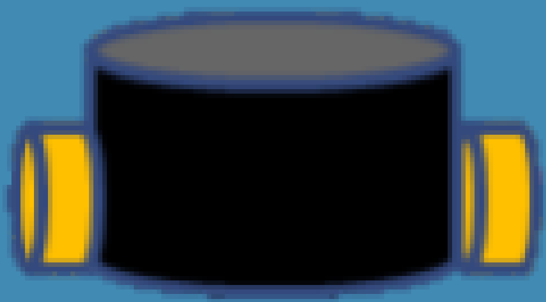
HOOVI PROGRAM REPORT



Oscar Bolton

CandNo: 253047

Introduction

The purpose of this program is to find the shortest path to the charger room from any room in which Hoovi operates. To achieve this a selection of functions were created, which work together achieving a correct and efficient result.

These are as follows:

- **generateSuccessors** – This function allows the program to understand where Hoovi can travel from any room in the building, considering various parameters. It then calls on the addToFringe function to evaluate these states.
- **addToFringe –** This function evaluates whether a state hasBeenVisited, and if not calls on the push function to 'push' the state to the waiting queue.
- **push** – this function 'pushes' a state (room) to the front of the queue
- **pop / pop_dfs –** pop retrieves the next state (room) in the waiting queue. Pop (bfs) takes from the front – FIFO. Pop_dfs takes from the rear – LIFO.
- **hasBeenVisited –** this function uses the equalStates function to check whether a state/room has been visited. If so, the function returns 1 (true), if not it returns 0 (false).
- **equalStates –** A function to check whether 2 states are the same.
- **isGoalState –** A simple function that checks whether the current state is the goal state. If so, the function returns 1 (true), if not it returns 0 (false).
- **printSolution –** A recursive function that uses parent index to track back from the goal state to the initial state and therefore print the solution path.

Program description

The program effectively solves the problem, answering all the marking criteria.

The state representation is shown in the struct, 'state' (lines 17 – 23). Given rooms are in the form ab.c: char a is the room letter indicator; int b is the floor level; int c is the room number indicator, where 2 = office & 1 = reception. I made the decision to convert the lobby and lift on each floor, and the charger room into this format: Lobby = Ib.0; Lift = Jb.0; Charger room = X1.0. This allows every room to be covered by the state representation, and a simple transition between rooms in the successor function. The final component of the struct is int p, which represents the parent index. A state's parent index corresponds to the visited array index of its parent. This means the program can trace its path from when it finds the goal, back to the initial state.

The initial state is where I allow the user to input any room in the floor plan in the form %c%d.%d, while not allowing any invalid input (lines 62 – 76). These values are assigned to three independent variables which are then assigned to the struct values in initialState on line 77. The parent index (s.p) = -1, as the initialState has no parent states.

The successor function (lines 243 – 308) generates all the possible paths from any given room in a non-trivial manner, which allows us to solve the problem of reaching the charger room. It would also allow HOOVI to search the building for its routine job as it includes every room in the building in the search. The successor function evaluates the possible states than can be reached from a particular state and passes them through the addToFringe function. This uses the hasBeenVistited function and therefore equalStates function to check whether we have visited this state already. If not, it will push the state to the waiting queue. The main search function will then use the pop function to take the state next in line in the waiting queue. If this state is not the goal. The successor function will run again.

The BFS strategy uses all the functions mentioned to achieve the solution. The key characteristic of the BFS is the first in first out ordering of the waiting queue. This means that the first state to enter the waiting queue is the first to be visited and analysed. This is achieved in the program by the pop function (219 – 228), which in turn uses the queue array, and in this case the front index counter. Pop will take the state which is at the front index then increment the index by one for the next pass. The program will then analyse the state and if it is not the goal will run the successor function, and afterwards, pop will once again take the state at the front of the queue.

The DFS strategy also uses the main functions, but this time uses the pop_dfs function (231 – 240). The difference here is that instead of taking the state at the front on the queue, pop_dfs uses the rear index to take the state at the rear of the queue. This satisfies the last in first out characteristic of DFS.

I believe for this problem the DFS strategy is the best suited for this task. Because there is only ever one goal and one solution, and the path must travel through the lobby to the lift to the charger room, the successor function can be ordered in such a way to optimise the search from any room to the charger room. Ensuring that forward steps are always at the rear of the waiting queue whenever the search function pops the next state means the program can find the solution without any redundant searches. This can be seen in my code. However, if parameters were to change and the floor plan allowed more than 1 solution then DFS has its drawbacks. Because states can only be visited once, if a path found included a state that had another, potentially shorter, path to the goal, the program wouldn't be able to find that solution as we have already visited that state. In addition, if the problem were to change and the goal state could change from use to use, the successor function could no longer be ordered in an optimum way for each goal, resulting in redundant searches in some cases.

