AI Assignment Analysis

February 14, 2024

Antonio Manuel Luque Molina - anluq0157@uit.no

[1]:
```
import matplotlib.pyplot as plt
import time
from memory_profiler import memory_usage

class CubeTower:
    def __init__(self, configuration, parent=None):
        """
        Initializes the cube tower with a given configuration.
        :param configuration: A list of the front-facing colors of the cubes in␣
↪the tower, starting from the bottom.
        :param parent: The parent node of the current node. (can be used for␣
↪tracing back the path)
        """
        self.order = ['red', 'blue', 'green','yellow']
        self.configuration = configuration
        self.height = len(configuration)
        self.parent = parent

    def visualize(self):
        """
        Visualizes the current state of the cube tower showing only the␣
↪front-facing side.
        """
        fig, ax = plt.subplots()
        cube_size = 1  # Size of the cube

        for i, cube in enumerate(self.configuration):
            # Draw only the front-facing side of the cube
            color = cube
            rect = plt.Rectangle((0.5 - cube_size / 2, i), cube_size,␣
↪cube_size, color=color)
            ax.add_patch(rect)

        ax.set_xlim(0, 1)
        ax.set_ylim(0, self.height)
        ax.set_aspect('equal', adjustable='box')
```

```
        ax.axis('off')
        plt.show()

    def visualize_path(self):
        """
        Visualizes the path taken to reach this state from the initial state.
        """
        path = self.get_path()
        fig, ax = plt.subplots(figsize=(len(path) * 2, self.height))
        cube_size = 1

        for i, configuration in enumerate(path):
            for j, cube in enumerate(configuration):
                color = cube
                rect = plt.Rectangle((i * (cube_size + 0.1), j), cube_size,␣
↪cube_size, color=color)
                ax.add_patch(rect)

        ax.set_xlim(0, len(path) * (cube_size + 0.1))
        ax.set_ylim(0, self.height)
        ax.set_aspect('equal', adjustable='box')
        ax.axis('off')
        plt.show()

    def visualize_bidirectional_path(self, path_forward, path_backward):
        path_backward.reverse()
        full_path = path_forward[:-1] + path_backward
        fig, ax = plt.subplots(figsize=(len(full_path) * 2, self.height))
        cube_size = 1
        for i, tower in enumerate(full_path):
            for j, cube_color in enumerate(tower.configuration):
                rect = plt.Rectangle((i * (cube_size + 0.1), j), cube_size,␣
↪cube_size, color=cube_color)
                ax.add_patch(rect)
        ax.set_xlim(0, len(full_path) * (cube_size + 0.1))
        ax.set_ylim(0, self.height)
        ax.set_aspect('equal', adjustable='box')
        ax.axis('off')
        plt.show()

    def get_path(self):
        """
        Retrieves the path taken to reach this state from the initial state.
        """
        path = [self.configuration]
        current = self
        while current.parent is not None:
```

```
            current = current.parent
            path.append(current.configuration)
        path.reverse()
        return path

    def check_cube(self):
        """
        Check if the cube tower is solved, i.e. all cubes are of the same color.
        """
        return len(set(self.configuration)) == 1

    def rotate_cube(self, ind, hold_index=None):
        """
        Rotates a cube and all cubes above it, or up to a held cube.
        :param index: The index of the cube to rotate.
        :param hold_index: The index of the cube to hold, if any.
        """
        if hold_index is not None:
            for i in range(ind, hold_index):
                i_in_order = self.order.index(self.configuration[i])
                self.configuration[i] = (self.order[(i_in_order + 1) % len(self.
 ↪order)])
                i += 1
            return self.configuration
        for i in range(ind, self.height):
            i_in_order = self.order.index(self.configuration[i])
            self.configuration[i] = (self.order[(i_in_order+1) % len(self.
 ↪order)])
            i += 1
        return self.configuration


# Implement the search algorithms here
def dfs_search(tower:CubeTower, visited=None, moves=0, depth=0):
    if visited is None:
        visited = set()
    config_tuple = tuple(tower.configuration)
    if config_tuple in visited:
        return None
    visited.add(config_tuple)
    if tower.check_cube():
        return {'solution': tower, 'moves': moves, 'depth': depth}
    for i in range(tower.height):    #i=ind
        for j in range(i + 1, tower.height + 1):  #j=hold_index
            new_tower = CubeTower(list(tower.configuration), parent=tower)
            new_tower.rotate_cube(i, j)
            result = dfs_search(new_tower, visited, moves + 1, depth + 1)
```

```
            if result:
                return result
    return None

def bfs_search(tower: CubeTower):
    visited = set()
    queue = [(tower, 0)]
    ind_queue = 0
    while ind_queue < len(queue):
        current_tower, current_moves = queue[ind_queue]
        ind_queue += 1  # Move to the next item in the queue
        config_tuple = tuple(current_tower.configuration)
        if config_tuple not in visited:
            visited.add(config_tuple)
            if current_tower.check_cube():
                return {'solution': current_tower, 'moves': current_moves}
            for i in range(current_tower.height):
                for j in range(i + 1, current_tower.height + 1):
                    new_tower = CubeTower(list(current_tower.configuration),␣
 ↪parent=current_tower)
                    new_tower.rotate_cube(i, j)
                    queue.append((new_tower, current_moves + 1))
    return None

def heuristic(configuration):
    """
    Heuristic: Count the number of cubes not matching the most common color.
    """
    most_common_color = max(set(configuration), key=configuration.count)
    return sum(1 for cube in configuration if cube != most_common_color)

def a_star_search(tower: CubeTower):
    visited = set()
    priority_queue = [(heuristic(tower.configuration), 0, tower)]
    ind_queue = 0
    while priority_queue:
        _, current_moves, current_tower = min(priority_queue, key=lambda x:␣
 ↪x[0])
        priority_queue.remove((_, current_moves, current_tower))    # Remove␣
 ↪selected element
        config_tuple = tuple(current_tower.configuration)
        if config_tuple not in visited:
            visited.add(config_tuple)
        if current_tower.check_cube():
            return {'solution': current_tower, 'moves': current_moves}
        for i in range(current_tower.height):
            for j in range(i + 1, current_tower.height + 1):
```

```
                new_configuration = list(current_tower.configuration)
                new_tower = CubeTower(new_configuration, parent=current_tower)
                new_tower.rotate_cube(i, j)
                g_score = current_moves + 1
                h_score = heuristic(new_tower.configuration)
                f_score = g_score + h_score
                if tuple(new_tower.configuration) not in visited:
                    priority_queue.append((f_score, g_score, new_tower))
    return None

# Additional advanced search algorithm
# Iterative Deepening Depth-First Search (IDDFS)
def iddfs_search(tower: CubeTower):
    moves = 0
    def dls(current_tower: CubeTower, depth):
        if depth == 0 and current_tower.check_cube():
            return {'solution': current_tower, 'moves': moves}
        elif depth > 0:
            for i in range(current_tower.height):
                for j in range(i + 1, current_tower.height + 1):
                    new_tower = CubeTower(list(current_tower.configuration),␣
 ↪parent=current_tower)
                    new_tower.rotate_cube(i, j)
                    found = dls(new_tower, depth -1)
                    if found:
                        return found
        return None
    for depth in range(0, 100):
        result = dls(tower, depth)
        moves += 1
        if result:
            return result
    return None

# Bidirectional Search
def bidirectional_search(tower: CubeTower):
    goal_config = [tower.configuration[0]] * tower.height
    goal_tower = CubeTower(goal_config)

    visited_forward = {tuple(tower.configuration)}
    visited_backward = {tuple(goal_tower.configuration)}

    queue_forward = [(tower, [tower])]
    queue_backward = [(goal_tower, [goal_tower])]

    while queue_forward and queue_backward:
        # Forward step
```

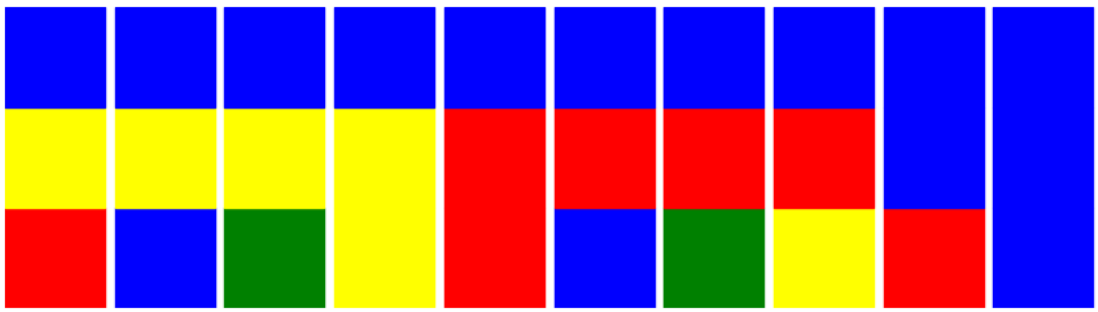
```
        current_forward, path_forward = queue_forward.pop(0)
        if tuple(current_forward.configuration) in visited_backward:
            # Finding matching node in backward path and concatenating paths
            matching_node = next((node for node, path in queue_backward if␣
↪tuple(node.configuration) == tuple(current_forward.configuration)), None)
            if matching_node:
                path_backward = next(path for node, path in queue_backward if␣
↪tuple(node.configuration) == tuple(current_forward.configuration))
                return {'solution' : (path_forward, path_backward[::-1]),␣
↪'moves' : len(path_forward + path_backward[::-1]) } # Reverse backward path␣
↪for correct order

        # Exploring neighbors in forward direction
        for i in range(current_forward.height):
            for j in range(i + 1, current_forward.height + 1):
                new_tower = CubeTower(list(current_forward.configuration),␣
↪parent=current_forward)
                new_tower.rotate_cube(i, j)
                if tuple(new_tower.configuration) not in visited_forward:
                    visited_forward.add(tuple(new_tower.configuration))
                    queue_forward.append((new_tower, path_forward +␣
↪[new_tower]))

        # Backward step
        current_backward, path_backward = queue_backward.pop(0)
        if tuple(current_backward.configuration) in visited_forward:
            # Finding matching node in forward path and concatenating paths
            matching_node = next((node for node, path in queue_forward if␣
↪tuple(node.configuration) == tuple(current_backward.configuration)), None)
            if matching_node:
                path_forward = next(path for node, path in queue_forward if␣
↪tuple(node.configuration) == tuple(current_backward.configuration))
                return {'solution' : (path_forward, path_backward[::-1]),␣
↪'moves' : len(path_forward + path_backward[::-1]) }

        # Exploring neighbors in backward direction
        for i in range(current_backward.height):
            for j in range(i + 1, current_backward.height + 1):
                new_tower = CubeTower(list(current_backward.configuration),␣
↪parent=current_backward)
                new_tower.rotate_cube(i, j)
                if tuple(new_tower.configuration) not in visited_backward:
                    visited_backward.add(tuple(new_tower.configuration))
                    queue_backward.append((new_tower, path_backward +␣
↪[new_tower]))
    return None
```

