CMPT417 Project Final Report

A+ group

Xiaolu Zhu (#301297674)

Yusong Cai (#301269793)

Joo-Young Shim (#301023141)

Introduction

In this project, we presented 5 different search algorithms for solving a sliding puzzle. We focused on breadth-first search(BFS), depth-first search(DFS), and iterative deepening depth-first search(IDDFS), A*, iterative deepening A*(IDA*). We tested the algorithm performance on a 3x3 puzzle. We compared these algorithms by their solution length, time cost, the number of nodes expanded, and memory usage. We also created different scenarios (puzzle types) to avoid contingency and increased data accuracy. Users can also get specific data reported about the performance of each algorithm at the end of each game round, which is saved as a .txt file.

Setup

We set up our development environment using Python (version 3.5 and above from https://www.python.org/downloads), Git(https://git-scm.com/downloads) and a text editor(e.g. Visual Studio Code). The environment can be set up cross-platform (Windows, macOS, Ubuntu Linux). Additionally, pygame(Python package) needs to be installed using pip. For detailed instruction, we created a README file at GitHub repository: https://github.com/j-shim/CMPT417GroupProject

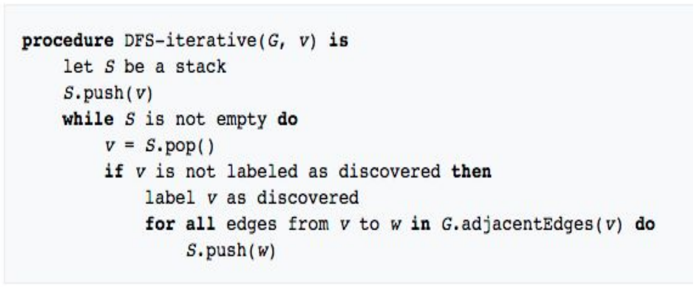
We tested the implementation on three different machines:

- macOS 10.13 High Sierra, 2.4 GHz Intel Core i5, 16 GB RAM
- Windows 10 Inspiron 13, Intel Core i5, 8 GB RAM
- Windows 10 Y520, Intel Core i7, 8 GB RAM

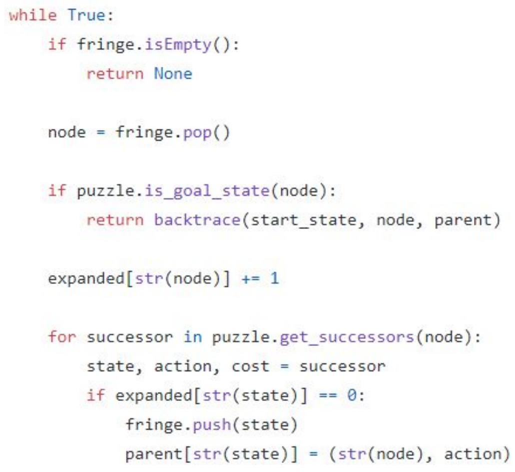
To test the implementation, clone or download the repository, open up a terminal and change directory to the root of the repository and make sure pygame package is installed by running command “pip install pygame”. Finally, run command “python src/main.py”.

Implementation

DFS search



```
procedure DFS-iterative(G, v) is
    let S be a stack
    S.push(v)
    while S is not empty do
        v = S.pop()
        if v is not labeled as discovered then
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                S.push(w)
```

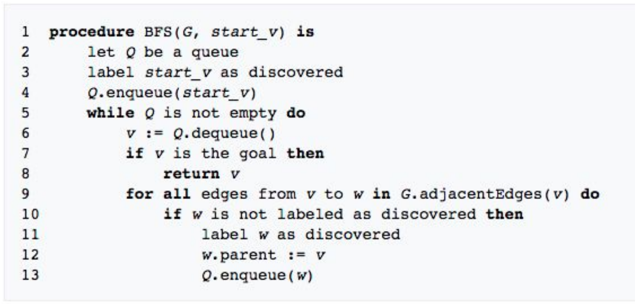


```
while True:
    if fringe.isEmpty():
        return None

    node = fringe.pop()

    if puzzle.is_goal_state(node):
        return backtrace(start_state, node, parent)

    expanded[str(node)] += 1

    for successor in puzzle.get_successors(node):
        state, action, cost = successor
        if expanded[str(state)] == 0:
            fringe.push(state)
            parent[str(state)] = (str(node), action)
```

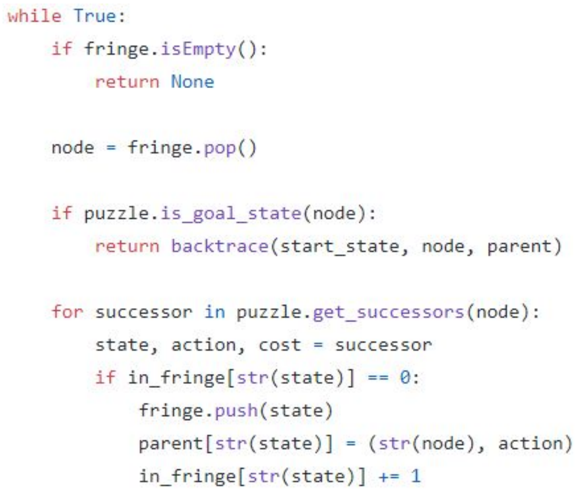
Figure 1. Pseudocode of DFS and depth_first_search

We make our depth_first_search by referring to the pseudocode of DFS. In the *depth_first_search* function we import stack class from util and push the root to the stack first. We considered the empty case of the puzzle and made sure we searched the node in order of depth first. When the node is the goal node, it will call the backtrace function, which will back trace each step in puzzle action and return a list of actions that leads to the goal state (e.g. [“Up”, “Left”, “Left”, “Down”, “Right”]).

