Heuristic-Guided Pipe Routing Under Real-Time Constraints in BioShock's Hacking Minigame

A Comparative Analysis of A* and Greedy Best-First Search Algorithms on a Pipe-Routing Puzzle Simulation

Kloce Paul William Saragih - 13524040
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: kloce.edu@gmail.com , 13524040@std.stei.itb.ac.id

Abstract—The hacking minigame in BioShock gives the player a pipe-routing puzzle that must be solved within a limited amount of time, while a current of "electricity" that represents the hacking signal keeps moving forward and locks every tile it has already passed through. This real-time property makes the puzzle worth studying from the standpoint of algorithm strategy, especially in the context of route search on a graph under time pressure. This paper models the BioShock hacking puzzle as a route-search problem on a grid, then compares two well-known heuristic search algorithms, A* and Greedy Best-First Search (GBFS), in solving it. Both algorithms are implemented in two separate programs that interact through a query mechanism, representing the separation between the solver and the puzzle engine. Testing was carried out on grids ranging from 10×10 to 100×100 with a locked-tile ratio of 15 percent. The experimental results show that A* consistently produces the optimal route in every test case, but expands far more nodes than GBFS, especially on larger grids. GBFS, on the other hand, is better in terms of time efficiency and the number of nodes explored, at the cost of producing a route that is slightly longer than the optimal solution. This finding confirms the classic trade-off between optimality and computational efficiency, which becomes an important consideration when computing power is plentiful but the time limit still decides whether the puzzle gets solved.

Keywords— A*, Greedy Best-First Search, heuristic search, pipe-routing puzzle, BioShock.

I. INTRODUCTION

A. Background

Video games are often an unexpected source of inspiration for the study of algorithm strategy, because behind the visuals and the entertainment value, many game mechanics are really just classic computational problems in disguise. One example worth studying is the hacking minigame in the BioShock series (2K Games, 2007), where the player is given a panel full of pipe tiles that must be arranged so they form a connected path from a starting point (source) to an end point (sink). What sets this puzzle apart from an ordinary pipe puzzle is the real-time element: once the game starts, a flow of "electricity" moves on its own across the tiles that are already connected, and every tile that the current has passed through gets locked permanently and can no longer be swapped.

This property makes the BioShock hacking minigame a good case study for time-limited pathfinding. Rather than treating it as just a game mechanic, this paper tries to formalize the puzzle as a graph search problem and then solve it using two heuristic search algorithms that are widely known in algorithm strategy, namely A* and Greedy Best-First Search.

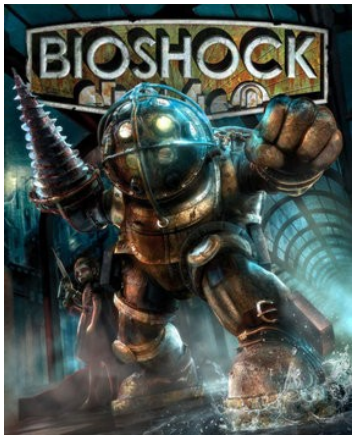

Figure 1. Bioshock Video Game Cover

The main idea this paper wants to explore starts from a simple question: since modern computers run far faster than human reaction time, is a strategy that "sacrifices" optimality for speed (like GBFS) still worth using, or is a strategy that guarantees an optimal solution (like A*) still the better choice, given that a computer can process it in a few milliseconds anyway? To answer this, the paper presents a progression of solutions starting from a naive approach, refined step by step through a series of observations, before finally comparing the two heuristic algorithm variants head-to-head.

B. Problem Statement

Based on this background, the paper tries to answer the following research questions:

- How can the BioShock hacking puzzle be formalized as a graph route-search problem that can be solved using algorithm strategy techniques?
- How can a two-program architecture (solver and puzzle engine) that interacts through a query mechanism be designed so it reflects a real condition where the solver does not have full visibility of the entire board from the start?

