SSN College of Engineering

Department of Computer Science and Engineering

CS1504 — Artificial Intelligence

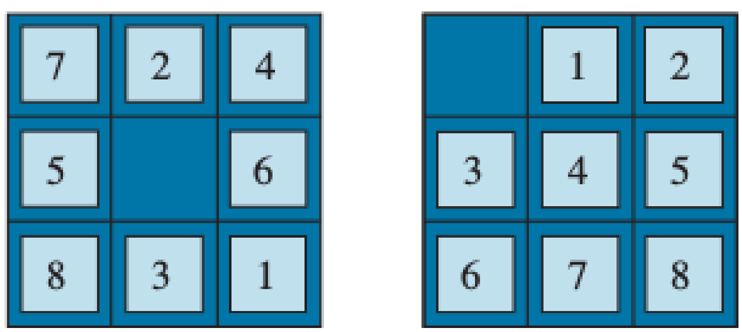
2021 – 2022

Assignment — 03
(State Space Search — Eight Queens Problem)

August 02, 2021

Problem Statement

In a 3x3 board, 8 of the squares are filled with integers 1 to 8, and one square is left empty. One *move* is sliding into the empty square the integer in any one of its adjacent squares.The start state is given on the left side of the figure and the goal state given on the right side. Find a sequence of moves to go from the start state to the goal state



Start State Goal State

1. Formulate the problem as a state-space search problem
2. Find a suitable representation for the states and the nodes
3. Solve the problem using any of the uninformed search strategies.

Responses

1. Formulated below (State Space Formulation)

2. States are represented as a nested tuple.
   a. The outer tuple contains three inner tuples, representing the three rows of the grid
   b. Each inner tuple contains three elements, representing the three blocs of each row
   c. The block numbers 1 to 8 are numbered the same. The empty block is represented as 0

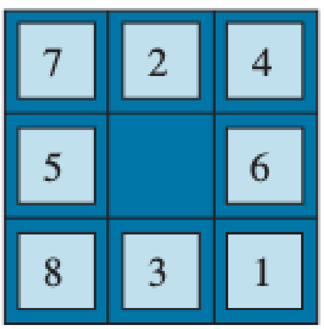
3. Two uniformed search approaches are adopted to solve the problem:

    a. Bidirectional-BFS: A forward search starting from the initial state and a reverse search starting from the goal state are implemented. The current state is compared with the fringe state of the other search is checked for equality to find the point of intersection between the two search trees

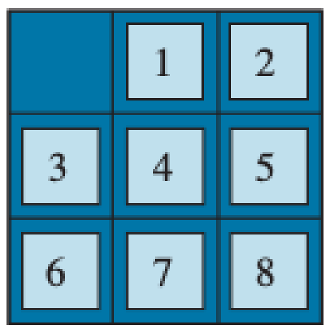
    b. Traditional-BFS: A single queue-driven BFS is implemented, starting from the initial state, until the goal state is reached

State Space Formulation

Initial State and Goal State



Start State



Goal State

Allowed actions to progress to Next State :

1. Interchange block-0 with its left block
2. Interchange block-0 with its right block
3. Interchange block-0 with its upper block
4. Interchange block-0 with its lower block

Constraint: Every block must have a distinct number between 0 and 8

Python Program Code

1. Queue.py - Script for Queue ADT

```python
class Queue:
   # [HEAD, ....... , TAIL]
   def __init__(self, data_list=None):
       self.data = list()
       self.size = 0
       if data_list is not None:
           self.data.extend(data_list)
           self.size = len(data_list)

   def enqueue(self, data_list):
       self.data.extend(data_list)
       self.size += len(data_list)

   def dequeue(self):
       if self.size == 0:
           return None
       self.size -= 1
       return self.data.pop(0)

   def get_contents(self):
       return self.data

   def is_empty(self):
       return self.size==0
```

