Eight Puzzle Solver

EECS 391 Progamming Project 1

William Koehrsen September 28, 2017

Introduction

The eight puzzle provides an ideal toy environment for exploring one of the fundamental problems in Artificial Intelligence: finding the optimal solution path through search. Although the eight puzzle presents a simplified environment, the general techniques of search employed to find a solution can translate into many domains, including laying out components on a computer chip or finding the best route between two cities. Two different search strategies were explored in this report: a-star and local beam.

In [1]:

```
from eight_puzzle import Puzzle
```

Approach

The complete project was coded in Python. The eight_puzzle was implemented as a class that contains an attribute to represent the game board as well, the two methods for solving the puzzle, and a number of utility methods to aid in the solution of the puzzle and visualizing the solution found. The solution of the eight puzzle is the series of actions that get the board from the starting state to the goal state. My approach therefore implemented dictionaries to keep track of all considered nodes. Each node must include the current state (the position of all tiles on the board), the parent state, and the action required to get from the parent state to the child state. When the goal state is reached, the algorithm can then transverse backwards through the dictionary starting at the goal state and linking each child state to its parent and recording the series of moves. This solution path can then be displayed and the length of the solution path can be determined. The efficiency of each search can be found from the solution path and the number of nodes generated during the search.

Files

eight_puzzle.py

Contains the eight puzzle class itself.

play_puzzle.py

Contains code for interacting with the puzzle and implementing the commands from the command line or a text file.

Required Commands

- **setState** [state]

Set the state of a puzzle to the specified argument. The format of the state is "b12 345 678" where each triple represents a row and the b is the blank tile (the only tile that can be moved).

- **randomizeState** [n]

  Randomly set the state of the puzzle by taking a random series of 'n' actions backward from the goal state. As only half of the initial states of the eight puzzle are solvable, taking a series of steps backwards from the goal state ensures that the resulting state will have a solution.

- **printState**

  Displays the current state of the puzzle in the form "b12 345 678"

- **move** [direction]

  Move the blank one place in the specified direction. The move must be one of the following: ["up", "down", "left", "right"] and admissible moves will be determined by the current state of the puzzle.

- **solveAStar** [hueristic]

  Solve the eight puzzle using the specified heuristic. The choices for the heuristic are ['h1'] or ['h2']. 'h1' counts the number of misplaced tiles from the goal state while h2 represents the sum of the Manhattan distances of each tile from its correct position in the goal state. After solving the puzzle, the solution path should be printed as a series of actions as well as the number of actions required to reach the solution.

- **solveBeam** [k]

  Solve the eight puzzle using the local beam search with the beam width equal to 'k'. In general, a larger value of k is more likely to find a solution becuase the search space is larger, but larger k values will require retaining more nodes in memory. After solving the puzzle, the solution path should be printed as a series of actions as well as the number of actions required to reach the solution.

- **maxNodes** [n]

  The maximum number of nodes to generate during a search. A node is generated when it is expanded by taking a move from its parent. When this limit is exceeded, the code must display an error message. If max nodes is not specified, a default number of 10000 nodes is passed to the method.

- **readCommands** [file.txt]

  Read and execute a series of commands from a text file. The commands are specified in the same format as the commands for the command line except without the dashes.

Additional Commands

- **prettyPrintState**

  Displays the current state in an aesthetic and understandable format.

- **prettyPrintSolution**

Displays the solution as a sequence of moves in an aesthetic and understandable format.