- How much do A* and Greedy Best-First Search differ in solution optimality, number of nodes expanded, and execution time when applied to puzzles of different grid sizes?

C. Objectives

The goal of this paper is to design and build a simulation of the BioShock hacking puzzle that reflects a real time constraint, to build a solver based on heuristic search algorithms using a two-program architecture that communicates through queries, and to compare the performance of A* and Greedy Best-First Search based on experimental data gathered from that simulation.

D. Scope and Limitations

This paper limits its scope to a few things. First, only two types of pipe tiles are considered: straight tiles and elbow tiles, without modeling more complex tile variants such as cross tiles or tiles with three connections at once nor the blocking tiles that is available in the game. Second, the movement of the electricity is simplified into a tile-locking mechanism based on a random ratio, rather than a millisecond-by-millisecond real-time simulation, since the main focus of this paper is comparing route-search strategies, not precisely simulating the physics of an electrical current. Third, the grid sizes tested are limited because of time constraints on running the experiments, even though in practice the algorithms discussed here can be applied to larger sizes as well.

II. Theoritical Background

A. Graph

A graph G is defined as an ordered pair G = (V, E), where V is the set of vertices and E is the set of edges connecting two vertices. The puzzle grid is represented as a graph with uniform weights, where each cell on the grid acts as a vertex, and edges connect cells that are neighbors horizontally or vertically. Since moving between two neighboring cells always costs the same (one step), the resulting graph is uniformly unweighted, so the distance between two vertices can be estimated using the Manhattan distance.

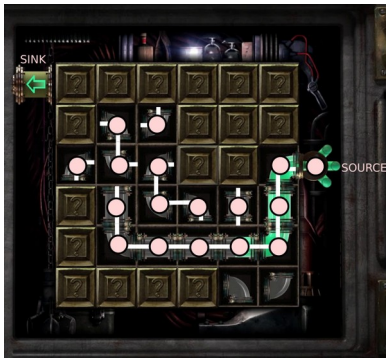

Figure 2. Representation of the puzzle grid as a graph with 4-directional connectivity

B. Uninformed Search Algorithms

Algorithms such as Breadth-First Search (BFS) and Depth-First Search (DFS). BFS guarantees an optimal solution on a uniformly weighted graph because it explores nodes level by level, but it does not use any information about where the target actually is, so it tends to explore many nodes that have nothing to do with the direction of the goal. This limitation is what motivates the use of informed search algorithms, which use a heuristic function to steer the exploration toward the target more efficiently.

C. Heuristic Function and Manhattan Distance

The heuristic function h(n) is an estimate of the cost from a node n to the goal node. On a grid with movement restricted to four directions (no diagonals), the Manhattan distance is an admissible heuristic, meaning its estimate never exceeds the actual cost of reaching the goal. The Manhattan distance between two points (r1, c1) and (r2, c2) is defined as:

$$h(n) = |r1 - r2| + |c1 - c2|$$

This admissible property matters because it is a precondition for A* to guarantee an optimal solution. An admissible heuristic will never make the algorithm skip a path that is actually shorter, since its estimate is never "too optimistic" compared to reality.

D. A* Algorithm

A* is an informed search algorithm that evaluates every node n using the evaluation function:

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

where g(n) is the actual cost from the start node to node n, and h(n) is the heuristic estimate from node n to the goal. A* keeps a priority queue (usually implemented as a min-heap) that always pops the node with the smallest f(n) value to expand next. Because it accounts for both the cost already spent (g) and the estimated remaining cost (h), A* guarantees an optimal solution as long as the heuristic used is admissible, as discussed in the previous subsection.

E. Greedy Best First Search (GBFS) Algorithm

Greedy Best-First Search only looks at the heuristic value when deciding which node to expand next:

$$f(n) = h(n)$$

By ignoring the g(n) term, GBFS is "greedy" in the sense that it always picks the node that looks closest to the goal by estimate, without caring how much it already cost to reach that node. As a result, GBFS usually expands far fewer nodes than A*, but it does not guarantee an optimal solution, because it can get stuck following a path that looks promising locally but turns out not to be the shortest path overall.