2. StateFormulation.py - Functions to formulate the state and its functions

```python
INITIAL_STATE = (
   (7, 2, 4),
   (5, 0, 6),
   (8, 3, 1),
)
```

```python
GOAL_STATE = (
   (0, 1, 2),
   (3, 4, 5),
   (6, 7, 8),
)

NUM_ROWS = len(INITIAL_STATE)
NUM_COLS = len(INITIAL_STATE[0])


def get_next_states(state):
   # Swap 0 with any of its neighbors

   def locate_empty_space():
       for i in range(NUM_ROWS):
           for j in range(NUM_COLS):
               if state[i][j] == 0:
                   return i,j

   def make_state(initial, final):
       # By moving 0 from initial to final
       new_state = [list(row) for row in state]
       new_state[initial[0]][initial[1]] = state[final[0]][final[1]]
       new_state[final[0]][final[1]] = 0
       new_state = tuple(map(tuple, new_state))
       return new_state

   result = list()
   row, col = locate_empty_space()

   # Move left
   if (col-1)>-1:
       result.append(make_state((row, col), (row, col-1)))
   # Move right
   if (col+1)<NUM_COLS:
       result.append(make_state((row, col), (row, col+1)))

   # Move up
```

```
    if (row-1)>-1:
        result.append(make_state((row, col), (row-1, col)))
    # Move down
    if (row+1)<NUM_ROWS:
        result.append(make_state((row, col), (row+1, col)))

    return result

'''
# Checking the next state generation function
for state in get_next_states(INITIAL_STATE):
    for row in state:
        print(row)
    print()
'''

def goal_test(state):
    for i in range(NUM_ROWS):
        for j in range(NUM_COLS):
            if state[i][j] != GOAL_STATE[i][j]:
                return False
    return True


# Compare the current state of `this` direction search
# to the fringe states of `that` direction search
def intersection_test(this_state, that_fringe):

    def are_states_same(state_A, state_B):
        for i in range(NUM_ROWS):
            for j in range(NUM_COLS):
                if state_A[i][j] != state_B[i][j]:
                    return False
        return True

    for state in that_fringe:
        if are_states_same(state, this_state):
            return True
    return False
```

3. BFS.py - Functions to perform traditional BFS

```
from Queue import Queue

from StateFormulation import (
   get_next_states,
   intersection_test,
   INITIAL_STATE,
   GOAL_STATE,
   NUM_ROWS,
   NUM_COLS
)

def deduce_path(connecting_state, f_parents, r_parents):

   def deduce_path_rec(f_state, r_state, path_seq, f_depth,
r_depth):
       f_parent = f_parents[f_state] if f_state is not None else
None
       r_parent = r_parents[r_state] if r_state is not None else
None
       recurse = False

       if f_parent is not None:
           path_seq = [f_parent] + path_seq
           f_depth += 1
           recurse = True
       if r_parent is not None:
           path_seq = path_seq + [r_parent]
           r_depth += 1
           recurse = True

       if recurse:
           return deduce_path_rec(f_parent, r_parent, path_seq,
f_depth, r_depth)
       else:
           return path_seq, f_depth, r_depth

   return deduce_path_rec(connecting_state, connecting_state, [], 0,
0)
```

```
def search():
   f_state_space = Queue([INITIAL_STATE])
   f_explored_states = set()
   f_parents = {INITIAL_STATE: None}
   r_state_space = Queue([GOAL_STATE])
   r_explored_states = set()
   r_parents = {GOAL_STATE: None}

   while(not f_state_space.is_empty() or not
r_state_space.is_empty()):
       # Forward Search
       f_state = f_state_space.dequeue()
       if f_state not in f_explored_states:
           # Explore now
           f_explored_states.add(f_state)
           if intersection_test(f_state,
r_state_space.get_contents()):
               return f_explored_states, r_explored_states, f_state,
f_parents, r_parents
           fringe = get_next_states(f_state)
           f_state_space.enqueue(fringe)
           for new_state in fringe:
               if new_state not in f_parents:
                   f_parents[new_state] = f_state
       # Reverse Search
       r_state = r_state_space.dequeue()
       if r_state not in r_explored_states:
           # Explore now
           r_explored_states.add(r_state)
           if intersection_test(r_state,
f_state_space.get_contents()):
               return f_explored_states, r_explored_states, r_state,
f_parents, r_parents
           fringe = get_next_states(r_state)
           r_state_space.enqueue(fringe)
           for new_state in fringe:
               if new_state not in r_parents:
                   r_parents[new_state] = r_state
   return False
```

