Artificial Intelligence I
Programming report
assignment 1

E.Bier A.Roorda
s3065979 s2973782
Lab Group: CS1

May 3, 2016

1 Theory Questions

1. PEAS description
   (a) Reversi:
   - Performance Measure: Number of right colored pieces.
   - Environment: Reversi Board.
   - Actuators: Placing pieces.
   - Sensors: Observe board position.
   - Other characteristics: fully observable, deterministic, sequential, static, discrete, multi-agent. Because the program only needs to look at the current scenario to make a decision, the best architecture for this agent is the Simple Reflex Agent.

   (b) Robot lawn mower:
   - Performance Measure: Area of grass with acceptable length.
   - Environment: Yard.
   - Actuators: Grass cutter, wheels.
   - Sensors: Camera, distance sensor, obstacle sensor,
   - Other characteristics: partially observable, stochastic, sequential, dynamic, continuous, multi-agent. This agent should be designed as a model-based goal-based agent, since it needs to track how the world looks at the moment of the decision. This includes tracking wherever it's recently been through, checking how long the grass is, etc.

2. Maze
   (a) The mazeDFS() is unable to find a path between the yellow and red squares because it gets stuck switching between squares 14 and 10. The reason for that is the fixed that is being used to choose the movements. Whenever the algorithm can't go north or east, it will go to the south or to the west. In case it does go south, the next movement will clearly be to the north and it'll be in the same situation as it is now.

   (b) To fix this issue, the code should keep track of the locations it has visited.

```
procedure mazeDFS(maze, start, goal):
stack = []
stack.push(start)
while stack is not empty:
  start.visited = true
```

```
    loc = stack.pop()
    if loc == goal:
      print "Goal found"
      return
    for move in [N,E,S,W]: # in this order!
      if allowedMove(loc, move) and not neighbour(maze, loc, move).visited:
        stack.push(neighbour(maze, loc, move))
print "Goal not found"
```

(c) The algorithm takes the following path for that call:
$1 \rightarrow 2 \rightarrow 6 \rightarrow 5 \rightarrow 9 \rightarrow 13 \rightarrow 14 \rightarrow 10 \rightarrow 7 \rightarrow 3 \rightarrow 4 \rightarrow 8 \rightarrow 12 \rightarrow 11 \rightarrow 15$

(d) If we change the order of visiting neighbors we have the following path:
$1 \rightarrow 2 \rightarrow 6 \rightarrow 7 \rightarrow 3 \rightarrow 4 \rightarrow 8 \rightarrow 12 \rightarrow 11 \rightarrow 15$

(e) The algorithm will always find a solution in that case because it will end up visiting all reachable states (until it finds the solution). All of the discovered states are added to the queue, so the algorithm is always moving forward, although it will visit the same states a lot of times.

(f) The states would be visited in the following order (repeating states are omitted):
$1 \rightarrow 2 \rightarrow 6 \rightarrow 7 \rightarrow 5 \rightarrow 3 \rightarrow 9 \rightarrow 4 \rightarrow 8 \rightarrow 13 \rightarrow 12 \rightarrow 14 \rightarrow 11 \rightarrow 10 \rightarrow 15$

(g) It is possible to reduce the number of visited states in the BFS algorithm. To do so, the algorithm should keep track of visited states so it doesn't have to go through them again.

```
procedure mazeBFS(maze, start, goal):
queue = []
queue.enqueue(start)
while queue is not empty:
    start.visited = true
    loc = queue.dequeue()
    if loc == goal:
      print "Goal found"
      return
    for move in [N,E,S,W]: # in this order!
      if allowedMove(loc, move) and not neighbour(maze, loc, move).visited:
        queue.enqueue(neighbour(maze, loc, move))
print "Goal not found"
```

(h) For large mazes, I would use a variation of the BFS algorithm. This algorithm always yields the shortest path, since it explores all branches at the same time, whereas the DFS algorithm may not give you the optimal solution because it may choose to explore first a branch that contains a very long path to the goal, when a short one does exist. Note that DFS and BFS have very similar running times, so there are no clear disadvantages to picking BFS over DFS.

