COMP3702/7702 Assignment 1, 2010

The 24 Puzzle

SU Sheng Loong (42397997) and TEE Lip Jian (42430942)

Parts completed:

1. Problem Representation (1)
2. Heuristic Functions (2)
3. Informed Search Algorithms (2)
4. Repeated States (1)
5. Evaluate (3)
6. Compare (2)

- Program documentation and report (1)

This work qualifies for 12 marks.

1. Problem Representation

Objects

24 numbered tiles (each numbered from 1 to 24) and a blank space on a 5x5 board.

State

The location of each of the 24 numbered tiles and the blank space on the 5x5 board.

State Space

The state space is represented in a tree whereby each node consists of a state and the directed edges between nodes are actions. There are approximately 10^{25} states in the state space which is considered quite huge.

Initial State

Any state, which is set up in random configuration.

Successor Function

Valid states as a result of performing actions. These actions include blank space moves Left, Right, Up, Down)

Goal State

The tile numbers are in order across and down the board and the blank tile in the bottom right hand corner.

Step Cost / Path Cost

Each step costs 1. The path cost is the number of steps in the path from the initial state to the goal state.

Representation in Data Structure

The states are represented in two-dimensional arrays. A two-dimensional array mimics the actual puzzle in such a way that the position of the numbered tiles and blank space are represented exactly as they are.

Constraints

Only one adjacent numbered tile can only be moved to the blank space at one time, which in turn blanks the original position.

24 Puzzle

Files changed: FifteenSearchApp.java, PuzzleState.java, Node.java, State.java

The implementation has been made so that the program will now generate a 5x5 puzzle states and run different informed search algorithms with the implemented heuristic functions. Before starting the program, the user may change the number of shuffle in the main() method of FifteenSearchApp.java. To run the program after compiling the source code, simply run the FifteenSearchApp class in eclipse or any other runtime environments.

2. Heuristic Functions

Formal Description of H1

The number of misplaced tiles is calculated. The counting is done by checking the position of each of every tile on the board. The fewer the number of misplaced tiles, the nearer a node is to the goal state.

Pseudocode for H1

```
function misplacedTiles(tiles of a state) returns number of misplaced tiles
    num = 0;
    for i = 0 to 4: //For each row
        for j = 0 to 4: //For each column
            if (tiles[i][j] is not 0) and (tiles[i][j] is not goalstate) value then
                num++;

    return num;
```

Description of Pseudocode for H1

The function misplacedTiles receives a two-dimensional array which represents tiles of a state and returns the number of misplaced tiles of the state. The function firstly initialises a number as zero which will be used as a counter. The function then loops through each row and column to check if the number of the tile is not zero and if the tile is misplaced. If yes, the num variable will be incremented by one. In other words, the function does not take consideration of misplacement of the blank space. At the end, the num variable will be returned.

Implementation of H1

This method is added in PuzzleState.java.

```java
/**
 * Heuristic function 1:
 * This method accepts tiles of a state in the two-
 * dimensional array form and loops through each of every
 * tile to check if it is misplaced. If it is, the counter
 * will be incremented. At the end of the method, the
 * counter will be returned.
 *
 * @param tiles Tiles of a state
 * @return number of misplaced tiles
 */
public static int misplacedTiles(int[][] tiles) {
	int num = 0; // initialise counter to zero
	// For each row
	for (int i=0; i<5; i++) {
		// for each column in the particular row
		for (int j=0; j<5; j++) {
			// if tile is not zero and is misplaced
			if (tiles[i][j]!=0 && tiles[i][j]!=i*5+j+1) {
				num++; // increment counter
			}
		}
	}
	return num;
}
```

Formal Description of H2

In this heuristic function, the total Manhattan distance is calculated. The counting is done by checking the position of each of every tile on the board, and sum up the distance of each tile to its goal position. The fewer the total distance, the nearer for current state to the goal state.

