Comparison of Heuristic Algorithms in A* Search via Sliding Puzzle Problem

Pei Xu
College of Science and Engineering
Univeristy of Minnesota
Minneapolis, Minnesota 55455
Email: xuxx0884@umn.edu

Abstract—**The efficiency of A* Search depends a lot on how to define its evaluation function. Usually, for a node, the cost from the start node to this node is certain. This means that the efficiency of A* Search, in general, depends on its heuristic algorithm. This paper mainly addresses on comparison of three heuristic algorithms, including Manhattan distance, straight-line distance and Squared Euclidean Distance. Meanwhile, the comparison of A* search to Greedy Best-First Search and to Bidirectional A* search is processed. A sliding puzzle solver using A* Search and Greedy Best-First Search written in Python is used as an experimental tool to demonstrate the theoretical comparison in this paper via solving sliding puzzle problem.**

Keywords*—*A* Search, Greedy Best-First Search, Bi-directional A* Search, Sliding Puzzle Solver.

I. Introduction

Many search problems, especially NP-complete and NP-hard problems, have high complexity like $O(n!)$ and $O(4^n)$, and thus usually cannot be solved in a reasonable amount of time and/or by using a reasonable amount of space. Informed search is designed to use an evaluation function to estimate the desirability of each node and try to reduce the time and/or space used through expanding the most desirable node every time.

A* search is a special case of informed search, whose evaluation function is defined as $f(n) = g(n) + h(n)$ where $g(n)$ represents the actual cost from the start node to the node n and $h(n)$ is a heuristic function representing the estimated cost from the node n to the goal node. This paper focuses on comparison of the efficiencies of three different ways to define $h(n)$ or rather three different heuristic algorithms, which include Manhattan Distance, Straight-Line Distance and Squared Euclidean Distance. As additionally work, in this paper, A* Search is compared to Greedy Best-First Search, another special case of informed search, and to Bidirectional A* Search, and the contrast of their efficiencies and optimality is shown.

In order to proceed the comparison, sliding puzzle problem, a NP-complete problem[7], is proposed as the experimental objective. A program using A* Search, Bidirectional A* Search and Greedy Best-First Search via Manhattan Distance, Straight-Line Distance and Squared Euclidean Distance was coded. The program record these algorithms' performance during solve the sliding puzzle problem for the sake of comparing these algorithms' optimality and time and space complexity.

In the following sections of this paper, introductions to A* Search that uses the heuristic algorithms mentioned above, to Greedy Best-First Search and to Bidirectional A* Search are given firstly, accompanied by a theoretical analysis of their optimality and time and space complexity. Then, in the experiment section, sliding puzzle problem is described and harnessed for testing these search algorithms' performance and demonstrating the conclusions we reach in the theoretical analysis section.

II. A* Search

Differentiating from uninformed search, A* search as a kind of informed search uses an evaluation function to estimate the cost from the start state to the goal state via a node n, and to decide which node will be expanded next. A pseudo-code for A* search is described in Algorithm 1.

The evaluation function of A* search is composed of two parts: $g(n)$ and $h(n)$, which are represented by $COST(start, path)$ and $HEURISTIC(n, goal)$ respectively in Algorithm 1. For a node n, $g(n)$, in general, is known and certain, since it is usually feasible for us to record the path through which we go from the start node to the current node n. Therefore, the heuristic algorithm we use is the key factor that decides the efficiency of A* search.

In the worst case, namely, where the heuristic algorithm cannot effectively play its role, A* search works mainly according to $g(n)$. In this case, A* search trends to be Breadth-First Search, since it always leans to expand the node with the lowest value of $g(n)$, and its space and time complexity is exponential like Breadth-First Search. For Graph-Search, A* search even may need to keep all nodes explored in memory for the sake of avoiding expanding the same node, like what is described in Algorithm 1. Nevertheless, with the optimal heuristic algorithm, which means that A* always can expand the optimal node firstly, its space and time complexity is only $O(bn)$, where n is the length of the path from the start to goal node.

Theorem 1: A* search is complete unless there are infinite acyclic paths in the search tree.