2 Programming Assignment

2.1 Answers

1. The program runs out of memory for the 0 100 call but not for the 0 99 and 0 102 calls. The reason for that is the fact that 99 and 102 are both multiples of 3 and therefore are much easier to reach and are not so deep in the search. On the other hand, the path to 100 is considerably longer which makes the number of states in the fringe way bigger. As for the path from 1 to 0 and 0 to 1 using DFS, the algorithm is runs out of memory for the 0 to 1 case because it gets stuck on 0 forever since the last item on the stack is always $0 \xrightarrow{*3} 0$.

2. The algorithm is much quicker now since it does not repeat previously visited states and does not change to states that will bring it further from the goal.

5. The optimal path from 0 to 42 are:
$0 \xrightarrow{+1} 1 \xrightarrow{+1} 2 \xrightarrow{*3} 6 \xrightarrow{+1} 7 \xrightarrow{*2} 14 \xrightarrow{*3} 42$
$0 \xrightarrow{+1} 1 \xrightarrow{+1} 2 \xrightarrow{*3} 6 \xrightarrow{+1} 7 \xrightarrow{*3} 21 \xrightarrow{*2} 42$

7. The IDS algorithm usually ends up visiting a large number of states and always provides the shortest path. Both BFS and DFS visit a similar number of states but sometimes, for specific inputs, end up visiting a lot of states. The priority implementation, on the other hand, seems to be really efficient, always providing the shortest path and visiting a reasonable, albeit sometimes slightly larger, amount of states than the BFS and DFS.

8. The two heuristics used for the A* algorithm use the same basic concept: The knight can always moves 3 positions **away** from it's current position in any shape. The first heuristic uses this by determining the amount of moves to get to the positions near (6 cell radius) the knight. The second heuristic expands on the first one by (under)estimating the amount of movements on positions further away from it. This estimation is done using the manhattan distance and the fact that the knight would only be able to get 3 blocks closer to its goal per move. It's clear to see that the second heuristic does a much better job than the first one, since the estimations on the second one are much closer to reality. As so, the first heuristic has a branching factor of 4.585463, while for the second one it's 4.010053.

2.2 Code

Listing 1: fringe.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

#include "fringe.h"

Fringe makeFringe(int mode) {
  /* Returns an empty fringe.
   * The mode can be LIFO(=STACK), FIFO, or PRIO(=HEAP)
   */
  Fringe f;
  if ((mode != LIFO) && (mode != STACK) && (mode != FIFO) &&
      (mode != PRIO) && (mode != HEAP)) {
    fprintf(stderr, "makeFringe(mode=%d): incorrect mode. ", mode);
    fprintf(stderr, "(mode <- [LIFO,STACK,FIFO,PRIO,HEAP])\n");
    exit(EXIT_FAILURE);
  }
  f.mode = mode;
  f.size = f.front = f.rear = 0; /* front+rear only used in FIFO mode */
  f.states = malloc(MAXF*sizeof(State));
  if (f.states == NULL) {
   fprintf(stderr, "makeFringe(): memory allocation failed.\n");
    exit(EXIT_FAILURE);
  }
  f.maxSize = f.insertCnt = f.deleteCnt = 0;
  return f;
}

void deallocFringe(Fringe fringe) {
  /* Frees the memory allocated for the fringe */
```

```c
  free(fringe.states);
}

int getFringeSize(Fringe fringe) {
  /* Returns the number of elements in the fringe
   */
  return fringe.size;
}

int isEmptyFringe(Fringe fringe) {
  /* Returns 1 if the fringe is empty, otherwise 0 */
  return (fringe.size == 0 ? 1 : 0);
}

