COMP 472

Rani Rafid[1], Gabriel Harris[2], and Manpreet Tahim[3]

[1] 26975852 rani.rafid@concordia.ca
[2] 40058821 gabrieltobanharris@hotmail.com
[3] 26592006 lvlanu29@gmail.com

1 Introduction

The project undertaken involved solving an Indonesian Dot Puzzle using different state space search algorithms, in python. Amongst the algorithms, there was one uniformed search algorithm (Depth-First or DFS) and two informed search algorithms (Best-First or BFS, and A*).

Before tackling coding, it was important that an overall architecture was set up, which evolved throughout several stages of the project. In the end, our project was divided into several entities: Nodes, Agents, and Puzzles. The Node can be considered as the basis of a graph tree, which in our case represented a board's state. Anything relating to the node, including its predecessor, descendent, and characteristics (depth, size, and length) would be located within this module. The Agents module was created specifically for each algorithm. In terms of exploration on a visited node, Best-First and A* both stem from a Breadth-First approach. However, due to the professor's request of ordering each children of a node in a specific format (boards containing 1's at the top leftmost spot would be selected first) for DFS, it essentially mirrored the Breadth-First approach. Hence, the Agent's module was customized to simply calculates a node's heuristic, specific to each algorithm. Lastly, the Puzzle module was implemented to handle bulk of the work. Traversal, solution handling and search handling was performed by this module.

For each algorithm, a basic structure was created in python upon which adjustments were performed in order to improve the algorithm's running efficiency and time. Since our code did not involve any complex computations and state handling was done by converting each state into an integer (considering it originally as a binary), we were able to forgo using Numpy, speeding up our solution time. In the end, our goal was to design a working modular environment in which we would be able to solve an Indonesian Dot puzzle as efficiently as possible for each algorithm.

2 Heuristics

In trying to determine a heuristic function, we created a simple heuristic based on Hamming distance in order to serve as a baseline, as shown in Figure 8. Essentially, h1(n) was the number of 1's (since our goal state is a board full of 0's). Given the nature of the problem, this heuristic makes a lot of sense, but fails to consider groups and ordering. For example, a board with 3 scattered 1's is definitely not closer to the goal state than a board with 5 1's in the shape of a plus (this would reach the goal state in one move).

From the heuristics above, we were able to stem some different variations and test them out. We created and tested heuristic functions that determined the number of pairs of 1's (h2) and number of three consecutive 1's (h3) as well (both overlapping and not) but this proved to yield a longer search result than our original heuristic. Both algorithms are shown in Figure 9 and 10, respectively. In retrospect, this makes sense as they don't take into consideration other alternatives that could prove to be better moves. Taking the same example as above, a board with some scattered pairs of 1's would be chosen over a board with 5 1's in the shape of a plus. Due to the size restriction of h2 and h3, you would have several boards that, while yielding the same heuristics, would not be comparable at all.

Another variation, shown in Figure 11, involved selecting the board containing a move that could clear the most amount of 1's (haround). However, this suffers from the same problem as mentioned above. This performs a more exhaustive search since it goes through every board that could potentially clear 5 1's first, and then 4, and so on.

Lastly, we tested a heuristic that would return the number of cleared rows (hrows) from top to bottom, as shown in Figure 12. The Indonesian Dot puzzle is based on a popular game called Light's Out, and one strategy involves performing such technique. However, this proved not as efficient as we had planned as towards the end, the algorithm involved going through each row once more in order to clear any remaining 1's.

From all these heuristics, the one selected was the initial one proposed (h1) that we will term h(n), which instead of returning the number of 1's, simply returns the number of potential moves that could be done, as shown in Figure 13. This involves first verifying that the modulus of the number of 1's for 5, 4 and 3, respectfully in that order, is not 1 and 2 has no moves could flip exactly 1 or 2 nodes. This heuristic is not without its fault as it merely works by counting the numbers of 1's and assumes their positions instead of take into account ordering.