Proof: Every time a node is added to $frontier$, all paths from this node to its neighbors are found by A*. Hence, in a search tree with finite acyclic paths, A* search will keep trying to visit all visitable nodes until it finds a solution if there is at least one solution. ■

Algorithm 1 A* Search

```
procedure COST($start, path$)
    return $actual\ cost\ from\ the\ start\ node\ along$
      $the\ path\ to\ the\ end\ node\ of\ the\ path$
end procedure
procedure HEURISTIC($n, goal$)
    return $estimated\ cost\ from\ the\ node\ n\ to\ the\ goal$
end procedure
procedure A*($problem$)
    $start \leftarrow problem.initial\_node$
    $start.path \leftarrow null$
    $frontier \leftarrow \{start\}$
    $explored \leftarrow an\ empty\ set$
    while $frontier$ do
        $c\_node \leftarrow node\ in\ frontier\ with\ lowest\ f\_val$
        $remove\ c\_node\ from\ frontier$
        if $c\_node\ is\ problem.goal$ then
            return $c\_node.path$
        end if
        $remove\ c\_node\ from\ frontier$
        $add\ c\_node\ to\ explored$
        for $each\ ch\_node\ of\ c\_node$ do
            if $ch\_node\ not\ in\ explored$ then
                $ch\_node.path \leftarrow c\_node.path + c\_node$
                $g\_val \leftarrow$ COST($start, ch\_node.path$)
                if $ch\_node\ in\ frontier$ then
                    $n \leftarrow frontier[ch\_node]$
                    if COST($start, n.path$) $> g\_val$ then
                        $remove\ n\ from\ frontier$
                    else
                        continue
                    end if
                end if
            end if
            $h\_val \leftarrow$ HEURISTIC($ch\_node, goal$)
            $ch\_node.f\_val \leftarrow g\_val + h\_val$
            $add\ ch\_node\ to\ frontier$
        end for
    end while
    return $failure$                              ▷ No solution found.
end procedure
```

Theorem 2: A* search is optimal when the heuristic function $h(n)$ is consistent.

Proof: A consistent $h(n)$ means

$$h(n) \leq c(n, n') + h(n') \tag{1}$$

where $c(n, n')$ represents the cost from the node n to its adjacent node n'.

This equation guarantees that

$$g(n) + h(n) \leq g(n) + c(n, n') + h(n') = g(n') + h(n') \tag{2}$$

Given that $f(n) = g(n) + h(n)$ and from the equation (2), when the heuristic function $h(n)$ is consistent, the evaluation function $f(n)$ is non-decreasing.

Considering A* search will always firstly expand the node whose evaluation function has the lowest value, the fist goal node A* finds must be the optimal solution. ■

Theorem 3: Suppose that the evaluation function f of the optimal solution S has the value C, namely, $f(S) = C$, when using A* search, all nodes whose evaluation function has a higher value than C must not be expanded.

Proof: Given that A* search will always firstly expand the node whose evaluation function has the lowest value, A* search will find the optimal solution S before any node whose evaluation function has a higher value than C is expanded, and then directly return the optimal solution. ■

Theorem 3 also means that A* Search is also optimally efficient. That is to say, for a given evaluation function, no other informed search can do better than A* search when used to find optimal solutions.

Theorem 4: Given two evaluation functions who have the same $g(n)$ and in which two heuristic functions $h_1(n)$ and $h_2(n)$ are both consistent, if, for a same node n, $h_1(n) \leq h_2(n)$, A* search using $h_2(n)$ is more efficient than that using $h_1(n)$.

Proof: From Theorem 3, we can conclude that A* search using $h_1(n)$ will not expand more nodes than that using $h_2(n)$. Hence the former one is more efficient than the latter. ■

In summary, A* search is complete, optimal and optimally efficient. Through increasing the accuracy of $h(n)$, a more efficient evaluation function can be obtained, if the price of computing $h(n)$ is considered free.

A. Heuristics of Manhattan Distance v.s. Straight-Line Distance

definition 1: The Manhattan Distance from a point $p(x, y)$ to a point $q(x, y)$ is defined as

$$d_M(p, q) = |p.x - q.x| + |p.y - q.y| \tag{3}$$

definition 2: The Straight-Line Distance or Euclidean distance from a point $p(x, y)$ to a point $q(x, y)$ is defined as

$$d_E(p, q) = \sqrt{(p.x - q.x)^2 + (p.y - q.y)^2} \tag{4}$$

If a point p can walk along any direction to the goal node q, $d_E(p, q)$ is obviously accurate; and, in this case, $d_M(p, q)$ would overestimate the distance between two points and, thus, is not consistent.