Fringe insertFringe(Fringe fringe, State s, ...) {
  /* Inserts s in the fringe, and returns the new fringe.
   * This function needs a third parameter in PRIO(HEAP) mode.
   */
  int priority;
  va_list argument;

  if (fringe.size == MAXF) {
    fprintf(stderr, "insertFringe(..): fatal error, out of memory.\n");
    exit(EXIT_FAILURE);
  }
  fringe.insertCnt++;
  switch (fringe.mode) {
  case LIFO: /* LIFO == STACK */
  case STACK:
    fringe.states[fringe.size] = s;
    break;
  case FIFO:
    fringe.states[fringe.rear++] = s;
    fringe.rear %= MAXF;
    break;
  case PRIO: /* PRIO == HEAP */
  case HEAP:
    /* Get the priority from the 3rd argument of this function.
     * You are not supposed to understand the following 5 code lines.
     */
    va_start(argument, s);
    priority = va_arg(argument, int);
    // printf("priority = %d ", priority);
    va_end(argument);
    s.cost = priority;
    fringe.states[fringe.size + 1] = s;
    siftUp(fringe.states, fringe.size + 1, fringe.size + 1);
    break;
  }
  fringe.size++;
  if (fringe.size > fringe.maxSize) {
    fringe.maxSize = fringe.size;
  }
  return fringe;
}

Fringe removeFringe(Fringe fringe, State *s) {
  /* Removes an element from the fringe, and returns it in s.
```

```
   * Moreover, the new fringe is returned.
   */
  if (fringe.size < 1) {
    fprintf(stderr, "removeFringe(..): fatal error, empty fringe.\n");
    exit(EXIT_FAILURE);
  }
  fringe.deleteCnt++;
  fringe.size--;
  switch (fringe.mode) {
  case LIFO: /* LIFO == STACK */
  case STACK:
    *s = fringe.states[fringe.size];
    break;
  case FIFO:
    *s = fringe.states[fringe.front++];
    fringe.front %= MAXF;
    break;
  case PRIO: /* PRIO == HEAP */
  case HEAP:
    *s = fringe.states[1];
    fringe.states[1] = fringe.states[fringe.size + 1];
    siftDown(fringe.states, fringe.size, 1);
    break;
  }
  return fringe;
}

void showStats(Fringe fringe) {
  /* Shows fringe statistics */
  printf("#### fringe statistics:\n");
  printf(" #size : %7d\n", fringe.size);
  printf(" #maximum size: %7d\n", fringe.maxSize);
  printf(" #insertions : %7d\n", fringe.insertCnt);
  printf(" #deletions : %7d\n", fringe.deleteCnt);
  printf("####\n");
}

void swap(State *s, int i, int j){
  State aux = s[i];
  s[i] = s[j];
  s[j] = aux;
}

void siftDown(State *s, int n, int i){

  int smallest = i;
  int left = 2 * i;
  int right = 2 * i + 1;

  if (left <= n && s[left].cost < s[smallest].cost)
    smallest = left;

  if (right <= n && s[right].cost < s[smallest].cost)
    smallest = right;

  if (smallest != i) {
    swap(s, i, smallest);
    siftDown(s, n, smallest);
```

```
  }
}

void siftUp(State *s, int n, int i){
  int child = i;
  int parent = i / 2;