BFS search



```
procedure BFS(G, start_v) is
    let Q be a queue
    label start_v as discovered
    Q.enqueue(start_v)
    while Q is not empty do
        v := Q.dequeue()
        if v is the goal then
            return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as discovered then
                label w as discovered
                w.parent := v
                Q.enqueue(w)
```

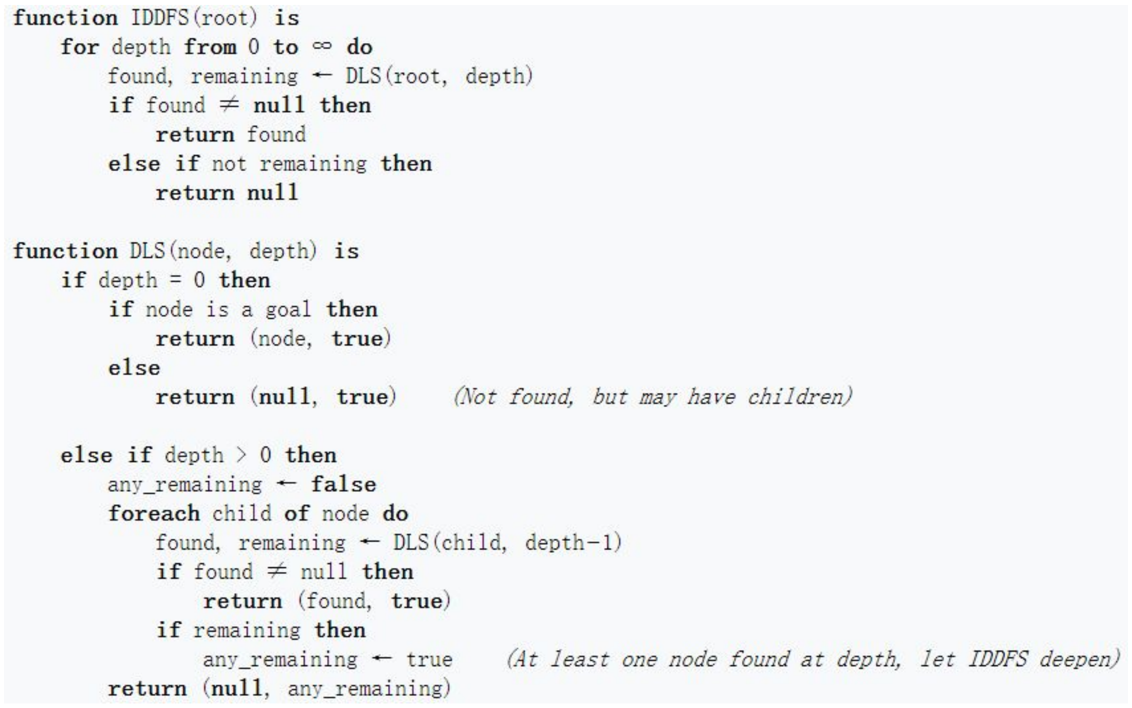


```
while True:
    if fringe.isEmpty():
        return None

    node = fringe.pop()

    if puzzle.is_goal_state(node):
        return backtrace(start_state, node, parent)

    for successor in puzzle.get_successors(node):
        state, action, cost = successor
        if in_fringe[str(state)] == 0:
            fringe.push(state)
            parent[str(state)] = (str(node), action)
            in_fringe[str(state)] += 1
```

Figure 2. Pseudocode of BFS and breadth_first_search

In BFS search, we first check the empty case of the puzzle. Different from DFS, in BFS we import queue class from util, and go through all the nodes in a breadth way.

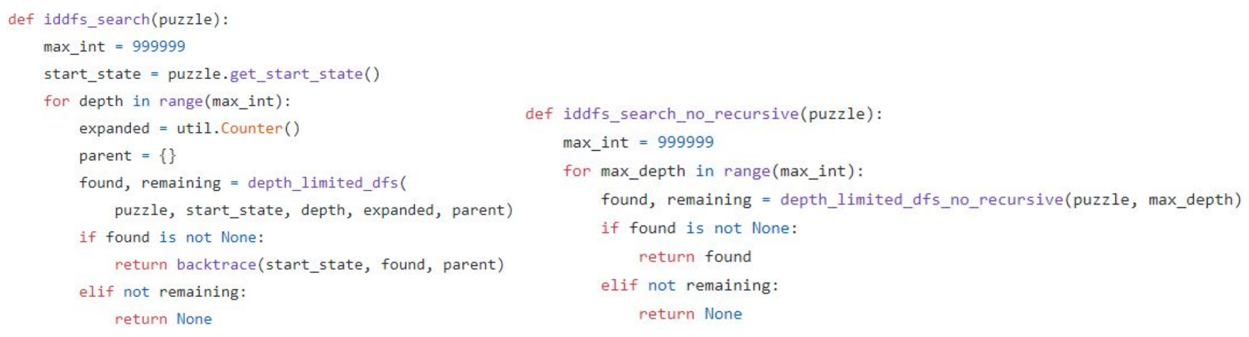
IDDFS search



```
function IDDFS(root) is
    for depth from 0 to ∞ do
        found, remaining ← DLS(root, depth)
        if found ≠ null then
            return found
        else if not remaining then
            return null

function DLS(node, depth) is
    if depth = 0 then
        if node is a goal then
            return (node, true)
        else
            return (null, true)    (Not found, but may have children)

    else if depth > 0 then
        any_remaining ← false
        foreach child of node do
            found, remaining ← DLS(child, depth−1)
            if found ≠ null then
                return (found, true)
            if remaining then
                any_remaining ← true    (At least one node found at depth, let IDDFS deepen)
        return (null, any_remaining)
```

Figure 3. iddfs_search pseudocode

In the IDDFS search section in search.py file,we implement two versions of IDDFS search which are the recursion version and non-recursion version of IDDFS by referring to the pseudocode for IDDFS.