If a point only can walk along four directions, namely, up, down, left and right, $d_M(p, q)$ is accurate; and, in this case, $d_E(p, q)$ still will not overestimate the distance between two points and is consistent. This means that, in such a case, the heuristics of Manhattan Distance and Straight-Line Distance both can be used to find the optimal solution (from Theorem 2) but the former one is obviously more efficient since it is accurate.

In general, given a case in which both of heuristics of Manhattan Distance and Straight-Line Distance are consistent, from Theorem 4, we can conclude that A* search using the heuristic of Manhattan Distance is more efficient than that using the heuristic of Straight-Line Distance, since $d_E(n) \leq h_M(n)$.

B. Heuristics of Squared Euclidean Distance

definition 3: The Squared Euclidean Distance from a point $p(x, y)$ to a point $q(x, y)$ is defined as

$$d_S(p, q) = (p.x - q.x)^2 + (p.y - q.y)^2 \tag{5}$$

When Squared Euclidean Distance is used as the heuristic function, $f(n)$ and $h(n)$ will have different scales, and $h(n)$ is likely to be much higher than the actual cost from the node n to the goal node and is also likely to be overestimated, which means not consistent. In this case, A* search is not optimal and but trend to generate fewer nodes during finding the solution, namely, trends to work like Greedy Best-Fist Search, which is introduced in the next subsection,

C. Conversion to Greedy Best-First Search

Greedy Best-First Search (GBFS) uses only the heuristic function $h(n)$ as its evaluation function. That is to say, A* Search can be converted into GBFS via defining $g(n) = 0$.

Suppose that $h(n)$ is the estimated cost from the node n to the goal node via a certain heuristic algorithm and that $h^*(n)$ represents the actual cost from the node n to the goal node, GBFS obviously cannot guarantee its solution's optimality even if the heuristic algorithm is consistent, since GBFS always trends to expand the node with the lowest estimated value to the goal node rather than the node who is really the most near to the goal node, unless $h(n)$ is accurate or, at least, directly proportional to $h^*(n)$.

In the worst case, GBFS will expand all nodes before it expands deepest nodes. In this worst case, its time and space complexity is the same to that of A* search. Nevertheless, without the consideration of optimality, in general, Greedy Best-First Search is better, since Greedy Best-First Search will not necessarily expand all nodes whose $f(n) < C$ where C is the value of evaluation function for the optimal solution, thereby expanding less nodes than A* will do (according to Theorem 3).

D. Bidirectional A* Search

The idea of Bidirectional A* Search is to run two A* search instances at once, usually, one of which is from the start point looking for the path to the goal point and the other one of which is from the goal point looking for the path to the start point.

Bidirectional A* Search will largely reduce the space and time complexity, especially in worse cases. In the worst case, each A* search instance processed by Bidirectional A* Search only needs $O(b^{d/2})$, in contrast with $O(b^d)$ of A* Search, where b is the branching factor and d is the depth of the solution.

However, Bidirectional A* search cannot guarantee the optimality. Suppose that $cost(a, b)$ is the cost from the node a to b and that the Bidirectional A* Search finds the solution S when its two A* search instances reach the node n, then the cost of the solution S can be obtained as