3 Challenges

3.1 Relational Modeling

A large portion of time leading to the completion of the project was attributed to understanding how subjects should be linked. This process was judged to be difficult since some domain requirements can be ambiguous. To resolve this issue, an experiment was conducted using the three main subjects of the domain: agents (search algorithms), puzzles (graphs), and nodes (board states).

(i) Agents use a puzzles and nodes.

(ii) Puzzles use agents and nodes.

(iii) Each agent uses puzzles and nodes.

(iv) Each puzzle uses agents and nodes.

When the models were compared on the same data set, it was observed that the average run-time for models (i) to (iv) was 2 minutes, 66 ms, 102 ms, and 18 ms respectively. Moreover, results of the experiment led to two important findings that helped us classify the domain requirements of the project.

The first finding resolves the concern of whether or not to use agent or puzzle as the primary subject of the system. For example, should (i/iii) be chosen over (i/iv)? If agent was the primary subject, then each agent must have a reference to a puzzle, and each agent would rely on the puzzle to provide it with a new state for every step of a traversal. The issue with this is that each agent had similar implementations for traversing the puzzle. This affected the overall performance of our system because there were redundancies for the traversal behaviour. To resolve this issue, we took away the responsibility of traversing nodes from the agent class and assigned it to the puzzle class. As a result of this change, the agent's responsibility was now limited to computing heuristic values. Overall, this finding suggests against using (i) and (iii) as a schema for modelling the system.

The first finding also suggests that agents should not have direct access to the node class. Yet, having access to this object is important for the system to function properly since it stores the values of each heuristic. A first approach to this problem was to include an accesssor method in the puzzle class. However, this increased the time complexity for state searching by a linear factor, and also contradicts the schema proposed in (ii) and (iv). Therefore, the puzzle needs to reference the agent.

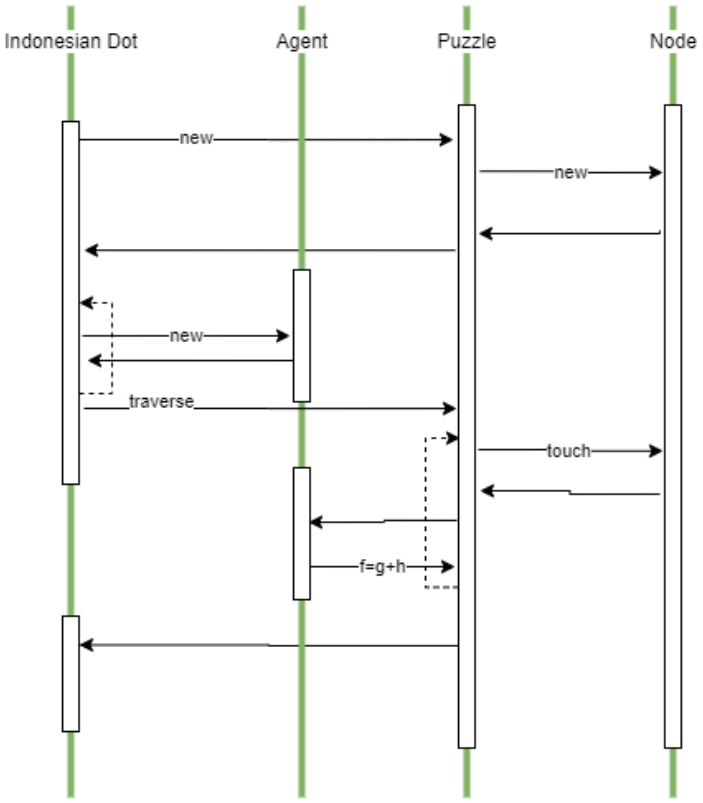

Fig. 1. Sequence Diagram of Indonesian Dot Solver

This leads to the second finding: A puzzle and an agent cannot have a many-to-many relational type because agent no longer has a reference to a puzzle. Moreover, many nodes are used by one puzzle. Therefore, the combination of both the agent and the node identifies the puzzle that it belongs to. This makes sense since each node points to it's predecessor, with the root node uniquely linked to the puzzle it belongs to (assuming there are no duplicate lines in the test file). Therefore, the second finding suggests that the logical architecture should be modelled after statement (iv).