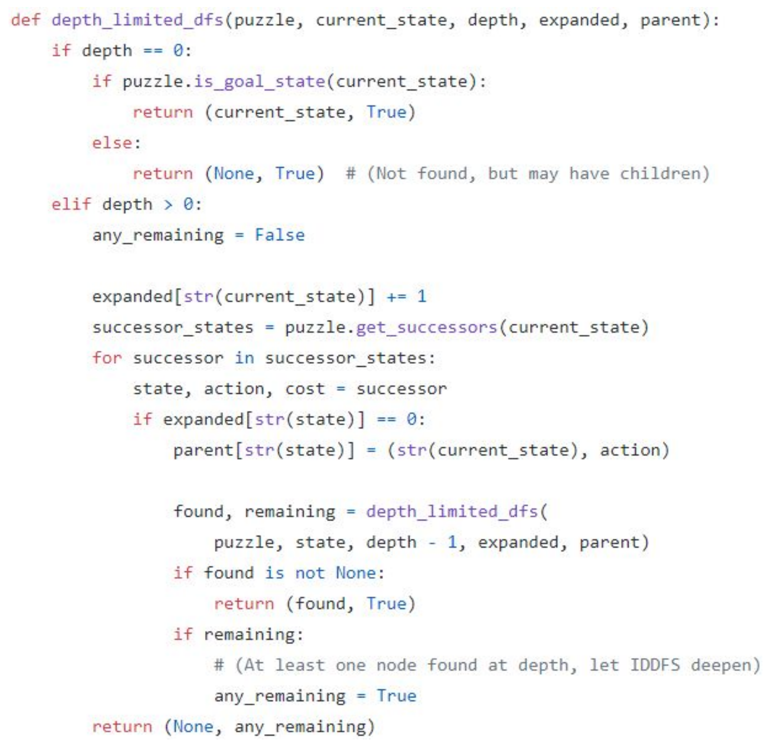


```python
def iddfs_search(puzzle):
    max_int = 999999
    start_state = puzzle.get_start_state()
    for depth in range(max_int):
        expanded = util.Counter()
        parent = {}
        found, remaining = depth_limited_dfs(
            puzzle, start_state, depth, expanded, parent)
        if found is not None:
            return backtrace(start_state, found, parent)
        elif not remaining:
            return None
```

```python
def iddfs_search_no_recursive(puzzle):
    max_int = 999999
    for max_depth in range(max_int):
        found, remaining = depth_limited_dfs_no_recursive(puzzle, max_depth)
        if found is not None:
            return found
        elif not remaining:
            return None
```

Figure 4. iddfs_search vs iddfs_search_no_recursive

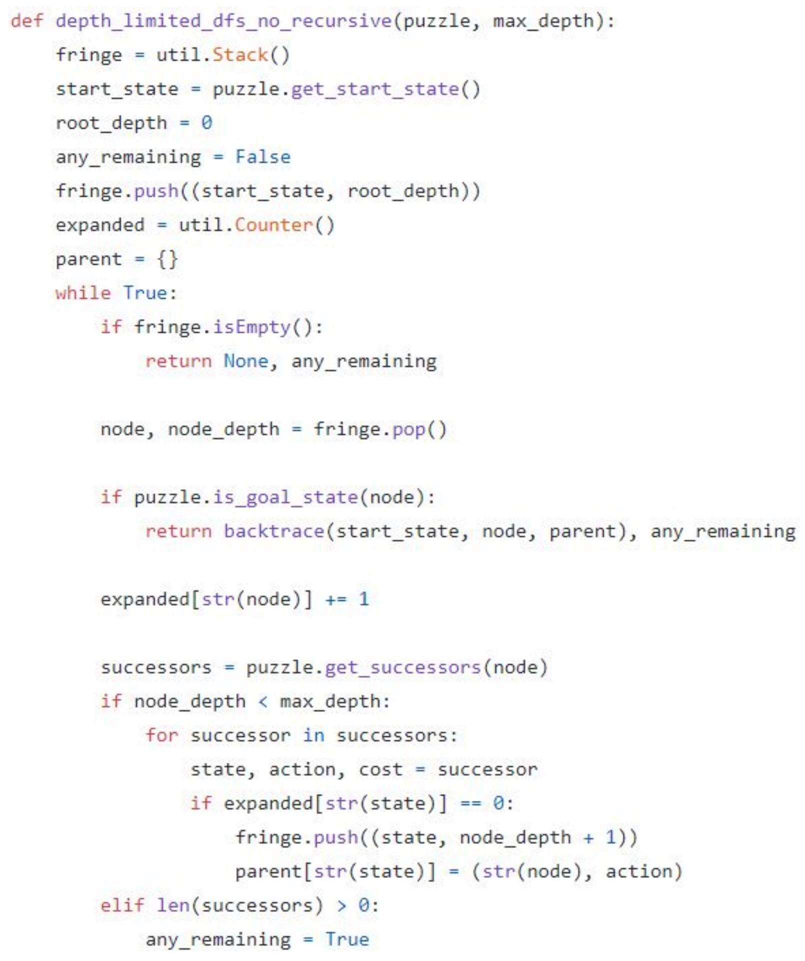
Compared with the recursion version and non-recursion version of IDDFS the difference is in the recursive version, the recursion helper function depth_limited_dfs will be called to return a tuple *(found, remaining)*. Found will either be *None* or the goal state, and *remaining* will let us know if we have some child nodes that are not expanded yet. We repeat search if *remaining* returns *True*. When *found* returns a goal state, we call backtrace function to return the solution as a list of actions.



```
def depth_limited_dfs(puzzle, current_state, depth, expanded, parent):
    if depth == 0:
        if puzzle.is_goal_state(current_state):
            return (current_state, True)
        else:
            return (None, True)  # (Not found, but may have children)
    elif depth > 0:
        any_remaining = False

        expanded[str(current_state)] += 1
        successor_states = puzzle.get_successors(current_state)
        for successor in successor_states:
            state, action, cost = successor
            if expanded[str(state)] == 0:
                parent[str(state)] = (str(current_state), action)

                found, remaining = depth_limited_dfs(
                    puzzle, state, depth - 1, expanded, parent)
                if found is not None:
                    return (found, True)
                if remaining:
                    # (At least one node found at depth, let IDDFS deepen)
                    any_remaining = True
        return (None, any_remaining)
```