Command Implementation

To demonstrate the functionality of the commands, I will show the code that implements the command as well as examples of running the command from the command line.

setState and printState

```
def set_state(self, state_string):
        # Set the state of the puzzle to state_string in format "b12
345 678"

        # Check string for correct length
        if len(state_string) != 11:
            print("String Length is not correct!")

        # Keep track of elements that have been added to board
        added_elements = []

        # Enumerate through all the positions in the string
        for row_index, row in enumerate(state_string.split(" ")):
            for col_index, element in enumerate(row):
                # Check to make sure invalid character not in string
                if element not in ['b', '1', '2', '3', '4', '5', '6',
'7', '8']:
                    print("Invalid character in state:", element)
                    break
                else:
                    if element == "b":
                        # Check to see if blank has been added twice
                        if element in added_elements:
                            print("The blank was added twice")
                            break

                        # Set the blank tile to a 0 on the puzzle
                        else:
                            self.state[row_index][col_index] = 0
                            added_elements.append("b")
                    else:
                        # Check to see if tile has already been adde
to board
                        if int(element) in added_elements:
                            print("Tile {} has been added
twice".format(element))
                            break

                        else:
                            # Set the correct tile on the board
                            self.state[row_index][col_index] = int(el
ment)
                              added_elements.append(int(element))
```

```
def print_state(self):
        # Display the state of the board in the format "b12 345 678"
        str_state = []

        # Iterate through all the tiles
        for row in self.state:
            for element in row:
                if element == 0:
                    str_state.append("b")
                else:
                    str_state.append(str(element))

        # Print out the resulting state
        print("".join(str_state[0:3]), "".join(str_state[3:6]), "".jo
in(str_state[6:9]))
```

The following code block shows an example of setting the state to "325 678 b14" and displaying the result in both string format and prettyPrint format. The "b" and the 0 are interchangeable and the puzzle converts b input into 0 to represent the blank tile.

In [2]:

```
%run play_puzzle.py -setState "325 678 b14" -printState -prettyPrintState
```

```
Current puzzle state:
325 678 b14

Current State
-------------
| 3 | 2 | 5 |
-------------
| 6 | 7 | 8 |
-------------
| 0 | 1 | 4 |
```

randomizeState

```
def randomize_state(self, n):
        # Take a random series of moves backwards from the goal
state
        # Sets the current state of the puzzle to a state thta is
guaranteed to be solvable

        current_state = (self.goal_state)

        # Iterate through the number of moves
        for i in range(n):
            available_actions, _, _ = self.get_available_actions(curr
ent_state)
            random_move = random.choice(available_actions)
            current_state = self.move(current_state, random_move)

        # Set the state of the puzzle to the random state
        self.state = current_state
```

The following code block shows an example of randomizing the state by taking 50 random steps from the goal state. The code also displays the state after randomizing in both string format and prettyPrint

In [3]:

```
%run play_puzzle.py -randomizeState 50 -printState -prettyPrintState
```

```
Current puzzle state:
154 3b2 678

Current State
-------------
| 1 | 5 | 4 |
-------------
| 3 | 0 | 2 |
-------------
| 6 | 7 | 8 |
```

move

``` python def move(self, state, action):

```
    # Move the blank in the specified direction
    # Returns the new state resulting from the move
    available_actions, blank_row, blank_column =
self.get_available_actions(state)

    new_state = copy.deepcopy(state)

    # Check to make sure action is allowed given board state
    if action not in available_actions:
        print("Move not allowed\nAllowed moves:", available_actions)
        return False

    # Execute the move as a series of if statements, probably not th
e most efficient method
    else:
        if action == "down":
            tile_to_move = state[blank_row + 1][blank_column]
            new_state[blank_row][blank_column] = tile_to_move
            new_state[blank_row + 1][blank_column] = 0
        elif action == "up":
```
```

```
                tile_to_move = state[blank_row - 1][blank_column]
                new_state[blank_row][blank_column] = tile_to_move
                new_state[blank_row - 1][blank_column] = 0
            elif action == "right":
                tile_to_move = state[blank_row][blank_column + 1]
                new_state[blank_row][blank_column] = tile_to_move
                new_state[blank_row][blank_column + 1] = 0
            elif action == "left":
                tile_to_move = state[blank_row][blank_column - 1]
                new_state[blank_row][blank_column] = tile_to_move
                new_state[blank_row][blank_column -1] = 0

        return new_state
```

The following two examples first illustrate taking a valid move and then an invalid move. (The puzzle is first set to the goal state for demonstration purposes.

