22-Route-Planning-(2026)-Bagian2.pdf [Accessed: 17 Jun. 2026]

[8] B. Thaddeus "Feature-Based Image Metamorphosis" available online: https://www.cs.princeton.edu/courses/archive/fall00/cs426/papers/beier92.pdf, [Accessed: 24 Dec. 2025]

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2025

Josh Reinhart Zidik