Overall, by performing this experimentation and discovering why the logical architecture performed well under different scenarios, we were able to create a classification plan for classifying each domain requirement into one that belongs to agents, puzzles, or nodes. This experimentation also helped us maximize the performance of the overall system.

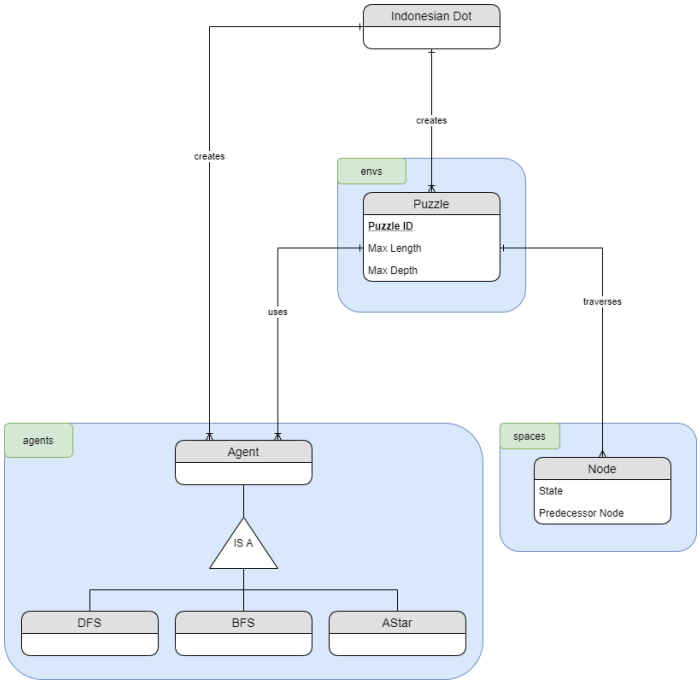

Fig. 2. Entity Relationship Diagram of Indonesian Dot Solver

3.2 Touch Redundancy

Every state of the Indonesian Dot puzzle has actions equivalent to the size of the board. For example, if the board size 25 (or 5x5), then there are 25 actions to choose from. Whenever a traversal was made, it used to be that the puzzle object would touch every action on a node and if it was visited, then it would ignore the action. Yet, checking is an expensive operation especially when the board size increases in size. The touch redundancy issue refers to the issue of having to check for $\sum_{i=0}^{n} \frac{n!}{(n-i)!}$ actions in the worst case, where n is the total size of the board. This is depicted in the following Figure:

```
Agent dfs average time is 1.21e+02 ms.
Agent astar average time is 2.76e+02 ms.
Agent bfs average time is 2.8e+02 ms.
```

Fig. 3. Average Run Time of DFS, BFS, and A* on 7 Puzzles with Touch Redundancy Issue

To resolve this issue, we discovered that the order of actions leading to the goal state did not matter. More formally, the solution path (in actions) can be any combination of the solution set. This also implies that the system would check for $\sum_{i=0}^{n} \binom{n}{i} = 2^n$ actions in the worst case. This discovery greatly helped reduce the run time of the system:

```
Agent astar average time is 1.09 ms.
Agent bfs average time is 1.19 ms.
Agent dfs average time is 31.3 ms.
```

Fig. 4. Average Run Time of DFS, BFS, and A* on 7 Puzzles without Touch Redundancy Issue

4 Analysis

4.1 Overview

The code is implemented such that the only difference between the solving algorithms lie in there heuristic values. As the rest of the code is shared. Though an important impact of the heuristic is how many iterations the main part of the program goes through. The iterations tend to take longer to calculate than the heuristics, thus is ideal to take a bit longer to calculate at to result in fewer iterations. Such is rather apparent with the various experiments, in which depending on the implementation some algorithms would perform better than others. Such can be seen in figures of the difficulties section.