Valid move: moving "right" from the goal state:

In [4]:

```
%run play_puzzle.py -setState "b12 345 678" -move "right" -printState
```

```
Current puzzle state:
1b2 345 678
```

Invalid move: moving "up" from the goal state:

In [5]:

```
%run play_puzzle.py -setState "b12 345 678" -move "up"
```

```
Move not allowed
Allowed moves: ['down', 'right']
```

solveAStar

```
def a_star(self, heuristic="h2", max_nodes=10000,
print_solution=True):
        # Performs a-star search
        # Prints the list of solution moves and the solution length

        # Need a dictionary for the frontier and for the expanded no
des
        frontier_nodes = {}
        expanded_nodes = {}

        self.starting_state = copy.deepcopy(self.state)
        current_state = copy.deepcopy(self.state)
        # Node index is used for indexing the dictionaries and to ke
```

```
ep track of the number of nodes expanded
        node_index = 0

        # Set the first element in both dictionaries to the starting
state
        # This is the only node that will be in both dictionaries
        expanded_nodes[node_index] = {"state": current_state, "parent
": "root", "action": "start",
                                      "total_cost": self.calculate_total
cost(0, current_state, heuristic), "depth": 0}

        frontier_nodes[node_index] = {"state": current_state, "parent
": "root", "action": "start",
                                      "total_cost": self.calculate_total
cost(0, current_state, heuristic), "depth": 0}


        failure = False

        # all_nodes keeps track of all nodes on the frontier and is
the priority queue. Each element in the list is a tuple consisting o
f node index and total cost of the node. This will be sorted by the
total cost and serve as the priority queue.
        all_frontier_nodes = [(0, frontier_nodes[0]["total_cost"])]

        # Stop when maximum nodes have been considered
        while not failure:

            # Get current depth of state for use in total cost calcu
ation
            current_depth = 0
            for node_num, node in expanded_nodes.items():
                if node["state"] == current_state:
                    current_depth = node["depth"]

            # Find available actions corresponding to current state
            available_actions, _, _ = self.get_available_actions(curr
ent_state)

            # Iterate through possible actions
            for action in available_actions:
                repeat = False

                # If max nodes reached, break out of loop
                if node_index >= max_nodes:
                    failure = True
                    print("No Solution Found in first {} nodes genera
ted".format(max_nodes))
                    self.num_nodes_generated = max_nodes
                    break
                # Find the new state corresponding to the action and 
calculate total cost
                new_state = self.move(current_state, action)
                new_state_parent = copy.deepcopy(current_state)

                # Check to see if new state has already been expande
                for expanded_node in expanded_nodes.values():
                    if expanded_node["state"] == new_state:
                        if expanded_node["parent"] == 
new_state_parent:
                            repeat = True

                # Check to see if new state and parent is on the fro
tier
                # The same state can be added twice to the frontier 
f the parent state is different
                for frontier_node in frontier_nodes.values():
                    if frontier_node["state"] == new_state:
                        if frontier_node["parent"] == 
new_state_parent:
                            repeat = True

                # If new state has already been expanded or is on the
frontier, continue with next action
                if repeat:
                    continue

                else:
                    # Each action represents another node generated
                    node_index += 1
                    depth = current_depth + 1

                    # Total cost is path length (number of steps fro
starting state) + heuristic
                    new_state_cost = self.calculate_total_cost(depth,
new_state, heuristic)

                    # Add the node index and total cost to the all_n
des list
                    all_frontier_nodes.append((node_index, 
new_state_cost))

                    # Add the node to the frontier
                    frontier_nodes[node_index] = {"state": new_state,
"parent": new_state_parent, "action": action, "total_cost": 
new_state_cost, "depth": current_depth + 1}

            # Sort all the nodes on the frontier by total cost
            all_frontier_nodes = sorted(all_frontier_nodes, key=lambd
a x: x[1])
```

```
            # If the number of nodes generated does not exceed max
nodes, find the best node and set the current state to that state
            if not failure:
                # The best node will be at the front of the queue
                # After selecting the node for expansion, remove it
rom the queue
                best_node = all_frontier_nodes.pop(0)
                best_node_index = best_node[0]
                best_node_state = frontier_nodes[best_node_index]["st
ate"]
                current_state = best_node_state

                # Move the node from the frontier to the expanded
nodes
                expanded_nodes[best_node_index] = (frontier_nodes.pop
(best_node_index))

                # Check if current state is goal state
                if self.goal_check(best_node_state):
                    # Create attributes for the expanded nodes and t
e frontier nodes
                    self.expanded_nodes = expanded_nodes
                    self.frontier_nodes = frontier_nodes
                    self.num_nodes_generated = node_index + 1

                    # Display the solution path
                    self.success(expanded_nodes, node_index,
print_solution)
                    break
```