The main function incorporates these functions with some other important functions and variables to solve the problem. The key variable so the program can find the path is the parentIndex. When a state gets popped and added to the visited queue it receives an array index which is tracked by the variable vfront, starting at 0. (Line 356). The parentIndex then gets assigned this value (line 137(BFS) / 173(DFS)), which then gets pushed into the successor function (155 / 191). This value then gets assigned to the p variable of every successor that's added to the waiting queue. Therefore, each states' p value = its parents index in the visited queue. This cycles until the goal is found and allows the program to find the path needed to reach the goal. The printSolution function uses a recursive call on itself, utilising the parent index. Using the p value of each state it finds its parent by taking the same value to visited queue index.

A representation of how this works can be seen in the table below, which shows a BFS search from room A1.2 to the charger room:

visited	A1.2	A1.1	I1.0	J1.0	H1.1	G1.1	F1.1	E1.1	D1.1	C1.1	B1.1	K1.1	L1.1	M1.1	N1.1	J2.0	X1.0
array index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
parentIndex (s.p)	-1	0	1	2	2	2	2	2	2	2	2	2	2	2	2	3	3

A1.2 -> A1.1 -> LOBBY 1.0 -> LIFT 1.0 -> CHARGER ROOM

The program can find solutions for the initial state scenarios given in the task brief using both BFS and DFS:

I) in room K3.2:

a. BFS:

```
*** WELCOME TO THE HOOVI INTERFACE ***

!!! HOOVI's battery level is critically low !!!
Please return to the charger room immediately.

Please enter your location
(Ib.0 = the lobby)
(Jb.0 = the lift)
(X1.0 = the charger room)
(Where b = floor level (1, 2, 3))
(All other room names are as the floor plan states).
: K3.2

Our starting position is K3.2.
Our goal position is the CHARGER ROOM (X1.0).

Would you like to receive detailed status messages regarding HOOVI's search activity? Y/N
N

Would you like to implement a Breadth First Search (BFS) or a Depth First Search (DFS)? BFS/DFS
BFS
...... searching for the fastest route using BFS ......


I found a solution (42 states explored):  K3.2 -> K3.1 -> LOBBY 3.0 -> LIFT 3.0 -> LIFT 2.0 -> LIFT 1.0 -> CHARGER ROOM


Searching remaining states...


No further solutions found. - (There should always only ever be one with this floor plan).
```

b. DFS:

```
*** WELCOME TO THE HOOVI INTERFACE ***

!!! HOOVI's battery level is critically low !!!
Please return to the charger room immediately.

Please enter your location
(Ib.0 = the lobby)
(Jb.0 = the lift)
(X1.0 = the charger room)
(Where b = floor level (1, 2, 3))
(All other room names are as the floor plan states).
: K3.2

Our starting position is K3.2.
Our goal position is the CHARGER ROOM (X1.0).

Would you like to receive detailed status messages regarding HOOVI's search activity? Y/N
N

Would you like to implement a Breadth First Search (BFS) or a Depth First Search (DFS)? BFS/DFS
DFS
...... searching for the fastest route using DFS ......


I found a solution (6 states explored):  K3.2 -> K3.1 -> LOBBY 3.0 -> LIFT 3.0 -> LIFT 2.0 -> LIFT 1.0 -> CHARGER ROOM

Searching remaining states...


No further solutions found. - (There should always only ever be one with this floor plan).
```

II) In the lift on the second floor:

a. BFS:

```
*** WELCOME TO THE HOOVI INTERFACE ***

!!! HOOVI's battery level is critically low !!!
Please return to the charger room immediately.

Please enter your location
(Ib.0 = the lobby)
(Jb.0 = the lift)
(X1.0 = the charger room)
(Where b = floor level (1, 2, 3))
(All other room names are as the floor plan states).
: J2.0

Our starting position is LIFT 2.0.
Our goal position is the CHARGER ROOM (X1.0).

Would you like to receive detailed status messages regarding HOOVI's search activity? Y/N
N

Would you like to implement a Breadth First Search (BFS) or a Depth First Search (DFS)? BFS/DFS
BFS
...... searching for the fastest route using BFS ......


I found a solution (18 states explored):  LIFT 2.0 -> LIFT 1.0 -> CHARGER ROOM

Searching remaining states...


No further solutions found. - (There should always only ever be one with this floor plan).
```