$$cost(S) = cost_{min}(start, n) + cost_{min}(n, goal) \tag{6}$$

Algorithm 2 Heuristic Algorithms for Sliding Puzzle Problem

```
procedure MANHATTAN(node)
    $distance \leftarrow 0$
    for $each\ slot\ of\ node.state$ do
        $pos \leftarrow slot.position$
        $num \leftarrow the\ number\ in\ the\ slot$
        $g\_pos \leftarrow the\ position\ of\ num\ in\ goal\ state$
        if $pos \neq g\_pos$ then
            $distance \leftarrow distance + |pos.x - g\_pos.x| + |pos.y - g\_pos.y|$
        end if
    end for
    return $distance$
end procedure
procedure STRAIGHT-LINE(node)
    $distance \leftarrow 0$
    for $each\ slot\ of\ node.state$ do
        $pos \leftarrow slot.position$
        $num \leftarrow the\ number\ in\ the\ slot$
        $g\_pos \leftarrow the\ position\ of\ num\ in\ goal\ state$
        if $pos \neq g\_pos$ then
            $distance \leftarrow distance + \sqrt{(pos.x - g\_pos.x)^2 + (pos.y - g\_pos.y)^2}$
        end if
    end for
    return $distance$
end procedure
procedure SQUARED-EUCLIDEAN(node)
    $distance \leftarrow 0$
    for $each\ slot\ of\ node.state$ do
        $pos \leftarrow slot.position$
        $num \leftarrow the\ number\ in\ the\ slot$
        $g\_pos \leftarrow the\ position\ of\ num\ in\ goal\ state$
        if $pos \neq g\_pos$ then
            $distance \leftarrow distance + (pos.x - g\_pos.x)^2 + (pos.y - g\_pos.y)^2$
        end if
    end for
    return $distance$
end procedure
```

Obviously, $cost(S)$ is not necessarily equal to $cost_{min}(start, goal)$. This means that the solution S is not necessarily the optimal solution. However, the solution S is a suboptimal solution, since it is composed of two optimal solutions if the heuristic algorithm is consistent.

III. EXPERIMENT

Sliding Puzzle Problem is used to test the comparison mentioned in the last section. In the experimented sliding puzzle problems, the blank slot only can move along four directions, namely, up, down, left and right, but cannot move across the four frontier sides; and the goal states are represented in Figure 1.

1	2	3
4	5	6
7	8	

Fig. 1. The Goal State of a 8-Sliding Puzzle

The heuristic algorithms of Manhattan Distance, Straight-Line Distance, and Squared Euclidean Distance used for solving sliding puzzle problems are described in Algorithm 2.

A. *Experimental Hypotheses*

According to the rule mentioned in the last paragraph, in the experiment, heuristics of Manhattan Distance and Straight-Line Distance are both consistent, but the heuristic of Squared Euclidean Distance is not consistent. According to the theoretical analysis in the last sections, we can get five hypotheses **for the sliding puzzle problem**.

Hypothesis 1: From Theorem 2, both of A* Search using Manhattan Distance and that using Straight-Line Distance should be optimal.

Hypothesis 2: From Theorem 4 and Section II-A Manhattan Distance v.s. Straight-Line Distance, A* Search using Manhattan Distance trends to generate fewer nodes than that using what Straight-Line Distance does.

Hypothesis 3.1: From Section II-B Squared Euclidean Distance, A* using Squared Euclidean Distance is not optimal, namely, not always able to find the optimal solution; and *Hypothesis 3.2:* A* using Squared Euclidean Distance trends to generate fewer nodes than that using Manhattan.

Hypothesis 4.1: From Section II-C Greedy Best-First Search, Greedy Best-First Search is not optimal; and

Hypothesis 4.2: Greedy Best-First Search, in general, is faster.

Hypothesis 5.1: From Section II-D Bidirectional A* Search, Bidirectional A* Search cannot guarantee the optimality; and

Hypothesis 5.2: Bidirectional A* Search, in general, is faster than A* Search.

B. *Metrics*

In sliding puzzle problems, all nodes explored need to be recorded for the sake of avoiding repeatedly visiting the same nodes, namely, it is a GRAPH-SEARCH. This means that all nodes generated during the process of solving a sliding puzzle problem always need to be kept in memory, and the time and space complexity are both directly proportional to the amount of nodes generated. Hence, in the experiment, the **amount of nodes generated** during solving a puzzle was record for the comparison of both the **time** and **space complexity**. As for the optimality, the **length** of the solution found for a puzzle was recorded for the comparison of these algorithms' **optimality**, and it represents the depth of solution d.

C. *Experimental Environment*

The experiment was run at a MacBook Pro with 2.7GHz Intel Core i7 and 16GB 1600 MHz DDR3.

The program was written in Python and run at Python v3.4.1.

