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

int main(int argc, char **argv){
        int i, j, a, dirt, x, y;
        int limit;
        dirt = 0;
        generated = 1;
        expanded = 0;

        //scan the size of the matrix
        scanf("%d %d", &m, &n);

        limit = 2*m*n;

        //scan the matrix itself
        getchar();
        for(i = 0; i < n; i++){

                for(j = 0; j < m; j++){
                        w[i][j] = getchar();

                        //if it is the robot start position
                        if(w[i][j] == '@'){
                                x = j;
                                y = i;
                        }

                        //counts the number of dirt and inserts in the list of d
irts
                        if(w[i][j] == '*'){
                                insertListDirt(j, i);
                                dirt++;
                        }
                }

                //throw away break line
                a = getchar();
        }

        //executes dfs algorithm
        if(strcmp(argv[1], "depth-first") == 0){
                if(dirt > 0){
                        if(dfs(x, y, dirt, 0, limit) == 1){
                                printList();
                                printf("%d nodes generated\n", generated);
                                printf("%d nodes expanded\n", expanded);
                        }
                        else
                                printf("no solution found\n");
                }
                //no dirt in the matrix, there is nothing to do
                else{
                        printf("%d nodes generated\n", generated);
                        printf("%d nodes expanded\n", expanded);
                }
        }

        //executes iterative dfs
        else if(strcmp(argv[1], "depth-first-id") == 0){
                if(dirt > 0){
                        for(i = 0; i < limit && dfs(x, y, dirt, 0, i) == 0; i++)
;

                        printList();
                        printf("%d nodes generated\n", generated);
                        printf("%d nodes expanded\n", expanded);
```



```
                }
                //no dirt in the matrix, there is nothing to do
                else{
                        printf("%d nodes generated\n", generated);
                        printf("%d nodes expanded\n", expanded);
                }
        }

        //executes djikstra algorithm
        else if(strcmp(argv[1], "uniform-cost") == 0)
                printAstar(x, y, dirt, 0);

        //executes a-star algorithm
        else if(strcmp(argv[1], "a-star") == 0){
                if(argv[2] == NULL)
                        printf("Missing heuristic\n");
                else if(strcmp(argv[2], "h0") == 0)
                        printAstar(x, y, dirt, 0);
                else if(strcmp(argv[2], "h1") == 0)
                        printAstar(x, y, dirt, 1);
                else if(strcmp(argv[2], "h2") == 0)
                        printAstar(x, y, dirt, 2);
                else
                        printf("Unknown heuristic\n");
        }
        //unknown command
        else
                printf("command unknown\n");

        return 0;
}
```

```
#include <stdint.h>

//structure of the list of actions taken
typedef struct list_aux{
        char action;
        struct list_aux *next;
} list_node, *list;

//structure of a list of dirt
typedef struct list_aux_dirt{
        int x;
        int y;
        struct list_aux_dirt *next;
} list_node_dirt, *list_dirt;

//structure of each node in a tree
typedef struct tree_aux{
        long int hashId;
        struct tree_aux *parent;
        struct tree_aux *N;
        struct tree_aux *S;
        struct tree_aux *W;
        struct tree_aux *E;
        char action;
        int dirt;
        int x;
        int y;
        int level;
        char **mat;

} tree_node, *tree;

//structure of a element of the queue
typedef struct queue_aux{
        double f;
        tree t;
} queue_vector;

//global variables
int n, m, n_queue;
int generated, expanded;
char w[1000][1000];
list actions;
queue_vector queue[1000000];
list_dirt dirts;