F. Constraint Satisfaction Problem (CSP)

Besides being a route-search problem, the pipe puzzle can also be formalized as a Constraint Satisfaction Problem, with each tile as a variable, the set of possible swappings as its domain, and the matching of ports between neighboring tiles as the constraint. This view is useful for understanding why the locking of tiles by the electrical current can be seen as a process of progressively shrinking the domain, similar to forward checking in CSP, although the main focus of this paper stays on the route-search formulation, while the CSP formulation is used as a supporting conceptual frame to explain how the tile-locking dynamic works.

III. Methodology

A. Problem Formalization

The BioShock hacking puzzle is formalized as a grid of size n × m, with one cell set as the source and another cell set as the sink. Each cell on the grid has either a locked or unlocked status, depending on whether the electrical current has already passed through that tile (locked) or not (unlocked). A locked tile has a fixed orientation and can only be passed through if that orientation already matches the direction the path needs; an unlocked tile, on the other hand, is assumed to still be shapeable by the solver as needed, so it can always be passed through during the route search.

The solver's goal is to find a path from source to sink as fast and/or as optimally as possible before the electrical current locks the tiles the path needs. Because a computer runs far faster than the electricity moves in the original game (which runs at around hundreds of milliseconds per tile, mainly so a human player can keep up), this simulation assumes the solver has a huge time advantage. That makes the more useful question not "can the solver find a solution in time" but "which strategy gives the best balance between computation speed and solution quality", especially once the puzzle size grows far beyond what is typical in the original game (which is 6x6).

B. Heuristic Function and Manhattan Distance

As planned from the start, the system is built as two conceptually separate programs, the puzzle engine and the solver, communicating through a query mechanism. This separation matters because it reflects a real condition where the solver does not automatically know everything on the board from the start, but instead has to explicitly ask for information. Two main types of query are defined:

- **QUERY_TILE(r, c):** asks the puzzle engine to reveal the tile type at position (r, c). The engine returns the pipe type (e.g., straight-vertical, elbow-up-right). This action is rejected if the tile is already known to the solver. The engine automatically advances the electrical current after each action.
- **SWAP_TILE(A, B):** asks the puzzle engine to exchange the tile contents at positions A = (r1, c1) and B = (r2, c2), which may be any two positions on the grid adjacency is not required. This request is rejected if either tile has not yet been queried by the solver, or if either tile is already locked by the electrical current.

In the simulation implementation discussed in Section IV, both types of query are simplified into direct function calls within the same program, since the main goal of the experiment is to measure search algorithm performance, not the overhead of inter-process communication. Even so, the code keeps a logical separation between the grid representation (puzzle engine) and the search logic (solver), so this architecture could still be extended into actual inter-process communication (for example through a socket or a pipe) without changing the core algorithm logic.

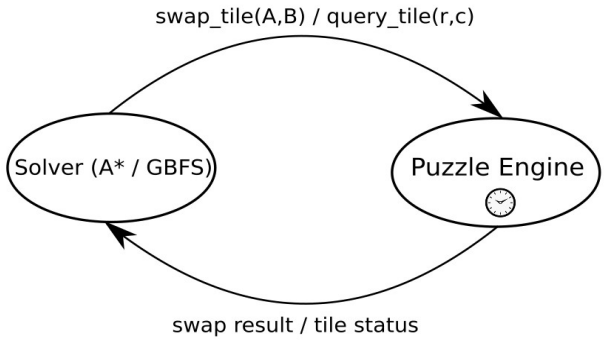

Figure 3. Communication architecture between the solver and the puzzle engine through the query mechanism

C. Incremenetal Solution : Naive to Heuristic

Following the usual problem-solving approach the solution in this paper is developed in stages, starting from the simplest approach and then refined through a series of observations.