The following example shows solving the puzzle from the state "312 475 68b" with a-star using the h1 heuristic and the default number of max nodes (10000). This puzzle can be solved in four moves and the solution is also printed out in a pretty format to see the sequence of moves.

In [6]:

```
%run play_puzzle.py -setState "312 475 68b" -solveAStar h1 -prettyPrintSolu
tion
```

```
Solution found!
Solution Length:  4
Solution Path ['start', 'left', 'up', 'left', 'up', 'goal']
Total nodes generated: 12

Starting State
-------------
| 3 | 1 | 2 |
-------------
| 4 | 7 | 5 |
-------------
| 6 | 8 | 0 |

Depth: 1
```

```
-------------
| 3 | 1 | 2 |
-------------
| 4 | 7 | 5 |
-------------
| 6 | 0 | 8 |

Depth: 2
-------------
| 3 | 1 | 2 |
-------------
| 4 | 0 | 5 |
-------------
| 6 | 7 | 8 |

Depth: 3
-------------
| 3 | 1 | 2 |
-------------
| 0 | 4 | 5 |
-------------
| 6 | 7 | 8 |

GOAL!!!!!!!!!
-------------
| 0 | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
```

As can be seen, a-star using the h1 heuristic finds the optimal (shortest path cost) solution.

The following example shows what happens if max nodes is specified and the puzzle is not able to find a solution within the maximum number of nodes to consider.

In [7]:

```
%run play_puzzle.py -setState "7b2 853 641" -solveAStar h1 -maxNodes 10
```

```
No Solution Found in first 10 nodes generated
```

This is an unsolvable puzzle, and as can be seen, no solution is found (try increasing the maximum number of nodes and give your computer some exercise!)

The following example demonstrates solving a randomly generated puzzle taking 10 moves backwards from the goal state. The puzzle is solved with a-star using the h2 hueristic and a maximum of 500 nodes.

In [8]:

```
%run play_puzzle.py -randomizeState 10 -solveAStar h2 -maxNodes 500
```

```
Solution found!
Solution Length:  4
Solution Path ['start', 'left', 'left', 'up', 'up', 'goal']
Total nodes generated: 10
```

The algorithm correctly solves the puzzle and finds the optimal path. The efficiency of the a-star algorithm and a comparison between the two heuristics will be discussed in the effective branching factor section of this report.

