**CS 461 – Artificial Intelligence**

**Homework #1**

This homework is done by a group of two people

Serkan Demirci – 21201619  
Batuhan Tüter – 21200624

Java Programming language is used.

**Listings**

**Note:** All codes are available at<https://github.com/tuterbatuhan/RiverProblem/>

Solver.java

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.Map.Entry;

**import** java.util.TreeMap;

**public** **class** Solver

{

// Missionaries initially in the left bank of the river

**public** **static** **final** **int** ***MAX\_MISSIONARY*** = 3;

// Cannibals initially in the left bank of the river

**public** **static** **final** **int** ***MAX\_CANNIBAL*** = 3;

// Size of the boat to cross the river

**public** **static** **final** **int** ***BOAT\_SIZE*** = 2;

**public** **static** **void** main(String[] args)

{

// Define start and goal states

State start = **new** State(***MAX\_MISSIONARY***, ***MAX\_CANNIBAL***, State.Boat.***LEFT***);

State goal = **new** State(0, 0, State.Boat.***RIGHT***);

// Creating a tree to calculate the number and the frequencies of the

// different solutions.

TreeMap<String, Integer> map = **new** TreeMap<>();

// Solve the program N times.

**final** **int** N = 1000;

**for** (**int** i = 0; i < N; i++)

{

Path p = *solve*(start, goal);// Creates the solution path

String str = p.toString();// Converts the solution path into string

**if** (map.containsKey(str))// If the path already found before, update

// the tree

map.put(str, map.get(str) + 1);

**else**// Add the new solution to the solution tree.

map.put(str, 1);

}

// Print solutions.

**int** solutionCount = 1;

**for** (Entry<String, Integer> s : map.entrySet())

{

System.***out***.println("---------------------------------------------------------");

System.***out***.println("Solution:" + solutionCount++);

System.***out***.println("Frequency: " + s.getValue() + "\n");

System.***out***.println(s.getKey());

}

}

**public** **static** Path solve(State start, State goal)

{

// An arraylist consisting of the paths

List<Path> agenda = **new** ArrayList<>();

agenda.add(**new** Path(start));

// An arraylist consisting of the travelled states

List<State> extendedList = **new** ArrayList<>();

// Pseudo code provided by Prof. Bob. Berwick.

**while** (!agenda.isEmpty())

{

Path currPath = agenda.remove(0);// Pop the first element from agenda

**if** (currPath.getLast().equal(goal))// Solution is found

**return** currPath;

// Solution not found

**else**

{

**if** (!extendedList.contains(currPath.getLast()))// New state is not visited before

{

// Add state to visited list

extendedList.add(currPath.getLast());

// All valid moves from last state

List<State> neighbours = *getNeighbours*(currPath.getLast());

// Add new paths to the agenda in random places

**for** (State s : neighbours)

agenda.add((**int**) (Math.*random*() \* (agenda.size() + 1)), **new** Path(currPath, s));

}

}

}

**return** **null**;

}

// Finds the reachable states from the last state of a path.

**private** **static** List<State> getNeighbours(State last)

{

List<State> result = **new** ArrayList<>();// Result list consisting of the next states

// Checks all different possibilities of crossing the river with the

// given size of a boat.

**for** (**int** passengerCount = 1; passengerCount <= ***BOAT\_SIZE***; passengerCount++)

{

**for** (**int** missionaryCount = 0; missionaryCount <= passengerCount; missionaryCount++)

{

// Checks if the next state is applicable

State s = *transport*(last, missionaryCount, passengerCount - missionaryCount);

**if** (s != **null**)

result.add(s);// Add next states to the result list.

}

}

**return** result;

}

// Checks if the given state is desired after it is ended

**public** **static** State transport(State s, **int** miss, **int** cann)

{

State.Boat position = s.boat;

**int** cannibals = s.cannibals;

**int** missionaries = s.missionaires;

// Checks if the given state is valid

**if** (miss + cann > ***BOAT\_SIZE*** || miss + cann <= 0)

**return** **null**;

// Concludes the state according to the boat position.

**if** (position == State.Boat.***LEFT***)

{

cannibals -= cann;

missionaries -= miss;

position = State.Boat.***RIGHT***;

} **else**

{

cannibals += cann;

missionaries += miss;

position = State.Boat.***LEFT***;

}

// Checks the states last status if it is desired.

