Rush Hour A* and BFS Analysis

GIANNA GALARD, AMENA FOSHANJI, AND AZEEM AKRAM

I. ABSTRACT

Rush Hour is a sliding block puzzle invented by Nob Yoshigahara in the 1970s. The goal of the game is to get only the red car out through the exit of the board by moving the other vehicles out of its way. However, the cars and trucks (set up before play, according to a puzzle card) obstruct the path which makes the puzzle even more difficult. The first algorithm used is A* (A-Star) which details the heuristics in three different ways. The second algorithm used is BFS (Breadth-First Search).

CCS Concepts

Social and professional topics → Computing education; • Computing Methodologies → Artificial intelligence

Additional Keywords and Phrases

Artificial intelligence

II. BACKGROUND

While Rush Hour seems like a simple puzzle game to most, programmers have found it helpful for testing search algorithms. Along with Rush Hour, many popular games and board games have received this same treatment. The coding community expects to take search algorithms and apply them to familiar settings. Standard search algorithms include Depth First Search, Breadth-First Search, Uniform Cost Search, and many more. Since there is an abundance

of search algorithms available, programmers have repeatedly tried to find which algorithm works best in specific environments. Upon a brief google search, you can find a version of the Rush Hour problem being analyzed with any standard search algorithm. With this information at our hands, we compared two different algorithms tackling the same puzzle, which has allowed us to highlight the differences between them and determine which was a better fit for the problem in hand.

III. INTRODUCTION AND OBJECTIVE

In most games, our objective often is to find our way to the end goal, which ideally in our scenario, we would want to see the shortest distance and take into account the obstacles blocking that will increase our travel time.

Our goal is to solve Rush Hour levels using the A* (A-Star) search algorithm for this project and compare it to the BFS (Breadth-First Search) algorithm. The A* algorithm uses an open and closed list to ensure that nodes are not visited twice, as our implementation uses a graph-search rather than a tree-search. Another characteristic of the A* is that it looks at heuristics and the cost while calculating a path. Taking both of these values into account makes A* a cost-effective algorithm as well as an optimal one. The BFS algorithm is an exhaustive algorithm that searches nodes at the shallowest depth first. Due to this method, it will generally expand more nodes than other search algorithms. However, BFS is still a complete and optimal algorithm because it will find the path at the shallowest depth. It is important to note that BFS doesn’t account for weighted paths like A*, possibly leading to a non-cost-effective result.

IV. OUR APPROACH

To compare the A* search algorithm and the BFS algorithm, there are a few key statistics we need to pay attention to. First and foremost, we needed to note how many nodes were expanded during the search, which is crucial because it shows how much work an algorithm might be doing behind the scenes to find its path. Lastly, we needed to keep track of the search duration to compare the runtime of the two different algorithms. This information is vital in giving us a clear picture of the strengths and weaknesses of each respective algorithm. For instance, an optimal and complete algorithm might expand more nodes than a sub-optimal and sometimes incomplete one.

To ensure the A* algorithm is appropriately implemented, we need to use heuristics. Heuristics estimate how many moves will be required to make it to the goal state when visiting specific nodes. The nodes with the lowest heuristic value are visited first as we go through the list. However, this is still an estimate, and the result found by the A* implementation may not be the most optimal path to the goal. In addition to the heuristic value associated with each node, the A* algorithm looks at the path's cost to find a cost-effective route. For the best comparison of the two algorithms, we chose to only pay attention to the advanced heuristic implementation of the A* algorithm.

The code we used for this project is written in Java and uses a heuristic approach to solve in the most time and cost-efficient way possible. The code for this project details the heuristics in three different ways: the nodes with the lowest heuristic values are visited first as we go through the list.

- Zero Heuristic
    - Returns the same value for every node.

- Blocking Heuristic
    - This heuristic counts the number of cars blocking the exit path directly.
- Advanced Heuristic
    - This heuristic counts the number of cars blocking other cars and each car blocking those cars. Essentially, it counts the number of cars blocking each other recursively.

V. CODE EXECUTION

Entry Point for Advanced Heuristic:

```
getValue(State state) {
    return 0
    if state.isGoal()
    visited.clear()
    visited.add(goalCar)
    value = 1;
    for (getInitialBlockingCars() as car) {
        value += getBlockingValue(car, getRequiredSpace(goalCar, car))
    }
    return value;
}
```

Recursive Identification of Blocking Cars:

```
getBlockingValue(car, requiredSpace) {
  visited.add(car)

  value = 1

  for (allCars as next) {
    continue if next == car
    continue if visited.contains(next)
    continue unless car.intersectsWith(next)

    forwardCosts = 0, backwardCosts = 0

    if (!canMoveForward(car, next, requiredSpace.forward) {
      forwardCosts = getBlockingValue(car, getRequiredSpace(car, next))
    } else if (isWallBlockingAhead(car, requiredSpace.forward)) {
      forwardCosts = INFINITY
    }

    if (!canMoveBackward(car, next, requiredSpace.backward) {
      backwardCosts = getBlockingValue(car, getRequiredSpace(car, next))
    } else if (isWallBlockingBehind(car, requiredSpace.backward)) {
      backwardCosts = INFINITY
    }

    value += min(forwardCosts, backwardCosts);
  }

  return value;
}
```