solveBeam

The evaluation function used for local beam search was

$$f(n) = h1(state) + h2(state)$$

Where f(state) represents the total cost of the state, h1(state) represents the number of misplaced tiles from the goal state of the state, and h2 represents the sum of the Manhattan distances of the tiles in the state from the goal state. This evaluation function was selected because the h1 and h2 heuristics are both consistent and admissible and work well for solving a-star. The difference between a-star and local beam is that a-star considers the path cost to each state when ranking the states whereas local beam only considers the evaluation function of the successor states. In effect, local beam is "blind" to the past and to any future beyond the next step (a-star is also "blind" to the future beyond the next step) At each iteration, local beam search will calculate the total cost of all successors of all current states and order the successor states by lowest to highest cost. The algorithm will then retain the "k" best states for the next iteration. This limits the amount of states that need to be kept in memory. Local beam search with $k = 1$ is equivalent to hill climbing because the algorithm will just select the next state with the lowest evaluation function. Local beam search with $k = \infty$ is equal to breadth first search because the algorithm will expand every single state at the current depth before moving to the next depth.

```
def local_beam(self, k=1, max_nodes = 10000, print_solution=True):
        # Performs local beam search to solve the eight puzzle
        # k is the number of successor states to consider on each it
eration
        # The evaluation function is h1 + h2, at each iteration, the
next set of nodes will be the k nodes with the lowest score

        self.starting_state = copy.deepcopy(self.state)
        starting_state = copy.deepcopy(self.state)
        # Check to see if the current state is already the goal
        if starting_state == self.goal_state:
            self.success(node_dict={}, num_nodes_generated=0)

        # Create a reference dictionary of all states generated
        all_nodes= {}

        # Index for all nodes dictionary
        node_index = 0

        all_nodes[node_index] = {"state": starting_state, "parent": "
root", "action": "start"}
```

```
        # Score for starting state
        starting_score = self.calculate_h1_heuristic(starting_state)
+ self.calculate_h2_heuristic(starting_state)

        # Available nodes is all the possible states that can be acc
essed from the current state stored as an (index, score) tuple
        available_nodes = [(node_index, starting_score)]

        failure = False

        while not failure:

            # Check to see if the number of nodes generated exceeds m
ax nodes
            if node_index >= max_nodes:
                failure = True
                print("No Solution Found in first {} generated nodes'
.format(max_nodes))
                break

            # Successor nodes are all the nodes that can be reached
from all of the available states. At each iteration, this is reset t
o an empty list
            successor_nodes = []

            # Iterate through all the possible nodes that can be
visited
            for node in available_nodes:

                repeat = False

                # Find the current state
                current_state = all_nodes[node[0]]["state"]

                # Find the actions corresponding to the state
                available_actions, _, _ = self.get_available_actions(
current_state)

                # Iterate through each action that is allowed
                for action in available_actions:
                    # Find the successor state for each action
                    successor_state = self.move(current_state, action
)


                    # Check if the state has already been seen
                    for node_num, node in all_nodes.items():
                        if node["state"] == successor_state:
                            if node["parent"] == current_state:
                                repeat = True

                    if not repeat:
                        node_index += 1
```

```
                            node_index += 1
                            # Calculate the score of the state
                            score = 
(self.calculate_h1_heuristic(successor_state) + 
self.calculate_h2_heuristic(successor_state))
                            # Add the state to the list of of nodes
                            all_nodes[node_index] = {"state": 
successor_state, "parent": current_state, "action": action}
                            # Add the state to the successor_nodes list
                            successor_nodes.append((node_index, score))
                        else:
                            continue


              # The available nodes are now all the successor nodes 
sorted by score
              available_nodes = sorted(successor_nodes, key=lambda x: x
[1])

              # Choose only the k best successor states
              if k < len(available_nodes):
                  available_nodes = available_nodes[:k]

              # If the best state is the goal, stop iteration
              if available_nodes[0][1] == 0:
                  self.expanded_nodes = all_nodes
                  self.num_nodes_generated = node_index + 1
                  self.success(all_nodes, node_index, print_solution)
                  break
```

The following example demonstrates solving the eight puzzle using local beam search with a beam width of 10 and the maximum number of default nodes (10000). This solution has a true length of 4.