[2]:
```
# Test your implementation here

# Example Usage
# self.order = ['red', 'blue', 'green','yellow']
initial_configurations = [
    ["red","yellow","blue"],
    ["red","yellow","blue","yellow"],
    ["blue","green","red","yellow"],
    ["blue","yellow","blue","yellow", "red"],
    ["yellow","red","blue","yellow", "green", "blue"]
]

def getPlots(results, algorithm_name):
    times = [result['time'] for result in results]
    moves = [result['moves'] for result in results]
    memory_usages = [result['memory_usage'] for result in results]

    configuration_indices = list(range(len(results)))

    plt.figure(figsize=(10, 8))

    # Plot Time
    plt.subplot(3, 1, 1)
    plt.plot(configuration_indices, times, label='Time (s)', marker='o')
    plt.ylabel('Time (s)')
    plt.title(f'{algorithm_name} Performance')
    plt.xticks(configuration_indices)

    # Plot Moves
    plt.subplot(3, 1, 2)
    plt.plot(configuration_indices, moves, label='Moves', marker='o')
    plt.ylabel('Moves')
    plt.xticks(configuration_indices)

    # Plot Memory Usage
    plt.subplot(3, 1, 3)
    plt.plot(configuration_indices, memory_usages, label='Memory Usage (MiB)',␣
 ↪marker='o')
    plt.ylabel('Memory Usage (MiB)')
    plt.xticks(configuration_indices)

    plt.xlabel('Configuration Index')
    plt.tight_layout()
    plt.legend()
    plt.show()

results_dfs = []
```

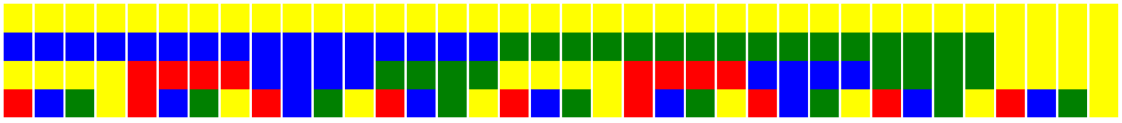
```
results_bfs = []
results_a_star = []
results_iddfs = []
results_bidirectional = []
```

[5]:
```
for config in initial_configurations:
    tower = CubeTower(config)
    print(f"Visualizing configuration: {config}")
    # tower.visualize()

    #~~~~~~~~DFS~~~~~~~~
    start_time = time.time()
    mem_usage_before = memory_usage(-1, interval=0.01, timeout=1,␣
 ↪max_usage=True)
    solution_dfs = dfs_search(tower)
    mem_usage_after = memory_usage(-1, interval=0.01, timeout=1, max_usage=True)
    end_time = time.time()
    if solution_dfs:
        print("    -> Visualizing DFS Solution")
        solution_dfs['time'] = end_time - start_time
        solution_dfs['memory_usage'] = mem_usage_after - mem_usage_before
        results_dfs.append(solution_dfs)
        print(f"Configuration: {config}, Moves: {solution_dfs['moves']}, Time:␣
 ↪{solution_dfs['time']:.4f}s, Memory Usage: {solution_dfs['memory_usage']:.
 ↪4f} MiB")
        solution_dfs['solution'].visualize_path()

    else:
        print("No solution of the DFS Algorithm found")
getPlots(results_dfs, "DFS")
```

```
Visualizing configuration: ['red', 'yellow', 'blue']
    -> Visualizing DFS Solution
Configuration: ['red', 'yellow', 'blue'], Moves: 9, Time: 2.0990s, Memory Usage:
0.0000 MiB
```



```
Visualizing configuration: ['red', 'yellow', 'blue', 'yellow']
    -> Visualizing DFS Solution
Configuration: ['red', 'yellow', 'blue', 'yellow'], Moves: 35, Time: 2.1131s,
Memory Usage: 0.0039 MiB
```



```
Visualizing configuration: ['blue', 'green', 'red', 'yellow']
    -> Visualizing DFS Solution
Configuration: ['blue', 'green', 'red', 'yellow'], Moves: 54, Time: 2.1108s,
Memory Usage: 0.0039 MiB
```

```
Visualizing configuration: ['blue', 'yellow', 'blue', 'yellow', 'red']
    -> Visualizing DFS Solution
Configuration: ['blue', 'yellow', 'blue', 'yellow', 'red'], Moves: 119, Time:
2.0896s, Memory Usage: 0.0000 MiB
```

```
Visualizing configuration: ['yellow', 'red', 'blue', 'yellow', 'green', 'blue']
    -> Visualizing DFS Solution
Configuration: ['yellow', 'red', 'blue', 'yellow', 'green', 'blue'], Moves: 902,
Time: 2.1120s, Memory Usage: 0.1211 MiB
```

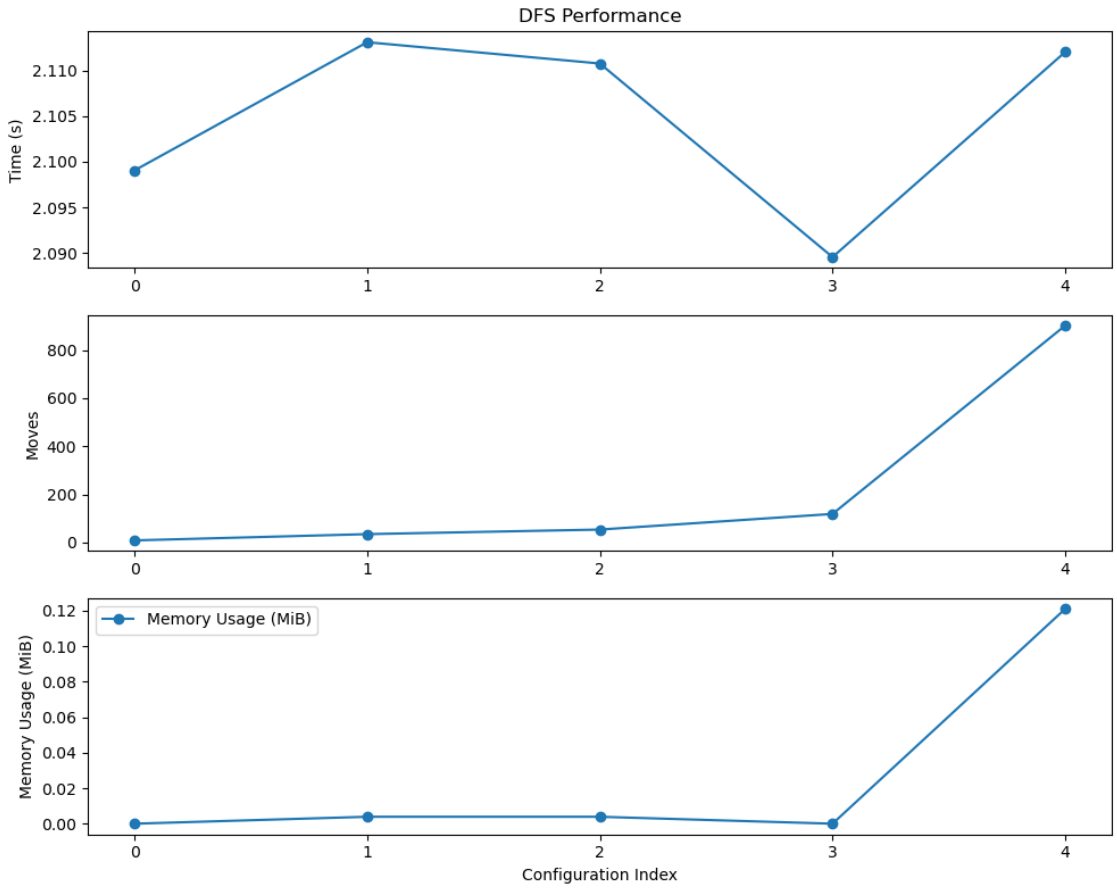
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
File ~\anaconda3\Lib\site-packages\IPython\core\formatters.py:340, in␣
 ↪BaseFormatter.__call__(self, obj)
    338     pass
    339 else:
--> 340     return printer(obj)
    341 # Finally look for special method names
    342 method = get_real_method(obj, self.print_method)

File ~\anaconda3\Lib\site-packages\IPython\core\pylabtools.py:152, in␣
 ↪print_figure(fig, fmt, bbox_inches, base64, **kwargs)
```

```
    149     from matplotlib.backend_bases import FigureCanvasBase
    150     FigureCanvasBase(fig)
--> 152 fig.canvas.print_figure(bytes_io, **kw)
    153 data = bytes_io.getvalue()
    154 if fmt == 'svg':

File ~\anaconda3\Lib\site-packages\matplotlib\backend_bases.py:2336, in␣
 ↪FigureCanvasBase.print_figure(self, filename, dpi, facecolor, edgecolor,␣
 ↪orientation, format, bbox_inches, pad_inches, bbox_extra_artists, backend,␣
 ↪**kwargs)
   2329     bbox_inches = rcParams['savefig.bbox']
   2331 if (self.figure.get_layout_engine() is not None or
   2332         bbox_inches == "tight"):
   2333     # we need to trigger a draw before printing to make sure
   2334     # CL works.  "tight" also needs a draw to get the right
   2335     # locations:
-> 2336     renderer = _get_renderer(
   2337         self.figure,
   2338         functools.partial(
   2339             print_method, orientation=orientation)
   2340     )
   2341     with getattr(renderer, "_draw_disabled", nullcontext)():
   2342         self.figure.draw(renderer)

File ~\anaconda3\Lib\site-packages\matplotlib\backend_bases.py:1598, in␣
 ↪_get_renderer(figure, print_method)
   1595     print_method = stack.enter_context(
   1596         figure.canvas._switch_canvas_and_return_print_method(fmt))
   1597 try:
-> 1598     print_method(io.BytesIO())
   1599 except Done as exc:
   1600     renderer, = exc.args

File ~\anaconda3\Lib\site-packages\matplotlib\backend_bases.py:2232, in␣
 ↪FigureCanvasBase._switch_canvas_and_return_print_method.<locals>.
 ↪<lambda>(*args, **kwargs)
   2228     optional_kws = {  # Passed by print_figure for other renderers.
   2229         "dpi", "facecolor", "edgecolor", "orientation",
   2230         "bbox_inches_restore"}
   2231     skip = optional_kws - {*inspect.signature(meth).parameters}
-> 2232     print_method = functools.wraps(meth)(lambda *args, **kwargs: meth(
   2233         *args, **{k: v for k, v in kwargs.items() if k not in skip}))
   2234 else:  # Let third-parties do as they see fit.
   2235     print_method = meth

File ~\anaconda3\Lib\site-packages\matplotlib\backends\backend_agg.py:509, in␣
 ↪FigureCanvasAgg.print_png(self, filename_or_obj, metadata, pil_kwargs)
    462 def print_png(self, filename_or_obj, *, metadata=None, pil_kwargs=None)
```

```
    463     """
    464     Write the figure to a PNG file.
    465
   (…)
    507         *metadata*, including the default 'Software' key.
    508     """
--> 509     self._print_pil(filename_or_obj, "png", pil_kwargs, metadata)

File ~\anaconda3\Lib\site-packages\matplotlib\backends\backend_agg.py:457, in␣
 ↪FigureCanvasAgg._print_pil(self, filename_or_obj, fmt, pil_kwargs, metadata)
    452 def _print_pil(self, filename_or_obj, fmt, pil_kwargs, metadata=None):
    453     """
    454     Draw the canvas, then save it using `.image.imsave` (to which
    455     *pil_kwargs* and *metadata* are forwarded).
    456     """
--> 457     FigureCanvasAgg.draw(self)
    458     mpl.image.imsave(
    459         filename_or_obj, self.buffer_rgba(), format=fmt, origin="upper"
    460         dpi=self.figure.dpi, metadata=metadata, pil_kwargs=pil_kwargs)

File ~\anaconda3\Lib\site-packages\matplotlib\backends\backend_agg.py:394, in␣
 ↪FigureCanvasAgg.draw(self)
    392 def draw(self):
    393     # docstring inherited
--> 394     self.renderer = self.get_renderer()
    395     self.renderer.clear()
    396     # Acquire a lock on the shared font cache.