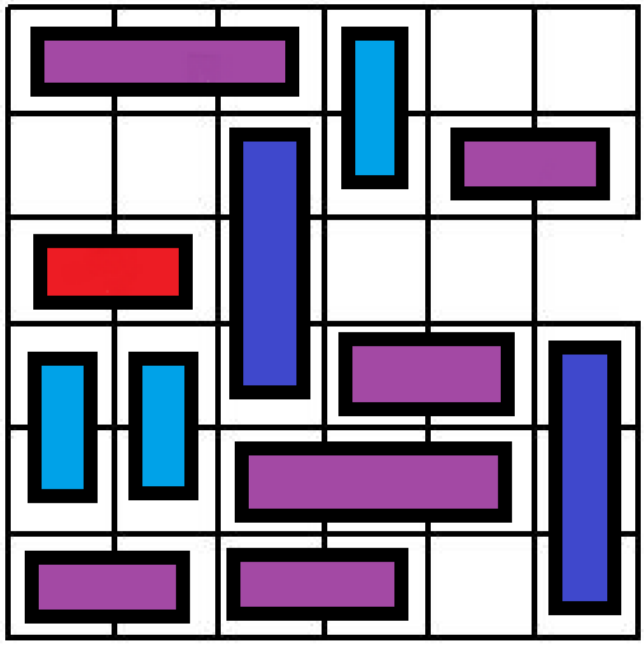
VI. RESULTS

There are a few drawbacks with our Advanced Heuristic that could result in the output not being the most optimal solution. The first is that the time needed to calculate the heuristic value of the node that is higher than that of the blocking heuristic. We can expect that with the

advanced heuristic, the A* algorithm will find a solution slower than other algorithms. The second is that cars cannot be counted twice. If the car was already counted, it is considered "moved" so it may or may not block another car. Typically, a BFS algorithm will expand more nodes than a A* algorithm in a given problem, due to its exhaustive nature of finding a solution. Even though it expands more nodes on average, BFS is also an optimal and complete algorithm because it finds the solution at the lowest depth. A* is also a complete and optimal algorithm but will generally run faster than BFS because of the difference in expanded nodes. It should be noted that the “duration” in the figures of the A* code is in milliseconds, while the BFS algorithm’s time was tracked in seconds. To give the most accurate performance statistics for each respective program the run time of each jam is an average of 5 runs for each algorithm.

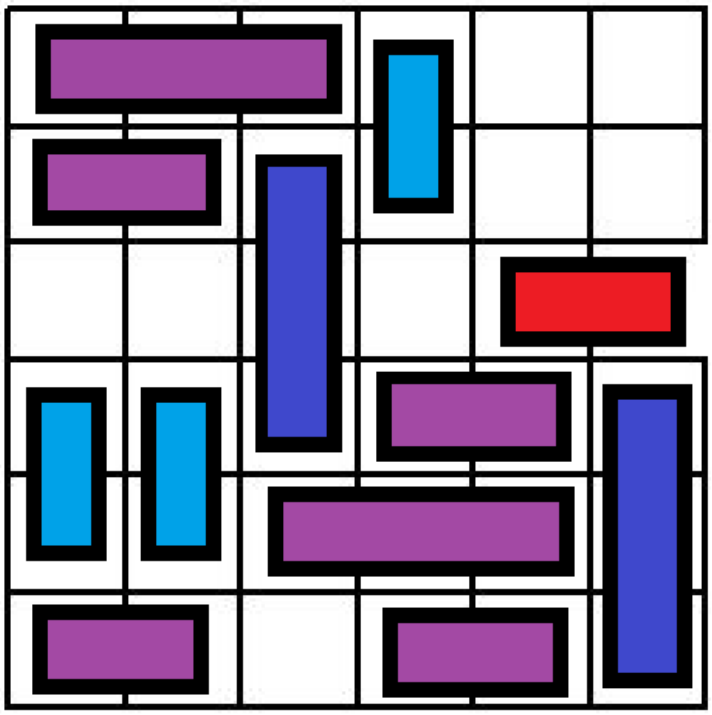
Jam 1 (A* + Advanced Heuristic)

Figure 1.1.1 - Initial State



```
================================================
puzzle = Jam-28

-----------------------------------

heuristic = Heuristics.AdvancedHeuristic
+-------------+
| < - > ^ . . |
| . . ^ v < > |
| < > | . . .
| ^ ^ v < > ^ |
| v v < - > | |
| < > < > . v |
+-------------+
```

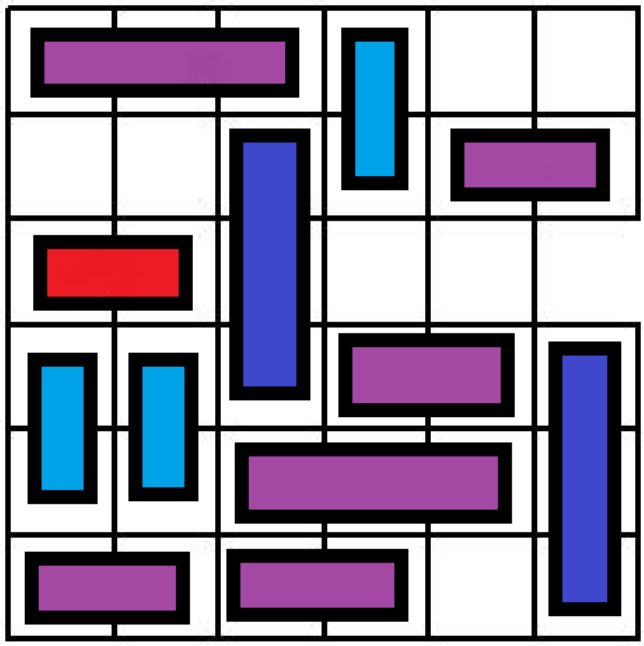
Figure 1.1.2 - End State



```
+-------------+
| < - > ^ . . |
| < > ^ v . . |
| . . | . . < >
| ^ ^ v < > ^ |
| v v < - > | |
| < > . < > v |
+-------------+

nodes expanded: 8022, soln depth: 30, duration: 19
================================================
```