During the solving of the puzzle, how the board is represented varies. Thus it is important to minimize conversion time. The indonesian_dot/spaces/node.py #touch(self, action) function performs said conversion to an integer. Initially the approach was to perform string concatenation then convert the binary representation to an integer. However it was found that directly calculating the integer value from the binary was faster. Most likely because the same calculations would still be performed in addition to the overhead of performing the concatenations.

With the aim of minimizing memory usage, indonesian_dot/spaces/node.py has the minimal number of features. As a simple way to have code run faster is for both the memory reading and the memory writing operations done by the operating system to take the minimal amount of time. Which requires the the operations to be taken out in the highest level of the memory hierarchy possible.

4.2 Algorithm Comparison

Regarding the three solving algorithms, in terms of time to complete, typically A* and BFS tended to have rather similar times with A* being slightly faster than BFS most of the time. Of the three DFS tended to be the slowest. The difference in times tended to increase as the size of the puzzles increased.

The calculating of g heuristic in terms of operation complexity and function calls is essentially the same, thus the h heuristic value is what sets them a part for speed. Though both A* and BFS have the exact same code for calculating h values; while DFS has a simple, though irrelevant, way to calculate DFS's h value.

4.3 Algorithm Contrast

In terms of function calls (raw data shown in Appendix section), DFS tended to have about 80 times as many as A*. Such helps to explain why DFS is so slow. Another reason for DFS's slowness is that the algorithm is essentially a brute force approach solution, while the others algorithms used tend to take educated guesses (reduces iteration count) to speed up their solving of the puzzle.

Due to DFS's minimalist approach in calculating heuristics, in the event that it would take the same search path as the other algorithms, it would have the best time. Though such would merely be the result of a specific puzzle configuration, than by design. As DFS is a rather brutish approach for solving.

To solve the puzzle no repeated moves are required, as they would simply end up canceling out. Also the order of the moves does not matter. The program is implemented so that for search paths, moves cannot be repeated, however the result is that the order that the moves are selected in is fixed. The drawback of the ordering is that the resulting search space is the power set of the remaining possible moves (excluding the empty set), starting from the last move in the order. While subsequent search spaces will be smaller, it can still be rather costly to located the first move in the order. As every move before it will be a waste of time, amounting only in determining that those starting moves are not in the solution. Therefore solutions that have consecutive moves in the ordering are ideal for both DFS and BFS, while A* prefers paths formed from sparse moves.

5 Contributions

5.1 Rani Rafid

Rani Rafid designed and implemented the logical architecture of the Indonesian Dot puzzle solver system. He assigned responsibilities to his teammates and managed the project throughout the iteration. He also set deadlines and organized team meetings.

5.2 Gabriel Harris

Gabriel Harris implemented the main (entry) program for the Indonesian Dot puzzle solver system. He contributed to the development of complementary features for the system and theorized creative solutions for past issues.

5.3 Manpreet Tahim

Manpreet Tahim implemented the search algorithms of the Indonesian Dot puzzle solver system. He explored and studied several heuristic functions. He also shared his findings with his teammates, and contributed to the development of the logical architecture.

References

1. Author, C. Larman: Applying UML and Patterns. 3rd edn. Prentice Hall
2. Wolfram MathWorld, http://mathworld.wolfram.com/LightsOutPuzzle.html. Last accessed 27 Feb 2020
3. Python 2.7.17 Documentation-Multiprocessing,https://docs.python.org/2/library/multiprocessing.html. Last accessed 27 Feb 2020
4. Python 2.7.17 Documentation-Heapq, https://docs.python.org/2/library/heapq.html. Last accessed 27 Feb 2020
5. Wikipedia - Hill Cipher, https://en.wikipedia.org/wiki/Hill_cipher. Last accessed 27 Feb 2020