Figure 5. Iddfs_search recursion version depth_limited_dfs helper function

In the helper function depth_limited_dfs, we first consider the empty case and move to the next step if the puzzle is not an empty puzzle. Each node goes through, will be added to the expanded list and it will go through all the branches in the search tree. When we are finding the target node, we use a recursive method here. If *found* is not *None* and the node is not found, we will call depth_limited_dfs again to do the next round search until we can find the solution.

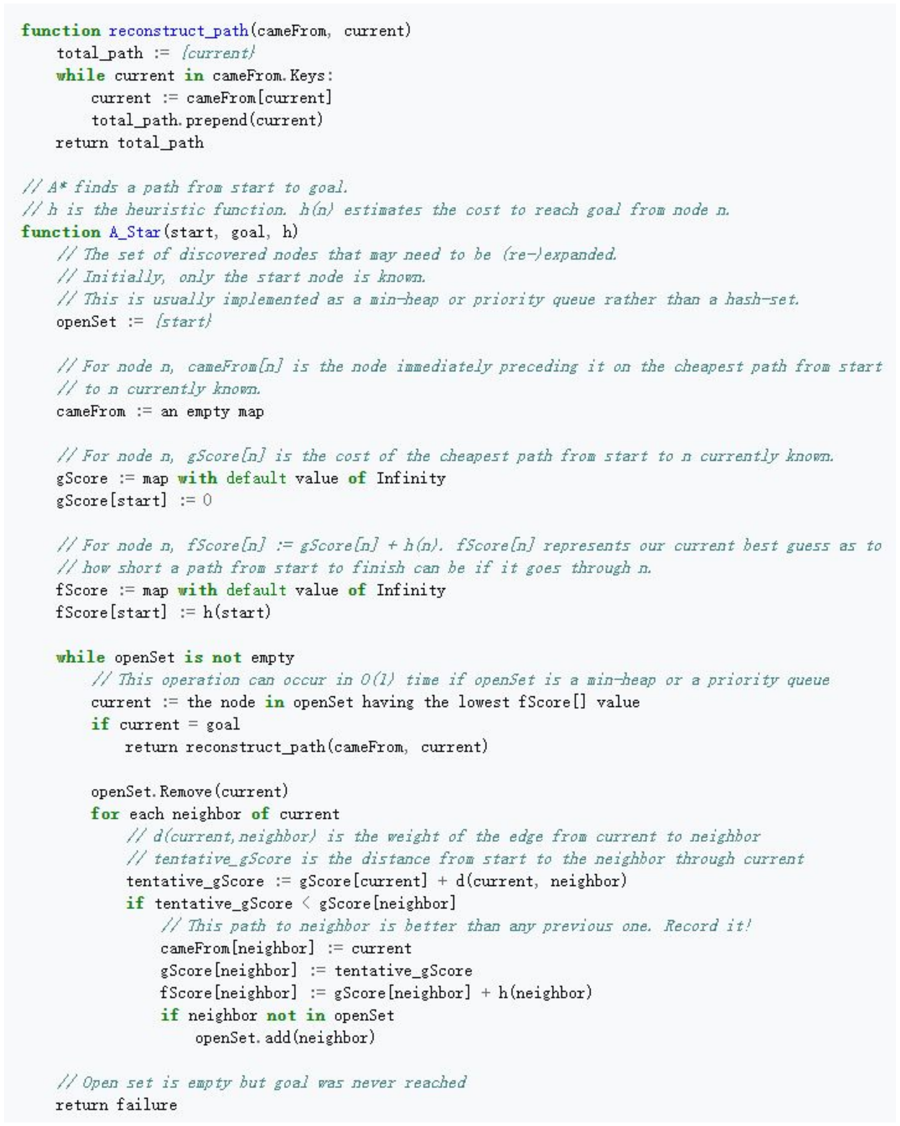


```
def depth_limited_dfs_no_recursive(puzzle, max_depth):
    fringe = util.Stack()
    start_state = puzzle.get_start_state()
    root_depth = 0
    any_remaining = False
    fringe.push((start_state, root_depth))
    expanded = util.Counter()
    parent = {}
    while True:
        if fringe.isEmpty():
            return None, any_remaining

        node, node_depth = fringe.pop()

        if puzzle.is_goal_state(node):
            return backtrace(start_state, node, parent), any_remaining

        expanded[str(node)] += 1

        successors = puzzle.get_successors(node)
        if node_depth < max_depth:
            for successor in successors:
                state, action, cost = successor
                if expanded[str(state)] == 0:
                    fringe.push((state, node_depth + 1))
                    parent[str(state)] = (str(node), action)
        elif len(successors) > 0:
            any_remaining = True
```