  if (child > 1 && s[child].cost < s[parent].cost) {
    swap(s, parent, child);
    siftUp(s, n, parent);
  }
}
```

Listing 2: search.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "state.h"
#include "fringe.h"

#define RANGE 1000000
typedef enum {NUL, ADD, DOUBLE, TRIPLE, SUBTRACT, HALF, THIRD} operation;

void printRoute(int target, int start);
int costOf(operation op);

int visited[RANGE];
operation operationToGetTo[RANGE] = {NUL};

Fringe insertValidSucc(Fringe fringe, int value, int parent, int op, int prevCost) {
  State s;
  if ((value < 0) || (value > RANGE) || visited[value] != -1) {
    /* ignore states that are out of bounds or that have already been reached */
    return fringe;
  }

  s.value = value;
  visited[value] = parent;
  operationToGetTo[value] = op;
  return insertFringe(fringe, s, costOf(op) + prevCost);
}

void search(int mode, int start, int goal) {
  Fringe fringe;
  State state;
  int goalReached = 0;
  int visited = 0;
  int value, cost;

  fringe = makeFringe(mode);
  state.value = start;
  fringe = insertFringe(fringe, state, 0);
  while (!isEmptyFringe(fringe)) {
    /* get a state from the fringe */
```

```
    fringe = removeFringe(fringe, &state);
    visited++;
    /* is state the goal? */
    value = state.value;
    cost = state.cost;
    if (value == goal) {
      goalReached = 1;
      printRoute(value, start);
      break;
    }
    /* insert neighbouring states */
    if (value > goal){
      fringe = insertValidSucc(fringe, value-1, value, SUBTRACT, cost); /* rule n->n - 1 */
      if (value != 0){
        fringe = insertValidSucc(fringe, value/2, value, HALF, cost); /* rule n->floor(n/2) */
        fringe = insertValidSucc(fringe, value/3, value, THIRD, cost); /* rule n->floor(n/3) */
      }
    }
    else {
      fringe = insertValidSucc(fringe, value+1, value, ADD, cost); /* rule n->n + 1 */
      if (value != 0){
        fringe = insertValidSucc(fringe, 3*value, value, TRIPLE, cost); /* rule n->3*n */
        fringe = insertValidSucc(fringe, 2*value, value, DOUBLE, cost); /* rule n->2*n */
      }
    }
  }
  if (goalReached == 0) {
    printf("goal not reachable ");
  } else {
    printf("goal reached ");
  }
  printf("(%d nodes visited)\n", visited);
  showStats(fringe);
  deallocFringe(fringe);
}

int main(int argc, char *argv[]) {
  int start, goal, fringetype;
  if ((argc == 1) || (argc > 4)) {
    fprintf(stderr, "Usage: %s <STACK|FIFO|HEAP> [start] [goal]\n", argv[0]);
    return EXIT_FAILURE;
  }
  fringetype = 0;

  if ((strcmp(argv[1], "STACK") == 0) || (strcmp(argv[1], "LIFO") == 0)) {
    fringetype = STACK;
  } else if (strcmp(argv[1], "FIFO") == 0) {
    fringetype = FIFO;
  } else if ((strcmp(argv[1], "HEAP") == 0) || (strcmp(argv[1], "PRIO") == 0)) {
    fringetype = HEAP;
  }
```

```
  if (fringetype == 0) {
    fprintf(stderr, "Usage: %s <STACK|FIFO|HEAP> [start] [goal]\n", argv[0]);
    return EXIT_FAILURE;
  }

  start = 0;
  goal = 42;
  if (argc == 3) {
    goal = atoi(argv[2]);
  } else if (argc == 4) {
    start = atoi(argv[2]);
    goal = atoi(argv[3]);
  }

  printf("Problem: route from %d to %d\n", start, goal);

  /* Initializing the visited array, not sure about the size though... */
  if (start < goal)
    for (int i = start / 3; i < 3 * goal && i < RANGE; i++)
      visited[i] = -1;
  else
    for (int i = goal / 3; i < 3 * start && i < RANGE; i++)
      visited[i] = -1;

  search(fringetype, start, goal);
  return EXIT_SUCCESS;
}

void printRoute(int target, int start){

  int current, x, length = 0, cost = 0;
  operation op;
  State s;
  Fringe operations = makeFringe(STACK);
  current = target;

  /* Checking what operations were done to get to the target value */
  while (current != start){
    s.value = operationToGetTo[current];
    /* Put the operations on a stack to print in the right order */
    operations = insertFringe(operations, s);
    current = visited[current];
    length++;
  }

  printf("\n%d ", start);
  x = start;
  while (!isEmptyFringe(operations)){
    operations = removeFringe(operations, &s);
    op = s.value;
    cost += costOf(op);
    switch(op){
      case ADD:
        printf("(+1)-> %d ", ++x);
        break;
      case SUBTRACT:
        printf("(-1)-> %d ", --x);
        break;
```

```
      case HALF:
        x /= 2;
        printf("(/2)-> %d ", x);
        break;
      case DOUBLE:
        x *= 2;
        printf("(*2)-> %d ", x);
        break;
      case THIRD:
        x /= 3;
        printf("(/3)-> %d ", x);
        break;
      case TRIPLE:
        x *= 3;
        printf("(*3)-> %d ", x);
        break;
      case NUL:
        printf("NULL ");
        break;
    }
  }
  printf("\n");
  printf("length: %d, cost: %d\n\n", length, cost);
  deallocFringe(operations);
}

int costOf(operation op) {
  int c;
  switch(op){
    case ADD:
    case SUBTRACT:
      c = 1;
      break;
    case THIRD:
    case HALF:
      c = 3;
      break;
    case DOUBLE:
    case TRIPLE:
      c = 2;
      break;
    default:
      c = 0;
      break;
  }
  return c;
}
```