6 Appendix

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
671	0.004	0.000	0.007	0.000	node.py:215(touch)
7	0.003	0.000	0.017	0.002	puzzle.py:73(traverse)
2369	0.002	0.000	0.003	0.000	node.py:174(__lt__)
671	0.002	0.000	0.003	0.000	node.py:95(__init__)
4828	0.001	0.000	0.001	0.000	node.py:126(f)
678	0.001	0.000	0.002	0.000	bfs_agent.py:9(h)
671	0.001	0.000	0.003	0.000	{built-in method _heapq.heappush}
14	0.001	0.000	0.001	0.000	{built-in method io.open}
1386	0.001	0.000	0.001	0.000	node.py:106(length)
678	0.001	0.000	0.001	0.000	node.py:201(__contains__)
678	0.000	0.000	0.000	0.000	{method 'count' of 'str' objects}
1349	0.000	0.000	0.000	0.000	node.py:145(h)
1349	0.000	0.000	0.000	0.000	node.py:140(g)
14	0.000	0.000	0.000	0.000	{method 'writelines' of '_io._IOBase' objects}
699	0.000	0.000	0.000	0.000	node.py:205(__len__)
1436	0.000	0.000	0.000	0.000	{built-in method builtins.len}
84	0.000	0.000	0.001	0.000	{built-in method _heapq.heappop}
755	0.000	0.000	0.000	0.000	node.py:122(depth)
678	0.000	0.000	0.000	0.000	node.py:161(state)
678	0.000	0.000	0.000	0.000	bfs_agent.py:6(g)
678	0.000	0.000	0.000	0.000	{built-in method builtins.isinstance}
90	0.000	0.000	0.000	0.000	node.py:238(search_artifact)
6	0.000	0.000	0.000	0.000	{method 'clear' of 'list' objects}
14	0.000	0.000	0.000	0.000	{built-in method _locale.nl_langinfo}
80	0.000	0.000	0.000	0.000	{built-in method math.sqrt}
22	0.000	0.000	0.000	0.000	node.py:241(solution_artifact)
7	0.000	0.000	0.000	0.000	puzzle.py:136(<listcomp>)
14	0.000	0.000	0.000	0.000	_bootlocale.py:33(getpreferredencoding)
84	0.000	0.000	0.000	0.000	node.py:154(previous_action)
126	0.000	0.000	0.000	0.000	{method 'append' of 'list' objects}
13	0.000	0.000	0.000	0.000	{method 'join' of 'str' objects}
14	0.000	0.000	0.000	0.000	codecs.py:186(__init__)
14	0.000	0.000	0.000	0.000	node.py:118(size)
7	0.000	0.000	0.000	0.000	puzzle.py:69(goal_state)
16	0.000	0.000	0.000	0.000	{built-in method builtins.chr}
14	0.000	0.000	0.000	0.000	{built-in method time.time}
16	0.000	0.000	0.000	0.000	{built-in method builtins.divmod}
22	0.000	0.000	0.000	0.000	node.py:150(predecessor)
14	0.000	0.000	0.000	0.000	bfs_agent.py:26(__str__)
7	0.000	0.000	0.000	0.000	puzzle.py:57(id)
16	0.000	0.000	0.000	0.000	{built-in method builtins.ord}
6	0.000	0.000	0.000	0.000	{method 'reverse' of 'list' objects}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Fig. 5. Agent BFS average time is 2.48ms, 20974 function calls in 0.019 seconds