// Assumption: MAX\_MISSIONAR = MAX\_CANNIBAL && BOAT\_SIZE <=2

**if** (s.missionaires == 0 || s.missionaires == ***MAX\_MISSIONARY*** || s.missionaires == s.cannibals)

{

**return** **new** State(missionaries, cannibals, position);

} **else**

**return** **null**;

}

}

Path.java

**import** java.util.ArrayList;

**import** java.util.List;

//Path is a list consisting of the states inside the state-space.

**public** **class** Path

{

**public** List<State> states;

// Default Constructors

**public** Path()

{

states = **new** ArrayList<>();

}

// Creates a path with initial starting state

**public** Path(State state)

{

states = **new** ArrayList<>();

states.add(state);

}

// Creates a new path, extending the given path with the next state.

**public** Path(Path copy, State nextState)

{

states = **new** ArrayList<>(copy.states);

states.add(nextState);

}

// Returns the last state of the path

**public** State getLast()

{

**return** states.get(states.size() - 1);

}

**public** String toString()

{

StringBuilder builder = **new** StringBuilder();

builder.append("Missionary\tCannibal\tBoat Position\n");

**for** (State s : states)

{

builder.append(s);

builder.append('\n');

}

**return** builder.toString();

}

}

State.java

//States represent the nodes of the state-space tree

**public** **class** State

{

**public** **enum** Boat

{

***LEFT***, ***RIGHT***

};

**public** **int** missionaires = 0;// Number of missionaries on the left bank of

// the river

**public** **int** cannibals = 0;// Number of cannibals on the left bank of the

// river

**public** Boat boat = Boat.***LEFT***;// Bank side of the boat

**public** State(**int** missionaires, **int** cannibals, Boat boat)

{

**super**();

**this**.missionaires = missionaires;

**this**.cannibals = cannibals;

**this**.boat = boat;

}

**public** **boolean** equal(State st)

{

**return** (st.missionaires == **this**.missionaires && st.cannibals == **this**.cannibals && st.boat == **this**.boat);

}

@Override

**public** **boolean** equals(Object O)

{

**return** O **instanceof** State && equal((State) O);

}

**public** String toString()

{

**int** remMiss = Solver.***MAX\_MISSIONARY***-missionaires;

**int** remCann = Solver.***MAX\_CANNIBAL***-cannibals;

**return** "|"+missionaires + "\t\t" + cannibals + "\t|\t" + boat.name()+"\t\t|" + remMiss + "\t\t" + remCann+"\t|";

}

}

**All Solutions with tabulation**

When the program is run, solution is printed to the console:

---------------------------------------------------------

Solution:1

Frequency: 252

Left Right

Missionaries Cannibals Boat Position Missionaries Cannibals

3 3 LEFT 0 0

2 2 RIGHT 1 1

3 2 LEFT 0 1

3 0 RIGHT 0 3

3 1 LEFT 0 2

1 1 RIGHT 2 2

2 2 LEFT 1 1

0 2 RIGHT 3 1

0 3 LEFT 3 0

0 1 RIGHT 3 2

0 2 LEFT 3 1

0 0 RIGHT 3 3

---------------------------------------------------------

Solution:2

Frequency: 255

Left Right

Missionaries Cannibals Boat Position Missionaries Cannibals

3 3 LEFT 0 0

2 2 RIGHT 1 1

3 2 LEFT 0 1

3 0 RIGHT 0 3

3 1 LEFT 0 2

1 1 RIGHT 2 2

2 2 LEFT 1 1

0 2 RIGHT 3 1

0 3 LEFT 3 0

0 1 RIGHT 3 2

1 1 LEFT 2 2

0 0 RIGHT 3 3

---------------------------------------------------------

Solution:3

Frequency: 245

Left Right

Missionaries Cannibals Boat Position Missionaries Cannibals

3 3 LEFT 0 0

3 1 RIGHT 0 2

3 2 LEFT 0 1

3 0 RIGHT 0 3

3 1 LEFT 0 2

1 1 RIGHT 2 2

2 2 LEFT 1 1

0 2 RIGHT 3 1

0 3 LEFT 3 0

0 1 RIGHT 3 2

0 2 LEFT 3 1

0 0 RIGHT 3 3

---------------------------------------------------------

Solution:4

Frequency: 248

Left Right

Missionaries Cannibals Boat Position Missionaries Cannibals

3 3 LEFT 0 0

3 1 RIGHT 0 2

3 2 LEFT 0 1

3 0 RIGHT 0 3

3 1 LEFT 0 2

1 1 RIGHT 2 2

2 2 LEFT 1 1

0 2 RIGHT 3 1

0 3 LEFT 3 0

0 1 RIGHT 3 2

1 1 LEFT 2 2

0 0 RIGHT 3 3