In [9]:

```
%run play_puzzle.py -setState "125 348 67b" -solveBeam 10 -prettyPrintSolut
ion
```

```
Solution found!
Solution Length:  4
Solution Path ['start', 'up', 'up', 'left', 'left', 'goal']
Total nodes generated: 38

Starting State
-------------
| 1 | 2 | 5 |
-------------
| 3 | 4 | 8 |
-------------
| 6 | 7 | 0 |

Depth: 1
-------------
```

```
| 1 | 2 | 5 |
-------------
| 3 | 4 | 0 |
-------------
| 6 | 7 | 8 |

Depth: 2
-------------
| 1 | 2 | 0 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |

Depth: 3
-------------
| 1 | 0 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |

GOAL!!!!!!!!!
-------------
| 0 | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
```

Local beam with a beam width of 10 is able to find the solution. However, local beam can get stuck in local optima if the beam width is too narrow.

readCommands

Another requirement of this project was that the program should be able to read a sequence of commands from a text file. This was implemented in the play_puzzle.py function using an argument parser. The following examples illustrate reading commands from a text file.

The following example illustrates using a text file to set the state of the puzzle, displaying the initial state of the puzzle, and then solving the puzzle using local beam with a beam width of 50 and 2000 nodes max.

In [10]:

```
%run play_puzzle.py -readCommands "tests\set_state_solve_beam.txt"
```

```
Current puzzle state:
3b2 615 748
Solution found!
Solution Length:  5
Solution Path ['start', 'down', 'down', 'left', 'up', 'up', 'goal']
Total nodes generated: 92
```

Contents of "set_state_solve_beam.txt":

```
setState "3b2 615 748" printState solveBeam 50 maxNodes 2000
```

The following example illustrates using a text file to randomize the state of the puzzle with 16 random steps back from the goal, solving the puzzle using a-star with the h2 heuristic, and pretty printing the solution path in an understandable format.

In [11]:

```
%run play_puzzle.py -readCommands "tests\randomize_solve_astar.txt"
```

```
Solution found!
Solution Length:  6
Solution Path ['start', 'left', 'down', 'down', 'left', 'up', 'up', 'goal']
Total nodes generated: 17

Starting State
-------------
| 3 | 2 | 0 |
-------------
| 6 | 1 | 5 |
-------------
| 7 | 4 | 8 |

Depth: 1
-------------
| 3 | 0 | 2 |
-------------
| 6 | 1 | 5 |
-------------
| 7 | 4 | 8 |

Depth: 2
-------------
| 3 | 1 | 2 |
-------------
| 6 | 0 | 5 |
-------------
| 7 | 4 | 8 |

Depth: 3
-------------
| 3 | 1 | 2 |
-------------
| 6 | 4 | 5 |
-------------
| 7 | 0 | 8 |

Depth: 4
-------------
| 3 | 1 | 2 |
-------------
| 6 | 4 | 5 |
-------------
| 0 | 7 | 8 |

Depth: 5
-------------
```

```
-------------
| 3 | 1 | 2 |
-------------
| 0 | 4 | 5 |
-------------
| 6 | 7 | 8 |

GOAL!!!!!!!!!
-------------
| 0 | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
```

Contents of "randomize_solve_astar.txt": `randomizeState 16 solveAStar h2 prettyPrintSolution`

The above examples demonstate that the program meets all of the requirements as specified in the project description. The final step of the project is to investigate the efficiency of the two algorithms and the two heuristics. This can best be quantified by the effective branching factor.

Efficiency Calculations

Effective Branching Factor

The branching factor for a specific node is the number of successor states generated by that node. The effective branching factor is an estimate for the number of successor states generated by a typical node. The formula to find the effective branching factor is:

$$N = b_{eff} + b_{eff}^2 + b_{eff}^3 + ... + b_{eff}^d$$

where N is the total number of nodes generated, b_eff is the effective branching factor, and d is the length of the solution path.