Figure 6. Iddfs_search non-recursion version depth_limited_dfs helper function

For non-recursion versions of IDDFS, we need to import stack from util. Similar to the recursion version, we need to check if the puzzle is empty first, and push the node to the fringe list. If we find the node we want, we return the list. We set up the *max_depth* value and compared it with *node_depth*. Compared with BFS, it saves time because we set the limit of the depth. The reason we implement IDDFS in two ways is because we found out that, in a small size puzzle such as 8-puzzle, non-recursive version of IDDFS takes less time and memory to finish the task and it's optimal for IDDFS.

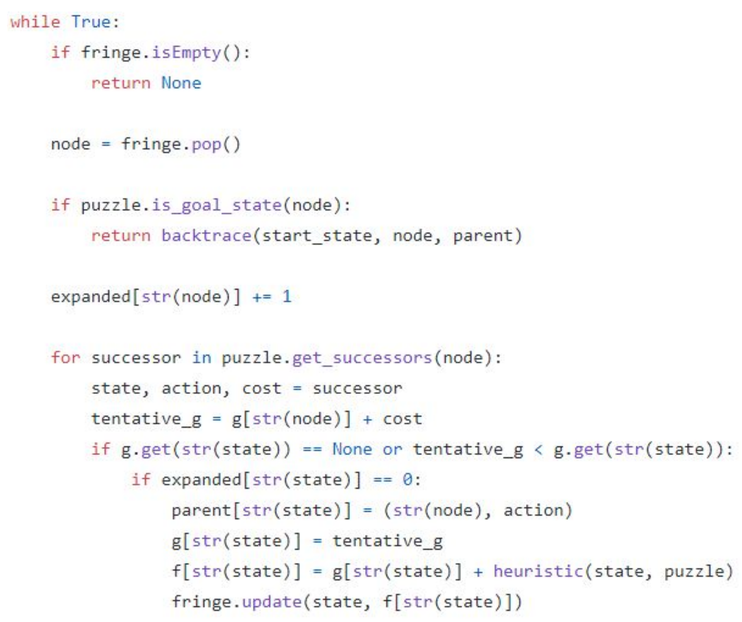
A* search



```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path

// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach goal from node n.
function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    // This is usually implemented as a min-heap or priority queue rather than a hash-set.
    openSet := {start}

    // For node n, cameFrom[n] is the node immediately preceding it on the cheapest path from start
    // to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from start to n currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0

    // For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our current best guess as to
    // how short a path from start to finish can be if it goes through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)

    while openSet is not empty
        // This operation can occur in O(1) time if openSet is a min-heap or a priority queue
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        for each neighbor of current
            // d(current,neighbor) is the weight of the edge from current to neighbor
            // tentative_gScore is the distance from start to the neighbor through current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
                // This path to neighbor is better than any previous one. Record it!
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := gScore[neighbor] + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)

    // Open set is empty but goal was never reached
    return failure
```

Figure 7. A search pseudocode*

We implemented a_star_ search by referring A* search pseudocode, A* search is another way of search method we used in puzzle game and it aims to find a way to the given node which takes the smallest cost (f = g + heuristic) and we will be comparing A* search with other search methods later in the Methodology and Experimental outcomes sections.

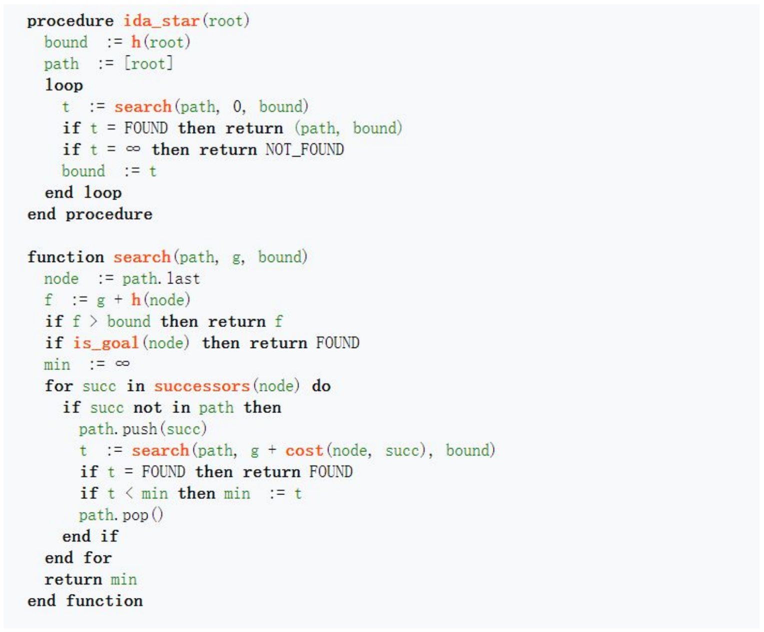


```
while True:
    if fringe.isEmpty():
        return None

    node = fringe.pop()

    if puzzle.is_goal_state(node):
        return backtrace(start_state, node, parent)

    expanded[str(node)] += 1

    for successor in puzzle.get_successors(node):
        state, action, cost = successor
        tentative_g = g[str(node)] + cost
        if g.get(str(state)) == None or tentative_g < g.get(str(state)):
            if expanded[str(state)] == 0:
                parent[str(state)] = (str(node), action)
                g[str(state)] = tentative_g
                f[str(state)] = g[str(state)] + heuristic(state, puzzle)
                fringe.update(state, f[str(state)])
```