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
673	0.004	0.000	0.008	0.000	node.py:215(touch)
14	0.003	0.000	0.003	0.000	{built-in method io.open}
7	0.003	0.000	0.017	0.002	puzzle.py:73(traverse)
673	0.002	0.000	0.003	0.000	node.py:95(__init__)
1353	0.001	0.000	0.001	0.000	node.py:145(h)
1436	0.001	0.000	0.002	0.000	node.py:174(__lt__)
680	0.001	0.000	0.001	0.000	astar_agent.py:8(h)
701	0.001	0.000	0.001	0.000	node.py:205(__len__)
2932	0.001	0.000	0.001	0.000	node.py:126(f)
673	0.001	0.000	0.002	0.000	{built-in method _heapq.heappush}
680	0.000	0.000	0.001	0.000	node.py:201(__contains__)
1390	0.000	0.000	0.000	0.000	node.py:106(length)
680	0.000	0.000	0.000	0.000	{method 'count' of 'str' objects}
14	0.000	0.000	0.000	0.000	{method 'writelines' of '_io._IOBase' objects}
1353	0.000	0.000	0.000	0.000	node.py:140(g)
680	0.000	0.000	0.000	0.000	astar_agent.py:5(g)
1407	0.000	0.000	0.000	0.000	node.py:122(depth)
1419	0.000	0.000	0.000	0.000	{built-in method builtins.len}
680	0.000	0.000	0.000	0.000	node.py:161(state)
680	0.000	0.000	0.000	0.000	{built-in method builtins.isinstance}
6	0.000	0.000	0.000	0.000	{method 'clear' of 'list' objects}
14	0.000	0.000	0.000	0.000	{built-in method _locale.nl_langinfo}
14	0.000	0.000	0.000	0.000	_bootlocale.py:33(getpreferredencoding)
54	0.000	0.000	0.000	0.000	{built-in method _heapq.heappop}
60	0.000	0.000	0.000	0.000	node.py:238(search_artifact)
22	0.000	0.000	0.000	0.000	node.py:241(solution_artifact)
59	0.000	0.000	0.000	0.000	{built-in method math.sqrt}
7	0.000	0.000	0.000	0.000	puzzle.py:136(<listcomp>)
96	0.000	0.000	0.000	0.000	{method 'append' of 'list' objects}
14	0.000	0.000	0.000	0.000	codecs.py:186(__init__)
54	0.000	0.000	0.000	0.000	node.py:154(previous_action)
13	0.000	0.000	0.000	0.000	{method 'join' of 'str' objects}
16	0.000	0.000	0.000	0.000	{built-in method builtins.chr}
14	0.000	0.000	0.000	0.000	{built-in method time.time}
14	0.000	0.000	0.000	0.000	node.py:118(size)
7	0.000	0.000	0.000	0.000	puzzle.py:69(goal_state)
16	0.000	0.000	0.000	0.000	{built-in method builtins.divmod}
14	0.000	0.000	0.000	0.000	astar_agent.py:25(__str__)
22	0.000	0.000	0.000	0.000	node.py:150(predecessor)
7	0.000	0.000	0.000	0.000	puzzle.py:57(id)
16	0.000	0.000	0.000	0.000	{built-in method builtins.ord}
6	0.000	0.000	0.000	0.000	{method 'reverse' of 'list' objects}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Fig. 6. Agent A* average time is 2.36ms, 18671 function calls in 0.020 seconds