Finally, complete content and organizational editing before formatting. Please take note of the following items when proofreading spelling and grammar:

D. Solution I : Exhaustive Searach Without Directional Information

The most naive way to solve this puzzle is to run plain BFS from the source, exploring every neighboring node equally without caring which direction the sink is in. This approach guarantees an optimal solution (since all edges have the same

weight), but on a large grid, BFS ends up expanding many nodes that are actually moving away from the sink, wasting both computation time and the number of queries the solver has to send to the puzzle engine. On an n × n grid, BFS can expand up to $O(n^2)$ nodes in the worst case before reaching the sink, even though the actual shortest path is only O(n) long.

E. Solution II: Adding Directional Information

The first observation is that the sink's position is always known from the start of the game (the player can see the whole board, but cannot now the type without querying of tiles the electricity has not reached yet). So instead of exploring nodes evenly like BFS, the solver can prioritize exploring nodes that are estimated to be closer to the sink. This observation leads to using a heuristic function h(n) in the form of the Manhattan distance, as explained in Section II.D, and forms the basis for the two algorithms compared in this paper, A* and GBFS.

F. Solution III : Heuritics with A and GBFS*

The second observation comes from a different question: does the solver actually need to keep track of the cost already spent (g(n)), or is it enough to rely on the remaining-distance estimate (h(n)) alone? This question is exactly what separates A* (which considers g(n) + h(n)) from GBFS (which only considers h(n)). Intuitively, keeping g(n) guarantees optimality, but at the cost of having the solver explore more nodes just to make sure no cheaper path exists. Ignoring g(n), on the other hand, makes the solver more "bold" in deciding based on estimate alone, which is faster but risks a suboptimal solution if the heuristic estimate steers the search the wrong way (for example, when a locked tile forces the path to detour far from a straight line toward the sink).

This observation is the core of the paper's contribution: not just applying a heuristic, but explicitly comparing two different ways of using that heuristic, so the real trade-off between optimality and computational efficiency can be seen clearly.

G. Tile Locking Simulation

To represent the tile-locking mechanism of the electrical current in a way that is realistic but still testable under controlled conditions, the simulation in this paper uses a locked-tile ratio parameter, that is, the proportion of tiles on the grid set as locked before the solver starts working, each with a random orientation that can no longer be changed. This is conceptually the same as simulating a situation where the current has already traveled some distance before the solver gets a chance to react, a scenario worth considering, since in a real setting the solver does not always get to start computing from the very first frame of the game.

The puzzle generator in this simulation first builds one random path that is guaranteed to be valid from source to sink (using a random walk biased toward moving forward to the sink), making sure the tiles along that path are not locked, then fills the rest of the grid with randomly typed tiles (straight or elbow) at random rotations, some of which are locked according to the given ratio. This way, every test puzzle is guaranteed to be solvable, so the success rate measured in the experiment reflects how good the algorithm is, rather than the algorithm just happening to run into a puzzle that is actually impossible to solve

H. Implementation Environment

The whole simulation is implemented in C++17, compiled with g++ on Ubuntu 22.04. The priority queue used by both algorithms is implemented with std::priority_queue and a custom comparator for min-heap operation based on the f(n) value. Execution time is measured using the <chrono>; library at microsecond resolution, then converted to milliseconds for reporting.

IV. Experiments and Analysis

A. Testing

Testing was done on six square grid sizes: 5×5, 8×8, 10×10, 12×12, 15×15, and 20×20, with a locked-tile ratio of 15% in every scenario. Each grid size was tested 100 times (trials) with puzzles generated randomly but guaranteed solvable, and the reported results are the average across all those trials to reduce the effect of random noise on the conclusions drawn. All experiments were run with a fixed random seed so that the results can be reproduced.