Listing 3: ids.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_DEPTH 100000
#define RANGE 1000000
#define UNDISCOVERED 0
#define DISCOVERED 1
```

```
#define VISITED 2

typedef enum {TRIPLE, DOUBLE, ADD, THIRD, HALF, SUBTRACT} operation;

int status[3*RANGE] = {0};
int verbose = 0;
int solution[RANGE];

int min(int x, int y){
  if (x < y) return x;
  return y;
}

int max(int x, int y){
  if (x > y) return x;
  return y;
}

/* Calculates the neighbors of x, only considers neighbors that move towards the goal */
void succ(int x, int *successors, int goal){
  if (x <= goal){
    successors[2] = x + 1;
    if (x != 0){
      successors[1] = x * 2;
      successors[0] = x * 3;
    }
    else {
      successors[1] = successors[0] = -1;
    }
    successors[5] = -1;
    successors[4] = -1;
    successors[3] = -1;
  }
  else {
    successors[2] = -1;
    successors[1] = -1;
    successors[0] = -1;
    successors[5] = x - 1;
    if (x != 0){
      successors[4] = x / 2;
      successors[3] = x / 3;
    }
    else {
      successors[4] = successors[3] = -1;
    }
  }
}

int dfsLimited(int start, int goal, int maxDepth, int currentDepth, int *nodeCount){
  int result, next;
  int successors[6] = {-1};

  status[start] = DISCOVERED;
  *nodeCount += 1;
  if (verbose)
    printf("%d discovered\n", start);
```

```c

  if (currentDepth > maxDepth){
    if (verbose)
      printf("Maximum depth reached (maxDepth = %d)\n", maxDepth);
    return -1;
  }

  if (start == goal) {
    if (verbose)
      printf("Goal reached!\n");
    return currentDepth;
  }
  /* Calculating neighbors */
  succ(start, successors, goal);

  /* Visiting neighbors */
  for (int i = 0; i < 6; i++){
    next = successors[i];
    result = -1;
    if (next > 0){
      if (status[next] == UNDISCOVERED)
        result = dfsLimited(next, goal, maxDepth, currentDepth + 1, nodeCount
            );
      if (result != -1){
        if (verbose)
          printf("Found it!\n");
        solution[currentDepth + 1] = i;
        return result;
      }
    }
  }

  status[start] = VISITED;
  if (verbose)
    printf("%d visited\n", start);

  return result;
}

int ids(int start, int goal, int *nodeCount){

  int result = -1;

  for (int depth = 0; depth < MAX_DEPTH; depth++){
    /* Reseting visited nodes */
    for (int i = min(start, goal) / 3; i < max(start, goal) * 3; i++)
      status[i] = UNDISCOVERED;

    result = dfsLimited(start, goal, depth, 0, nodeCount);
    if(verbose)
      printf("\n");
    if (result != -1)
      return result;
  }

  return result;