//functions declaration
void insertList(char a);
void insertListDirt(int x, int y);
void printList();

void insertQueue(tree t, double f);
void removeQueue(tree *t, double *f);
void orderQueue();

char** newMatrix();
void freeMatrix(char **mat);
void copyMatrix(char** from, char** to);
int compareMatrix(char** m1, char** m2);

tree insertTree(tree p, int dirt, int x, int y, char** mat, char a, int level);
void freeTree(tree t);
int checkDuplicate(tree t, char** mat, long hashId, int dirt);
long hashFunction(int x, int y);
uint32_t int_hash(uint32_t v);

int dfs(int x, int y, int dirt, int level, int limit);

double computeHeuristic(int x, int y, int g, int heuristic);
```



```
double computeDistancesDirts(int x, int y, int heuristic);
double euclidean(int x1, int y1, int x2, int y2);
double manhattan(int x1, int y1, int x2, int y2);

tree astar(int x, int y, int d, int heuristic);
void printAstar(int x, int y, int dirt, int heuristic);
```

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

//insert on front of the list
void insertList(char a){
	list new;

	new = malloc(sizeof(list_node));
	if(new == NULL)
		printf("error alocating memory for list node\n");
	new -> next = actions;
	new -> action = a;

	actions = new;
}

//insert on front of the dirts list
void insertListDirt(int x, int y){
	list_dirt new;

	new = malloc(sizeof(list_node_dirt));
	if(new == NULL)
		printf("error alocating memory for dirt node\n");
	new -> next = dirts;
	new -> x = x;
	new -> y = y;

	dirts = new;
}

//removes from the dirt list
void removeListDirt(int x, int y){
	list_dirt aux;

	if(dirts != NULL){
		//checks first node
		if(dirts -> x == x && dirts -> y == y)
			dirts = dirts -> next;

		//checks the rest of the list
		else{
			for(aux = dirts; aux -> next != NULL; aux = aux -> next)
{
				if(aux -> next -> x == x && aux -> next -> y ==
y)
					aux -> next = aux -> next -> next;
					break;
			}
		}
	}
}

//prints a list
void printList(){
	list aux = actions;

	while(aux != NULL){
		printf("%c\n", aux -> action);
		aux = aux -> next;
	}
}

//insert in the end of the queue
void insertQueue(tree t, double f){
	queue[n_queue].t = t;
	queue[n_queue].f = f;
```



```c
	n_queue++;
}

//remove the first element from the queue
void removeQueue(tree *t, double *f){
	int i;

	*t = queue[0].t;
	*f = queue[0].f;
	//shift the elements
	for(i = 0; i < n_queue; i++){
		queue[i].t = queue[i+1].t;
		queue[i].f = queue[i+1].f;

	}

	n_queue--;
}

//creates a new matrix
char** newMatrix(){
	int i;
	char **mat;

	mat = (char **)malloc(n * sizeof(char*));
	if(mat == NULL)
		printf("error allocating memory for a matrix\n");
	for(i = 0; i < n; i++){
		mat[i] = (char *)malloc(m * sizeof(char));
		if(mat[i] == NULL)
			printf("error allocating memory for a matrix\n");
	}

	return mat;
}

//prints a matrix
void printMatrix(char **mat){
	int i, j;

	for(i = 0; i < n; i++){
		for(j = 0; j < m; j++)
			printf("%c ", mat[i][j]);
		printf("\n");
	}
}

//free the space of a matrix
void freeMatrix(char **mat){
	int i;
	for (i = 0; i < n; i++)
		free(mat[i]);

	free(mat);
}

//copy one matrix to another
void copyMatrix(char** from, char** to){
	int i, j;

	for(i = 0; i < n; i++){
		for(j = 0; j < m; j++)
			to[i][j] = from[i][j];
	}
}

//compare matrices
int compareMatrix(char** m1, char** m2){
	int i, j;
```

```
        for(i = 0; i < n; i++){
                for(j = 0; j < m; j++){
                        if(m1[i][j] != m2[i][j])
                                return 0;
                }
        }

        return 1;
}

//Knuth multiplicative method
uint32_t int_hash(uint32_t v)
{
    return v * UINT32_C(2654435761);
}

//hash function to map the position of the robot
long hashFunction(int x, int y){
        return (51 + int_hash(y)) * 51 + int_hash(x);
}

//insert a node in the tree
tree insertTree(tree p, int dirt, int x, int y, char** mat, char a, int level){
        tree new;
        int i;

        new = malloc(sizeof(tree_node));
        if(new == NULL)
                printf("error allocating memory for tree\n");
        new -> parent = p;
        new -> dirt = dirt;
        new -> x = x;
        new -> y = y;
        new -> mat = mat;
        new -> action = a;
        new -> level = level;
        new -> N = NULL;
        new -> S = NULL;
        new -> W = NULL;
        new -> E = NULL;
        new -> hashId = hashFunction(x, y);

        return new;
}

//free a tree
void freeTree(tree t){
        if(t != NULL){
                freeTree(t -> N);
                freeTree(t -> S);
                freeTree(t -> E);
                freeTree(t -> W);
                freeMatrix(t -> mat);
                free(t);
        }
}

//checks if it find a duplicate state
int checkDuplicate(tree t, char** mat, long hashId, int dirt){
        tree new;

        if(t != NULL){
