BLG 336E – ANALYSIS OF ALGORITHMS II

Project 1

Muhammed Burak Buğrul
150140015

Introduction

We are wanted to implement **DFS** and **BFS** algorithms for solving sliding blocks problem.

Environment

Project implemented in an **Ubuntu 16** machine with **C++**.

Structures

Node: Represents current state. Contains block information in a vector and a hash value for cycle detection.

- **Node::fill(bool ar[][MAXN]):** Fills ar according to block information.
- **Node::print()**: Prints blocks in right format
- **Node::getHash():** Calculates hash value of node. If it is calculated before(if it is not -1) returns that value. Hash function gives a number transformed from 107(prime number) base modulo 1000000007(prime also).

Block: Represents blocks. It just stores block information.

Code

In the code there are some explanations(comments).

BFS

Breadth First Search is an algorithm that uses a **queue** in order to reach nodes by their distances to start node(when wights of edges are 1 like in this project). Pseudo-code of BFS:

```
Queue.push(start)

while there is at least one element in queue:

      current ← next element in queue

      if current is target node:

            cnt ← 0 // This will be path length
            write ← create stack
            cur ← current.getHash()

            while until finish the solution path:

                  cnt++
                  write.push(visited[cur]) // next node, found by hash value

            print( all elemets of write stack )



      for each block in current:

            if block is vertical:
                  if neighbouring upper cell is empty:
                        create → new_state
                        if new_state.getHash() is not visited: // Cycle control
                              queue.push(new_state)
                              make → new_state_getHash() visited
                  if neighbouring down cell is empty:
                        create → new_state
                        if new_state.getHash() is not visited: // Cycle control
                              queue.push(new_state)
                              make → new_state_getHash() visited

            else:
                  if neighbouring right cell is empty:
                        create → new_state
                        if new_state.getHash() is not visited: // Cycle control
                              queue.push(new_state)
                              make → new_state_getHash() visited
                  if neighbouring left cell is empty:
                        create → new_state
                        if new_state.getHash() is not visited: // Cycle control
                              queue.push(new_state)
                              make → new_state_getHash() visited
```

DFS

Depth First Search is an algorithm that uses a **stack(in my code function calls represents stack)** in order to reach nodes. It doesn’t always give the shortest distance like BFS. Pseudo-code of BFS:

```
dfs(node):

      S.push(node) // S is a stack that holds current path
      visited_dfs.insert(node.getHash()) // To avoid cycles

      if node is target node:

            length ← size of S // This is path length
            write ← create stack

            while until finish the solution path:

                  write.push(S.top()) // parent node of last removed node
                  write.pop()

            print( all elemets of write stack )


      for each block in node:

            if block is vertical:
                  if neighbouring upper cell is empty:
                        create → new_state
                        if new_state.getHash() is not visited // Cycle control
                           and dfs(new_state):
                              return true // We found a way
                  if neighbouring down cell is empty:
                        create → new_state
                        if new_state.getHash() is not visited // Cycle control
                           and dfs(new_state):
                              return true // We found a way

            else:
                  if neighbouring right cell is empty:
                        create → new_state
                        if new_state.getHash() is not visited // Cycle control
                           and dfs(new_state):
                              return true // We found a way
                  if neighbouring left cell is empty:
                        create → new_state
                        if new_state.getHash() is not visited // Cycle control
                           and dfs(new_state):
                              return true // We found a way

      S.pop() // removing current node in order to keep the correctly
      return false // We tried everythong and couldn’t find a way
```

Analysis

For the sake of simplicity, let N be the number of nodes(a state of table) and M be the number of edges(state transitions).

BFS: Time complexity of BFS is **O(N+MlgM),** because every node goes into queue at most once(**O(N)**). And every edge is to be tried by program in order to create a new state(**O(M)**). In addition, cycle detection costs **O(lgM)** (find function of STL set).So total complexity is **O(N+M).** According to pseudo-code time complexity of while loop is **O(N)** and in the while loop there are state transition operations. Sum of every state transition operations is **O(M).** So we have **O(N+M)** again.

DFS: Time complexity of DFS is **O(N+M),** because every node calls with dfs(node) function at most once(**O(N)**). And every edge is to be tried by program in order to create a new state(**O(M)**). In addition, cycle detection costs **O(lgM)** (find function of STL set) So total complexity is **O(N+MlgM).** According to pseudo-code, time complexity of dfs(node) calls are **O(N)** and in the function there are state transition operations. Sum of every state transition operations is **O(MlgM).** So we have **O(N+MlgM)** again.

Cycle Searches: Checking hash values in visited set costs **O(lgM)** time(STL set). There is no need to go on a node that visited before.

Adjacency List Representation: Complexity of this case depends on implementation. Trying to find every valid node and querying every pair if they have edge between them will be **O(N^2)** because of calculation of all node pairs. But picking every node individually and finding its’ neightbours is **O(N+M)**, it is like the situation above. We are trying transition operations to every node. There are N nodes and M edges(transitions). Plus there will be complexity of DFS-BFS.