CS 461 Homework 2

Serkan Demirci - 21201619

Batuhan Tüter - 21200624

Section 1

We used Java programming to implement for solving 8 puzzle problem.

*Code is available at:

https://github.com/tuterbatuhan/EightPuzzle/

We tested our program for 4 different beam width (W) values in order of W = 1,2,3,4. We applied these test cases into 2 different implementation of beam search, since it was not stated clearly that which implementation is required from us:

1. When our algorithm finds a previously visited state, it discards that state and it does not include that state into count of beam width (W).

   Our results for 1000 board configurations:

	W = 1	W = 2	W = 3	W = 4
# of solution found	155	1000	1000	1000

2. When our algorithm finds a previously visited state, it discards that state but it includes that state into count of beam width (W).

   Our results for 1000 board configurations:

	W = 1	W = 2	W = 3	W = 4
# of solution found	1	991	1000	1000

An example output taken from our program:

```
1000 distinct board configuration will be tested using beam sort (w = 1):
Example Solution :
Inital Configuration of the puzzle:


---------
|1||3||6|
---------
| ||2||8|
---------
|4||7||5|
---------

Move 1 : DOWN

---------
|1||3||6|
---------
|4||2||8|
---------
| ||7||5|
---------

Move 2 : RIGHT

---------
|1||3||6|
---------
|4||2||8|
---------
|7|| ||5|
---------

Move 3 : RIGHT

---------
|1||3||6|
---------
|4||2||8|
---------
|7||5|| |
---------

Move 4 : UP

---------
|1||3||6|
---------
|4||2|| |
---------
|7||5||8|
---------

Move 5 : UP

---------
|1||3|| |
---------
|4||2||6|
```

```
---------
|7||5||8|
---------
```

Move 6 : LEFT

```
---------
|1||  ||3|
---------
|4||2||6|
---------
|7||5||8|
---------
```

Move 7 : DOWN

```
---------
|1||2||3|
---------
|4||  ||6|
---------
|7||5||8|
---------
```

Move 8 : DOWN

```
---------
|1||2||3|
---------
|4||5||6|
---------
|7||  ||8|
---------
```

Move 9 : RIGHT

```
---------
|1||2||3|
---------
|4||5||6|
---------
|7||8||  |
---------
```

Successfully solved 1 of 1000 puzzles

CODES:

EightPuzzle.java

```
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;
import java.util.Set;

/**
 * Driver class that tests beam search on the Eight Puzzle problem
 *
 */
public class EightPuzzle
{
	public static void main(String[] args)
	{
		final int TEST_SIZE = 1000; //# of tests
		final int W = 2;			//Beam size

		System.out.println(TEST_SIZE + " distinct board configuration will be tested"
				+ " using beam sort (w = " + W + "): ");


		//Create a new state factory with given
		StateFactory factory = new StateFactory();

		//Create startStates and goal state using factory
		Set<State> set = factory.getRandomStates(TEST_SIZE);
		State goalState = factory.getSolvedState();

		int solvedCount=0; // # of solved puzzles
		Path solutionPath;
		boolean solutionPrinted = false;
		for(State s : set)
		{
			solutionPath = beamSearch(s, goalState, W);
			if(solutionPath != null)//If solved
				solvedCount++;

			if (!solutionPrinted && solutionPath!=null)
			{
				System.out.println("Example Solution : ");
				System.out.println(solutionPath);
				solutionPrinted = true;
			}
		}

		System.out.println("Successfully solved " + solvedCount  + " of " + TEST_SIZE + "
puzzles");
	}

	/**
	 * Performs beam search with beam width w to find a path from start state and goal state
	 * It might not be able to find any path when the width of the beam is small or heuristic
	 *   is not guiding the goal state or there is no path from start state to goal state.
	 * If solution cannot be found it returns null, otherwise it returns the path that starts
from startState
	 * and lead to the goal state.
	 *
	 * @param startState Starting state for the search.
	 * @param goalState Goal state of the search
	 * @param w Width of the beam
	 * @return Path if solution found, otherwise null.
	 */
	public static Path beamSearch(State startState, State goalState, int w){
		//Hash set is used to check whether any state is already visited
		//Eliminates cycles
		Set<State> visitedStates = new HashSet<>();

		//Agenda holds paths that is visited.
		//It is used to determine next states that will be searched.
		Queue<Path> agenda = new LinkedList<>();

		agenda.offer(new Path(startState));//Enqueue start state (standard search starting)
```

```
		//Search loop. Runs until a solution is found or no solution is found.
		while(!agenda.isEmpty())
		{
			Path curPath = agenda.poll();
			State curState = curPath.getLastState();

			//Gets the possible moves from current state
			//Returned neighbour states are sorted according to their heuristic scores.
			List<State> neighbours = curState.getNeighbours();

			int remainingBests = w;//Stores how many beam is added in this iteration

			for (State s : neighbours)
			{
				//If all of the beams is added exit the for loop
				if(remainingBests==0)
					break;

				if(!visitedStates.contains(s))//If not already visited
					if(s.equals(goalState))
						return new Path(curPath,s); // Solution found
					else
					{
						agenda.offer(new Path(curPath,s));
						remainingBests--; //Implementation 1
					}
		//			remainingBests--; //Implementation 2

			}//for all neighbours

			// Add last visited state into visitedStates
			visitedStates.add(curState);

		}//while agenda is not empty

		return null;//Solution is not found
	}

}
```