Pseudocode for H2

```
function mDistance(tiles of current state) returns total Manhattan Distance
    totalDistance = 0;
    for row = 0 to 4: //For each row
        for col = 0 to 4: //For each column
            if (tiles[row][col] != 0) then
                totalDistance  = totalDistance + | row – tiles[row][col] / 6 |;
                if (tiles[row][col] % 5 != 0) then
                    totalDistance = totalDistance + | col – (dCol -1) |;
                else then
                    totalDistance = totalDistance + | (col – 4) |;
    return totalDistance;
```

Description of Pseudocode for H2

The function mDistance takes a two-dimensional array which represents tiles of a state as argument and returns the total Manhattan Distance of the state. The function firstly initialises a zero totalDistance that will be used as a total distance counter. The function then loops through each row and column to check if the tile is misplaced. If the tile is misplaced, the function will calculate the distance of the current tile to its desired position, excluding the blank tile, which does not carry any distance value, and add it to the variable totalDistance. After all, the totalDistance variable will be returned.

Implementation of H2

This method is added in PuzzleState.java.

```java
/**
 * Heuristic function 2:
 * The function loops through each row and column to check if the tile is
 * misplaced. If the tile is misplaced, the function will calculate the
 * distance of the current tile to its desired position, excluding the
 * blank tile.
 * @param states  tiles of current state
 * @return int          the total Manhattan Distance
 */
public static int mDistance(int[][] states) {
      int totalDist = 0,dRow,dCol;
      for(int row = 0; row < 5; row++) {
            for(int col = 0; col < 5; col++) {
                  //If current tile is not a blank tile
                  if(states[row][col] != 0) {
                        //Determine Row Distance (dRow)
                        totalDist += Math.abs(row - states[row][col] / 6);
                        //Determine Column distance (dCol)
                        if((dCol = states[row][col] % 5) != 0) {
                              //for value that is a factor of 1 to 4
                              totalDist += Math.abs(col - (dCol - 1));
                        } else {
                              //for value that is factor of 5
                              totalDist += Math.abs(col - 4);
                        }
                  } //End if
            }//End nested for loop
      }//End for loop
      return totalDist;

}
```

Invented Heuristic Functions

Formal Description of H3

In this heuristic function, the total number of tiles that is placed wrongly adjacent to their above tiles is calculated. However, it can only be done from second row until the last row. As for first row's tiles, they obviously have no above tiles. Therefore, they will be checked against the goal state values. By returning total number of wrongly placed adjacent tiles, the function is able to whether they are closer to goal state since goal state results 0 wrongly placed adjacent tiles.

Pseudocode for H3

```
function wrongAboveAdjacent (tiles of current state) returns number of wrongly adjacent tiles
    wrongAboveAdjacentTiles = 0;
    for row = 0 to 4:
        for col = 0 to 4:
            if (tiles[row][col] is 0) then
                continue;
            if (row is 0) then
                if (tiles[row][col] != row * 5 + col +1) then
                    wrongAboveAdjacentTiles++;
            else if (tiles[previousRow][col] + 5 != tiles[row][col]) then
                    wrongAboveAdjacentTiles++;
    return wrongAboveAdjacentTiles;
```

Description of Pseudocode for H3

The function wrongAboveAdjacent takes a two dimensional array which represents tiles of a state as argument and returns the total number of wrongly above adjacent tiles in the state. The function firstly initialises a zero valued wrongAboveAdjacentTiles that used as a counter. It will then loops through each row and column for checking every tiles. If the function encounters a blank tile, it will skip through it, otherwise, the function will check if it currently examines the first row's tiles, it will check the tiles against the goal state value, else, the numbers in other rows will be check whether they are correctly adjacent to its above tile. Finally, the function returns wrongAboveAdjacentTiles value as an indicator for the inform search.