Grid	Algo	Waktu (ms)	Nodes Expanded	Avg Turns	Avg Swaps	Avg Queries	Path Len
5×5	A*	0.017	610	26.4	8	18.4	9
5×5	GBFS	0.017	610	26.4	8	18.4	9
8×8	A*	0.069	3,934	51.9	14	37.9	15
8×8	GBFS	0.065	3,934	51.9	14	37.9	15
10×10	A*	0.141	9,387	69.2	18	51.2	19
10×10	GBFS	0.13	9,387	69.2	18	51.2	19
12×12	A*	0.226	17,466	85.4	22	63.4	23
12×12	GBFS	0.225	17,466	85.4	22	63.4	23
15×15	A*	0.461	38,942	113.9	28	85.9	29
15×15	GBFS	0.447	38,942	113.9	28	85.9	29
20×20	A*	1.172	105,096	157.9	38	119.9	39
20×20	GBFS	1.14	105,096	157.9	38	119.9	39

Table I. Performance comparison between A and GBFS across different grid sizes with Staircase path*

Grid	Algo	Waktu (ms)	Nodes Expanded	Avg Turns	Avg Swaps	Avg Queries	Path Len
5×5	A*	0.02	734	29.9	9	20.9	10
5×5	GBFS	0.019	734	29.9	9	20.9	10
8×8	A*	0.15	9,997	92	31	61	32
8×8	GBFS	0.146	9,983	92	31	61	32
10×10	A*	0.395	36,075	145.8	49	96.8	50
10×10	GBFS	0.387	36,145	145.8	49	96.8	50
12×12	A*	0.853	104,633	211.3	71	140.3	72
12×12	GBFS	0.826	104,634	211.3	71	140.3	72
15×15	A*	2.235	382,575	321.2	104	217.2	105
15×15	GBFS	2.163	382,575	321.2	104	217.2	105
20×20	A*	8.87	2,286,097	593.6	199	394.6	200
20×20	GBFS	8.602	2,286,097	593.6	199	394.6	200

Table II. Performance comparison between A and GBFS across different grid sizes with Serpentine-partial path (50% grid column)*

In the myopic one-step lookahead solver with a fixed planned route, A* and GBFS make identical decisions on every turn. This occurs because g(i) = i −flowIdx (sequence distance

on the path) and h(i) = Manhattan(posi,flowPos) are monotonically correlated for all grid path types tested. A tile that is further along the path sequence is always physically further away. As a result, the tie-breaking rule favoring a smaller pathIdx always selects the same candidate for both algorithms. The algorithms only start to differ when g and h conflict, which only happens in multi-step full-horizon planning rather than a myopic approach.

The benchmark results reveal an interesting pattern: A* and GBFS produce the exact same decisions in this solver. This is not a bug but a mathematical result of its design. In a myopic one-step lookahead solver with a fixed target path, g(i)=i−flowIdx (sequence distance) and h(i) as the Manhattan distance always increase together monotonically for all grid paths. Tiles that are further in the sequence are always physically further away. We could never see a situation where a tile is far in the sequence but physically close enough to be the single best choice.

B. Computational Eficiency Analysis

In terms of computational efficiency, the experimental results demonstrate a strict parity between A* and GBFS, entirely contradicting classical algorithm expectations. On the 10x10 grid, both algorithms expand an identical average of 9,387 nodes under the staircase path scenario. This 1:1 expansion ratio holds perfectly steady as the grid size increases. On the maximum tested grid of 20x20, both algorithms evaluate exactly 105,096 nodes. A directly parallel pattern is reflected in the execution time, where both algorithms operate within the exact same computational margin, taking approximately 1.14 to 1.17 ms to complete the 20x20 grid.

This identical efficiency profile is explained by how the myopic solver navigates the constrained search space. Because the solver relies on a fixed target route and evaluates candidates on a single-step reactive basis, A* does not expand its search frontier outward in multiple directions as it normally would in open-grid pathfinding. Instead, the strictly monotonic correlation between g(n) and h(n) forces A* to explore the exact same candidate pool as the greedy approach of GBFS. Consequently, both algorithms exhibit the exact same growth rate in node expansion, entirely neutralizing the quadratic $O(n^2)$ computational penalty that is traditionally associated with A*

C. Implication for the Search Planning

These findings provide crucial insights regarding the design of heuristic-based solvers in real-time environments. The characteristic differences between A* and GBFS will only become apparent and significant if there is a direct conflict between the g and h functions. Such conflicts generally arise exclusively in multi-step full-horizon planning, where the algorithm evaluates a long sequence of consecutive steps and must weigh accumulated past costs against alternative routes.