                if(hashId == t -> hashId && dirt == t -> dirt && compareMatrix(m
at, t -> mat) == 1)
                        return 1;
                else{
                        return (checkDuplicate(t -> N, mat, hashId, dirt) == 1)
|| (checkDuplicate(t -> S, mat, hashId, dirt) == 1)
```



```
                        || (checkDuplicate(t -> W, mat, hashId, dirt) == 1) || (
checkDuplicate(t -> E, mat, hashId, dirt) == 1);
                }
        }

        return 0;
}

//order queue  according to f value
void orderQueue(){
        int i, j;
        queue_vector a;

        for(i = 0; i < n_queue; i++){
        for(j = i + 1; j < n_queue; j++){
            if(queue[i].f > queue[j].f){
                a.f = queue[i].f;
                a.t = queue[i].t;
                queue[i].f = queue[j].f;
                queue[i].t = queue[j].t;
                queue[j].f = a.f;
                queue[j].t = a.t;
            }
        }
    }
}

//compute manhattan distance of 2 points
double manhattan(int x1, int y1, int x2, int y2){
        return abs(x1-x2) + abs(y1-y2);
}

//compute euclidean distance of 2 points
double euclidean(int x1, int y1, int x2, int y2){
        return sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
}

//compute distances(manhattan if h2 and euclidean if h1) to all dirts, return th
e smaller
double computeDistancesDirts(int x, int y, int heuristic){
        double smaller, d;
        list_dirt aux;

        smaller = 0;

        if(dirts != NULL){
                if(heuristic == 1)
                        smaller = euclidean(x, y, dirts -> x, dirts -> y);
                else
                        smaller = manhattan(x, y, dirts -> x, dirts -> y);

                for(aux = dirts -> next; aux != NULL; aux = aux -> next){
                        if(heuristic == 1){
                                d = euclidean(x, y, aux -> x, aux -> y);

                                if(d < smaller)
                                        smaller = d;
                        }
                        else{
                                d = manhattan(x, y, aux -> x, aux -> y);

                                if(d < smaller)
                                        smaller = d;
                        }
                }
        }

        return smaller;
}
```

```
//compute the f = g + h function according to the heuristic
double computeHeuristic(int x, int y, int g, int heuristic){
        //h0
        if (heuristic == 0)
                return g;

        //h1 or h2
        else
                return computeDistancesDirts(x, y, heuristic) + g;
}
```



```
#include <stdio.h>
#include <stdlib.h>
#include "functions.h"

/*executes depth first search on the world to find the plan to clean it
it returns 1 if founded a way to collect all dirt, else it returns 0*/
int dfs(int x, int y, int dirt, int level, int limit){
        int instruction = 0; //0 -> U, 1 -> D, 2 -> L, 3 -> R
        int next_x, next_y;
        char temp = w[y][x];

        if(level < limit){
                //increments the number of nodes expanded
                expanded++;

                //calculates the number of nodes that can be generated
                if(y != 0 && w[y-1][x] != '#' && w[y-1][x] != dirt + '0')
                        generated++;
                if(y != n - 1 && w[y+1][x] != '#' && w[y+1][x] != dirt + '0')
                        generated++;
                if(x != 0 && w[y][x-1] != '#' && w[y][x-1] != dirt + '0')
                        generated++;
                if(x != m - 1 && w[y][x+1] != '#' && w[y][x+1] != dirt + '0')
                        generated++;
                if(w[y][x] == '*')
                        generated++;

                //vacuum the dirt
                if(w[y][x] == '*'){
                        dirt--;
                        expanded++;

                        //check to see if it is the last one
                        if(dirt == 0){
                                insertList('V');
                                return 1;
                        }
                }

                //changes the cell value to the number of dirt left
                w[y][x] = dirt + '0';

                while(instruction < 4){
                        //calculate the next cell to be seen acording to the ins
truction
                        next_y = y;
                        next_x = x;

                        if(instruction == 0 && y != 0 && w[y-1][x] != '#')
                                next_y = y - 1;
                        else if(instruction == 1 && y != n - 1 && w[y+1][x] != '
#')
                                next_y = y + 1;
                        else if(instruction == 2 && x != 0 && w[y][x-1] != '#')
                                next_x = x - 1;
                        else if(instruction == 3 && x != m - 1 && w[y][x+1] != '
#')