Implementation of H3

This method is added in PuzzleState.java.

```java
/**
 * Heuristic function 3:
 * The function loops through each row to check each tile in current are
 * correctly adjacent to its above tile. In addition, since first row has no
 * above adjacent tiles, they will check against the goal state values.
 * @param states  tiles of current state
 * @return int          the total number wrongAboveAdjacentTiles
 */
 public static int wrongAboveAdjacent(int[][] states) {
    int wrongAboveAdjacentTiles = 0;
    for (int row = 0; row < 5; row++) {
      for (int col = 0; col < 5; col++) {
            //Ignore blank tiles
            if (states[row][col] == 0)
                  continue;
            if (row == 0) {
                  //Check first row tiles against goal state value
                  if (states[row][col] != row * 5 + col + 1)
                        wrongAboveAdjacentTiles++;
                  //other rows' tile checks against its above adjacent tile
                  } else if (states[row-1][col] + 5 != states[row][col])
                        wrongAboveAdjacentTiles++;
      }
    }
    return wrongAboveAdjacentTiles;

 }
```

Formal Description of H4

In this heuristic function, the first and last column of the tiles will be checked against the tiles in the goal states. If any of these tiles are not the same as what they are in the goal states, the number of pieces will be incremented. For the rest of the tiles, checking will be done based on whether the tiles are in sequence or not. What heuristic is trying to achieve is to break the puzzle state into different pieces so that the pieces contains tiles in sequence. This heuristic believes that the fewer the number of pieces, the more number of tiles which are already in sequence or correct order.

Pseudocode for H4

```
function countPieces(tiles of current state) returns number of pieces
    numOfPieces = 0;
    for row = 0 to 4: //For each row
        previousNumber = first value for current row;
        for col = 0 to 4: //For each column
            if it is the first column then
                if previousNumber != goalstate value then
                    numOfPieces++;
                end nested-IF
            else if it is the last column and the tile value is not 0 then
                if tiles[row][col] != goalstate value then
                    numOfPieces++;
                end nested-if
            else then
                if tile[row][col] is a blank tile then
                    if previousNumber != 24 then
                        numOfPieces++;
                        continue;
                    end inner-if
                end nested-if
                if tile[row][col] != previousNumber +1 then
                    numOfPieces++;
                end nested-if
            end if
            previousNumber = tiles[row][col];
        end nested-for-loop
    end for-loop
    return numOfPieces;
end function
```

Description of Pseudocode for H4

The function countPieces takes a two dimensional array which represents tiles of a state as argument and returns the total number of pieces in the state. The function firstly initialises a zero valued numOfPieces that used as a pieces counter. It will then loops through each row, stores the current first number of each row into a variable previousNumber and start checking every number in the row. If the currently examined number is the first or last number of the row, it will check it against the goal state value. If the check fails, it indicates that the current number is not in the correct row and hence numOfPieces is incremented. Other than first and last number in the row, if the currently examined number is a blank tile, we will assume its value is 25 and check whether previous number is 24, if it is not, numOfPieces will be incremented. Otherwise, the currently examined number will be checked against previousNumber, which will be updated after each number examination, in order to check whether they are in sequence. If they are not in sequence, numOfPieces will be incremented. At the end, the numOfPieces variable will be returned.

Implementation of H4

```
/**
 * Heuristic function 4:
 * The function loops through each row to check whether the tiles in the
 * row are arranged orderly, and is currently sitting at correct row
 * position. If it violates the rule above, it will be counted as a piece
 * in the row. Finally, function will return the number of pieces of the
 * board.
 * @param states  tiles of current state
 * @return int          the total number of pieces
 */
 public static int countPieces(int[][] states) {
    int numOfPieces = 0;
    for(int row = 0; row < 5; row++) { //For each row
      int previousNumber = states[row][0];
      for(int col = 0; col < 5; col++) { //For each column
            if (col == 0) { //First number in the row
                  //previousNumber check against goalstate value
                  if(previousNumber != (row * 5 + 1))
                              numOfPieces++;
            } else if (col == 4 && states[row][col] != 0) {//Last number in row
                  //current examine number check against goalstate value
                  if(states[row][col] != (row * 5 + 5))
                              numOfPieces++;
            } else {
                  //If is a blank tile
                  if(states[row][col]== 0)
                        if(previousNumber != 24){
                              numOfPieces++;
                              continue;
                        }
                  //Second, Third and fourth number in the row
                  if(states[row][col] != previousNumber + 1)
                        numOfPieces++;
            }
            //update previous number
            previousNumber = states[row][col];
      }
      return numOfPieces;
 }
```