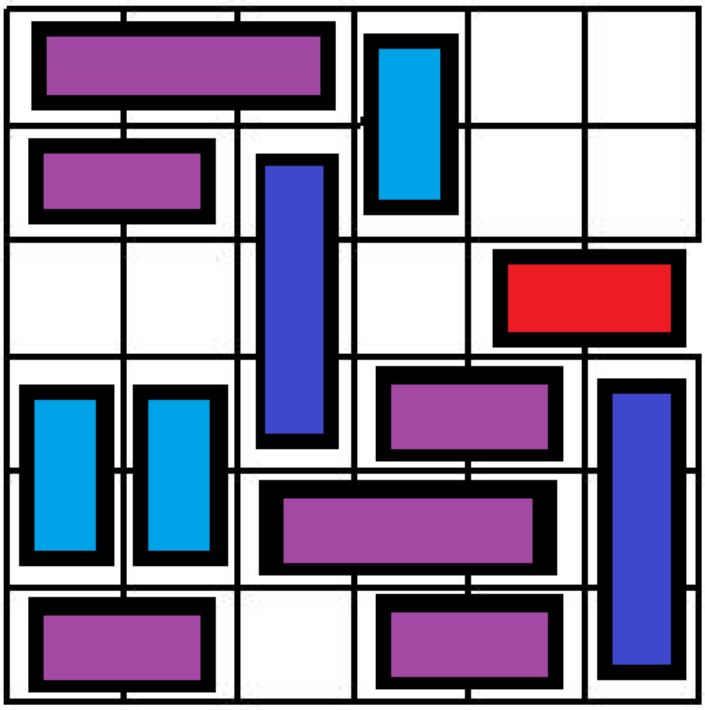
Jam 1 (BFS)

Figure 1.2.1 - Initial State



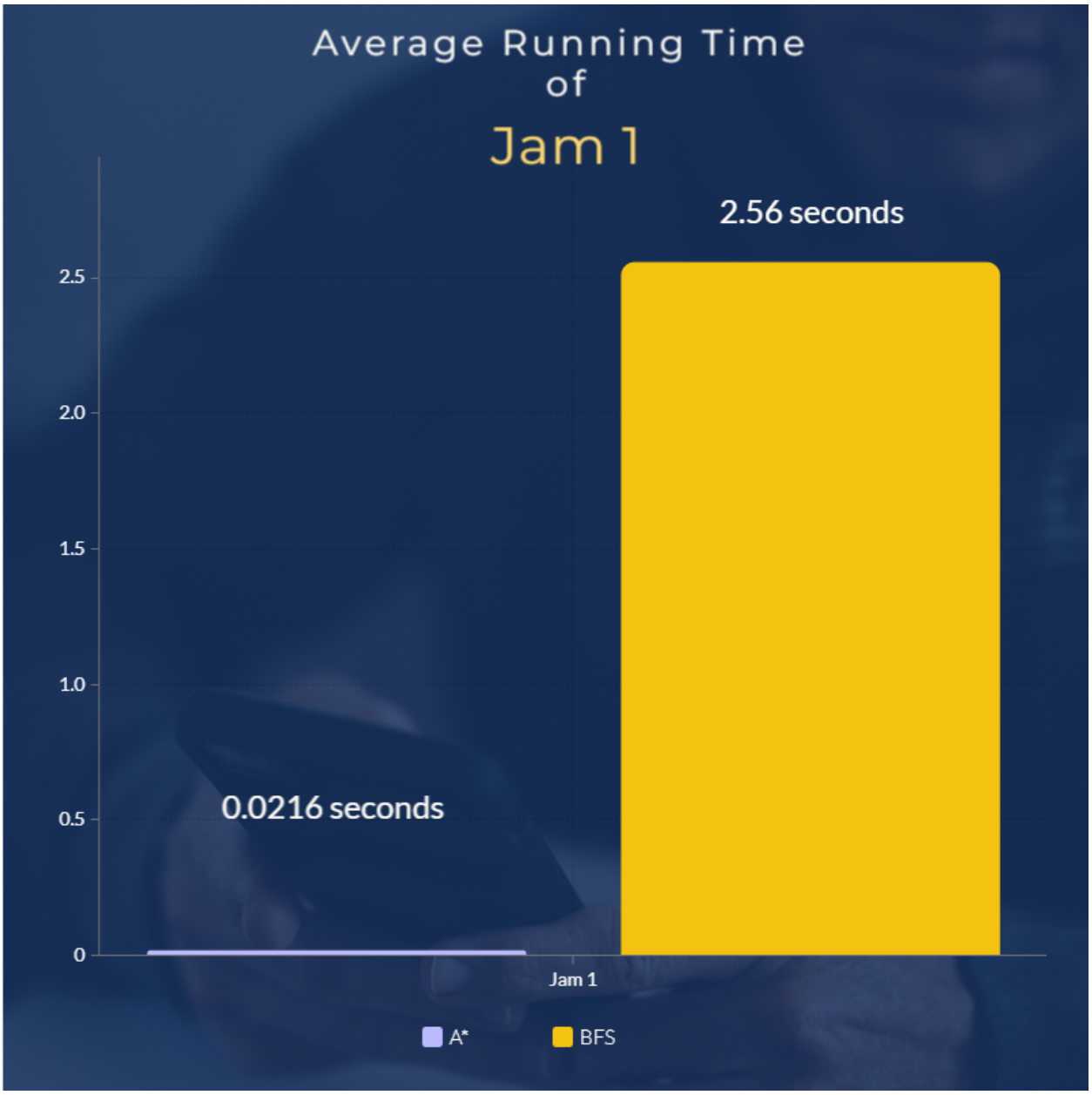
```
STARTING BOARD STATE:

 02 02 02 010 _  _
 _  _  09 010 01 01
 X  X  09 _  _  _
 07 08 09 06 06 011
 07 08 05 05 05 011
 03 03 04 04 _  011
```