To calculate the effective branching factor, I generated 100 starting states for each solution length from 2 moves to 22 moves (by 2). This was done by taking moves backwards from the goal state with the condition that no state could be repeated twice as shown in the following code.

```
# Generate a list of integers from 2 to 22 by 2
solution_lengths = list(np.arange(2,24,2))

# Create a dictionary to store the starting states
# The format of the dictionary is number of actions in solution path
: starting state
starting_states = {}

# Iterate through the wanted solution path lengths
for solution_length in solution_lengths:
    print(solution_length)
    starting_states[solution_length] = []

    # Store 100 starting states for each solution length
    while len(starting_states[solution_length]) < 100:
        puzzle.set_goal_state()
        states = [puzzle.goal_state]
        while len(states) < solution_length + 1:
            available_actions, _, _ = puzzle.get_available_actions(pu
zzle.state)
            next_action = random.choice(available_actions)
            new_state = puzzle.move(state = puzzle.state, action = ne
xt_action)
            # If this move is not already in the sequence add it on
            # This ensures that the solution length will be exactly
equal to the wanted solution length
            if new_state not in states:
                states.append(new_state)
                puzzle.state = new_state

        # Append the starting state to the dictionary
        starting_states[solution_length].append(states[-1])
```

For each solution path length, there are 100 starting states. To evaluate both algorithms and the two heuristics, we can then iterate through all of these starting states, recording the number of nodes generated by each algorithm. We then take the average number of nodes generated for each solution.

The average number of nodes generated for each algorithm are shown in the table below:

Nodes Generated for Algorithms on the Eight Puzzle

	A* using h1	A* using h2	Local Beam k = 100	Local Beam k = 10
d				
2	6	6	13	13
4	12	12	47	39
6	24	19	162	70
8	44	29	421	101
10	101	45	678	155
12	238	80	976	236
14	580	144	1236	427
16	1276	276	1459	615
18	2814	435	1735	603
20	5210	688		
22		1072		

	A* using h1	A* using h2	Local Beam k = 100	Local Beam k = 10
24		1544		

The effective branching factor can then be calculated by solving the equation described. This equation was solved by graphing both sides and finding the intersection of the two graphs.

The branching factors of the algorithms are summarized in the table below:

Branching Factors of Search Algorithms on the Eight Puzzle

	A* using h1	A* using h2	Local Beam k = 100	Local Beam k = 10
d				
2	2	2	3.14	3.14
4	1.49	1.49	2.29	2.17
6	1.42	1.34	2.10	1.79
8	1.38	1.28	1.95	1.58
10	1.41	1.26	1.77	1.48
12	1.43	1.27	1.64	1.43
14	1.45	1.28	1.54	1.41
16	1.45	1.30	1.47	1.38
18	1.46	1.29	1.41	1.32
20	1.45	1.29		
22		1.32		
24		1.35		

Visualization of Efficiency

Using the R programming language, we can quickly visualize the efficiency of the two algorithms and the heuristics. a_star_h1 refers to the a-star algorithm using the h1 heuristic, a_star_h2 refers to the a-star algorithm using the h2 heuristic, local_beam_100 refers to the local beam algorithm with a beam width of 100, and local_beam_10 refers to the local beam algorithm with a beam width of 10.

```
ggplot(generated, aes(d, nodes_generated, color = algorithm)) +
geom_jitter(size = 3) + geom_line(aes(color = algorithm)) + xlab("Sol
ution Depth") + ylab("Nodes Generated") + ggtitle("Nodes Generated
vs Solution Depth Broken Out by Algorithm")
```

```
ggplot(effective, aes(d, effective_branching, color = algorithm)) +
geom_jitter(size = 2) + geom_line(aes(color = algorithm)) + xlab("Sol
ution Depth") + ylab("Effective Braching Factor") +
ggtitle("Effective Branching Factor vs Solution Depth Broken Out by
Algorithm")
```

In [ ]:

```

```