D. *Experimental Process*

In the experiment, the program firstly generated 10000 solvable 8-sliding puzzles randomly, and solved these puzzles via A* Search, Bidirectional A* Search and GBFS using Manhattan, Straight-Line and Squared Euclidean Distance. Namely, 9 kinds of algorithms were tested.

For each algorithm, the program recorded the amount of nodes generated and the running time taken by the solver to solve a puzzle, and the length of the solutions found by the solver; and wrote them into a log file.

Finally, the program compared these algorithms via analyzing the log file.

E. *Experimental Results*

The average lengths of solutions, the average amount of nodes generated and the average time taken by each algorithm during the experiment are shown in Table 1, 2, 3 respectively. Comparing Table 2 and 3, we can spot that a same sequence can be obtained no matter we order the three heuristics via the average amount of nodes generated or via the average time taken. This demonstrates that it is reasonable to use the amount of nodes generated as the criterion to measure time complexity, which we discussed in Section III-B Metrics.

TABLE I. AVERAGE LENGTHS OF SOLUTIONS

	A*	Bi-A*	GBFS
Straight Line Distance	21.9814	22.9464	49.5216
Manhattan Distance	21.9814	23.2764	43.9784
Squared Euclidean Distance	22.5080	23.7728	40.2700

TABLE II. AVERAGE NODES GENERATED

	A*	Bi-A*	GBFS
Straight Line Distance	3806.3337	829.0245	449.8505
Manhattan Distance	2450.1913	783.5216	479.8310
Squared Euclidean Distance	589.3139	432.9511	280.9169

TABLE III. AVERAGE TIME TAKEN

	A*	Bi-A*	GBFS
Straight Line Distance	5.4040	0.1627	0.0518
Manhattan Distance	2.0178	0.1442	0.0611
Squared Euclidean Distance	0.1185	0.0551	0.0235

Unit: second

1) Optimality: The steps of solutions found by each algorithm are used to demonstrate the optimality of the algorithm.

In Table 1, it can be recognized that A* Search using Manhattan Distance and that using Straight-Line Distance had the same and fewest average length of solutions. In fact, the author compared the lengths of the solution found by the nine algorithms for each puzzle, recorded the number of times when each algorithm found optimal solutions (shown in Table 4), and found that A* Search using Manhattan Distance and A* Search using Straight-Line Distance always could find the same solutions, which were the shortest, namely, the optimal, solutions as well.

TABLE IV. TIMES TO FIND OPTIMAL SOLUTIONS

	A*	Bi-A*	GBFS
Straight Line Distance	10000	6703	395
Manhattan Distance	10000	6029	447
Squared Euclidean Distance	7827	5595	804

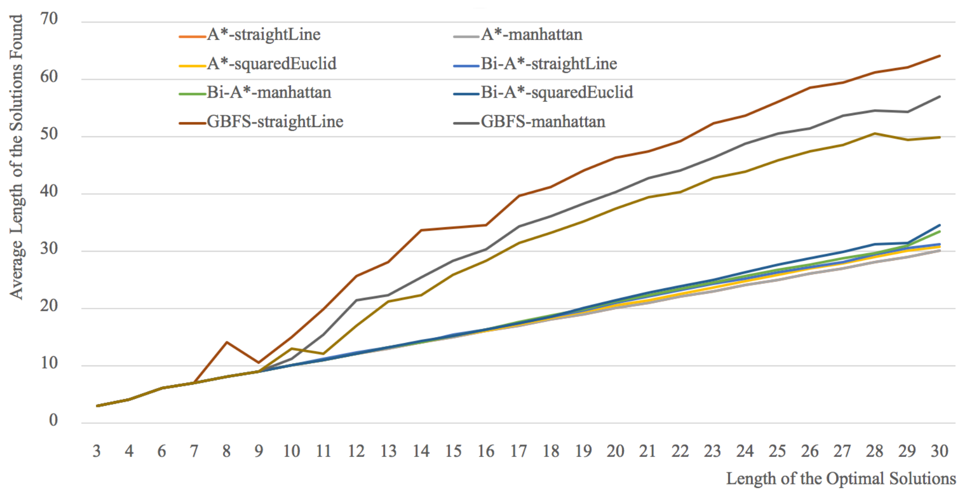

Fig. 2. Average Lengths of Solutions Found by each Algorithm in the case where the the puzzle has the given Lengths of Optimal Solutions