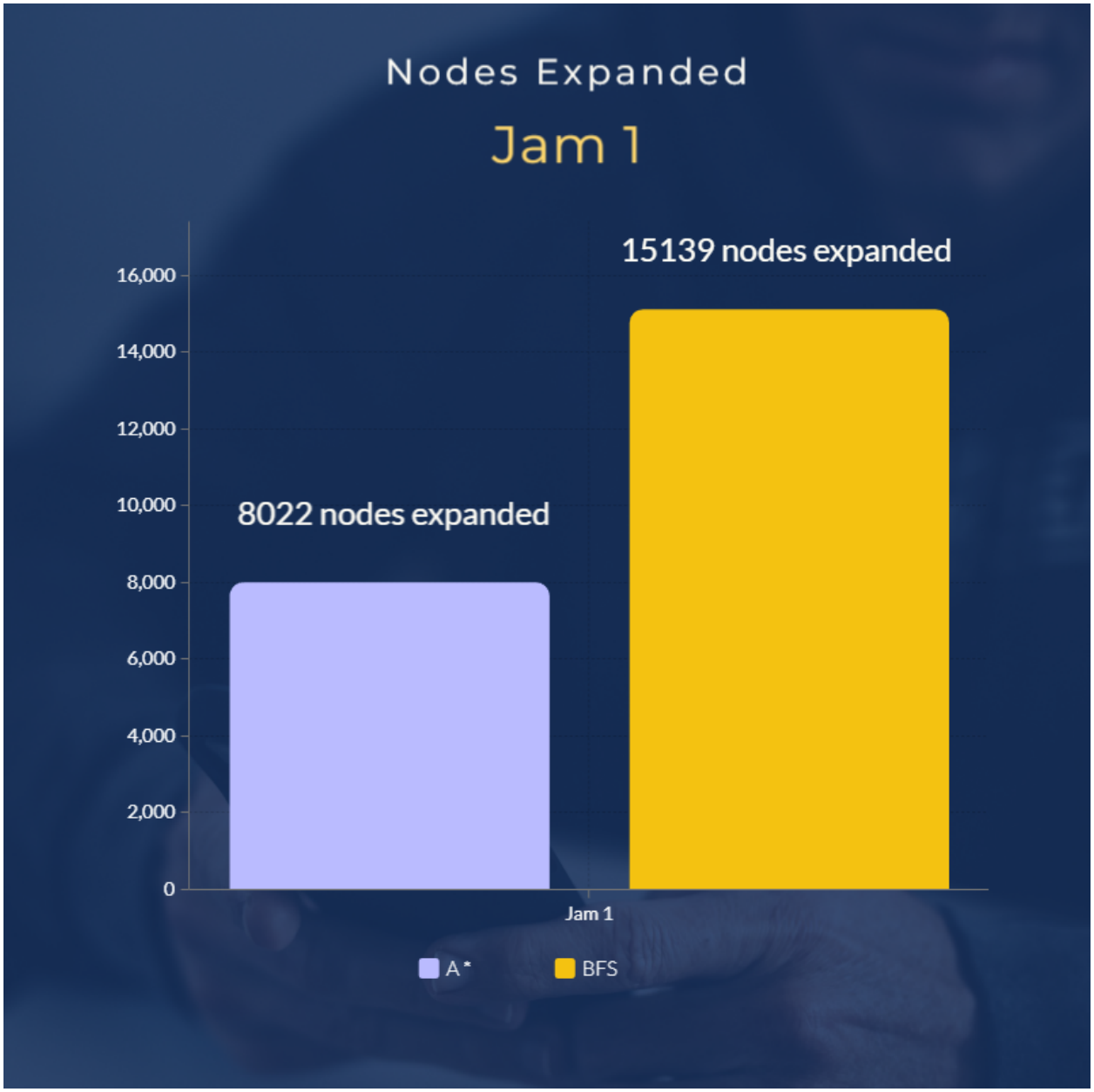
Figure 1.2.2 - End State



```
The shortest sequence file output4_bfs.txt is created.
TOTAL SOLUTIONS:24
TOTAL CYCLES:0
MOVE # OF THE SHORTEST SOLUTION:51
TOTAL NODES GENERATED:2400
MAXIMUM # OF NODES KEPT IN MEMORY:15139
TOTAL bfs TIME:2.28125 seconds
```

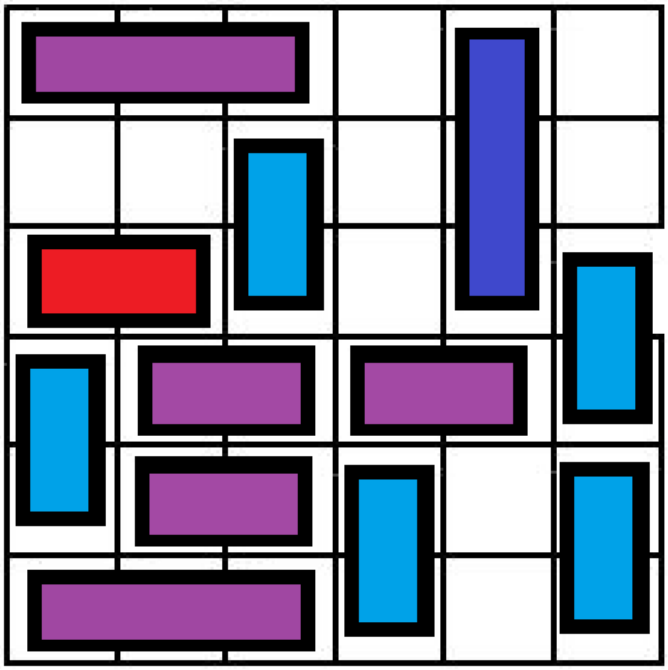
Jam 1 Average Results




Nodes Expanded
Jam 1
15139 nodes expanded
8022 nodes expanded
16,000
14,000
12,000
10,000
8,000
6,000
4,000
2,000
0
Jam 1
A*
BFS