3. Informed Search Algorithms

Formal Description of Greedy Search

Greedy search is a type of best first search algorithm. Basically greedy search tries to expand the node closest to the goal. Therefore the evaluation function, f(n) for greedy search is simply the heuristic function, h(n) which estimates the distance from the node to the goal state. Greedy search will always expand node where f(n) = h(n) is minimised. The implementation of greedy search requires a priority queue in which the nodes are sorted according to the evaluation value. Greedy search generally has more efficient search but can stuck in the loop which in turn makes the search incomplete and not optimal.

Pseudocode for Greedy Search

```
function greedySearch(Initial State) returns actions
    goal = genericSearch(Initial State, priority queue with GreedyComparator);
    return Actions that lead from initial state to goal;


GreedyComparator
    function compare(nodeA, nodeB) returns integer value
        integer = heuristic(state of nodeA) – heuristic(state of nodeB);
        return integer;


function genericSearch(initial state, priorityQueue with defined comparator) returns Goal node
    add initial state into priority queue;
    while priority queue is not empty:
        head = first element(highest priority) in priority queue;
        if (head represents goal state) then
            return head;
        add all head’s expandable child to priority queue;
    return null;
```

Description of Pseudocode for Greedy Search

Firstly, greedy search will further call the genericSearch function to search to the goal state by given the initial state and an empty priority queue with defined comparator, which is GreedyComparator in this case. GreedyComparator is used by the priority queue to determine all its elements’ priority, differentiating by their heuristic values, which is the value return by heuristic function only.

While inside genericSearch function, firstly, it will insert the initial state into the priority queue, and loop for searching to the goal state by expanding all the possible moves from current move until the priority queue is empty or goal state is found. Null will be returned if loop was finished but goal state is not found.

Implementation of Greedy Search

This method is added in FifteenSearchApp.java.

```
/**
 * Greedy implementation with heuristic function 1.
 * @param state initial puzzle state
 * @return moves that lead to goal state
 */
 public static Action[] solveH2G(PuzzleState state){
   /* call generic search by given initial state and priority queue with
      h(n1)-h(n2) comparator
   */
   Node goal = Node.genericSearch(state,
      new PriorityQueue<Node>(11, new Comparator<Node>() {
            //priority queue with comparator( h(a) - h(b) )
            public int compare(Node lhs,Node rhs) {
                  int lhsH2 = PuzzleState.mDistance(lhs.getState().getTiles());
                  int rhsH2 = PuzzleState.mDistance(rhs.getState().getTiles());
                  return lhsH2 - rhsH2;
            }
         }
      ));
      Action[] actions = goal.getActions();
      return actions;
 }
```

This method is added in Node.java.

```
/**
 * Generic search function takes initial state and a priority queue as argument.
 * Node will be inserted into queue based on their priority, particularly their
 * heuristic value. The smaller heuristic value the node has, the higher
 * priority the node has in the queue.Hence, a generic search can be implemented
 * for both greedy search and A* search algorithm, but have to distinguish by
 * their heuristic value. As given below:
 * Greedy Search: f(n) = h(n)
 * A* search: f(n) = h(n) + g(n)
 * @param initial initial state
 * @param fringe  the pqueue of all nodes that should be expanded, usually
 * empty.
 * @return Node   the solution node if one is found, null otherwise
 */
 public static Node genericSearch(State initial, PriorityQueue<Node> fringe) {
      fringe.add(new Node(initial));
      while(!fringe.isEmpty()) {
            //pick the lowest heuristic value move
            Node head = fringe.poll();
            State state = head.getState();
            //check whether it is a goal state
            if(state.goal()){
                  return head;
            }
            for (Object child:Arrays.asList(head.expand())) {
                        fringe.add((Node)child);
            }
      }
      return null;

 }
```

Formal Description of A* Search

A* search is also a kind of best first search algorithm. Unlike greedy search, A* search aims to avoid expanding nodes which already have high path costs. Therefore, the evaluation function f(n) is equal to g(n) + h(n) where g(n) is the path cost to the node and h(n) is the heuristic value. The implementation of A* search also requires a priority queue except the criteria of sorting the nodes is different. Compared to greedy search, A* search is better in terms of completeness and optimality but is worse in terms of time and space required.