Figure 8. Partial a_star_search implementation

In a_star_search, we also first check the empty case and return the value by calling the backtrace function if the current node is the goal node. We used tentative_g to calculate the cost for the current state and use helper function heuristic to calculate the approximate distance between the current node and the goal state.

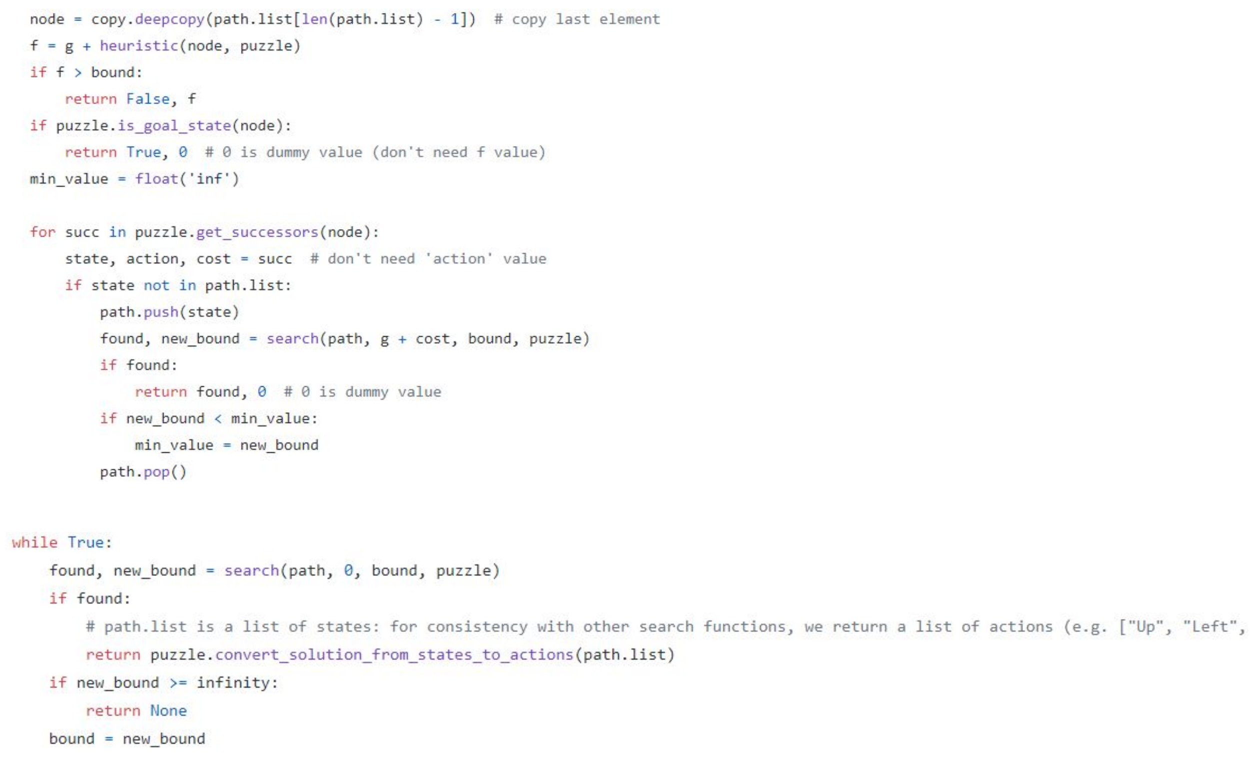
IDA* search



```
procedure ida_star(root)
  bound  := h(root)
  path  := [root]
  loop
    t  := search(path, 0, bound)
    if t = FOUND then return (path, bound)
    if t = ∞ then return NOT_FOUND
    bound  := t
  end loop
end procedure

function search(path, g, bound)
  node  := path.last
  f  := g + h(node)
  if f > bound then return f
  if is_goal(node) then return FOUND
  min  := ∞
  for succ in successors(node) do
    if succ not in path then
      path.push(succ)
      t  := search(path, g + cost(node, succ), bound)
      if t = FOUND then return FOUND
      if t < min then min  := t
      path.pop()
    end if
  end for
  return min
end function
```

Figure 9. IDA search pseudocode*

IDA* search is Iterative Deepening way of A* search. In IDA* search iteration will be used and at each iteration, the threshold used for the next iteration is the minimum cost of all values that exceeded the current threshold. We use IDA* search to reduce unnecessary searches compared with A* search.



```python
node = copy.deepcopy(path.list[len(path.list) - 1])  # copy last element
f = g + heuristic(node, puzzle)
if f > bound:
    return False, f
if puzzle.is_goal_state(node):
    return True, 0  # 0 is dummy value (don't need f value)
min_value = float('inf')

for succ in puzzle.get_successors(node):
    state, action, cost = succ  # don't need 'action' value
    if state not in path.list:
        path.push(state)
        found, new_bound = search(path, g + cost, bound, puzzle)
        if found:
            return found, 0  # 0 is dummy value
        if new_bound < min_value:
            min_value = new_bound
        path.pop()
```

```python
while True:
    found, new_bound = search(path, 0, bound, puzzle)
    if found:
        # path.list is a list of states: for consistency with other search functions, we return a list of actions (e.g. ["Up", "Left",
        return puzzle.convert_solution_from_states_to_actions(path.list)
    if new_bound >= infinity:
        return None
    bound = new_bound
```