                                next_x = x + 1;

                        //it just makes sense to pass through the same cell if t
he number of dirt left in the second time is different than the first
                        if((next_y != y || next_x != x) && w[next_y][next_x] !=
w[y][x]){
                                if(dfs(next_x, next_y, dirt, level + 1, limit) =
= 1){
                                        if(instruction == 0)
                                                insertList('N');
                                        else if(instruction == 1)
                                                insertList('S');
```

```
                                        else if(instruction == 2)
                                                insertList('W');
                                        else if(instruction == 3)
                                                insertList('E');

                                        if(temp == '*')
                                                insertList('V');

                                        return 1;
                                }
                        }

                                instruction++;
                }

                w[y][x] = temp;
                return 0;
        }

        return 0;
}
```



```
#include <stdio.h>
#include <stdlib.h>
#include "functions.h"

/*implements A* algorithm*/
tree astar(int x, int y, int d, int heuristic){
        char a = 'X';//X simbolize that no action was taken to get there
        tree t, t1, t2;
        char **mat1, **mat2;
        int i, j;
        int level;
        double f;
        queue_vector aux;
        int k;

        //copy the starting matrix
        mat1 = newMatrix();
        for(i = 0; i < n; i++){
                for(j = 0; j < m; j++)
                        mat1[i][j] = w[i][j];
        }

        t1 = insertTree(NULL, d, x, y, mat1, a, 0);
        t = t1;

        //compute the f value according to the heuristic
        f = computeHeuristic(x, y, 0, heuristic);

        insertQueue(t1, f);

        while(n_queue > 0){
                //remove the first element of the queue
                removeQueue(&aux.t, &aux.f);
                t1 = aux.t;

                d = t1 -> dirt;
                mat1 = t1 -> mat;
                a = t1 -> action;
                x = t1 -> x;
                y = t1 -> y;
                level = t1 -> level;

                //increments the number of nodes expanded
                expanded++;

                //adds into the queue
                if(y != 0 && mat1[y-1][x] != '#'){
                        mat2 = newMatrix();
                        copyMatrix(mat1, mat2);

                        a = mat1[y-1][x];

                        mat2[y-1][x] = '@';
                        mat2[y][x] = '_';

                        if(a == '*')
                                k = d-1;
                        else
                                k = d;

                        if(checkDuplicate(t, mat2, hashFunction(x, y-1), k) == 0
){
                                //inserts in lowercase if there is dirt
                                if(a == '*'){
                                        removeListDirt(x, y-1);
                                        t2 = insertTree(t1, d-1, x, y-1, mat2, '
n', level + 1);

                                        if(d-1 == 0)
```

```
                                        return t2;
                                }
                                else
                                        t2 = insertTree(t1, d, x, y-1, mat2, 'N
', level + 1);

                                t1 -> N = t2;

                                //compute the f value according to the heuristic
                                f = computeHeuristic(x, y-1, level+1, heuristic)
;

                                insertQueue(t2, f);
                                generated++;
                        }
                }

                if(y != n - 1 && mat1[y+1][x] != '#'){
                        mat2 = newMatrix();
                        copyMatrix(mat1, mat2);

                        a = mat1[y+1][x];

                        mat2[y+1][x] = '@';
                        mat2[y][x] = '_';

                        if(a == '*')
                                k = d-1;
                        else
                                k = d;

                        if(checkDuplicate(t, mat2, hashFunction(x, y+1), k) == 0
){
                                //inserts in lowercase if there is dirt
                                if(a == '*'){
                                        removeListDirt(x, y+1);
                                        t2 = insertTree(t1, d-1, x, y+1, mat2, '
s', level + 1);

                                        if(d-1 == 0)
                                                return t2;