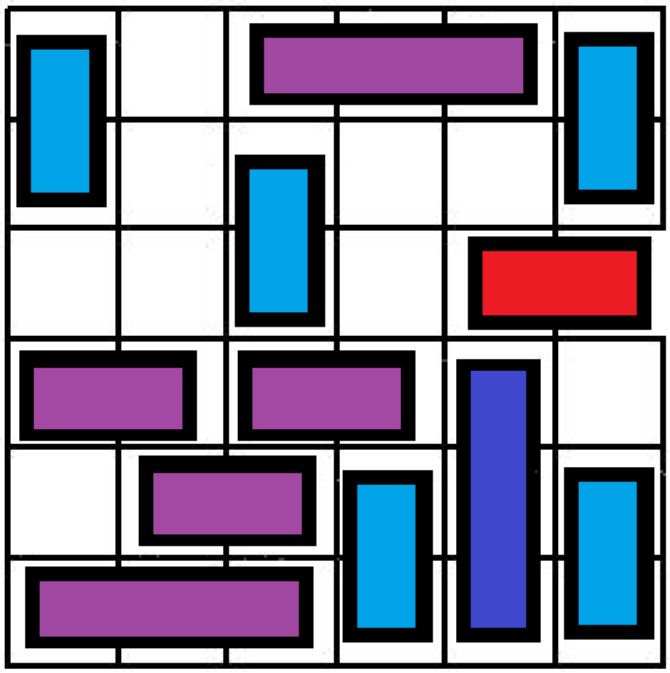
Jam 2 (A* + Advanced Heuristic)

Figure 2.1.1 - Initial State



```
================================================
puzzle = Jam-29

--------------------------------------

heuristic = Heuristics.AdvancedHeuristic
+-------------+
| < - > . ^ . |
| . . ^ . | . |
| < > v . v ^
| ^ < > < > v |
| v < > ^ . ^ |
| < - > v . v |
+-------------+
```

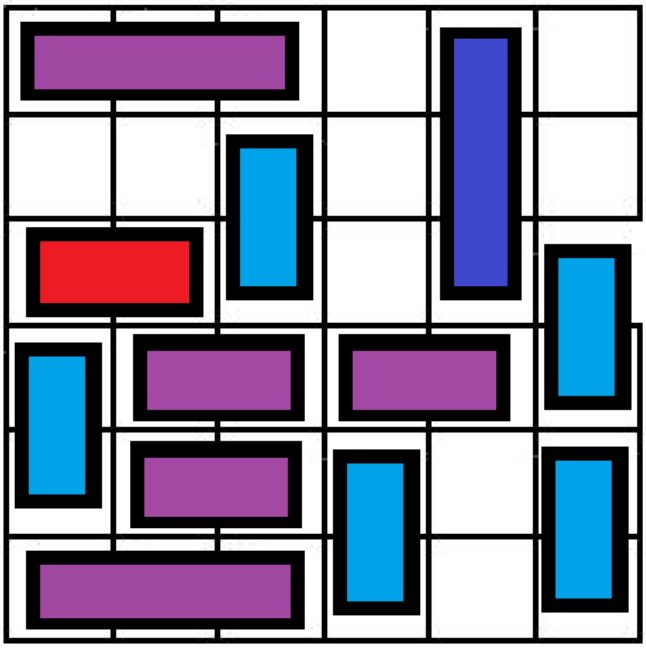
Figure 2.1.2 - End State



```
+-------------+
| ^ . < - > ^ |
| v . ^ . . v |
| . . v . . < >
| < > < > ^ . |
| . < > ^ | ^ |
| < - > v v v |
+-------------+

nodes expanded: 38082, soln depth: 31, duration: 197
==================================================
```

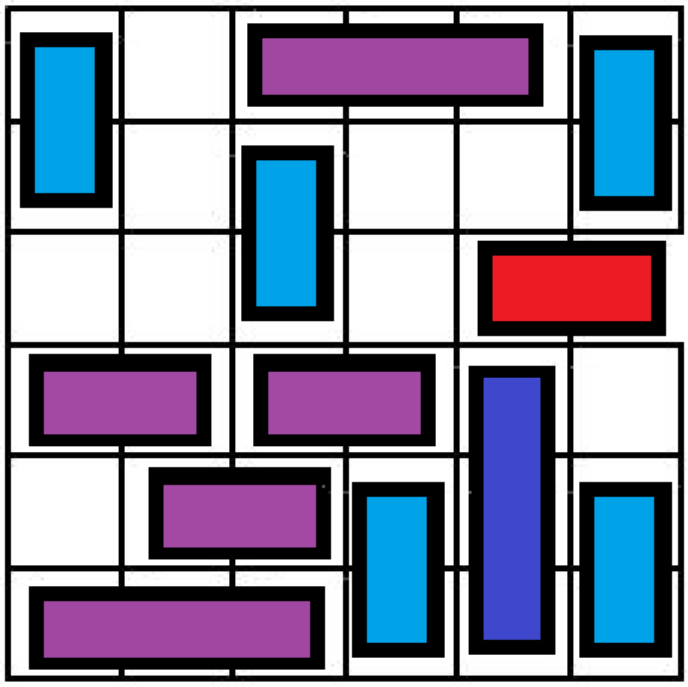
Jam 2 (BFS)