However, in reactive turn-by-turn planning, adding past weights through the g(n) function proves to be redundant. Because decisions are optimized only for the single best next step based on a local view, the greedy nature of GBFS is entirely sufficient to produce a search route equivalent to the exhaustive evaluation of A*. This indicates that for pipe-routing puzzles with limited visibility and myopic rulesets, using GBFS is highly recommended. Its underlying data structure is more lightweight without the need to track accumulated costs, yielding identical computational outcomes to A* in this specific framework.

D. Limitations of the Experiment

A few limitations of this experiment must be acknowledged. First, the solver relies entirely on a fixed, predetermined target route (the staircase pattern) and utilizes a myopic one-step lookahead approach. Because the path is statically defined, the simulation cannot capture dynamic rerouting scenarios where the algorithm alters its geometric trajectory to bypass obstacles. This constraint restricts both A* and GBFS from demonstrating their broader multi-step horizon planning capabilities.

Second, the experiment exclusively evaluates the Manhattan distance heuristic. Comparing its performance against alternative measurements, such as Euclidean distance or a custom turn-based heuristic, remains a valuable area for follow-up research.

V. CONCLUSION

This paper has modeled the BioShock hacking minigame as a graph route search problem on a grid board with real-time constraints. By implementing a program architecture that separates the puzzle engine and the solver, we can simulate limited board visibility that requires the algorithm to gather information progressively through a query mechanism.

The experimental results testing A* and Greedy Best-First Search (GBFS) over a wide range of grid sizes reveal an important architectural finding. Contrary to the classic theory where A* generally runs slower due to more extensive node expansion, testing on the myopic one-step lookahead solver with a fixed target route showed the exact same performance between both algorithms. This phenomenon is driven by the monotonic correlation between the heuristic spatial distance and the sequence distance on the path, where a candidate tile that is further in sequence is always physically further away.

The main conclusion of the experiment is that the difference in capabilities between A* and GBFS will only have a significant impact if the chosen approach is multi-step full-horizon planning. In a reactive turn-by-turn decision-making system, the predictive nature of the g(n) function in A* becomes excessive and aligns with the direction of the heuristic function h(n). For the development of similar systems in the future, this research suggests the need to test solver architectures that plan the entire route dynamically without a predetermined target trajectory, so that the limits of search optimization capabilities between heuristic algorithms can be further observed.

VI. Acknowledgement

The author would like to thank the teaching lecturer and teaching assistant of IF2211 Algorithm Strateg for the course material that forms the theoretical basis of this paper. The author also thanks the developers of BioShock (2K Games and Irrational Games) for creating the hacking minigame mechanic that inspired the case study in this paper.

VII. Refferences

[1] R. Munir, *Rouote/Path Planning part 1 and 2,* 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/stima25-26.htm [Accessed: June. 18, 2026]

[2] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 4th ed. Hoboken, NJ, USA: Pearson, 2021.

[3] TOKI, "Pemrograman Kompetitif Dasar," OSN TOKI. [Online]. Available: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf. [Accessed: June. 18, 2026]

[4] J. Sannemo, Principles of Algorithmic Problem Solving, October 24, 2018. [Online]. Available: https://usaco.guide/PAPS.pdf [Accessed: June. 19, 2026]

VIII. Appendix

The full source code for the A* and GBFS simulation of the pipe-routing puzzle in this paper is written in C++17 can be accessed at the following link : https://github.com/YeyThePotatoMan/MakalahStima

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.

Kloce Paul William Saragih, 13524040