File ~\anaconda3\Lib\site-packages\matplotlib\_api\deprecation.py:384, in␣
 ↪delete_parameter.<locals>.wrapper(*inner_args, **inner_kwargs)
    379 @functools.wraps(func)
    380 def wrapper(*inner_args, **inner_kwargs):
    381     if len(inner_args) <= name_idx and name not in inner_kwargs:
    382         # Early return in the simple, non-deprecated case (much faster␣
 ↪than
    383         # calling bind()).
--> 384         return func(*inner_args, **inner_kwargs)
    385     arguments = signature.bind(*inner_args, **inner_kwargs).arguments
    386     if is_varargs and arguments.get(name):

File ~\anaconda3\Lib\site-packages\matplotlib\backends\backend_agg.py:411, in␣
 ↪FigureCanvasAgg.get_renderer(self, cleared)
    409 reuse_renderer = (self._lastKey == key)
    410 if not reuse_renderer:
--> 411     self.renderer = RendererAgg(w, h, self.figure.dpi)
    412     self._lastKey = key
    413 elif cleared:
```

```
File ~\anaconda3\Lib\site-packages\matplotlib\backends\backend_agg.py:84, in␣
 ↪RendererAgg.__init__(self, width, height, dpi)
     82 self.width = width
     83 self.height = height
---> 84 self._renderer = _RendererAgg(int(width), int(height), dpi)
     85 self._filter_renderers = []
     87 self._update_methods()

ValueError: Image size of 180600x600 pixels is too large. It must be less than␣
 ↪2^16 in each direction.
```

```
<Figure size 180600x600 with 1 Axes>
```



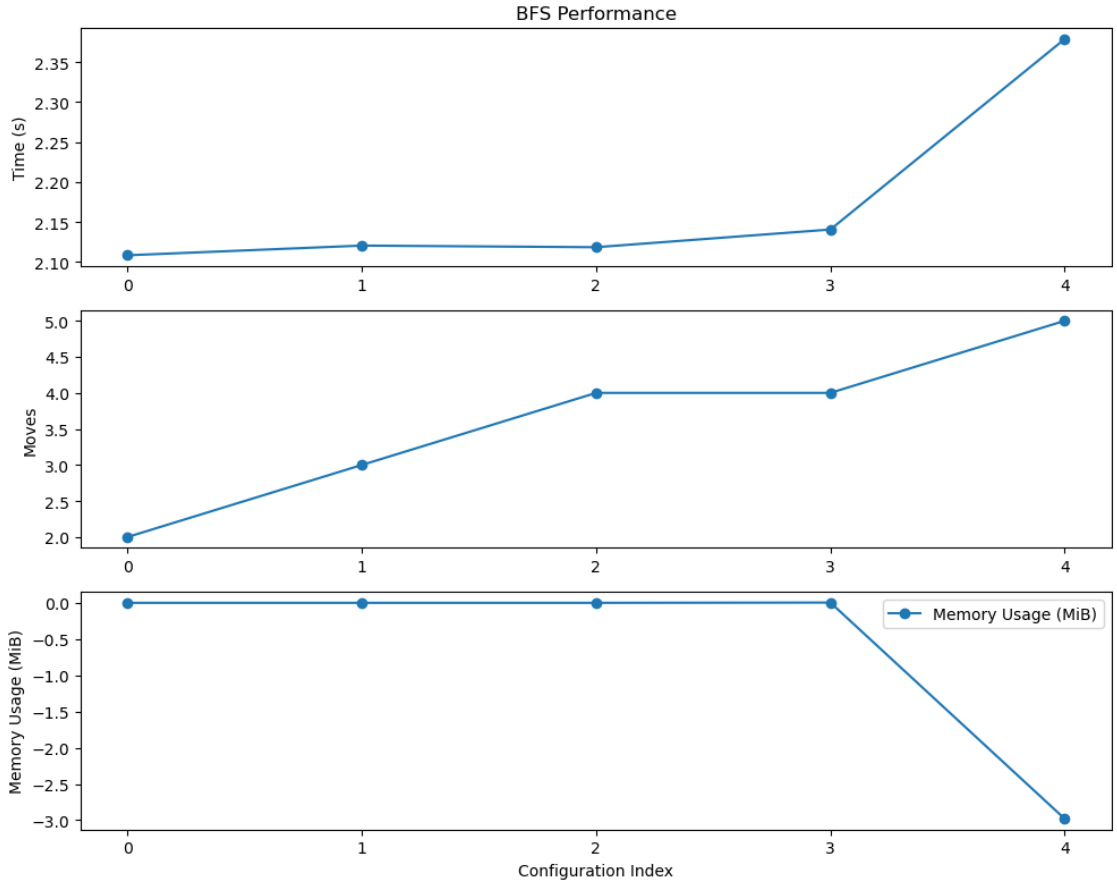
The error we encountered above is due to the image being so large that it cannot be rendered.

[20]:
```
results_bfs = []
initial_configurations = [
    ["red","yellow","blue"],
    ["red","yellow","blue","yellow"],
```

```
    ["blue","green","red","yellow"],
    ["blue","yellow","blue","yellow", "red"],
    ["yellow","red","blue","yellow", "green", "blue"]
]
for config in initial_configurations:
    tower = CubeTower(config)
    print(f"Visualizing configuration: {config}")
    # tower.visualize()

    #~~~~~~~~BFS~~~~~~~~
    start_time = time.time()
    mem_usage_before = memory_usage(-1, interval=0.01, timeout=1,␣
 ↪max_usage=True)
    solution_bfs = bfs_search(tower)
    mem_usage_after = memory_usage(-1, interval=0.01, timeout=1, max_usage=True)
    end_time = time.time()
    if solution_bfs:
        print("    -> Visualizing BFS Solution")
        solution_bfs['time'] = end_time - start_time
        solution_bfs['memory_usage'] = mem_usage_after - mem_usage_before
        results_bfs.append(solution_bfs)
        print(f"Configuration: {config}, Moves: {solution_bfs['moves']}, Time:␣
 ↪{solution_bfs['time']:.4f}s, Memory Usage: {solution_bfs['memory_usage']:.
 ↪4f} MiB")
        # solution_bfs['solution'].visualize_path()
    else:
        print("No solution of the BFS Algorithm found")
getPlots(results_bfs, "BFS")
```

```
Visualizing configuration: ['red', 'yellow', 'blue']
    -> Visualizing BFS Solution
Configuration: ['red', 'yellow', 'blue'], Moves: 2, Time: 2.1085s, Memory Usage:
0.0000 MiB
Visualizing configuration: ['red', 'yellow', 'blue', 'yellow']
    -> Visualizing BFS Solution
Configuration: ['red', 'yellow', 'blue', 'yellow'], Moves: 3, Time: 2.1206s,
Memory Usage: 0.0000 MiB
Visualizing configuration: ['blue', 'green', 'red', 'yellow']
    -> Visualizing BFS Solution
Configuration: ['blue', 'green', 'red', 'yellow'], Moves: 4, Time: 2.1186s,
Memory Usage: 0.0000 MiB
Visualizing configuration: ['blue', 'yellow', 'blue', 'yellow', 'red']
    -> Visualizing BFS Solution
Configuration: ['blue', 'yellow', 'blue', 'yellow', 'red'], Moves: 4, Time:
2.1407s, Memory Usage: 0.0039 MiB
Visualizing configuration: ['yellow', 'red', 'blue', 'yellow', 'green', 'blue']
    -> Visualizing BFS Solution
```

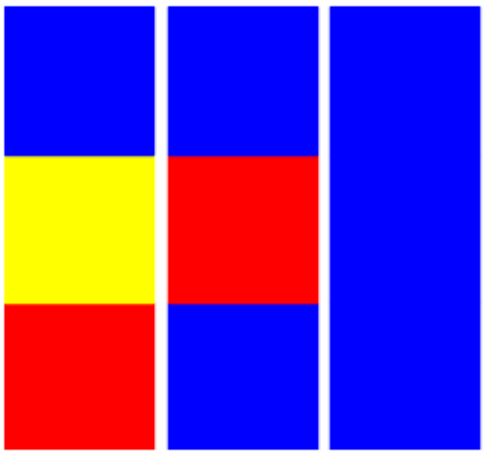
```
Configuration: ['yellow', 'red', 'blue', 'yellow', 'green', 'blue'], Moves: 5,
Time: 2.3785s, Memory Usage: -2.9805 MiB
```



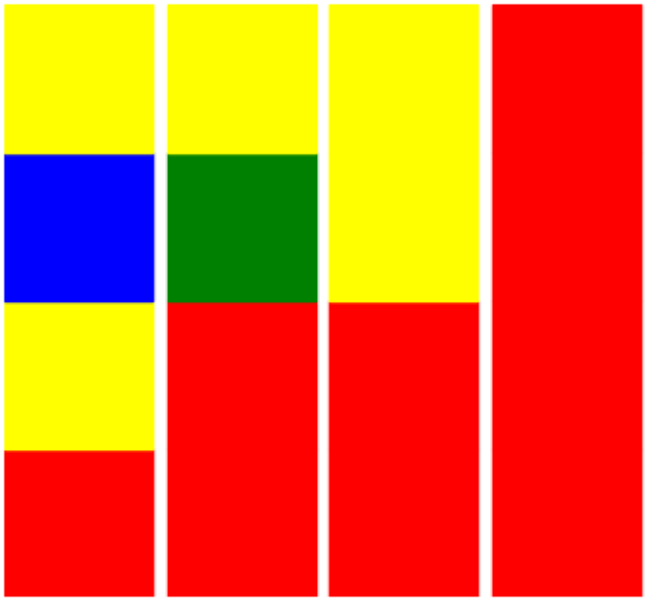
[4]:
```
results_bfs = []
initial_configurations = [
    ["red","yellow","blue"],
    ["red","yellow","blue","yellow"],
    ["blue","green","red","yellow"],
    ["blue","yellow","blue","yellow", "red"]
]
results_bfs = []
for config in initial_configurations:
    tower = CubeTower(config)
    print(f"Visualizing configuration: {config}")
    # tower.visualize()

    #~~~~~~~~BFS~~~~~~~~
    start_time = time.time()
```

```
    mem_usage_before = memory_usage(-1, interval=0.01, timeout=1,␣
 ↪max_usage=True)
    solution_bfs = bfs_search(tower)
    mem_usage_after = memory_usage(-1, interval=0.01, timeout=1, max_usage=True)
    end_time = time.time()
    if solution_bfs:
        print("    -> Visualizing BFS Solution")
        solution_bfs['time'] = end_time - start_time
        solution_bfs['memory_usage'] = mem_usage_after - mem_usage_before
        results_bfs.append(solution_bfs)
        print(f"Configuration: {config}, Moves: {solution_bfs['moves']}, Time:␣
 ↪{solution_bfs['time']:.4f}s, Memory Usage: {solution_bfs['memory_usage']:.
 ↪4f} MiB")
        solution_bfs['solution'].visualize_path()
    else:
        print("No solution of the BFS Algorithm found")
getPlots(results_bfs, "BFS")
```

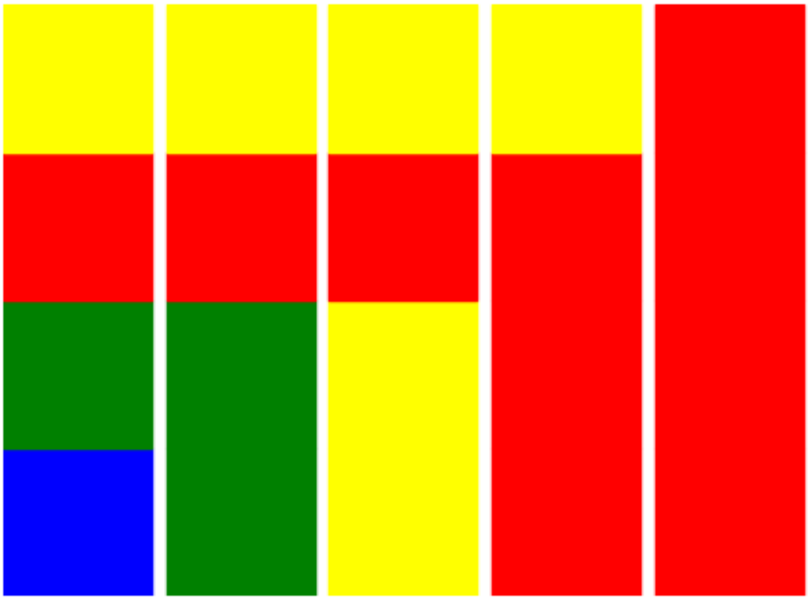
```
Visualizing configuration: ['red', 'yellow', 'blue']
    -> Visualizing BFS Solution
Configuration: ['red', 'yellow', 'blue'], Moves: 2, Time: 2.1010s, Memory Usage:
0.0000 MiB
```