Pseudocode for A* Search

```
function A*Search(Initial State) returns actions
    goal = genericSearch(Initial State, priority queue with A*Comparator);
    return Actions that lead from initial state to goal;


A*Comparator
    function compare(nodeA, nodeB) returns integer value
        integer = (heuristic(state of nodeA) + totalPathCost(nodeA))
                  – (heuristic(state of nodeB) + totalPathCost(nodeB));
        return integer;


function genericSearch(initial state, priorityQueue with defined comparator) returns Goal node
    add initial state into priority queue;
    while priority queue is not empty:
        head = first element(highest priority) in priority queue;
        if (head represents goal state) then
            return head;
        add all head’s expandable child to priority queue;
    return null;
```

Description of Pseudocode for A* Search

Firstly, A* search will further call the genericSearch function to search to the goal state by given the initial state and an empty priority queue with defined comparator, which is A*Comparator in this case. A*Comparator is used by the priority queue to determine all its elements’ priority, differentiating by their heuristic values, which is the value return by the sum of heuristic function’s value and total path costs leads to the move.

While inside genericSearch function, firstly, it will insert the initial state into the priority queue, and loop for searching to the goal state by expanding all the possible moves from current move until the priority queue is empty or goal state is found. Null will be returned if loop was finished but goal state is not found.

Implementation of A* Search

This method is added in FifteenSearchApp.java.

```
/**
 * A* implementation with heuristic function 2.
 * @param state initial puzzle state
 * @return moves that lead to goal state
 */
 public static Action[] solveH2A(PuzzleState state){
  /*  call generic search by given initial state and priority queue with
      [h(n1) + g(n1)] - [h(n2) + g(n2)] comparator.
  */
  Node goal = Node.genericSearch(state, new PriorityQueue<Node>(11,
      new Comparator<Node>() {
            //priority queue with comparator( [h(a) + g(a)] - [h(b) + g(b)] )
            public int compare(Node lhs,Node rhs) {
                  int lhsH2 = PuzzleState.mDistance(lhs.getState().getTiles());
                  int rhsH2 = PuzzleState.mDistance(rhs.getState().getTiles());
                  return (lhsH2 + lhs.getDepth()) - (rhsH2 + rhs.getDepth());
            }
      }
  ));
  Action[] actions = goal.getActions();
  return actions;

}
```

This method is added in Node.java.

```
/**
 * Generic search function takes initial state and a priority queue as argument.
 * Node will be inserted into queue based on their priority, particularly their
 * heuristic value. The smaller heuristic value the node has, the higher
 * priority the node has in the queue.Hence, a generic search can be implemented
 * for both greedy search and A* search algorithm, but have to distinguish by
 * their heuristic value. As given below:
 * Greedy Search: f(n) = h(n)
 * A* search: f(n) = h(n) + g(n)
 * @param initial initial state
 * @param fringe  the pqueue of all nodes that should be expanded, usually
 * empty.
 * @return Node   the solution node if one is found, null otherwise
 */
 public static Node genericSearch(State initial, PriorityQueue<Node> fringe) {
      fringe.add(new Node(initial));
      while(!fringe.isEmpty()) {
            //pick the lowest heuristic value move
            Node head = fringe.poll();
            State state = head.getState();
            //check whether it is a goal state
            if(state.goal()){
                  return head;
            }
            for (Object child:Arrays.asList(head.expand())) {
                        fringe.add((Node)child);
            }
      }
      return null;

 }
```

4. Repeated States

Formal Description of Repeated States

The repeated states can be detected by following the three rules – never return to the state which has been traversed previously, never form a cycle within the nodes and never generate a state which has already been generated. This is to avoid the searches stuck in a loop.