4. Bidirectional_BFS.py - Functions to perform bidirectional BFS

```
from Queue import Queue

from StateFormulation import (
   get_next_states,
   intersection_test,
   INITIAL_STATE,
   GOAL_STATE,
   NUM_ROWS,
   NUM_COLS
)


def deduce_path(connecting_state, f_parents, r_parents):

   def deduce_path_rec(f_state, r_state, path_seq, f_depth,
r_depth):
       f_parent = f_parents[f_state] if f_state is not None else
None
       r_parent = r_parents[r_state] if r_state is not None else
None
       recurse = False

       if f_parent is not None:
           path_seq = [f_parent] + path_seq
           f_depth += 1
           recurse = True
       if r_parent is not None:
           path_seq = path_seq + [r_parent]
           r_depth += 1
           recurse = True

       if recurse:
           return deduce_path_rec(f_parent, r_parent, path_seq,
f_depth, r_depth)
       else:
           return path_seq, f_depth, r_depth

   return deduce_path_rec(connecting_state, connecting_state, [], 0,
0)


def search():
```

```
   f_state_space = Queue([INITIAL_STATE])
   f_explored_states = set()
   f_parents = {INITIAL_STATE: None}

   r_state_space = Queue([GOAL_STATE])
   r_explored_states = set()
   r_parents = {GOAL_STATE: None}

   while(not f_state_space.is_empty() or not
r_state_space.is_empty()):
       # Forward Search
       f_state = f_state_space.dequeue()
       if f_state not in f_explored_states:
           # Explore now
           f_explored_states.add(f_state)
           if intersection_test(f_state,
r_state_space.get_contents()):
               return f_explored_states, r_explored_states, f_state,
f_parents, r_parents
           fringe = get_next_states(f_state)
           f_state_space.enqueue(fringe)
           for new_state in fringe:
               if new_state not in f_parents:
                   f_parents[new_state] = f_state

       # Reverse Search
       r_state = r_state_space.dequeue()
       if r_state not in r_explored_states:
           # Explore now
           r_explored_states.add(r_state)
           if intersection_test(r_state,
f_state_space.get_contents()):
               return f_explored_states, r_explored_states, r_state,
f_parents, r_parents
           fringe = get_next_states(r_state)
           r_state_space.enqueue(fringe)
           for new_state in fringe:
               if new_state not in r_parents:
                   r_parents[new_state] = r_state

   return False
```

5. main.py - Driver function for BFS

```
import time

import BFS as BFS
import Bidirectional_BFS as BiBFS
from StateFormulation import INITIAL_STATE, GOAL_STATE

line = "----------------------------"
runs = 10

print("\nINITIAL STATE")
for row in INITIAL_STATE:
   print(row)

print("\nGOAL STATE")
for row in INITIAL_STATE:
   print(row)


print("\n"+line)
print("Performing Biderictional BFS...")
total_time = 0
for i in range(runs):
   start_time = time.time()
   f_explored_states, r_explored_states, conn_state, f_parents,
r_parents = BiBFS.search()
   total_time += time.time() - start_time
   # Display the output after the first run
   # The rest of the runs are used to average the runnning-time
   if i==0:
       print("\nSearch Complete. Path obtained is:\n")
       goal_path, f_depth, r_depth = BiBFS.deduce_path(conn_state,
f_parents, r_parents)
       for state in goal_path:
           for row in state:
               print(row)
           print()
```

```
        print("\nCompleting {} runs of Bidrectional-BFS for run-time
averaging...".format(runs))
BiBFS_time = total_time/runs
print("\nNo. of Distinct States Explored: ",
len(f_explored_states.union(r_explored_states)))
print("Time Taken (avg. of {} runs): {} seconds".format(runs,
BiBFS_time))
print("Depth of Goal State: ", f_depth+r_depth+1)
print("Depth of Forward Search: ", f_depth)
print("Depth of Reverse Search: ", r_depth)
print("\n"+line)
print("Performing Traditional BFS...")
total_time = 0
for i in range(runs):
    start_time = time.time()
    explored_states, parents = BFS.search()
    total_time += time.time() - start_time
    # Display the output after the first run
    # The rest of the runs are used to average the runnning-time
    if i==0:
        print("\nSearch Complete. Path Obtained")
        goal_path = BFS.deduce_path(GOAL_STATE, parents)
        for state in goal_path:
            for row in state:
                print(row)
            print()
        print("\nCompleting {} runs of BFS for run-time
averaging...".format(runs))
BFS_time = total_time/runs
print("\nTime Taken (avg. of {} runs): {} seconds".format(runs,
BFS_time))
print("No. of Distinct States Explored: ", len(explored_states))
print("Depth of Goal State: ", len(goal_path)-1)
print("\n"+line)
print("Time taken by traditional BFS: {} seconds".format(BFS_time))

print("Time taken by bidirectional BFS: {}
seconds".format(BiBFS_time))
```