```
Visualizing configuration: ['red', 'yellow', 'blue', 'yellow']
    -> Visualizing BFS Solution
Configuration: ['red', 'yellow', 'blue', 'yellow'], Moves: 3, Time: 2.0957s,
Memory Usage: 0.0391 MiB
```

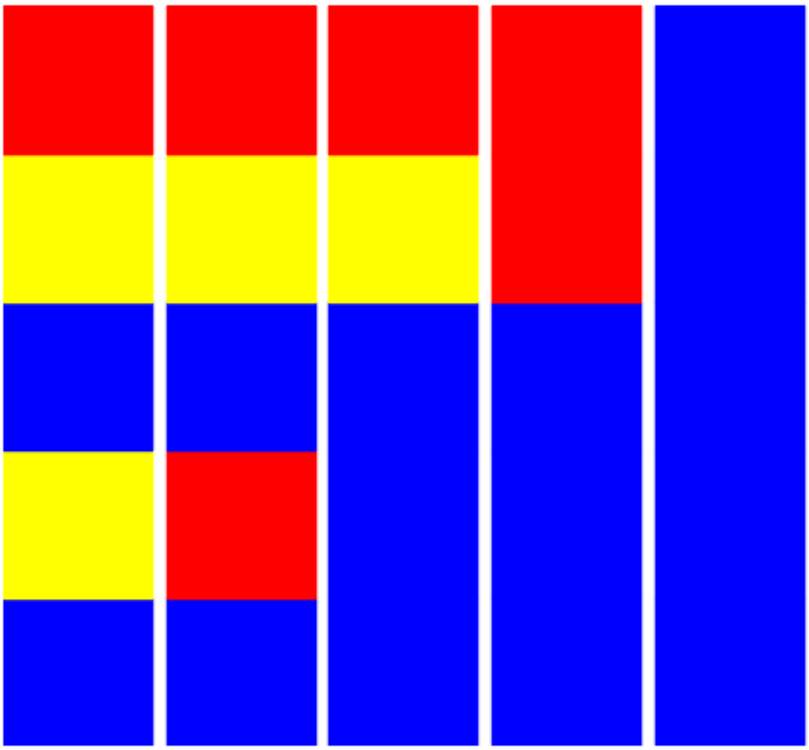


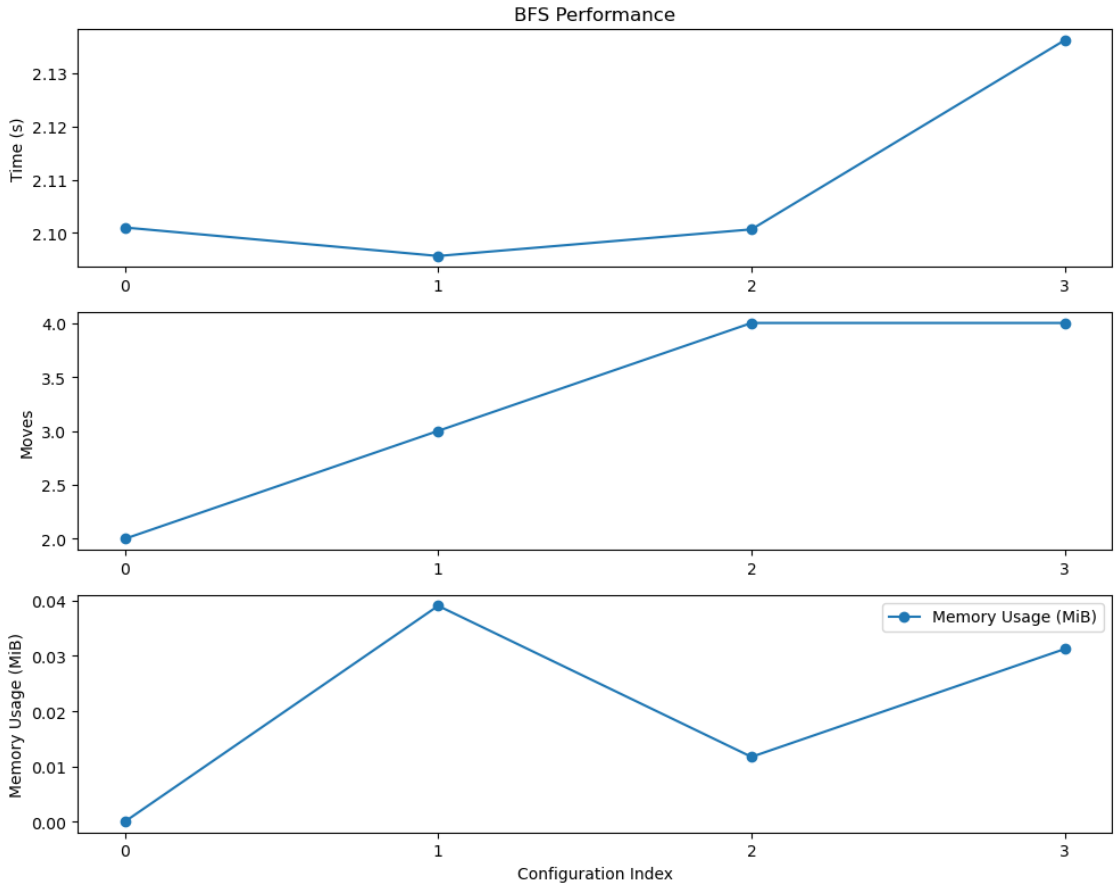
```
Visualizing configuration: ['blue', 'green', 'red', 'yellow']
    -> Visualizing BFS Solution
Configuration: ['blue', 'green', 'red', 'yellow'], Moves: 4, Time: 2.1007s,
Memory Usage: 0.0117 MiB
```



```
Visualizing configuration: ['blue', 'yellow', 'blue', 'yellow', 'red']
    -> Visualizing BFS Solution
```

Configuration: ['blue', 'yellow', 'blue', 'yellow', 'red'], Moves: 4, Time: 2.1362s, Memory Usage: 0.0312 MiB

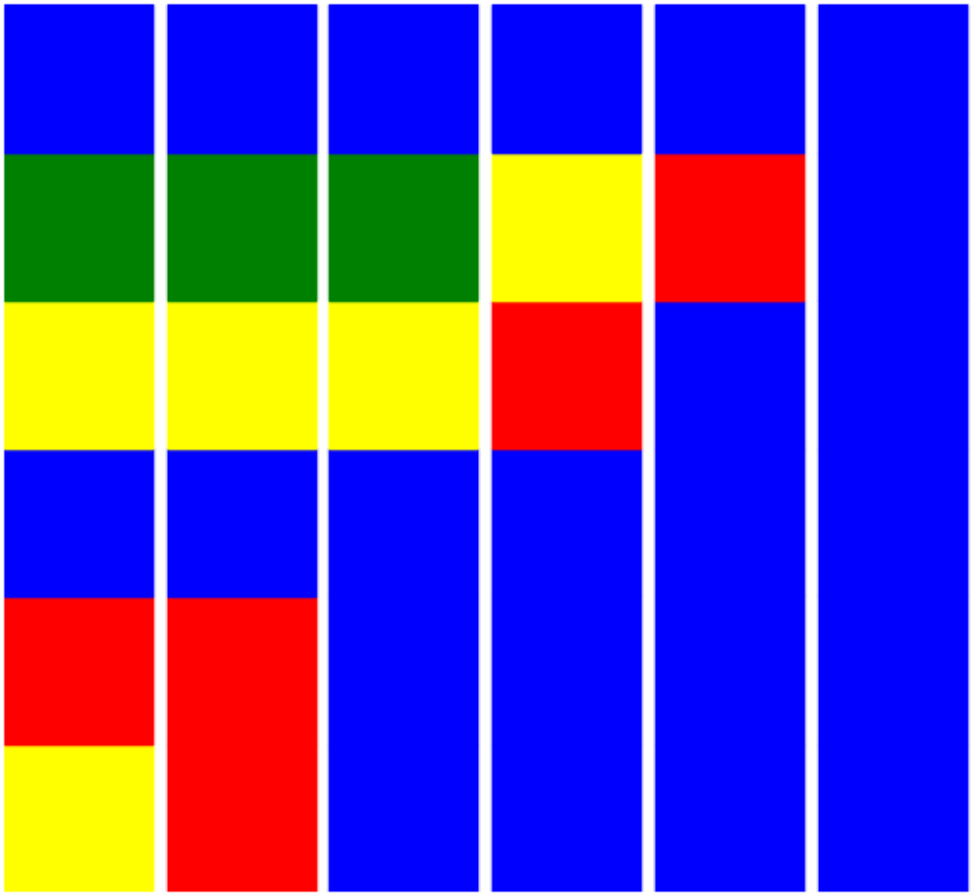


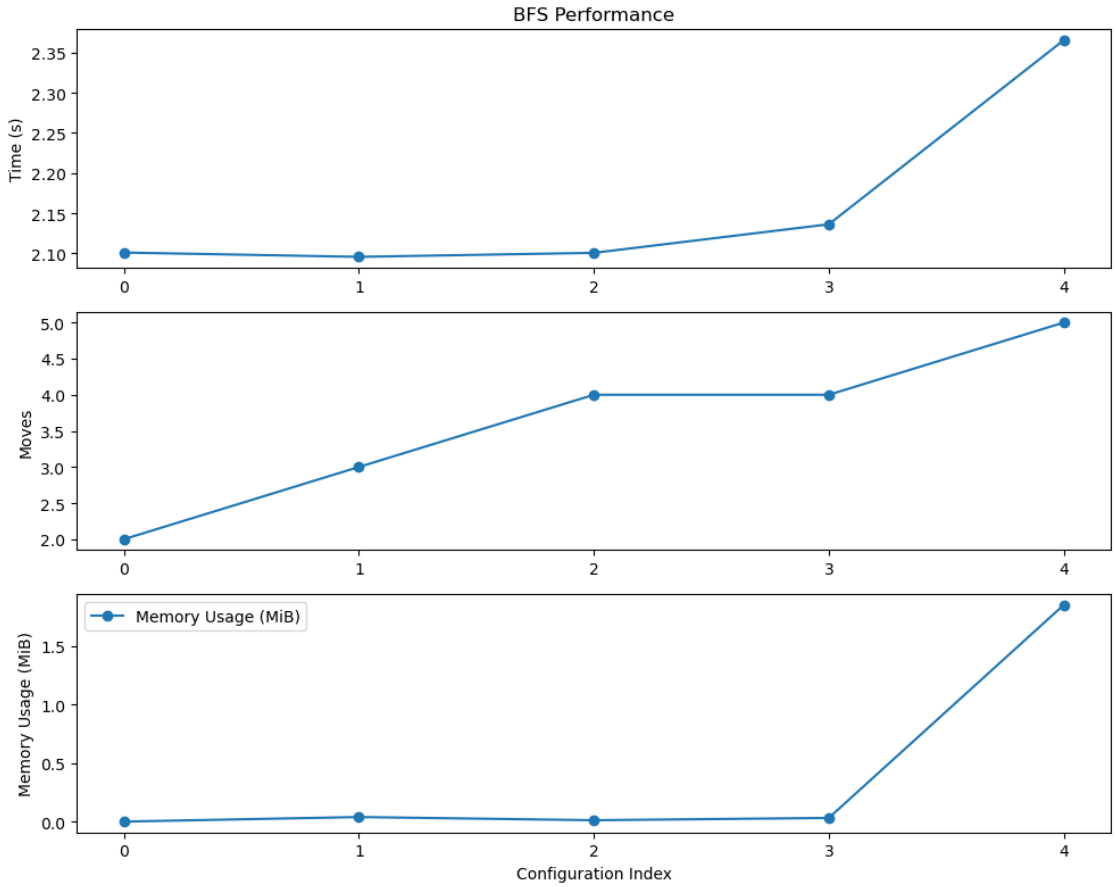


[5]:
```
initial_config = ["yellow","red","blue","yellow", "green", "blue"]
tower = CubeTower(initial_config)
print(f"Visualizing configuration: {initial_config}")
# tower.visualize()