Pseudocode for Repeated States

```
function genericSearch(initial state, priorityQueue with defined comparator) returns Goal node
    add initial state into priority queue;
    initialize allNode list to keep track all generated nodes
    insert initial state into allNode
    while priority queue is not empty:
        head = first element(highest priority) in priority queue;
        if (head represents goal state) then
            return head;
        for each move in all possible moves:
            if (the move is not in allNode) then
                insert the move into priority queue;
                insert the move into allNode;
    return null;
```

Description of Pseudocode for Repeated States

The repeated state checking is a fraction of codes that added into the genericSearch function. Generally, before entering the while loop, it is necessary to initialise a list to keep track all generated nodes, and insert the first node (initial state) into it. While inside the looping process, there is another for loop added to check whether all possible moves leaded from current move are not generated before. If they were generated before, they will not insert to the priority queue, because they may cause a sequence of moves that having a loop inside. Otherwise, the possible move will be inserted to the priority queue and allNode list.

Implementation of Repeated States

This method is added in Node.java.

```java
/**
 * Generic search function takes initial state and a priority queue as argument.
 * Node will be inserted into queue based on their priority, particularly their
 * heuristic value. The smaller heuristic value the node has, the higher
 * priority the node has in the queue.Hence, a generic search can be implemented
 * for both greedy search and A* search algorithm, but have to distinguish by
 * their heuristic value. As given below:
 * Greedy Search: f(n) = h(n)
 * A* search: f(n) = h(n) + g(n)
 * @param initial initial state
 * @param fringe  the pqueue of all nodes that should be expanded, usually
 * empty.
 * @return Node         the solution node if one is found, null otherwise
 */
 public static Node genericSearch(State initial, PriorityQueue<Node> fringe) {
    fringe.add(new Node(initial));
    //Keep track of all generated nodes
    ArrayList<Node> allNode = new ArrayList<Node>();
    allNode.add(new Node(initial));
    while(!fringe.isEmpty()) {
      //pick the lowest heuristic value move
      Node head = fringe.poll();
      State state = head.getState();
      //check whether it is a goal state
      if(state.goal()){
            return head;
      }
      //Check for repeated state
      for (Object child:Arrays.asList(head.expand())) {
            //do not repeat examining the node which had generated before
            if(!allNode.contains((Node)child)){
                  fringe.add((Node)child);
                  allNode.add((Node)child);
            }
      }
   }
   return null;

}
```

5. Evaluate

Effective Branching Factors

Depth	H1 A*	H1 Greedy	H2 A*	H2 Greedy	H3 A*	H3 Greedy	H4 A*	H4 Greedy
2	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.1800	1.1800
4	1.0182	1.0000	1.0338	1.0000	1.0364	1.0364	1.0728	1.0364
8	1.0942	1.1360	1.0704	1.0244	1.1362	1.1434	1.4098	1.0426
16	1.2720	1.5010	1.2802	1.2094	1.2750	1.3330	1.0688	1.4404
32	nil	nil	nil	nil	nil	nil	nil	nil

The method for generating random puzzle has been modified so that the blank square will not return to the previous position. At depth 32, the searches cannot complete in the time length of 5 minutes. However, the searches would yield results if much longer time is given to run the searches.

Strengths and Weaknesses of the Heuristics

Evaluation of EBFs

EBF is used to measure how good a particular heuristic function is. A well designed heuristic function should have EBF of close to 1.

H1 Greedy: The EBF is very impressive if the depth of the optimal solution is shallow. However, as the depth increases, the EBF also increases at a relatively fast rate.

H1 A*: The EBF is also good and is in fact more ideal compared to H1 greedy search as the depth increases.

H2 Greedy: The EBF is good and is better than the H2 A* search. Also it is noticeable that H2 can provide more accurate heuristic value than any other heuristic functions. By comparing the EBF of searches using H2 with other EBFs, H2 clearly made the searches faster and the EBF closer to 1.