                                }
                                else
                                        t2 = insertTree(t1, d, x, y+1, mat2, 'S'
, level + 1);

                                t1 -> S = t2;

                                //compute the f value according to the heuristic
                                f = computeHeuristic(x, y-1, level+1, heuristic)
;

                                insertQueue(t2, f);
                                generated++;
                        }
                }

                if(x != 0 && mat1[y][x-1] != '#'){
                        mat2 = newMatrix();
                        copyMatrix(mat1, mat2);

                        a = mat1[y][x-1];

                        mat2[y][x-1] = '@';
                        mat2[y][x] = '_';

                        if(a == '*')
                                k = d-1;
                        else
```



```
                                k = d;

                        if(checkDuplicate(t, mat2, hashFunction(x - 1, y), k) ==
 0){
                                //inserts in lowercase if there is dirt
                                if(a == '*'){
                                        removeListDirt(x-1, y);
                                        t2 = insertTree(t1, d-1, x-1, y, mat2, '
w', level + 1);

                                        if(d-1 == 0)
                                                return t2;
                                }
                                else
                                        t2 = insertTree(t1, d, x-1, y, mat2, 'W
', level + 1);

                                t1 -> W = t2;

                                //compute the f value according to the heuristic
                                f = computeHeuristic(x, y-1, level+1, heuristic)
;

                                insertQueue(t2, f);
                                generated++;
                        }
                }

                if(x != m - 1 && mat1[y][x+1] != '#'){
                        mat2 = newMatrix();
                        copyMatrix(mat1, mat2);

                        a = mat1[y][x+1];

                        mat2[y][x+1] = '@';
                        mat2[y][x] = '_';

                        if(a == '*')
                                k = d-1;
                        else
                                k = d;

                        if(checkDuplicate(t, mat2, hashFunction(x + 1, y), k) ==
 0){
                                //inserts in lowercase if there is dirt
                                if(a == '*'){
                                        removeListDirt(x+1, y);
                                        t2 = insertTree(t1, d-1, x+1, y, mat2, '
e', level + 1);

                                        if(d-1 == 0)
                                                return t2;
                                }
                                else
                                        t2 = insertTree(t1, d, x+1, y, mat2, 'E
', level + 1);

                                t1 -> E = t2;

                                //compute the f value according to the heuristic
                                f = computeHeuristic(x, y-1, level+1, heuristic)
;

                                insertQueue(t2, f);
                                generated++;
                        }
                }

                orderQueue();
```

```
        }

        return NULL;
}

//prints the path of the a-star
void printAstar(int x, int y, int dirt, int heuristic){
        tree t;

        if(dirt > 0){
                t = astar(x, y, dirt, heuristic);

                if(t == NULL)
                        printf("no solution found\n");

                else{
                        //inserts the actions in a list
                        while(t -> action != 'X'){
                                //lowercase mean it can vacuum
                                if(t -> action > 96){
                                        insertList('V');
                                        t -> action = t -> action - 32;
                                }

                                //inserts in the list
                                insertList(t -> action);

                                t = t -> parent;
                        }

                        //free the memory
                        freeTree(t);

                        //print it
                        printList();
                        printf("%d nodes generated\n", generated);
                        printf("%d nodes expanded\n", expanded);
                }
        }
        //no dirt in the matrix, there is nothing to do
        else{
                printf("%d nodes generated\n", generated);
                printf("%d nodes expanded\n", expanded);
        }
}
```