Figure 2.2.1 - Initial State



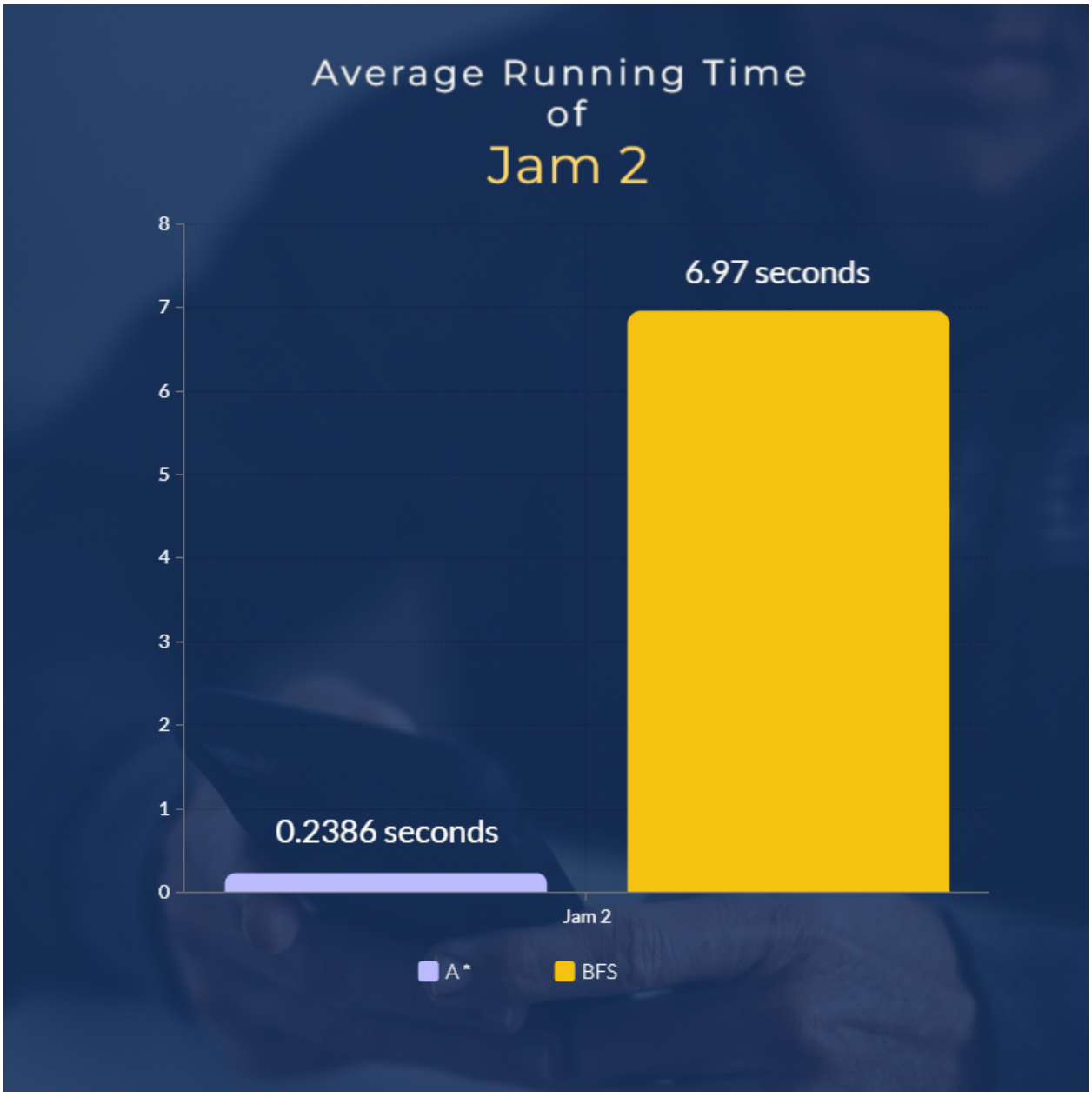
```
STARTING BOARD STATE:

01 01 01 _  06 _
_  _  07 _  06 _
X  X  07 _  06 08
09 02 02 03 03 08
09 04 04 010 _  011
05 05 05 010 _  011
```

Figure 2.2.2 - End State



```
The shortest sequence file output5_bfs.txt is created.
TOTAL SOLUTIONS:20
TOTAL CYCLES:0
MOVE # OF THE SHORTEST SOLUTION:54
TOTAL NODES GENERATED:4373
MAXIMUM # OF NODES KEPT IN MEMORY:28849
TOTAL bfs TIME:6.32812 seconds
```

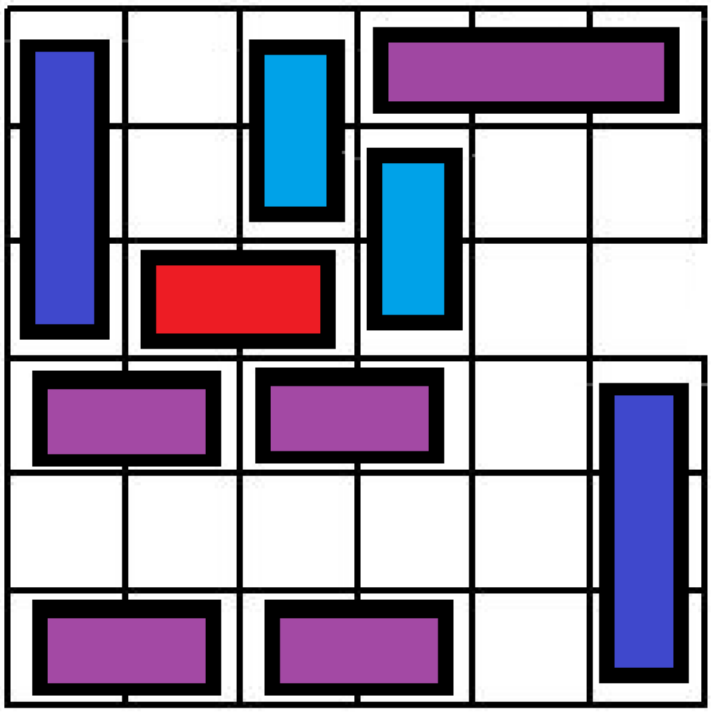
Jam 2 Average Results




Nodes Expanded
Jam 2
38082 nodes expanded
28849 nodes expanded
40,000
35,000
30,000
25,000
20,000
15,000
10,000
5,000
0
Jam 2
A*
BFS