Path.java

```
import java.util.LinkedList;
import java.util.List;
import java.util.Stack;

/**
 * Describes a path of states that is found by valid moves.
 *
 *
 */
public class Path {
	//Path is an immutable stack data structure

	private Path tail; //Tail stores rest of the path
	private State head; //Last state in path

	/**
	 * Constructs a new Path by adding a state to the end of the given path
	 * @param p Path that will be used
	 * @param s State that will be added at the end of the path
	 */
	public Path(Path p,State s){
		tail=p;
		head=s;
	}

	/**
	 * Creates a new path with initial state
	 * @param s Initial state
	 */
	public Path(State s)
	{
		this(null,s);
	}
```

```
	/**
	 * Returns last state added in the path
	 */
	public State getLastState(){
		return head;
	}

	/**
	 * Returns the list of States in the Path.
	 * First item of the list is the first added State.
	 * Last item is the last state added.
	 * @return
	 */
	public List<State> toList()
	{
		//Since items are ordered reverse in stack, A reversal is needed.
		List<State> list = new LinkedList<>();
		Stack<State> stack = new Stack<>();

		Path cur = this;
		while(cur!=null)
		{
			stack.push(cur.head);
			cur=cur.tail;
		}

		while(!stack.isEmpty())
			list.add(stack.pop());

		return list;
	}

	/**
	 * Returns a string that describes path.
	 */
	public String toString(){
		Stack<State> stack = new Stack<>();
		Path cur = this;
		while(cur!=null)
		{
			stack.push(cur.head);
			cur=cur.tail;
		}

		System.out.println("Inital Configuration of the puzzle:");

		StringBuilder builder = new StringBuilder();

		int moveCounter = 0;
		while(!stack.isEmpty())
		{
			if (moveCounter != 0)
				builder.append("\nMove " + moveCounter + " : ");

			appendMove(builder, stack.pop());
			moveCounter++;
		}

		return builder.toString();
	}

	/**
	 * Utility function for toString method of the Path
	 * @param builder StringBuilder that move will be recorded
	 * @param state Current State that will be recorded
	 */
	private static void appendMove(StringBuilder builder,State state)
	{
		State.MoveType lastMove = state.getLastMove();
		String move = "";
		if (lastMove != null)
			move = lastMove.name();

		builder.append(move + "\n");
		builder.append(state.toString());
	}
}
```

