Artificial Intelligence :
1st Optional Task 2022

Yfantidis Dimitris, AEM: 3938

1. Introduction

This work was implemented in the C++ language, in the C++20 standard. All the necessary code files for compilation and production of the executable are located in the "Code" folder.

The implementation used several modern STL structures for ease of writing and understanding code, as well as older C features to save space and time. The performance and execution metrics of the algorithm, as well as the visual representation of the puzzle solution are recorded, without any unexpectedness, in a "results.txt" file.

In the following technical report, the modelling of the given problem will be briefly and progressively reported. First, the modeling of the colored liquids, then the containers, then the states, then the BFS algorithm and main will be reported. In each paragraph, the relevant files will also be mentioned.

Note that in case there may be some ambiguity, there are several more detailed comments in the code files that provide minor details of the implementation.

2. The coloured liquids

The maximum number of bottles is 19 (pronunciation: N < 20), so the maximum number of different colours is 17 ($N_{colours} = N_{bottles} - 2$). Each color corresponds to an integer of data type char, from 0 (black) to 16 (yellow). Absence in a layer of the container is represented by the value -1 (macro: NO_COLOR). All colors are defined (and absence of color) are defined as macros in the header file "colors.h", along with the

string array COLOR_STR, where COLOR_STR[**i**] returns the color coded **i** as a string.

3. The containers

Each container is represented by an object of the **Bottle** class, which is declared and implemented in the files "Bottle.h" and "Bottle.cpp" respectively. Each container consists of a static table of 4 characters (**contents[4]**). This is where the colored liquids are stored. The class contains methods such as:

- **bool hasFreeSpace():** returns whether it is filled or not
- **bool isEmpty():** returns whether it is empty or not
- **bool isComplete():** returns true if it is filled to the top with the same liquid or empty.
- **bool shouldPourTo(const Bottle &):** Returns whether or not it is possible for the bottle to pour liquid into the other one. The bottle can pour k mL of continuous liquid if the other one is empty or has the same color on top and has k mL of free space.
- **color_t top():** returns the color of the liquid in the first layer, the corresponding overload "top(int &)" returns with reference to the free space. *(color_t = char using typedef)*
- **color_t pour(Bottle &):** If done, pours the liquid of its top into the referenced bottle and returns the color exchanged, otherwise returns NO_COLOR and nothing happens.
- **color_t getColor(size_t i):** returns contents[i].

+ Operators (==, =, [])

The above class and its functions contribute to the ease of reading and writing the code to model the puzzle states.

4. The situations

Each snapshot of the puzzle is represented by an object of the **State** class of the "State.h" and "State.cpp" files. Each state consists of a pointer to a container table, the number of bottles, the verbal description of the transition from the previous state to the current state, and a pointer to the previous state. Important methods of the State class are:

- **void init():** initializes the first N - 2 bottles of the initial state with N - 2 random liquids (4 mL each) in random order. The random selections are made using the Mersenne Twister generator (std::mt19937 of the <random> library).
- **hash_t hashValue():** returns the hash value of the object. Its calculation mechanism was done according to the pattern of overloading the hashCode() function of Java objects. *(hash_t = long long with typedef).*
- **std::string toString():** returns a visual representation of a problem state, e.g:

Figure 1

Figure 2

- **bool isVictorious():** returns true if the state is such that the puzzle is completed, otherwise false. This function is the termination condition of the BFS.
- **int getDepth():** returns the depth of the instance in the state tree.

- **color_t pour(State &n, int i, int j):** checks if liquid can be poured from the i-th container to the j-th container. If so, a new state identical to the current state where container **i** has poured into container **j is** created and stored in **n**, inserted into the state tree, and the color of the poured liquid is returned. Otherwise nothing is done and NO_COLOR is returned.
  This function is the transition operator between states.

- **std::vector<State *> *expand():** returns the child states, i.e. any state where the transition operator can be successfully applied from any bottle to any other bottle.

5. The BFS and the main programme

The main program (**int main() { ... }**) is located together with the implementation function of the breadth-first search (BFS) algorithm in the file "main.cpp".

The structures used to implement the BFS were:

std::queue<State *> frontier for the search front (queue)

std::unordered_map<hash_t, State *> closed for the closed set as a hashmap.