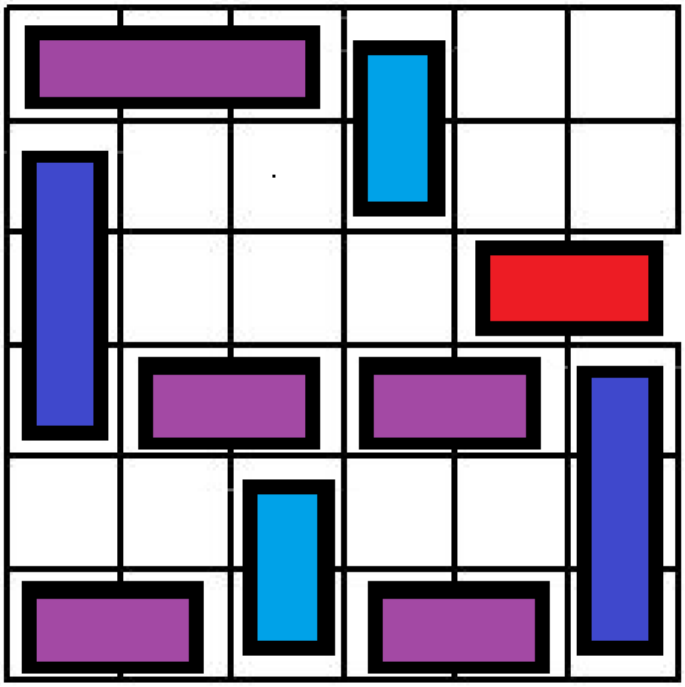
Jam 3 (A* + Advanced Heuristic)

Figure 3.1.1 - Initial State



```
================================================
puzzle = Jam-30

--------------------------------------

heuristic = Heuristics.AdvancedHeuristic
+-------------+
| ^ . ^ < - > |
| | . v ^ . . |
| v < > v . . |
| < > < > . ^ |
| . . . . . | |
| < > < > . v |
+-------------+
```

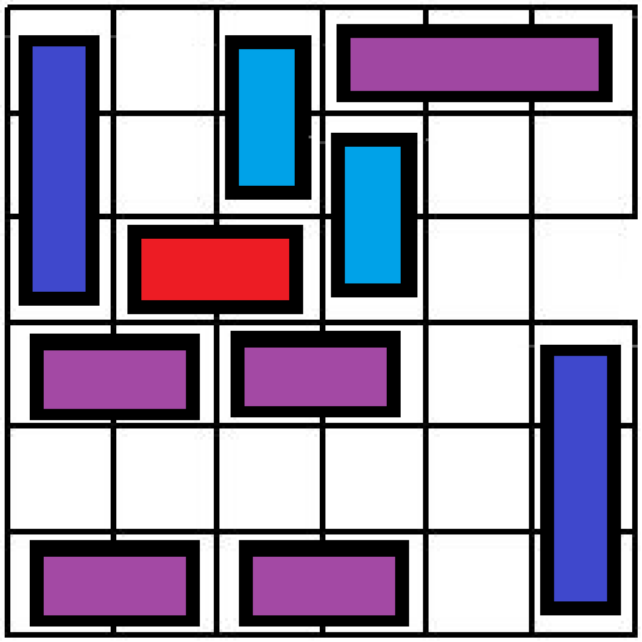
Figure 3.1.2 - End State



```
+------------+
| < - > ^ . . |
| ^ . . v . . |
| | . . . . < > |
| v < > < > ^ |
| . . ^ . . | |
| < > v < > v |
+------------+

nodes expanded: 7316, soln depth: 32, duration: 10
================================================
```

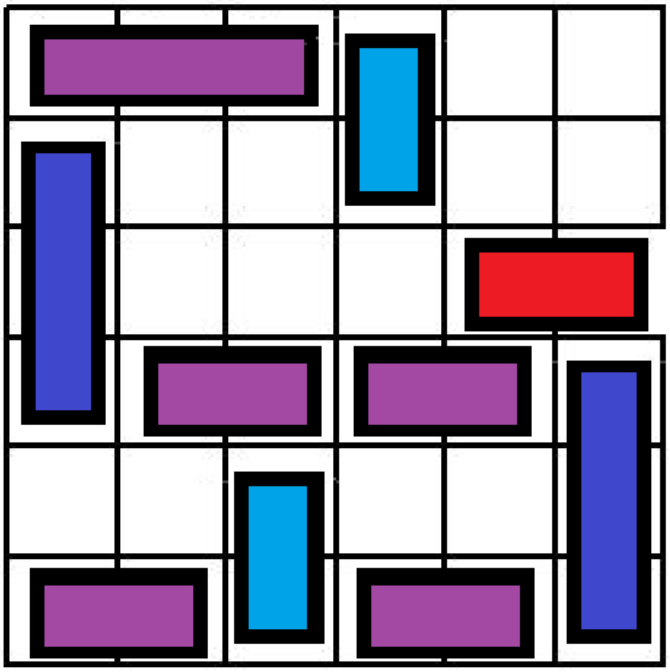
Jam 3 (BFS)