}
```

```c

int costOf(operation op) {
  int c;
  switch(op){
    case ADD:
    case SUBTRACT:
      c = 1;
      break;
    case THIRD:
    case HALF:
      c = 3;
      break;
    case DOUBLE:
    case TRIPLE:
      c = 2;
      break;
    default:
      c = 0;
      break;
  }
  return c;
}

void printSolution(int start, int d, int nodeCount){
   int x, cost = 0;

   printf("%d ", start);
   x = start;
   for (int i = 1; i <= d; i++){
      cost += costOf(solution[i]);
      switch(solution[i]){
         case ADD:
            printf("(+1)-> %d ", ++x);
            break;
         case SUBTRACT:
            printf("(-1)-> %d ", --x);
            break;
            case HALF:
            x /= 2;
            printf("(/2)-> %d ", x);
            break;
         case DOUBLE:
            x *= 2;
            printf("(*2)-> %d ", x);
            break;
         case THIRD:
            x /= 3;
            printf("(/3)-> %d ", x);
            break;
         case TRIPLE:
            x *= 3;
            printf("(*3)-> %d ", x);
            break;
         default:
            printf("ERROR ");
            break;
      }

```

```c
  }
  printf("\nlength: %d, cost %d",d, cost);
  printf("\n\n");
  printf("Goal reached! (%d nodes visited)\n", nodeCount);

}

int main(int argc, char *argv[]){

  int start, goal, d, nodeCount = 0;

  if (argc < 3) {
    fprintf(stderr, "Usage: %s start goal [-verbose/-v]\n", argv[0]);
    return EXIT_FAILURE;
  }

  if (argc == 4 && (strcmp(argv[3], "-v") == 0 || strcmp(argv[3], "-verbose")
      == 0)){
    verbose = 1;
  }

  start = atoi(argv[1]);
  goal = atoi(argv[2]);

  printf("Problem: route from %d to %d\n\n", start, goal);

  d = ids(start, goal, &nodeCount);