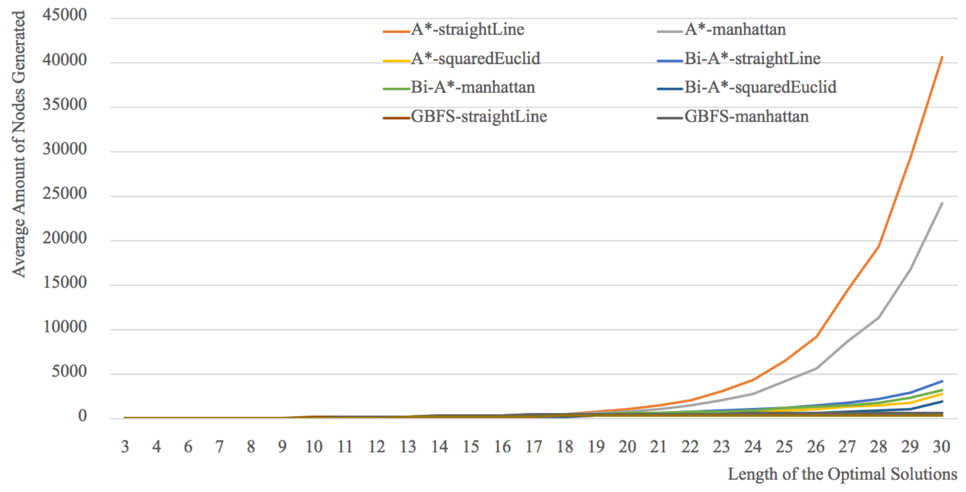

Fig. 3. Average Amounts of Nodes Generated by each Algorithm in the case where the the puzzle has the given Lengths of Optimal Solutions

Besides, Bidirectional A* Search using Manhattan Distance and that using Straight-Line Distance were not always able to find the shortest solutions, like A* Search using Squared Euclidean Distance that is not consistent for this problem. This means that Bidirectional A* Search is not optimal even if its heuristic algorithm is consistent.

This demonstrates *Hypothesis 1*, *Hypothesis 3.1*, *Hypothesis 4.1* and *Hypothesis 5.1*. Namely, among these nine algorithms, A* Search using Manhattan Distance and that using Straight-Line Distance are the only two algorithms who are optimal.

2) *Time and Space Complexity:* The average amount of nodes generated by each algorithm is used to compare time and space complexity.

TABLE V. TIMES TO GENERATE FEWER NODES

A* Manhattan v.s. A* Straight Line

A* Manhattan	A* Straight-Line
9934	0

Table 5 records how many times A* Search using Manhattan Distance and A* Search using Straight-Line Distance generated fewer nodes during solving a sliding puzzle respectively. From this table, it is clear that A* Search using Manhattan Distance is better than A* Search using Straight-Line Distance. This demonstrates *Hypothesis 2*.

From Table 2, we can find that A* Search using Squared Euclidean, in general, generated fewer nodes than that using Manhattan or Straight-Line Distance did, which demonstrates *Hypothesis 3.2*; but generated more nodes compared to GBFS or Bidirectional A* Search, which demonstrates *Hypothesis 4.2* and *Hypothesis 5.2*.

This result also means that A* Search using Squared Euclidean has lower time and space complexity compared to that using Manhattan Distance and that Bidirectional A* Search and GBFS have more lower time and space complexity.

Figure 2 shows the average lengths of the solutions found by each algorithm in the case where the optimal solution has the given lengths. From Figure 2, we can identify that as the length of the optimal solutions increase, the average length of the solutions found by each algorithm, basically, increases linearly.

Figure 3 shows the average amount of nodes generated by each algorithm in the case where the optimal solution has the given lengths. From Figure 3, we can identify that the time and space complexity of A* Search using Manhattan Distance and that using Straight-Line Distance have a distinct exponential increasing trend. This demonstrates our analysis in Section II A* Search. Nevertheless, for the other eight algorithms, there is no significant increase as the lengths of the optimal solutions increases, especially for GBFS, although A* Search using Squared Euclidean Distance and Bidirectional A* Search have an exponential increasing trend in time and space complexity as well.