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
233278	0.137	0.000	0.204	0.000	puzzle.py:81(lt)
22744	0.070	0.000	0.128	0.000	node.py:215(touch)
7	0.059	0.008	0.471	0.067	puzzle.py:73(traverse)
743761	0.058	0.000	0.058	0.000	node.py:122(depth)
14	0.040	0.003	0.040	0.003	{method 'writelines' of '_io._IOBase' objects}
22744	0.028	0.000	0.047	0.000	node.py:95(__init__)
22531	0.020	0.000	0.140	0.000	{built-in method _heapq.heappop}
22744	0.019	0.000	0.103	0.000	{built-in method _heapq.heappush}
152992	0.013	0.000	0.013	0.000	node.py:161(state)
22537	0.009	0.000	0.013	0.000	node.py:238(search_artifact)
45536	0.009	0.000	0.011	0.000	node.py:106(length)
45495	0.007	0.000	0.007	0.000	node.py:140(g)
45495	0.007	0.000	0.007	0.000	node.py:145(h)
22751	0.006	0.000	0.008	0.000	node.py:201(__contains__)
22774	0.006	0.000	0.009	0.000	node.py:205(__len__)
50694	0.004	0.000	0.004	0.000	{built-in method builtins.len}
22537	0.004	0.000	0.004	0.000	node.py:126(f)
7	0.003	0.000	0.017	0.002	puzzle.py:136(<listcomp>)
22751	0.002	0.000	0.002	0.000	{built-in method builtins.isinstance}
22751	0.002	0.000	0.002	0.000	dfs_agent.py:6(g)
22575	0.002	0.000	0.002	0.000	{method 'append' of 'list' objects}
22751	0.002	0.000	0.002	0.000	dfs_agent.py:9(h)
6421	0.001	0.000	0.001	0.000	node.py:154(previous_action)
14	0.001	0.000	0.001	0.000	{built-in method io.open}
5192	0.001	0.000	0.001	0.000	{built-in method math.sqrt}
13	0.001	0.000	0.001	0.000	{method 'join' of 'str' objects}
14	0.000	0.000	0.000	0.000	{built-in method _locale.nl_langinfo}
14	0.000	0.000	0.000	0.000	_bootlocale.py:33(getpreferredencoding)
24	0.000	0.000	0.000	0.000	node.py:241(solution_artifact)
14	0.000	0.000	0.000	0.000	node.py:118(size)
6	0.000	0.000	0.000	0.000	{method 'clear' of 'list' objects}
7	0.000	0.000	0.000	0.000	puzzle.py:69(goal_state)
14	0.000	0.000	0.000	0.000	codecs.py:186(__init__)
18	0.000	0.000	0.000	0.000	{built-in method builtins.chr}
14	0.000	0.000	0.000	0.000	{built-in method time.time}
18	0.000	0.000	0.000	0.000	{built-in method builtins.divmod}
14	0.000	0.000	0.000	0.000	dfs_agent.py:12(__str__)
24	0.000	0.000	0.000	0.000	node.py:150(predecessor)
7	0.000	0.000	0.000	0.000	puzzle.py:57(id)
18	0.000	0.000	0.000	0.000	{built-in method builtins.ord}
6	0.000	0.000	0.000	0.000	{method 'reverse' of 'list' objects}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Fig. 7. Agent DFS average time is 67.4ms, 1601322 function calls in 0.513 seconds

```
def h1(self, n) -> float:
    # Count numbers of 1
    return n.state.count("1")
```

Fig. 8. Heuristic Function h1(n)

```
def h2(self, n) -> float:
    # Count numbers of pair of 1
    return n.state.count("11")
```

Fig. 9. Heuristic Function h2(n)

```
def h3(self, n) -> float:
    # Count numbers of three consecutive 1
    return n.state.count("111")
```

Fig. 10. Heuristic Function h3(n)

```
def haround(self, n) -> float:

    count = 0
    size = int(math.sqrt(len(n.state)))

    for i in range(len(n.state)):
        new_count = 0
        if n.state[i] == '1':
            new_count += 1
        if ((i % size) - 1) >= 0:
            if n.state[i - 1] == '1':
                new_count += 1
        if ((i % size) + 1) < size:
            if n.state[i + 1] == '1':
                new_count += 1
        if i - size > 0:
            if n.state[i - size] == '1':
                new_count += 1
        if i + size < len(n.state):
            if n.state[i + size] == '1':
                new_count += 1
        if new_count > count:
            count = new_count
    return count
```

Fig. 11. Heuristic Function haround(n)

```python
def hrows(self, n) -> float:

    count = 0
    size = math.sqrt(len(n.state))
    for i in range(len(n.state)):
        if n.state[i] == '1':
            return count
        elif ((i + 1) % size) == 0:
            count += 1
    return count
```

Fig. 12. Heuristic Function hrows(n)

```python
def h(self, n) -> int:
    x = n.state.count('1')
    count = 0

    if not (0 < x % 5 < 3):
        count += x // 5
        x = x % 5

    if not (0 < x % 4 < 3):
        count += x // 4
        x = x % 4

    count += x // 3
    x = x % 3
    count += x
    return count
```

Fig. 13. Heuristic Function h(n)