#~~~~~~~~BFS~~~~~~~~
start_time = time.time()
mem_usage_before = memory_usage(-1, interval=0.01, timeout=1, max_usage=True)
solution_bfs = bfs_search(tower)
mem_usage_after = memory_usage(-1, interval=0.01, timeout=1, max_usage=True)
end_time = time.time()
if solution_bfs:
    print("    -> Visualizing BFS Solution")
    solution_bfs['time'] = end_time - start_time
    solution_bfs['memory_usage'] = mem_usage_after - mem_usage_before
    results_bfs.append(solution_bfs)
    print(f"Configuration: {initial_config}, Moves: {solution_bfs['moves']},␣
 ↪Time: {solution_bfs['time']:.4f}s, Memory Usage:␣
 ↪{solution_bfs['memory_usage']:.4f} MiB")
```

```
    solution_bfs['solution'].visualize_path()
else:
    print("No solution of the BFS Algorithm found")
getPlots(results_bfs, "BFS")
```

```
Visualizing configuration: ['yellow', 'red', 'blue', 'yellow', 'green', 'blue']
    -> Visualizing BFS Solution
Configuration: ['yellow', 'red', 'blue', 'yellow', 'green', 'blue'], Moves: 5,
Time: 2.3653s, Memory Usage: 1.8516 MiB
```





[35]:
```
results_bfs.pop(4)
```

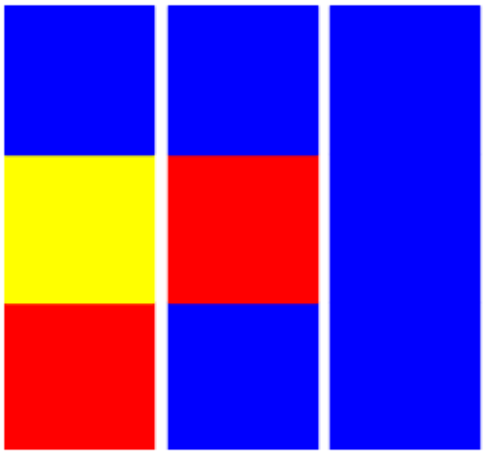
[35]:
```
{'solution': <__main__.CubeTower at 0x266da7b1c10>,
 'moves': 5,
 'time': 2.4831020832061768,
 'memory_usage': -3.80859375}
```

I managed to know the approximate value of the used memory by restarting the kernel and more methods to free the memory before it was released in the middle of analysis

[3]:
```
results_a_star = []
initial_configurations = [
    ["red","yellow","blue"],
    ["red","yellow","blue","yellow"],
    ["blue","green","red","yellow"],
    ["blue","yellow","blue","yellow", "red"],
    ["yellow","red","blue","yellow", "green", "blue"]
]
for config in initial_configurations:
    tower = CubeTower(config)
```

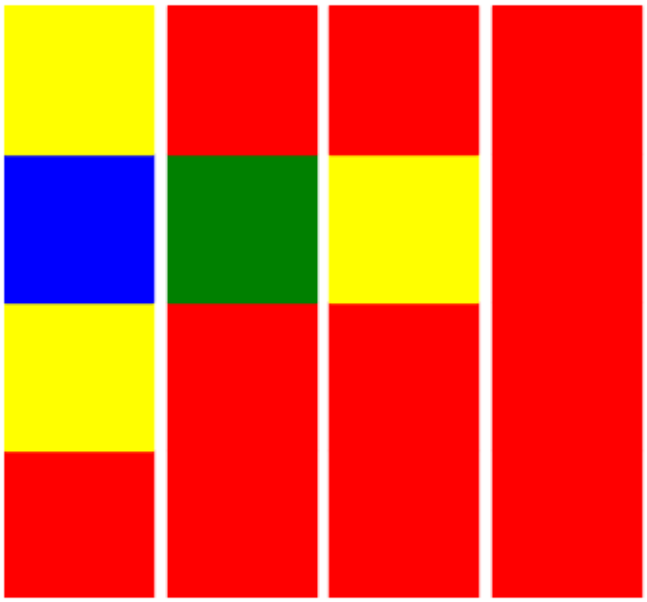
```
    print(f"Visualizing configuration: {config}")
    # tower.visualize()

    #~~~~~~~~A*~~~~~~~~
    start_time = time.time()
    mem_usage_before = memory_usage(-1, interval=0.01, timeout=1,␣
 ↪max_usage=True)
    solution_a_star = a_star_search(tower)
    mem_usage_after = memory_usage(-1, interval=0.01, timeout=1, max_usage=True)
    end_time = time.time()
    if solution_a_star:
        print("    -> Visualizing A Star Solution")
        solution_a_star['time'] = end_time - start_time
        solution_a_star['memory_usage'] = mem_usage_after - mem_usage_before
        results_a_star.append(solution_a_star)
        print(f"Configuration: {config}, Moves: {solution_a_star['moves']},␣
 ↪Time: {solution_a_star['time']:.4f}s, Memory Usage:␣
 ↪{solution_a_star['memory_usage']:.4f} MiB")
        solution_a_star['solution'].visualize_path()
    else:
        print("No solution found using A* Search")
getPlots(results_a_star, "A*")
```

```
Visualizing configuration: ['red', 'yellow', 'blue']
    -> Visualizing A Star Solution
Configuration: ['red', 'yellow', 'blue'], Moves: 2, Time: 2.1000s, Memory Usage:
0.0000 MiB
```

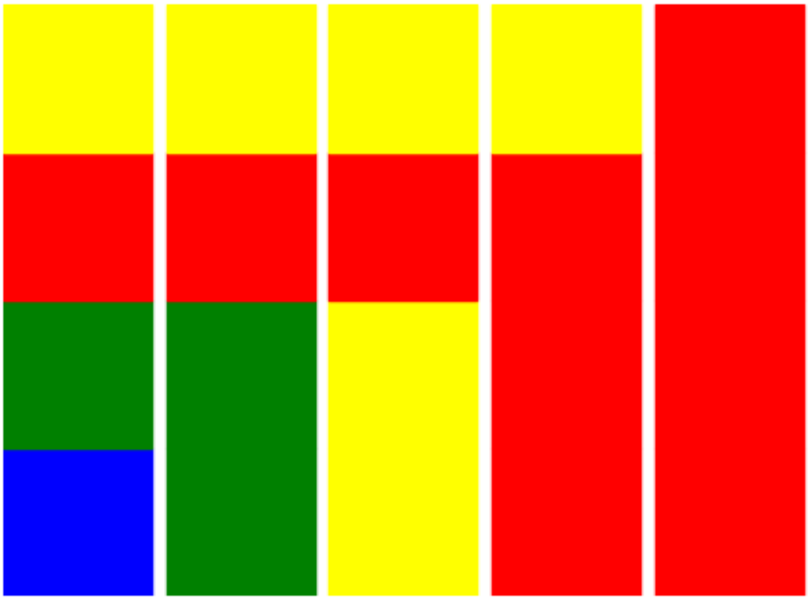