Synthesizing the data shown in Figure 2 and 3, we reach the result that as the optimal solution's length increases, namely, as the factor d increases, using the eight algorithms mentioned above, although cannot guarantee optimality still, but will not need to spend significantly more time or memory. Especially Bidirectional A* Search, it is able to find the suboptimal solution in pretty lower need of time and memory (this illustrates our analysis in Section II-D Bidirectional A* Search), and, meanwhile, has a better increasing curve of the time and space complexity, although it is also exponential, as d increase.

IV. Conclusions and Future Work

This paper introduced A* Search, Greedy Best-First Search and Bidirectional A* Search, and analyzed their optimality and time and space complexity. Besides, three heuristics, including Manhattan Distance, Straight-Line Distance and Squared Euclidean Distance, are introduced. The author speculated these algorithms' properties and demonstrated the hypotheses via analyzing their performance during solving sliding puzzles.

In Section II Introduction to A* Search, we proved that A* Search is complete, and is optimal and optimally efficient if the heuristic algorithm is consistent. Furthermore, A* Search's time and space complexity in GRAPH-SEARCH situation is exponential. In contrast, GBFS and Bidirectional A* Search are not optimal even if the heuristic algorithm used is consistent.

In Section III Experiment, basically, we reached five conclusions. First, Heuristics of Manhattan Distance and Straight-Line Distance can be used for A * Search to find the optimal solution. Second, Manhattan Distance is better than Straight-Line Distance for A* Search solving sliding puzzle problem. Third, A* Search using Squared Euclidean Distance is not optimal. Fourth, Greedy Best-First Search is not optimal as well, but it, in general, is faster. Fifth, Bidirectional A* Search cannot guarantee the optimality, but it, in general, is faster than A* Search. We argued these conclusions minutely in Section III-E Experimental Result. We further observed the increasing curve of these algorithms' time and space complexities as the length of the optimal solutions increases, and found that in the experimental environment A* Search and Bidirectional A* Search both have an exponential time and space complexity but that it is much more obvious for the former one. Through these observation, considering the optimality and time and space complexity, the author supposed that Bidirectional A* may perform better than A* Search and GBFS as the factor d increases. These conclusions we reached and observation we got in Section III Experiment are consistent with our analysis in Section II Introduction to A* Search.

From the experiment results, we find that A* Search using Squared Euclidean Distance performed not so bad. In fact, in most situations, it is able to find optimal solutions and it generated much lower nodes compared to A* Search using Manhattan Distance and that using Straight-Line Distance. if only considering the cases in which A* Search using Squared Euclidean Distance found the optimal solutions, we can find that this algorithm generated fewer nodes in more than 98% cases (shown in Table 6). In the experiment, $g(n)$ and $h(n)$ of A* using Squared Euclidean Distance have two scales, which makes $h(n)$ have a higher specific weight in $f(n)$. The author suppose that A* Search may perform better if we advisably increase the weight of $h(n)$. We will focus our work on this supposition in the next phase.

TABLE VI. Times to Generate Fewer Nodes in the case both of the two algorithms found the optimal solutions

A* Manhattan v.s. A* Squared Euclidean Distance

A* Manhattan	A* Squared Euclidean Distance
122	7705

References

[1] Stuart Russell and Peter Norvig, *Artificial Intelligence - A Modern Approach*, 3rd Edition. 2010.

[2] Carlos Hernandez, Roberto Asin and Jorge A. Baier, *Time-Bounded Best-First Search*. 2014.

[3] Ivica Martinjak and Marin Golub, *Comparison of Heuristic Algorithms for the N-Queen Problem*. 2007.

[4] Christopher Wilt, Jordan Thayer and Wheeler Ruml, *A comparison of greedy search algorithms*. 2010.

[5] Amit Patel, *Introduction to A**. Available at: http://www.redblobgames.com

[6] Wikipedia, *A* search algorithm*. Available at: https://en.wikipedia.org/wiki/A*_search_algorithm

[7] D. Ratner and M. Warmuth, *Finding A Shortest Solution for the N*N-Extension of the 15-Puzzle Is Intractable*. 1990.