The initial state is passed to the BFS and inserted into the search front. Also passed by reference are the variables **examined** and **memory** where the number of states examined and the number of total states are stored respectively. The implementation is shown on the next page.

```
State *BFS(8tate& initial, unsigned long &examined, unsigned long &memory)

    std:::queue<State+> fnonntien;
    std:::unordered_nap<hash_t,State+> clo se
    d;std::vector<State+> Fchildneen;std::vect
    or<State+> Fchildnen;

    frontier.push(&initial);
    examined = 0;
    memory = l;

    while (!frontier.empty()))

        if   (fnontieo.size()+ctosed.size()>memoy) I
             memoy = fnontieo.size()+ctosed.size();

        State *s = frontier.front();

        frontier.pop();

        if (closed.count(s->hashVaIue()) == 0)

            examined += 1;

            if (s->isVictorious())
                // Lori s*r*.°
                return s;          reached.

            closed.insent(/s->hash Value(),s});;

            child              s->expand();;

            for (const auto& child          : Fchildnen)

                if   (clo sed.count(child->ha sh Value()) == 0)
                     fnontien.pu sh (child);


            delete childnen;;


    return nullptr;
```

Regarding the main programme:

- The **BOTTLES_N** macro specifies the number of bottles. In the deliverable it is set to 8, for another number of bottles the program must be redefined and recompiled.
- The variables **memory** and **examined** are declared.
- A file "results.txt" is created and opened where the metrics and the visual representation of the states and transitions of the solution will be printed. (In case something goes wrong with the file, the results are printed to the console).
- Some additional objects are declared to record the start and end timestamps of the algorithm, along with the **duration** variable that stores the execution time of the BFS algorithm in milliseconds. This is an extra performance metric of the algorithm.

6. Results

The results vary from run to run for a fixed number of containers. The results are recorded below 3
of different runs for N = 6, N = 8, N = 10 and N = 11.

For 6 containers:

Average Space: negligible

Average Time: ½ second

```
METRICS FOR 6 BOTTLES:
-> Depth:           9
-> Total Nodes:     5233
-> Examined Nodes:  2353
-> Elapsed Time:    00h : 00m : 00s : 025ms
METRICS FOR 6 BOTTLES:
-> Depth:           13
-> Total Nodes:     11621
-> Examined Nodes:  5661
-> Elapsed Time:    00h : 00m : 00s : 062ms
```

```
METRICS FOR 6 BOTTLES:
-> Depth:           9
-> Total Nodes:     5233
-> Examined Nodes:  2353
-> Elapsed Time:    00h : 00m : 00s : 025ms
```

For 8 containers:

Average Space (based on task management): ~100 to 300 MB

Average Time: ~1 to 2 seconds

```
METRICS FOR 8 BOTTLES:
-> Depth:           15
-> Total Nodes:     200878
-> Examined Nodes:  90601
-> Elapsed Time:    00h : 00m : 01s : 573ms
```

```
METRICS FOR 8 BOTTLES:
-> Depth:           16
-> Total Nodes:     237982
-> Examined Nodes:  93267
-> Elapsed Time:    00h : 00m : 01s : 630ms
```

```
METRICS FOR 8 BOTTLES:
-> Depth:           17
-> Total Nodes:     308356
-> Examined Nodes:  127079
-> Elapsed Time:    00h : 00m : 02s : 182ms
```

For 10 containers:

Average Space (based on task management): ~3.5 to 5 GB of memory Average Time: ~1 to 1.5 minutes

```
METRICS FOR 10 BOTTLES:
-> Depth:           20
-> Total Nodes:     2184986
-> Examined Nodes:  687382
-> Elapsed Time:    00h : 00m : 18s : 191ms
```

```
METRICS FOR 10 BOTTLES:
-> Depth:           23
-> Total Nodes:     9350294
-> Examined Nodes:  3202919
-> Elapsed Time:    00h : 01m : 23s : 650ms
```

```
METRICS FOR 10 BOTTLES:
-> Depth:           25
-> Total Nodes:     12268311
-> Examined Nodes:  5015040
-> Elapsed Time:    00h : 02m : 08s : 291ms
```

For 11 containers:

Average Space (based on task management and SSD capacity reporting): ~28 GB of memory (12 GB peak RAM + 16 on SSD)

Average Time: ~12 to 15 minutes

```
METRICS FOR 11 BOTTLES:
-> Depth:           25
-> Memory:          67066607
-> Examined:        22334518
-> Elapsed Time:    00h : 12m : 16s : 100ms
```

For more containers:

The time difference grows a lot from 8 bottles to 10 and from 10 to 11. Given these metrics, it would be very difficult to find a solution for N = 12 and practically impossible in terms of time and memory for N > 14. However the BFS algorithm is blind therefore other AI algorithms perform better. Moreover for more containers, there may be more conflicts with the hash values of the states.