Output

```
INITIAL STATE
(7, 2, 4)
(5, 0, 6)
(8, 3, 1)

GOAL STATE
(7, 2, 4)
(5, 0, 6)
(8, 3, 1)

----------------------------
Performing Biderictional BFS...

Search Complete. Path obtained is:

(7, 2, 4)
(5, 0, 6)
(8, 3, 1)

(7, 2, 4)
(0, 5, 6)
(8, 3, 1)

(0, 2, 4)
(7, 5, 6)
(8, 3, 1)

(2, 0, 4)
(7, 5, 6)
(8, 3, 1)

(2, 5, 4)
(7, 0, 6)
(8, 3, 1)

(2, 5, 4)
(7, 6, 0)
(8, 3, 1)
```

```
(2, 5, 4)
(7, 6, 1)
(8, 3, 0)

(2, 5, 4)
(7, 6, 1)
(8, 0, 3)

(2, 5, 4)
(7, 6, 1)
(0, 8, 3)

(2, 5, 4)
(0, 6, 1)
(7, 8, 3)

(2, 5, 4)
(6, 0, 1)
(7, 8, 3)

(2, 5, 4)
(6, 1, 0)
(7, 8, 3)

(2, 5, 4)
(6, 1, 3)
(7, 8, 0)

(2, 5, 4)
(6, 1, 3)
(0, 7, 8)

(2, 5, 4)
(0, 1, 3)
(6, 7, 8)
```

```
(2, 5, 4)
(1, 3, 0)
(6, 7, 8)

(2, 5, 0)
(1, 3, 4)
(6, 7, 8)

(2, 0, 5)
(1, 3, 4)
(6, 7, 8)

(0, 2, 5)
(1, 3, 4)
(6, 7, 8)

(1, 2, 5)
(0, 3, 4)
(6, 7, 8)

(1, 2, 5)
(3, 0, 4)
(6, 7, 8)

(1, 2, 5)
(3, 4, 0)
(6, 7, 8)

(1, 2, 0)
(3, 4, 5)
(6, 7, 8)

(1, 0, 2)
(3, 4, 5)
(6, 7, 8)

(0, 1, 2)
(3, 4, 5)
(6, 7, 8)
```

```
Completing 2 runs of Bidirectional-BFS for run-time averaging...

No. of Distinct States Explored:  3679
Time Taken (avg. of 10 runs): 2 seconds
Depth of Goal State:  27
Depth of Forward Search:  13
Depth of Reverse Search:  13
```

```
----------------------------
Performing Traditional BFS...

Search Complete. Path Obtained
(7, 2, 4)
(5, 0, 6)
(8, 3, 1)

(7, 2, 4)
(0, 5, 6)
(8, 3, 1)

(0, 2, 4)
(7, 5, 6)
(8, 3, 1)

(2, 0, 4)
(7, 5, 6)
(8, 3, 1)

(2, 5, 4)
(7, 0, 6)
(8, 3, 1)

(2, 5, 4)
(7, 6, 0)
(8, 3, 1)

(2, 5, 4)
(7, 6, 1)
(8, 3, 0)
```

```
(2, 5, 4)
(7, 6, 1)
(8, 0, 3)

(2, 5, 4)
(7, 6, 1)
(0, 8, 3)

(2, 5, 4)
(0, 6, 1)
(7, 8, 3)

(2, 5, 4)
(6, 0, 1)
(7, 8, 3)

(2, 5, 4)
(6, 1, 0)
(7, 8, 3)

(2, 5, 4)
(6, 1, 3)
(7, 8, 0)

(2, 5, 4)
(6, 1, 3)
(7, 0, 8)
```

```
(2, 5, 4)
(6, 1, 3)
(0, 7, 8)

(2, 5, 4)
(0, 1, 3)
(6, 7, 8)

(2, 5, 4)
(1, 0, 3)
(6, 7, 8)

(2, 5, 4)
(1, 3, 0)
(6, 7, 8)

(2, 5, 0)
(1, 3, 4)
(6, 7, 8)

(2, 0, 5)
(1, 3, 4)
(6, 7, 8)

(0, 2, 5)
(1, 3, 4)
(6, 7, 8)
```

```
(1, 2, 5)
(0, 3, 4)
(6, 7, 8)

(1, 2, 5)
(3, 0, 4)
(6, 7, 8)

(1, 2, 5)
(3, 4, 0)
(6, 7, 8)

(1, 2, 0)
(3, 4, 5)
(6, 7, 8)

(1, 0, 2)
(3, 4, 5)
(6, 7, 8)
```

```
Completing 2 runs of BFS for run-time averaging...

Time Taken (avg. of 10 runs): 2 seconds
No. of Distinct States Explored:  164919
Depth of Goal State:  25

----------------------------
Time taken by traditional BFS: 12.256423354148865 seconds
Time taken by bidirectional BFS: 3.023608088493347 seconds
```