```
Visualizing configuration: ['red', 'yellow', 'blue', 'yellow']
    -> Visualizing A Star Solution
```

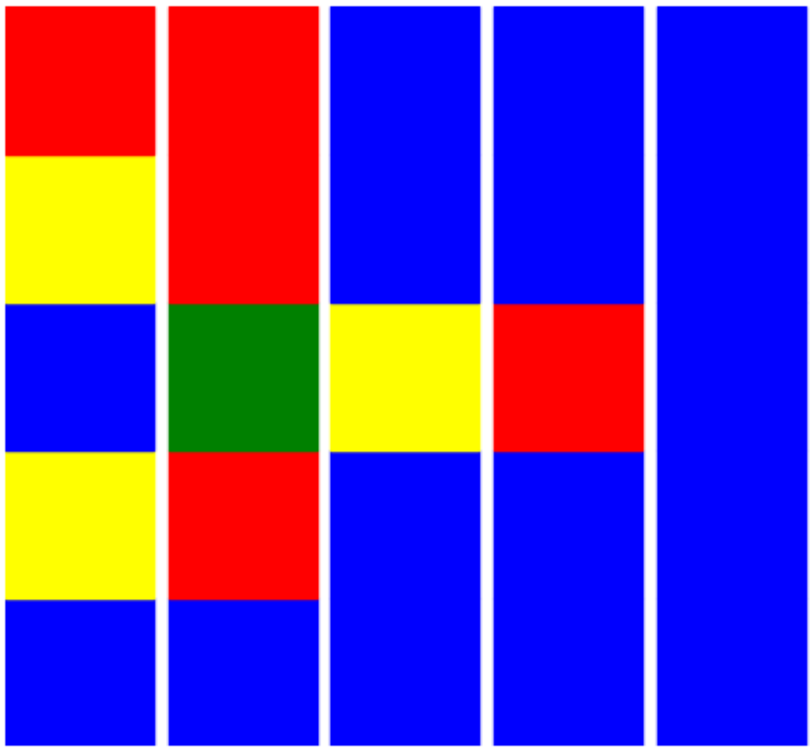
```
Configuration: ['red', 'yellow', 'blue', 'yellow'], Moves: 3, Time: 2.0948s,
Memory Usage: 0.0508 MiB
```



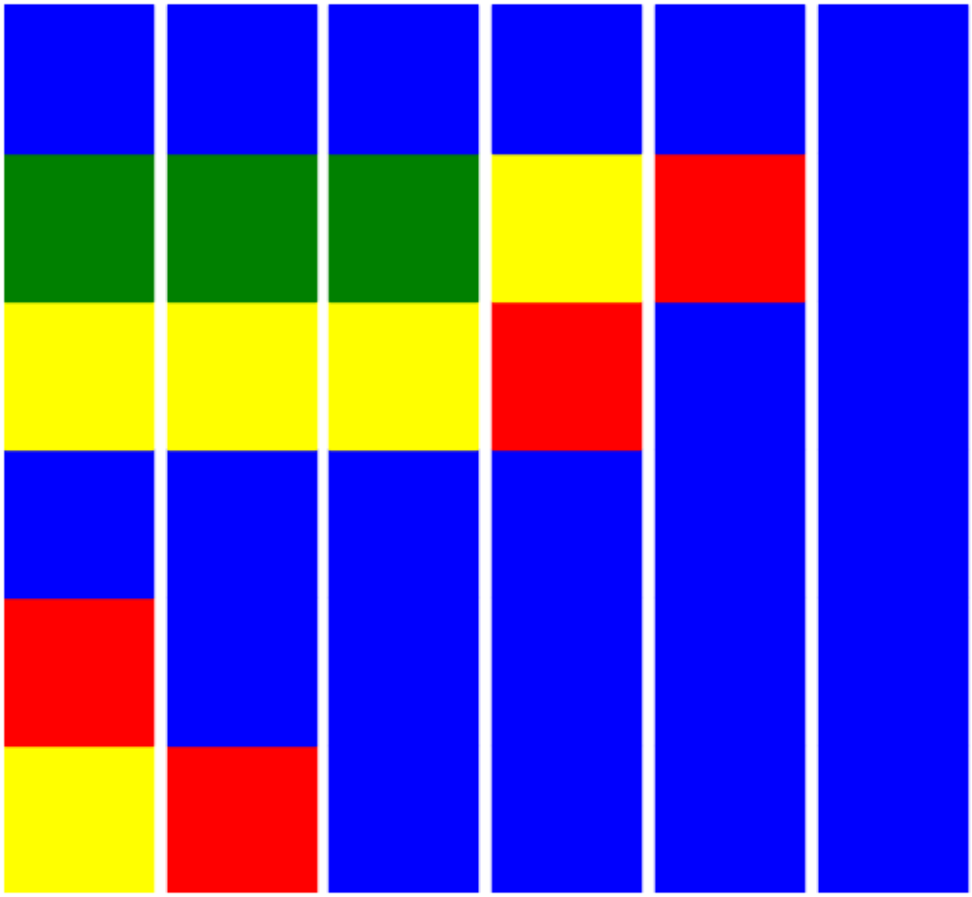
```
Visualizing configuration: ['blue', 'green', 'red', 'yellow']
    -> Visualizing A Star Solution
Configuration: ['blue', 'green', 'red', 'yellow'], Moves: 4, Time: 2.1357s,
Memory Usage: 0.3242 MiB
```

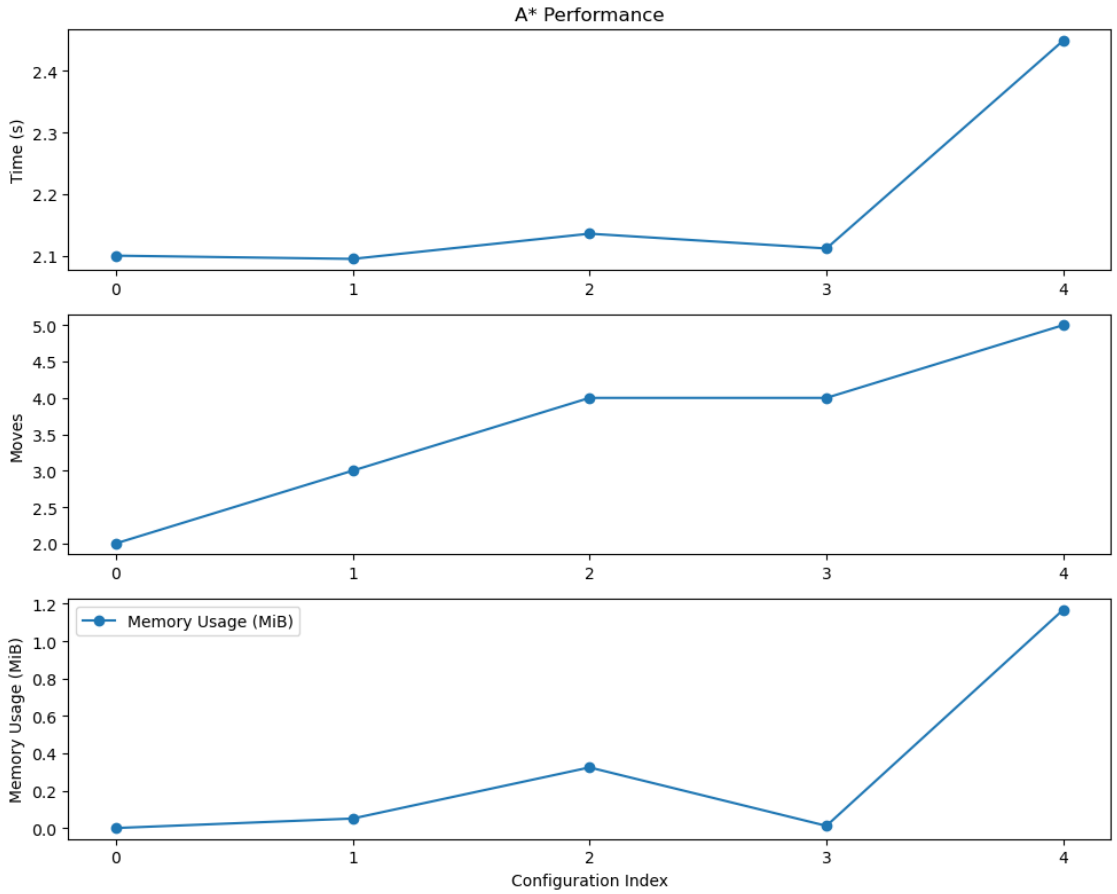


```
Visualizing configuration: ['blue', 'yellow', 'blue', 'yellow', 'red']
    -> Visualizing A Star Solution
Configuration: ['blue', 'yellow', 'blue', 'yellow', 'red'], Moves: 4, Time:
2.1117s, Memory Usage: 0.0117 MiB
```



```
Visualizing configuration: ['yellow', 'red', 'blue', 'yellow', 'green', 'blue']
    -> Visualizing A Star Solution
Configuration: ['yellow', 'red', 'blue', 'yellow', 'green', 'blue'], Moves: 5,
Time: 2.4489s, Memory Usage: 1.1680 MiB
```

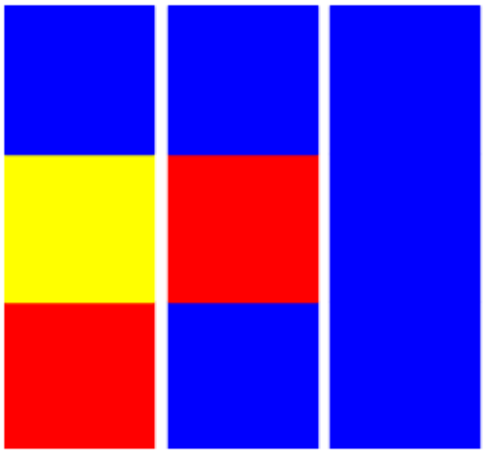




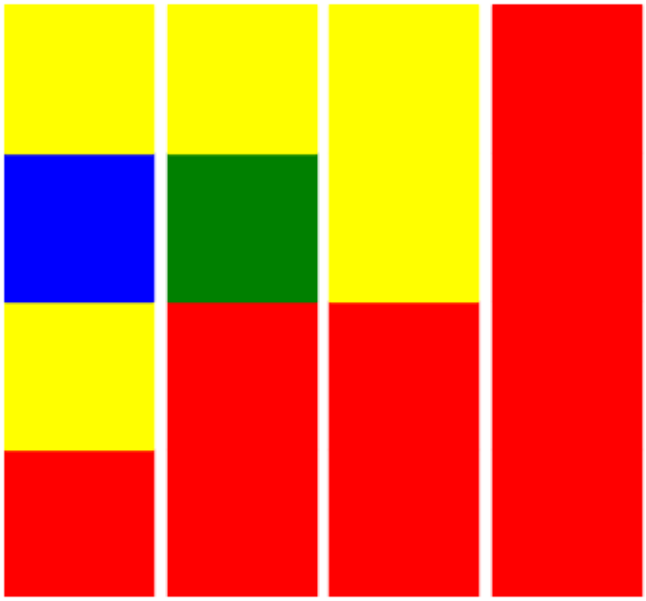
[5]:
```
results_iddfs = []
for config in initial_configurations:
    tower = CubeTower(config)
    print(f"Visualizing configuration: {config}")
    # tower.visualize()

    #~~~~~~~~IDDFS~~~~~~~~
    start_time = time.time()
    mem_usage_before = memory_usage(-1, interval=0.01, timeout=1,␣
 ↪max_usage=True)
    solution_iddfs = iddfs_search(tower)
    mem_usage_after = memory_usage(-1, interval=0.01, timeout=1, max_usage=True)
    end_time = time.time()
    if solution_iddfs:
        print("    -> Visualizing IDDFS Solution")
        solution_iddfs['time'] = end_time - start_time
        solution_iddfs['memory_usage'] = mem_usage_after - mem_usage_before
        results_iddfs.append(solution_iddfs)
```

```
        print(f"Configuration: {config}, Moves: {solution_iddfs['moves']}, Time:
 ↪ {solution_iddfs['time']:.4f}s, Memory Usage:␣
 ↪{solution_iddfs['memory_usage']:.4f} MiB")
        solution_iddfs['solution'].visualize_path()
    else:
        print("No solution found using IDDFS")
getPlots(results_iddfs, "IDDFS")
```

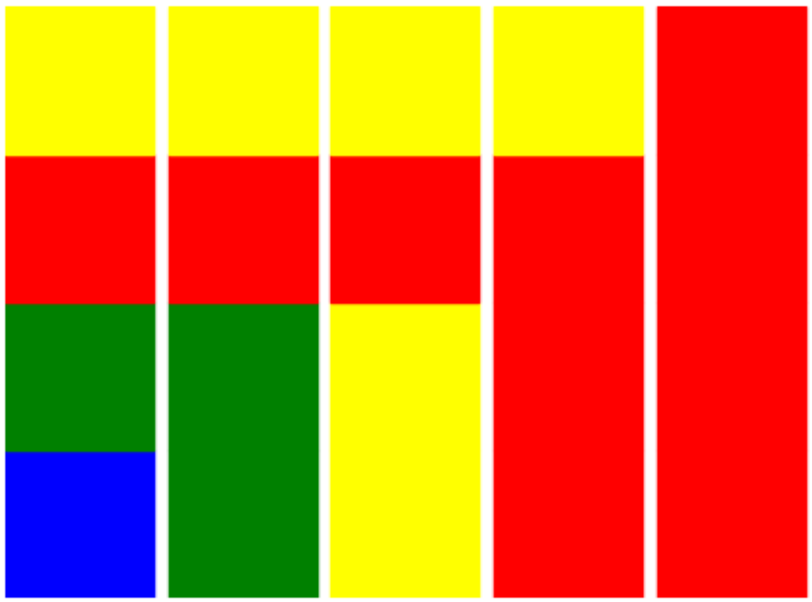
```
Visualizing configuration: ['red', 'yellow', 'blue']
    -> Visualizing IDDFS Solution
Configuration: ['red', 'yellow', 'blue'], Moves: 2, Time: 2.1152s, Memory Usage:
0.0000 MiB
```