Figure 3.2.1 - Initial State



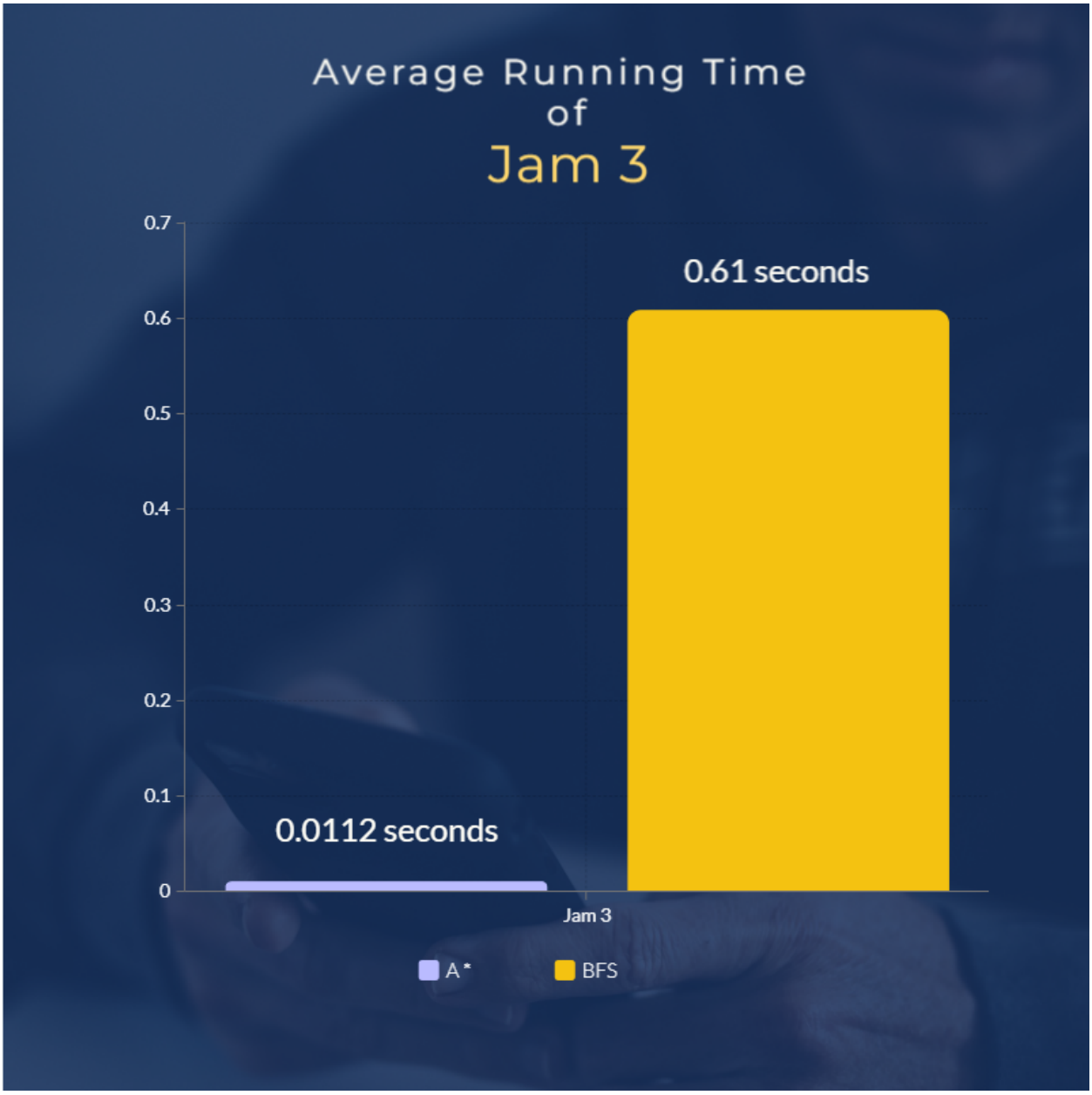
```
STARTING BOARD STATE:

 06 _  08 01 01 01
 06 _  08 07 _  _
 06 X  X  07 _  _
 02 02 03 03 _  09
 _  _  _  _  _  09
 04 04 05 05 _  09
```

Figure 3.2.2 - End State



```
The shortest sequence file output6_bfs.txt is created.
TOTAL SOLUTIONS:2
TOTAL CYCLES:0
MOVE # OF THE SHORTEST SOLUTION:55
TOTAL NODES GENERATED:1171
MAXIMUM # OF NODES KEPT IN MEMORY:6335
TOTAL bfs TIME:0.578125 seconds
```

Jam 3 Average Results



VII. DISCUSSION

After comparing the performance of A* and BFS in solving Rush Hour jams, A* is clearly the best option out of the two. Across the board we see that A* outperforms BFS in virtually every way. However, there is context that must be taken into account. Generally, we can

assume that a BFS algorithm will expand more nodes than an A* algorithm and our results supported that assumption. This in itself explains the massive time difference between both algorithms solving the same jam. While we could not compare the depth level of the BFS algorithm's solutions, we can assume that it produced a similar or more shallow solution than the A* algorithm. Upon further inspection, BFS seemed to have more worst case scenarios, leading to many more nodes being expanded. Compared to BFS, The A* algorithm can more reliably find an optimal and cost-effective solution in a shorter amount of time with the use of heuristics. However, it is important to note that while A* was the better choice in this scenario, it is not the best suited algorithm for any given problem.

VIII. REFERENCES

Rush Hour (puzzle). (2022, January 30). Wikipedia.

https://en.wikipedia.org/wiki/Rush_Hour_(puzzle)

Zarhuber, S. (2022, March 2). *Rush Hour - An A* Implementation*. GitHub.

https://github.com/saschazar21/rushhour

Özgen, A. C. (2019, December 4). Rush Hour with BFS and DFS. GitHub.

https://github.com/azmiozgen/rush_hour