State.java

```
import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedList;
import java.util.List;

/**
 * Represents a configuration of the 8 puzzle
 *
 *
 */
public class State {
	public static final byte EMPTY = 9; // Id of the empty puzzle piece
	byte emptyIndice = 0; //stores place of the empty puzzle piece
	private byte [] board = new byte [9];//Stores id's of the pieces in board

	private int score = -1; //Heuristic value that shows how does it close to the solution
	private int hash = -1; //Hash value of the state

	public enum MoveType {UP,DOWN,LEFT,RIGHT}; // Possible moves of the empty puzzle piece

	private MoveType lastMove;    //Shows the last move that generated this state.
                                                        //lastMove is not considered when
checking whether 2 state is
                                                        //equal.

	/**
	 * Initializes a state that is solved.
	 */
	public State()
	{
		for (byte i=0; i<9;i++){
			board[i] = (byte) (i+1);
		}
		emptyIndice = 8;
		this.lastMove = null;
	}

	/**
	 * Creates a new state that is clone of given state
	 * @param state State to be copied
	 */
	public State(State state) {
		for (byte i=0; i<9;i++){
			board[i] = state.board[i];
		}
		emptyIndice = state.emptyIndice;
		this.lastMove = state.lastMove;
	}

	/**
	 * Initializes a state whose board configuration is given as board
	 * @param board Configuration of board
	 */
	public State(byte[] board) {
		this.score = 0;
		for (byte i=0; i<9;i++)
		{
			this.board[i] = board[i];
			if (board[i]==EMPTY)
				emptyIndice = i;
		}
		this.lastMove = null;
	}

	/**
	 * Utility method that generates a new state created from a move.
	 * NOTE: This method does not check whether move is valid.
	 * @param type Type of the move
	 * @return New State generated from current state
	 */
	private State move(MoveType type){
		State s = new State(this);
		switch(type){
		case UP:
			s.emptyIndice-=3;
			break;
```

```
        case DOWN:
                s.emptyIndice+=3;
                break;
        case LEFT:
                s.emptyIndice-=1;
                break;
        case RIGHT:
                s.emptyIndice+=1;
                break;
        }

        s.lastMove = type;
        s.board[this.emptyIndice]=this.board[s.emptyIndice];
        s.board[s.emptyIndice]=this.board[this.emptyIndice];
        return s;
}

/**
 * Utility method for sorting a list of states with respect to their heuristic scores.
 * @param list List of State to be sorted.
 */
private static void sort(List<State> list){
        Collections.sort(list,new Comparator<State>() {

                @Override
                public int compare(State s1, State s2) {
                        return s1.getScore() - s2.getScore();
                }
        });
}

/**
 * Returns the States that can be generated from this state using valid moves.
 *
 * @return List of States whose elements are ordered with respect to heuristic score
 */
public List<State> getNeighbours(){
        List <State> list = new LinkedList<State>();

        switch(emptyIndice){
        case 0:
                list.add(this.move(MoveType.RIGHT));
                list.add(this.move(MoveType.DOWN));
                break;
        case 1:
                list.add(this.move(MoveType.LEFT));
                list.add(this.move(MoveType.RIGHT));
                list.add(this.move(MoveType.DOWN));
                break;
        case 2:
                list.add(this.move(MoveType.LEFT));
                list.add(this.move(MoveType.DOWN));
                break;
        case 3:
                list.add(this.move(MoveType.UP));
                list.add(this.move(MoveType.RIGHT));
                list.add(this.move(MoveType.DOWN));
                break;
        case 4:
                list.add(this.move(MoveType.UP));
                list.add(this.move(MoveType.LEFT));
                list.add(this.move(MoveType.RIGHT));
                list.add(this.move(MoveType.DOWN));
                break;
        case 5:
                list.add(this.move(MoveType.UP));
                list.add(this.move(MoveType.LEFT));
                list.add(this.move(MoveType.DOWN));
                break;
        case 6:
                list.add(this.move(MoveType.UP));
                list.add(this.move(MoveType.RIGHT));
                break;
        case 7:
                list.add(this.move(MoveType.UP));
                list.add(this.move(MoveType.RIGHT));
                list.add(this.move(MoveType.LEFT));
                break;
        case 8:
                list.add(this.move(MoveType.UP));
```

```
                list.add(this.move(MoveType.LEFT));
                break;
        }
        sort(list);//Sort state wrt their heuristic score
        return list;
}

/**
 * Calculates and returns the heuristic score of the state
 * @return An integer value that is obtained from heuristic.
 */
private int getScore()
{

        if (score == -1) // If method called first time
        {
                //Calculate score
                score = 0;

                for (int i = 0 ; i < board.length ; i++)
                        if (board[i] != i + 1)
                                score += 1;

        }

        return score;
}

/**
 * Compares the state with an object.
 * @return True if they are equivalent
 */
@Override
public boolean equals(Object o){
        if(o instanceof State)
                return equals((State)o);
        else
                return false;
}

/**
 * Compares 2 states.
 * It compares hash values of the states since they are unique
 * @param s2 other state
 * @return True if states are same
 */
public boolean equals(State s2)
{
        return this.hashCode() == s2.hashCode();
}

/**
 * Calculates a unique hash for state.
 * Hash code is unique for every configuration
 *
 * @return Hash value of the state
 */
@Override
public int hashCode()
{
        if (hash == -1) // If method called first time
        {
                //Calculate hash
                hash = 0;
                for (int i = 0 ; i < board.length ; i++)
                        hash = hash*10 + board[i];
        }
        return hash;
}

/**
 * Returns the last move that is done to the previous state to get this state
 * @return Last move. If the state does not have an ancestor State it will return null
 */
public MoveType getLastMove()
{
        return lastMove;
}
```

```
	/**
	 * Returns a string that describes state
	 */
	public String toString(){
		StringBuilder builder = new StringBuilder();
		builder.append("\n---------\n");

		for(int i=0;i<this.board.length;i++){
			builder.append('|');
			builder.append(this.board[i] == EMPTY ? " ":this.board[i] );
			builder.append('|');
			if(i%3==2)
				builder.append("\n---------\n");
		}
		return builder.toString();
	}

}
```