```
Visualizing configuration: ['red', 'yellow', 'blue', 'yellow']
    -> Visualizing IDDFS Solution
Configuration: ['red', 'yellow', 'blue', 'yellow'], Moves: 3, Time: 2.1154s,
Memory Usage: 0.0000 MiB
```

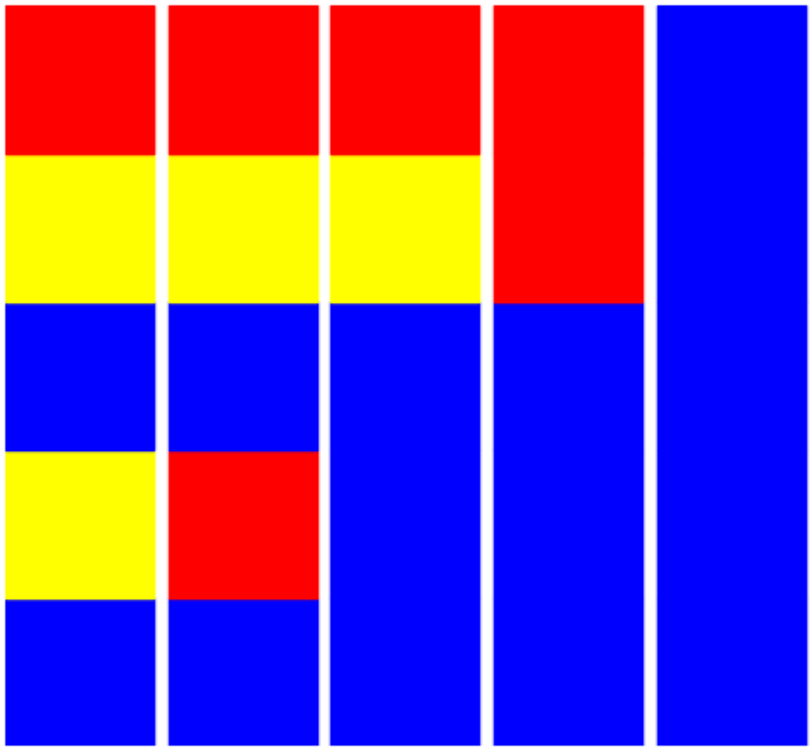


```
Visualizing configuration: ['blue', 'green', 'red', 'yellow']
    -> Visualizing IDDFS Solution
Configuration: ['blue', 'green', 'red', 'yellow'], Moves: 4, Time: 2.1012s,
Memory Usage: 0.0000 MiB
```

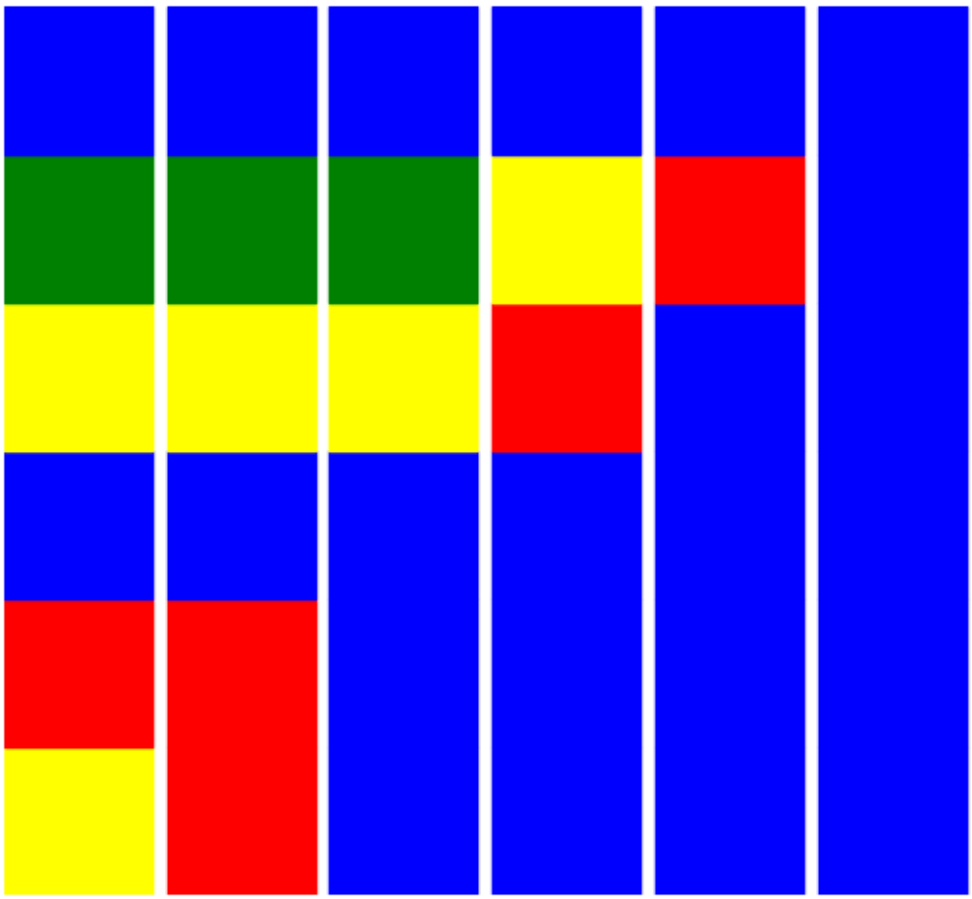


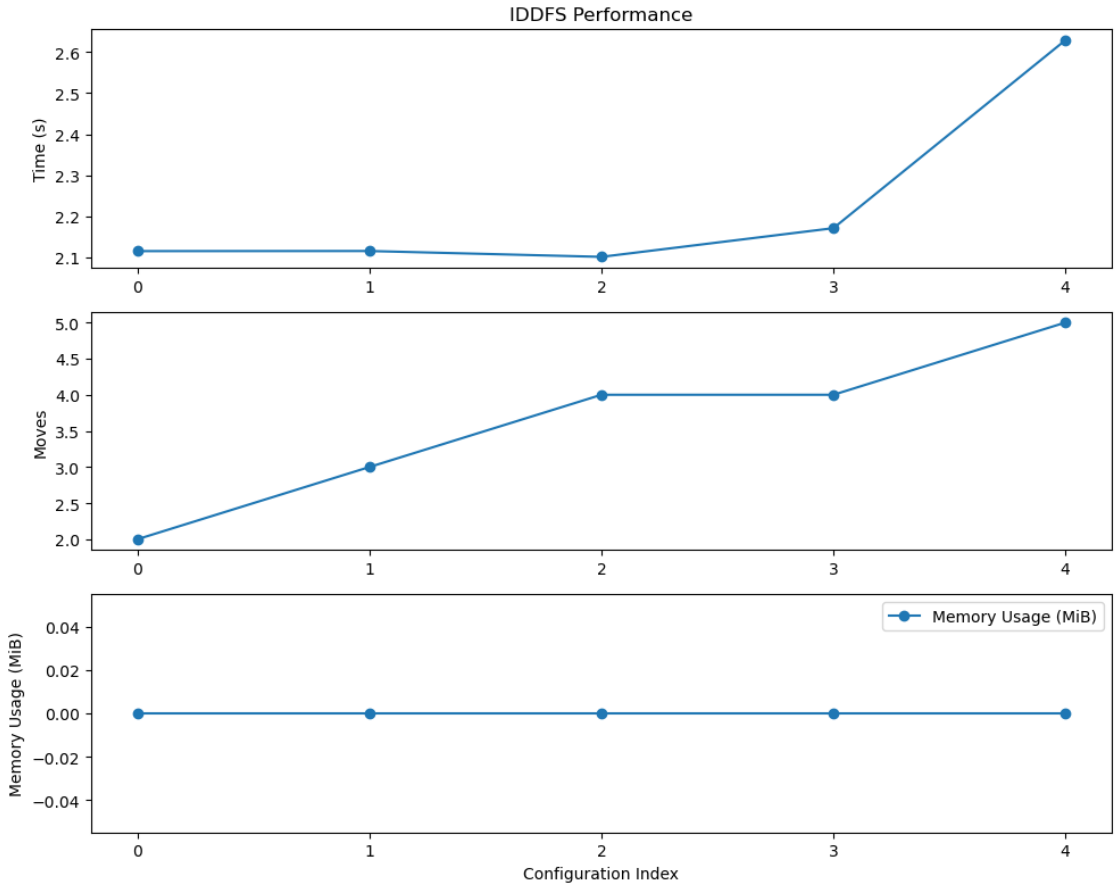
```
Visualizing configuration: ['blue', 'yellow', 'blue', 'yellow', 'red']
    -> Visualizing IDDFS Solution
```

```
Configuration: ['blue', 'yellow', 'blue', 'yellow', 'red'], Moves: 4, Time:
2.1711s, Memory Usage: 0.0000 MiB
```



```
Visualizing configuration: ['yellow', 'red', 'blue', 'yellow', 'green', 'blue']
    -> Visualizing IDDFS Solution
Configuration: ['yellow', 'red', 'blue', 'yellow', 'green', 'blue'], Moves: 5,
Time: 2.6288s, Memory Usage: 0.0000 MiB
```

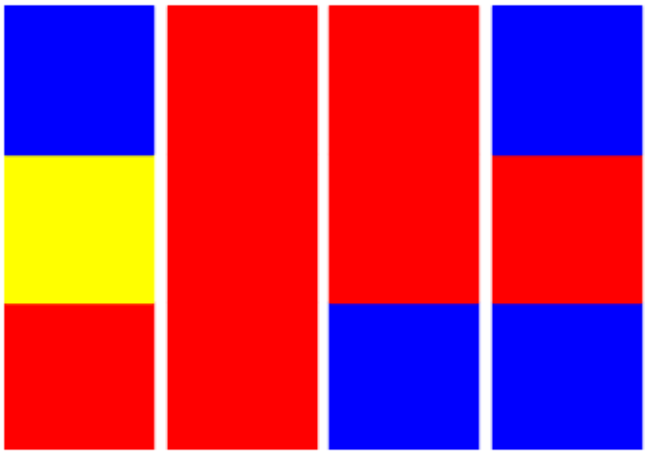




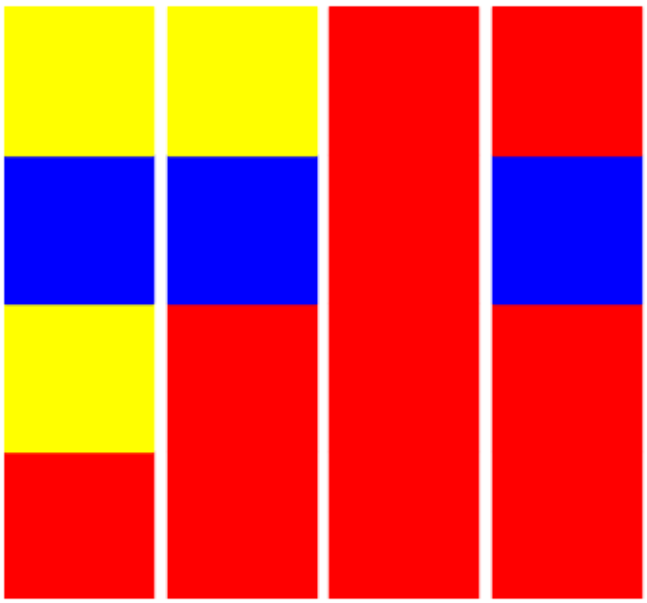
[3]:
```
results_bidirectional = []
for config in initial_configurations:
    tower = CubeTower(config)
    print(f"Visualizing configuration: {config}")
    # tower.visualize()

    #~~~~~~~~Bidirectional~~~~~~~~
    start_time = time.time()
    mem_usage_before = memory_usage(-1, interval=0.01, timeout=1,␣
 ↪max_usage=True)
    solution_bidirectional = bidirectional_search(tower)
    mem_usage_after = memory_usage(-1, interval=0.01, timeout=1, max_usage=True)
    end_time = time.time()
    if solution_bidirectional:
        print("    -> Visualizing Bidirectional Search Solution")
        solution_bidirectional['time'] = end_time - start_time
        solution_bidirectional['memory_usage'] = mem_usage_after -␣
 ↪mem_usage_before
        results_bidirectional.append(solution_bidirectional)
```