  if (d >= 0){
    printSolution(start, d, nodeCount);
  }

  else {
    printf("Solution not found.\n");
  }

  return 0;
}
```

Listing 4: astar.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "fringe.h"
#include "state.h"

#define N 500 /* N times N chessboard */
#define UNKNOWN 0
#define DISCOVERED 1
#define VISITED 2

int actions[8][2] = { /* knight moves */
  {-2, -1}, {-2, 1}, {-1, -2}, {-1, 2}, {1,-2}, {1,2}, {2, -1}, {2, 1}
};
int costShortestPath[N][N];
int fScore[N][N];
int status[N][N];
```

```
unsigned long statesVisited = 0;

int isValidLocation(int x, int y) {
  return (0<=x && x < N && 0<= y && y < N);
}

void initialize() {
  int r, c;
  statesVisited = 0;
  for (r=0; r < N; r++) {
    for (c=0; c < N; c++) {
      costShortestPath[r][c] = 999999; /* represents infinity */
      fScore[r][c] = 999999;
      status[r][c] = UNKNOWN;
    }
  }
}

int max(int x, int y){
  if (x > y) return x;
  return y;
}

int min(int x, int y){
  if (x < y) return x;
  return y;
}

/* Heuristic 1 */
int h1(int row, int column, int rowGoal, int columnGoal){
  int deltaRow, deltaColumn;
  int M, m;

  deltaRow = abs(rowGoal - row);
  deltaColumn = abs(columnGoal - column);

  M = max(deltaRow, deltaColumn);
  m = min(deltaRow, deltaColumn);

  /* Too far away, Knight can move max 3 squares per turn,
     so it would have to make at least (M + m) / 3 moves */
  if (M + m >= 6)
    return ceil((float) (M + m) / 3);

  /* Right spot */
  if (M + m == 0)
    return 0;

  /* Close enough, knight can move 3 squares in any shape */
  return abs(M + m - 3) + 1;
}

/* Heuristic 2 */
int h2(int row, int column, int rowGoal, int columnGoal){
  int deltaRow, deltaColumn;
  int M, m;

  deltaRow = abs(rowGoal - row);
```

```c
  deltaColumn = abs(columnGoal - column);

  M = max(deltaRow, deltaColumn);
  m = min(deltaRow, deltaColumn);

  /* Too far away, Knight can move max 3 squares per turn,
     so it would have to make at least (M + m) / 3 moves */
  if (M + m >= 6)
    return 2;

  /* Right spot */
  if (M + m == 0)
    return 0;

  /* Close enough, knight can move 3 squares AWAY from its current position in any shape */
  return abs(M + m - 3) + 1;
}

/* Implements the A* algorithm. Receives a function to be used as a heuristic */
int aStar(int row, int column, int rowGoal, int columnGoal, int (*heuristic) (int, int, int, int)) {

  State s;
  Fringe fringe = makeFringe(PRIO);

  initialize();

  fScore[row][column] = heuristic(row, column, rowGoal, columnGoal);
  costShortestPath[row][column] = 0;

  s.row = row;
  s.column = column;
  fringe = insertFringe(fringe, s, fScore[row][column]);

  while (!isEmptyFringe(fringe)) {
    fringe = removeFringe(fringe, &s);
    statesVisited++;

    /* Goal found? */
    if (s.row == rowGoal && s.column == columnGoal){
      deallocFringe(fringe);
      return costShortestPath[s.row][s.column];
    }

    /* Insert neighbors */
    for (int act = 0; act < 8; act++){
      int r = s.row + actions[act][0];
      int c = s.column + actions[act][1];

      if (isValidLocation(r, c) && status[r][c] != VISITED){
        int gScore = costShortestPath[s.row][s.column] + 1;

        if (gScore < costShortestPath[r][c]){ /* Path better than the previous one? */
          costShortestPath[r][c] = gScore;
          fScore[r][c] = costShortestPath[r][c] + heuristic(r, c, rowGoal,
```

```
            columnGoal);
        }

        if (status[r][c] == UNKNOWN){
          State newState;
          newState.row = r;
          newState.column = c;
          status[r][c] = DISCOVERED;
          fringe = insertFringe(fringe, newState, fScore[r][c]);
        }
      }
      status[s.row][s.column] = VISITED;
    }
  }
  deallocFringe(fringe);
  return 0;
}

double effectiveBranchingFactor(unsigned long states, int d) {
  /* approximates such that N=\Sum+{i=1}^{d} b^i */
  double lwb = 1;
  double upb = pow(states, 1.0/d);
  while (upb - lwb > 0.000001) {
    double mid = (lwb + upb) / 2.0;
    /* the following test makes use of the geometric series */
    if (mid*(1-pow(mid, d))/(1-mid) <= states) {
      lwb = mid;
    } else {
      upb = mid;
    }
  }
  return lwb;
}

int main(int argc, char *argv[]) {
  int x0,y0, x1,y1;
  do {
    printf("Start location (x,y) = "); fflush(stdout);
    scanf("%d %d", &x0, &y0);
  } while (!isValidLocation(x0,y0));
  do {
    printf("Goal location (x,y) = "); fflush(stdout);
    scanf("%d %d", &x1, &y1);
  } while (!isValidLocation(x1,y1));

  printf("\nHeuristic 1:\n");
  printf("Length shortest path: %d\n", aStar(x0,y0, x1,y1, h1));
  printf("#visited states: %lu\n", statesVisited);
  printf("#effective branching factor: %f\n", effectiveBranchingFactor(
      statesVisited, 8));

  printf("\nHeuristic 2:\n");
  printf("Length shortest path: %d\n", aStar(x0,y0, x1,y1, h2));
  printf("#visited states: %lu\n", statesVisited);
  printf("#effective branching factor: %f\n", effectiveBranchingFactor(
      statesVisited, 8));
  return 0;
}
```