StateFactory.java

```
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Random;
import java.util.Set;

/**
 * State factory class is used to generate different states for 8 puzzle problem
 *
 */
public class StateFactory
{
	private final Random rng;//Random Number Generator

	/**
	 * Initializes StateFactory class
	 *
	 * @param seed is the starting seed of the random number generator
	 *	that is used for random operations off this class.
	 *
	 */
	public StateFactory(long seed)
	{
		rng = new Random(seed);
	}

	/**
	 * Initializes StateFactory class
	 * Randomly selects a seed for the random number generator
	 *
	 */
	public StateFactory()
	{
		rng = new Random();
	}

	/**
	 * When called, it creates a new solvable random State(board) for 8 puzzle problem.
	 * @return Solvable state.
	 */
	public State getRandomState(){

		byte[] board = {1,2,3,4,5,6,7,8,9};
		do
			shuffle(board);//Like a boss
		while(!isSolvable(board));

		return new State(board);
	}

	/**
	 * Checks whether given board is solvable.
	 * -> This method is inherited from; Solvability of the Tiles Game by Mark Ryan
	 * http://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.html
	 *
	 * @param board Location of the pieces in board
	 * @return True if puzzle is solvable using valid moves
	 */
```

```
	private boolean isSolvable(byte[] board)
	{
		byte inversionVariable = 0;

		for (int i=0;i<board.length;i++)
			if (board[i]!=State.EMPTY)
				for(int k=i+1;k<board.length;k++)
					if(board[k]!=State.EMPTY && board[k]>board[i])
						++inversionVariable;

		return inversionVariable%2==0;
	}

	/**
	 * Shuffles an array.
	 * @param array
	 */
	private  void shuffle (byte [] array){
		byte n = (byte) array.length;
		for (int i = 0; i < array.length; i++) {
		    // Get a random index of the array past i.
			byte random = (byte) (i + rng.nextInt(n - i));
		    // Swap the random element with the present element.
		    byte randomElement = array[random];
		    array[random] = array[i];
		    array[i] = randomElement;
		}
	}

	/**
	 * Returns Set of random solvable states.
	 *  Since it it returns a set, elements of the set is distinct.
	 * @param size Size of the List that will be returned.
	 * @return Set that containing distinct puzzle board states
	 */
	public Set<State> getRandomStates(int size)
	{
		//181440 is the max number of valid(solvable) board configurations
		//181440 = 9! / 2;
		if (size > 181440)
		{
			System.out.println("Warning: Size cannot be bigger than max number"
					+ " of distinct states which is 181440. Setting size to max
value");
			Thread.dumpStack();
			size = 181440;
		}

		Set<State> set = new HashSet<>();

		while(set.size()<size)
			set.add(this.getRandomState());

		return set;
	}

	/**
	 * Returns the solved state
	 * @return State that represent solved puzzle
	 */
	public State getSolvedState()
	{
		return new State();
	}
}
```