```
        print(f"Configuration: {config}, Moves:␣
 ↪{solution_bidirectional['moves']}, Time: {solution_bidirectional['time']:.
 ↪4f}s, Memory Usage: {solution_bidirectional['memory_usage']:.4f} MiB")
        path_forward, path_backward = solution_bidirectional['solution']
        tower.visualize_bidirectional_path(path_forward, path_backward)
    else:
        print("No solution found using Bidirectional Search")
getPlots(results_bidirectional, "Bidirectional")
```

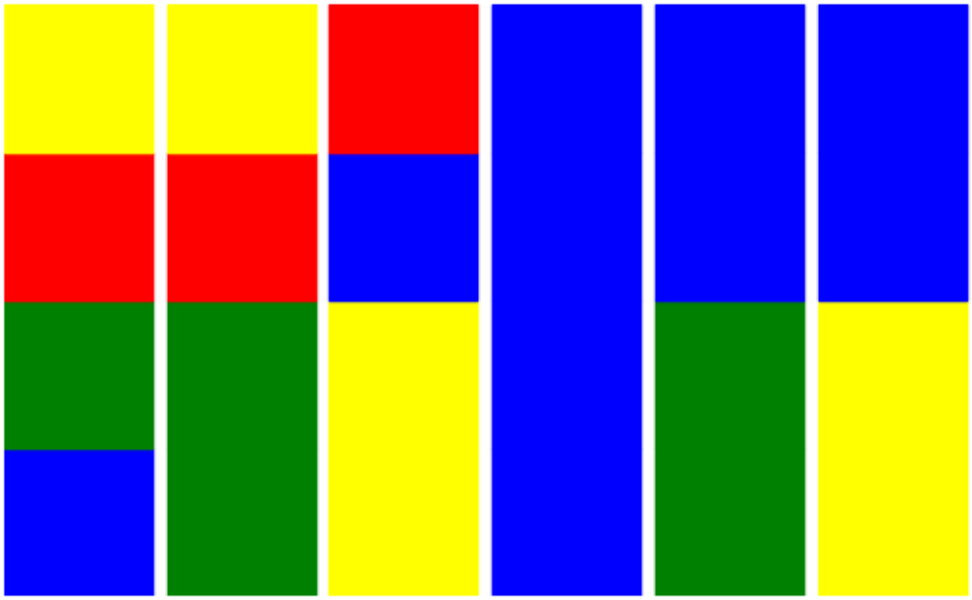
```
Visualizing configuration: ['red', 'yellow', 'blue']
    -> Visualizing Bidirectional Search Solution
Configuration: ['red', 'yellow', 'blue'], Moves: 5, Time: 2.0983s, Memory Usage:
0.0000 MiB
```



```
Visualizing configuration: ['red', 'yellow', 'blue', 'yellow']
    -> Visualizing Bidirectional Search Solution
Configuration: ['red', 'yellow', 'blue', 'yellow'], Moves: 5, Time: 2.1129s,
Memory Usage: 0.0820 MiB
```

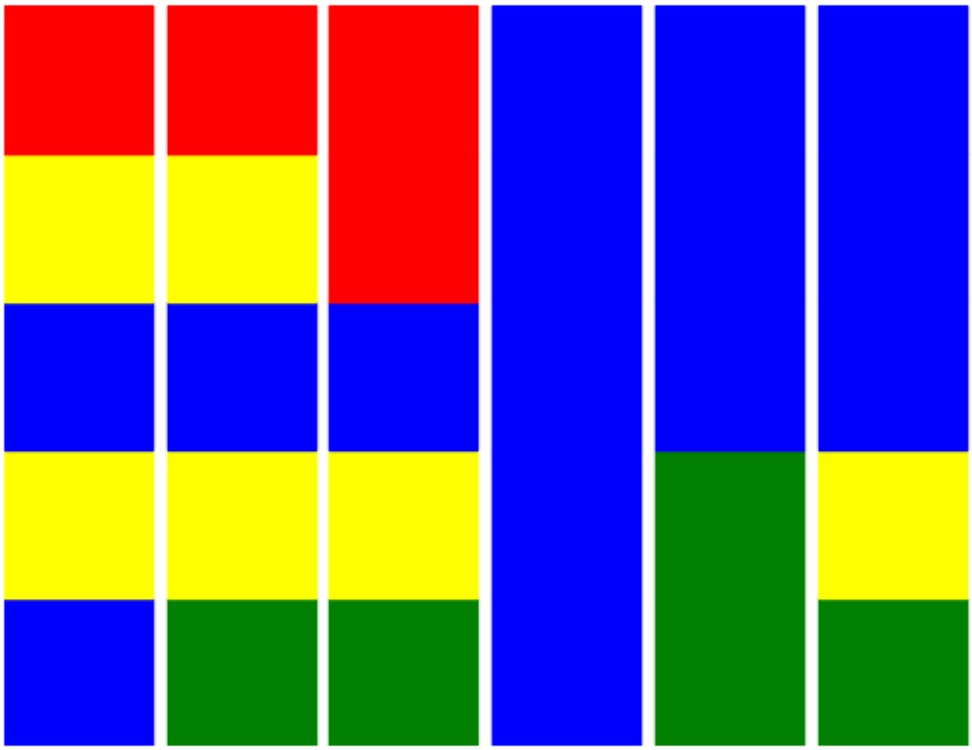


```
Visualizing configuration: ['blue', 'green', 'red', 'yellow']
    -> Visualizing Bidirectional Search Solution
Configuration: ['blue', 'green', 'red', 'yellow'], Moves: 7, Time: 2.1179s,
Memory Usage: 0.0000 MiB
```

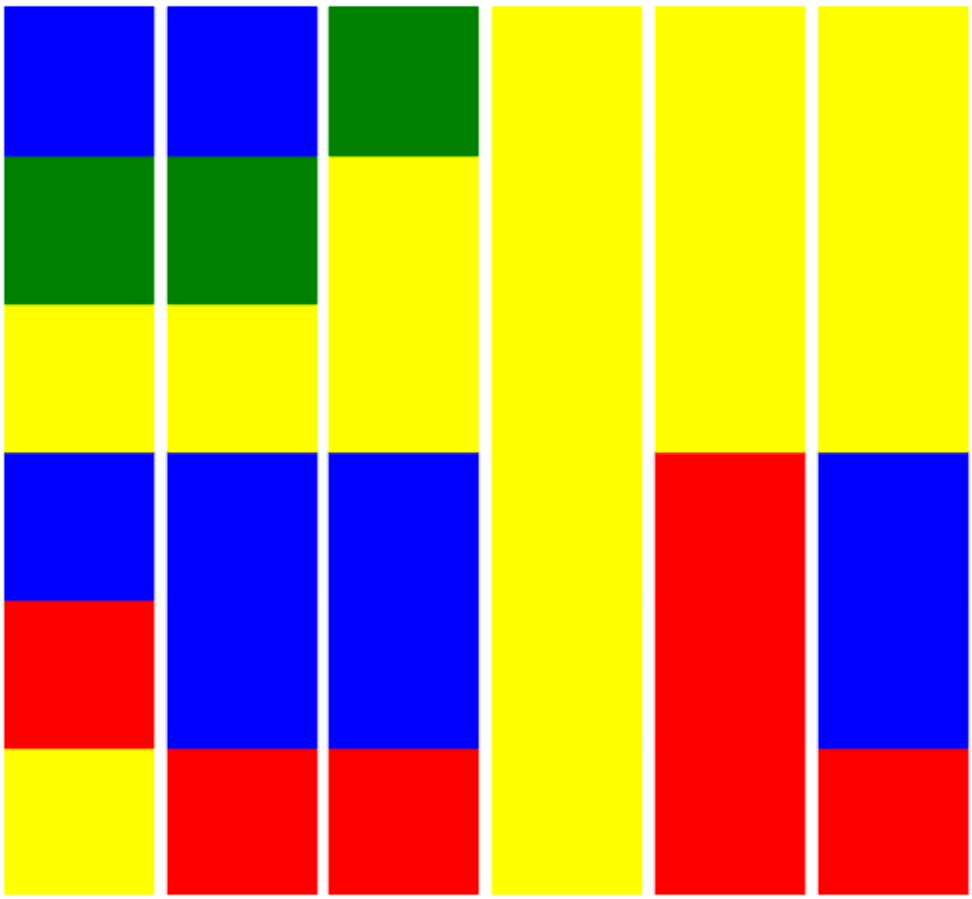


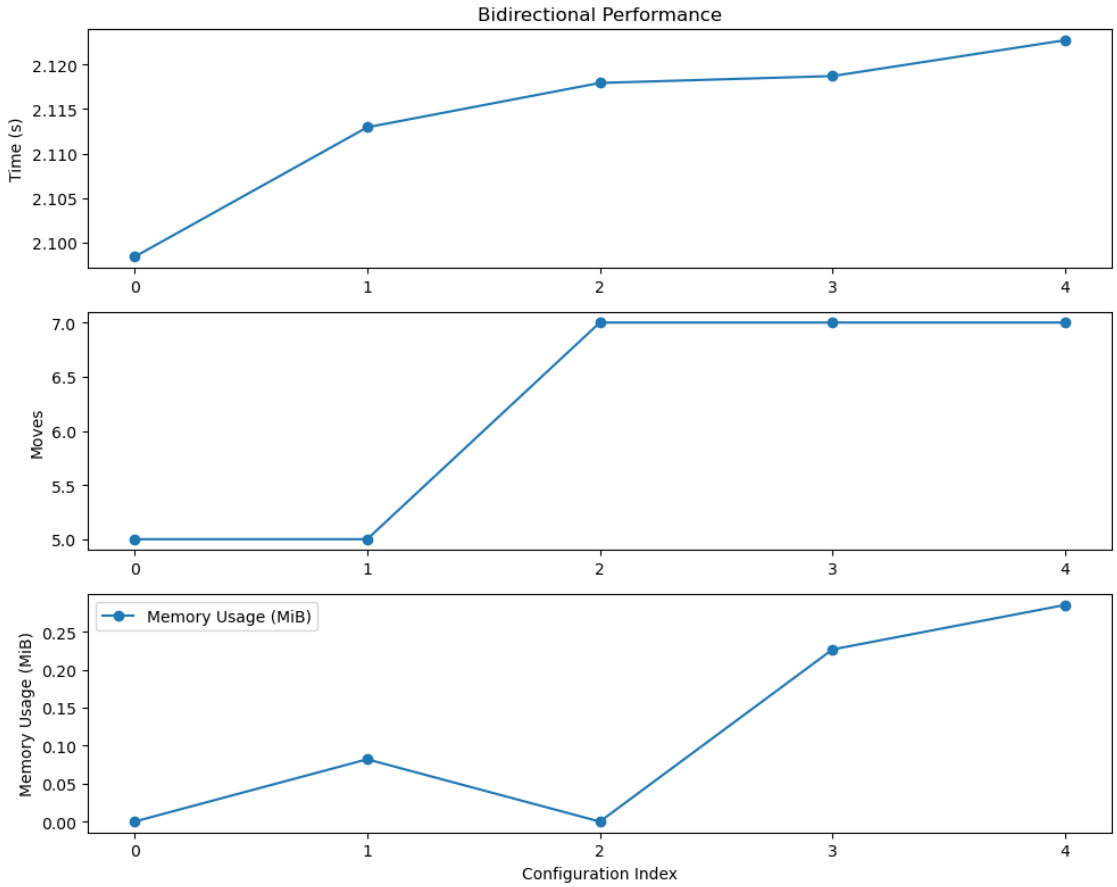
```
Visualizing configuration: ['blue', 'yellow', 'blue', 'yellow', 'red']
    -> Visualizing Bidirectional Search Solution
```

```
Configuration: ['blue', 'yellow', 'blue', 'yellow', 'red'], Moves: 7, Time:
2.1187s, Memory Usage: 0.2266 MiB
```



```
Visualizing configuration: ['yellow', 'red', 'blue', 'yellow', 'green', 'blue']
    -> Visualizing Bidirectional Search Solution
Configuration: ['yellow', 'red', 'blue', 'yellow', 'green', 'blue'], Moves: 7,
Time: 2.1228s, Memory Usage: 0.2852 MiB
```





Up to this point the analysis, thank you for your attention.