Figure 10. Partial search and ida_star implementation

In IDA* search, we will invoke search function. Search function will copy the last element in the path list first and we will set bound in the function to limit the depth of the search tree, which is the estimated distance from the initial state to the target state. If the cost of finding a goal exceeds the limit, a tuple of *(False, cost)* will be returned. We keep adjusting the maximum depth to *new_bound* until we find the goal node. If the search tree is infinitely long, *None* will be returned.

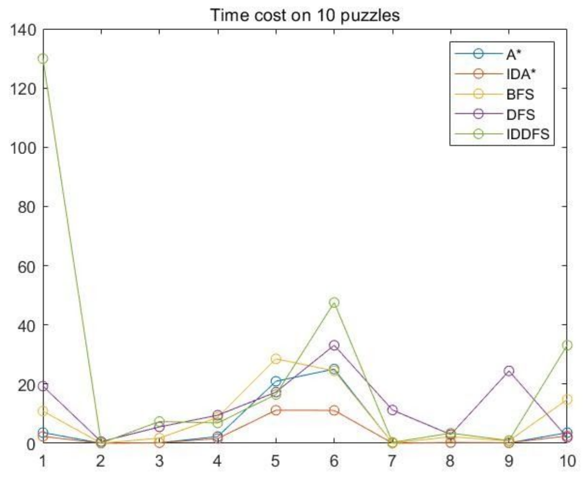
Methodology

1. A* algorithm (Admissible heuristic)
2. Iterative Deepening A*(IDA) algorithm
3. Breadth-First Search(BFS) algorithm
4. Depth-First Search(DFS) algorithm
5. Iterative Deepening Depth-first Search(IDDFS) algorithm

Our instance is a 3x3 sliding puzzle which is generated by our method, it shuffles to another random state at each game round. It guarantees that each puzzle will have a solution to the goal state, rather than wasting time to run algorithms then get to infinity waiting.

Out of our experiments, the time cost and memory usage of IDA* always outperforms other algorithms under for all instances. Therefore, IDA* is a improved and memory-saved version of A* algorithm, even with around 2 times more nodes expansion.

The overhead of all algorithms increases when it comes to a difficult puzzle. But even with some simple puzzles, while other algorithms only use a few moves to reach the goal state, the DFS algorithm takes tens of thousands of moves and it is the most time-consuming and memory-intensive one. Compared to IDDFS, which is the iterative version of DFS, IDDFS expands more nodes DFS has maximum fluctuation depends on different scenarios of puzzle, at the meantime, the performance of IDA* has always been stable (figure 11). In conclusion, the DFS algorithm is the last choice for solving the sliding puzzle problem.



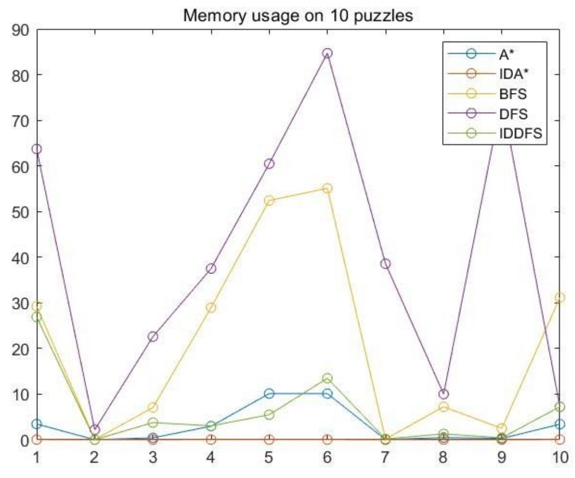


Figure 11. Time cost and memory usage direct view

From the sample data, we observed that IDDFS and DFS sometimes don't return an optimal solution. However, the solution provided by IDDFS is very close to the optimal one (only a few moves more), while DFS is still far away. A* and IDA* are the optimal algorithms that always return the shortest path to the goal state. BFS is also optimal under the puzzle problem, since every move of the puzzle costs the same and there is node duplication check to make sure it finds the shortest path.

Experimental outcomes

We ran our program and searched 10 times for different puzzles solutions on different algorithms, collected and calculated their averaged performance on the below table and figures.

Table 1. Averaged experimental data from 10 different puzzles

	A*	**IDA***	**BFS**	**DFS**	**IDDFS**
Solution length (moves)	~17	~17	~17	46458	~25
Time cost (s)	5.6150	2.9490	9.2380	12.5910	24.610
Expanded nodes	~2560	13050	48407	97363	199140
Memory usage (MB)	3.1380	0.0360	21.3914	40.1775	6.1895