H2 A*: Unfortunately the EBF of H2 A* search is less favourable compared to H2 greedy search. This could be due to the fact that A* generally expand more nodes and can therefore yields complete and optimal solution.

H3 Greedy: The EBF shows that H3 is also a well-designed heuristic function by having EBF of close to 1. The EBF at each depth is quite close to H3 A* search, though relatively low.

H3 A*: The EBFs calculated for H3 A* search is ideal and is better than H3 Greedy search. However, the EBF value is generally less favourable compared to searches using H1 and H2.

H4 Greedy: Searches using H4 yield the highest overall EBFs. So it might be due to the fact that H4 is a less effective heuristic function.

H4 A*: H4 A* search has generally better EBFs compared to H4 Greedy search except at depth 4. Having said that, overall EBFs of less than 1.5 is still considered good.

Space and Time Complexity

H1: H1 has space complexity of O(1) because it does not require storing of all states which have been checked or are misplaced into the memory. What this function stores is only the number of misplaced tiles of which the memory required is not affected by the number of input. H1 has time complexity of O(n) because it has to iterate through each tile in each row and column.

H2: Same as H1, H2 also has space complexity of O(1) because it only stores one state in the memory at one time during checking. H2 has time complexity of O(n) because it has to calculate Manhattan distance for each of the tiles.

H3: H3 has space complexity of O(1) because it only stores one state to be checked in the memory at a time and no matter how large the puzzle state is, the heuristic will always just require storing of one state at a time. H3 also has time complexity of O(n) since it has to check every tile in the puzzle state to see if the tiles are not adjacent to the tiles that they are supposed to be adjacent to in the goal state.

H4: H4 also has space complexity of O(1) because it does not store any extra state except the one state to be checked in each round of the loop. H4 has time complexity of O(n) as it also needs to check through each of the tile to check if they are in sequence.

Accuracy

Accuracy shows how good or accurate a heuristic function is at representing the goodness of a state. H1 is fairly accurate, though less accurate than H2 and H3. H1 is a relaxed heuristic function where a tile can be moved without taking consideration of the distance and also whether it is adjacent to the blank space. H2 is always the most accurate among the four heuristics since it produces highest value by calculating the Manhattan distance of all tiles. H3 is more accurate than H1 and H4 because H3 takes consideration of the adjacency between the tiles vertically on top of checking whether the tiles in the first row are in the goal state. This generally produces higher value compared to H1 and H4 but certainly not H2 since the number of wrongly placed vertically adjacent tile is less than the value of calculating Manhattan distance. H4 is the least accurate among these four heuristic functions since it breaks the puzzle into different pieces without actually considering the distance between the current position and the goal state position of the tiles. Also, H4 is not an admissible function, the reason being will be discussed in the optimality section.

Completeness

H1 Greedy: Complete because repeated-state checking is applied.

H1 A*: Complete

H2 Greedy: Complete because repeated-state checking is applied.

H2 A*: Complete

H3 Greedy: Complete because repeated-state checking is applied.

H3 A*: Complete

H4 Greedy: Complete because repeated-state checking is applied.

H4 A*: Complete

Optimality

H1 Greedy: Not optimal.

H1 A*: Optimal

H2 Greedy: Not optimal.

H2 A*: Optimal

H3 Greedy: Not optimal.

H3 A*: Optimal

H4 Greedy: Not optimal.

H4 A*: Not always optimal because heuristic function H4 is not admissible after comparing with other heuristic functions because it sometimes has larger A* solution as compared to the others. For instance, A* search using H1, H2 and H3 have optimal solution at depth 12 but A* search using H4 has solution at depth 14 and thus it is not optimal as compared to the former three searches.

6. Reference

ORACLE(2010). *Java™ Platform, Standard Edition 6 API Specification* [Online]. Oracle. Available: http://download.oracle.com/javase/6/docs/api/ [Accessed 03/09/2010].

Russel, Stuart & Norvig, Peter (2009). *Artificial Intelligence: A Modern Approach*, Prentice Hall.