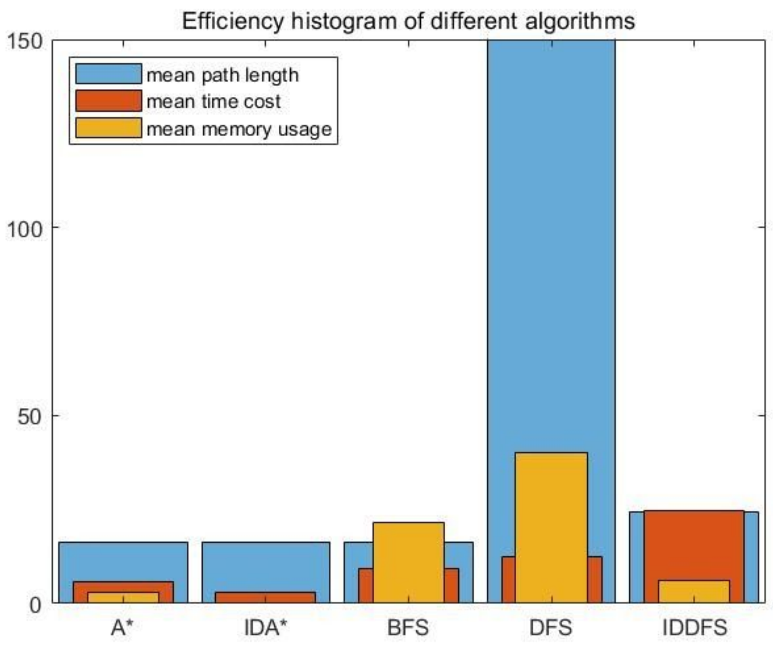


Figure 12. Efficiency histogram of different algorithms[1]

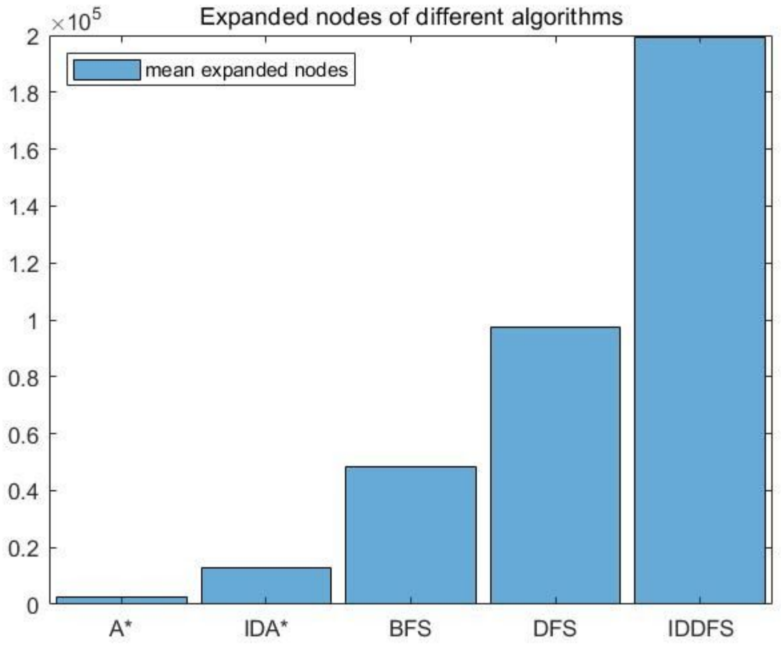


Figure 13. Expanded nodes of different algorithms

[1] The y-value of “mean path length” of DFS algorithm has been compressed into range [0,150] for better viewing, its actual value here should be 464589 (moves).

Conclusion

Table 2. Comparison and conclusion of each algorithms

	A*	**IDA***	**BFS**	**DFS**	**IDDFS**
Time efficiency	2nd	1st	3rd	4th	5th
Memory usage	2nd	1st	4th	5th	3rd
Optimality	✓	✓	✓	×	×
Completeness	✓	✓	✓	✓	✓

In terms of time efficiency of the algorithms, IDA* scored the best, A* came second, followed by BFS, DFS and IDDFS. This is because A* is a best-first search, which should be faster than BFS or DFS, and IDA* is the improved version of A*, it prunes some unnecessary explorations and saves lots of time and space.

For memory usage, IDA* scored 0.036 MB which is far ahead of the second efficient algorithm A* (3.138 MB). IDDFS came third with 6.1895 MB, BFS with 21.3914 MB and DFS with 40.1775 MB.

All five algorithms showed to be complete. For BFS, DFS and IDDFS, this is because we implemented duplicate detection, so the algorithm will run similar to a tree search, instead of graph search. A* and IDA* are guaranteed to be complete, given that the heuristic is admissible.

In terms of optimality of the algorithms, A*, IDA* and BFS resulted in an optimal solution. There are two reasons for BFS to be optimal. First one is that every edge has a cost of 1, and the second reason is that we implemented duplicate detection, as explained above. DFS and IDDFS didn't result in an optimal solution, because they expand deepest nodes first, regardless of the path costs or the heuristic values.

In summary, IDA* proved to be the most time efficient, memory efficient, optimal and complete search algorithm, while DFS and IDDFS were the least time and memory efficient. Based on the experience given to us by this project, we have designed a great platform to test the performance of various algorithms. We expect to add and implement more path-finding algorithms and incorporate them into the game, create a convenient platform and allow user to play and learn different algorithms in a more intuitive way.

Reference

A* search algorithm pseudocode:
https://en.wikipedia.org/wiki/A*_search_algorithm

IDA* search algorithm pseudocode:
https://en.wikipedia.org/wiki/Iterative_deepening_A*

BFS search algorithm pseudocode:
https://en.wikipedia.org/wiki/Breadth-first_search

DFS search algorithm pseudocode:
https://en.wikipedia.org/wiki/Depth-first_search

IDDFS search algorithm pseudocode:
https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search

src/util.py file adapted from:
http://ai.berkeley.edu/search.html