b. DFS:

```
*** WELCOME TO THE HOOVI INTERFACE ***

!!! HOOVI's battery level is critically low !!!
Please return to the charger room immediately.

Please enter your location
(Ib.0 = the lobby)
(Jb.0 = the lift)
(X1.0 = the charger room)
(Where b = floor level (1, 2, 3))
(All other room names are as the floor plan states).
: J2.0

Our starting position is LIFT 2.0.
Our goal position is the CHARGER ROOM (X1.0).

Would you like to receive detailed status messages regarding HOOVI's search activity? Y/N
N

Would you like to implement a Breadth First Search (BFS) or a Depth First Search (DFS)? BFS/DFS
DFS
...... searching for the fastest route using DFS ......


I found a solution (2 states explored):  LIFT 2.0 -> LIFT 1.0 -> CHARGER ROOM

Searching remaining states...


No further solutions found. - (There should always only ever be one with this floor plan).
```

III) In room F1.1:

a. BFS:

```
*** WELCOME TO THE HOOVI INTERFACE ***

!!! HOOVI's battery level is critically low !!!
Please return to the charger room immediately.

Please enter your location
(Ib.0 = the lobby)
(Jb.0 = the lift)
(X1.0 = the charger room)
(Where b = floor level (1, 2, 3))
(All other room names are as the floor plan states).
: F1.1

Our starting position is F1.1.
Our goal position is the CHARGER ROOM (X1.0).

Would you like to receive detailed status messages regarding HOOVI's search activity? Y/N
N

Would you like to implement a Breadth First Search (BFS) or a Depth First Search (DFS)? BFS/DFS
BFS
...... searching for the fastest route using BFS ......


I found a solution (27 states explored):  F1.1 -> LOBBY 1.0 -> LIFT 1.0 -> CHARGER ROOM

Searching remaining states...


No further solutions found. - (There should always only ever be one with this floor plan).
```

b. DFS

```
*** WELCOME TO THE HOOVI INTERFACE ***

!!! HOOVI's battery level is critically low !!!
Please return to the charger room immediately.

Please enter your location
(Ib.0 = the lobby)
(Jb.0 = the lift)
(X1.0 = the charger room)
(Where b = floor level (1, 2, 3))
(All other room names are as the floor plan states).
: F1.1

Our starting position is F1.1.
Our goal position is the CHARGER ROOM (X1.0).

Would you like to receive detailed status messages regarding HOOVI's search activity? Y/N
N

Would you like to implement a Breadth First Search (BFS) or a Depth First Search (DFS)? BFS/DFS
DFS
...... searching for the fastest route using DFS ......


I found a solution (3 states explored):  F1.1 -> LOBBY 1.0 -> LIFT 1.0 -> CHARGER ROOM

Searching remaining states...


No further solutions found. - (There should always only ever be one with this floor plan).
```

Appendix

```
/*
HOOVI is an automated vacuum cleaner which operates in an office building.
The building has three floors with a lobby as well as 12 offices and 12 reception rooms on each floor.
There is also a lift for travelling to the next floor below or above.

HOOVI is battery powered and can only be recharged at the charging room on the ground floor.
Once HOOVI's battery level becomes critically low, it must return to the charging room immediately.
This means that HOOVI could be any room on any floor when it must return to the charging room.

The algorithm below allows the user to find the shortest path to the charging room from any room in the building
*/

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

typedef struct // state representation
{
    char a; //room indicator
    int b; //floor inticator
    int c; //office or reception indicator
    int p; // parent index
}state;


//Variables at global scope
int vfront = -1; // next pos in the visited list
int front = 0; // front pos of the queue
int rear = -1; // rear pos of the queue   //front and rear can be seen as the scope of the queue/array. Shifting up and down as states get added and visited.
int stateCount = 0; // amount of rooms in the waiting list
int searchCost = 0; // a counter to measure the number of search iterations
state queue[1000]   = {};   // This is the waiting queue - we put all states yet to be examined in here (BFS = FIFO | DFS = LIFO)
state visited[1000] = {};   // This is the visited states list - we put all states we have already examined in here
int i;
int verbatim = 0; // switch this to 1 to give detailed messages of the search process
char v; // verbatim Y/N?


//Functions
void addToFringe(char, int, int, int);
int equalStates(state, state);
void generateSuccessors(state, int);
int hasBeenVisited(state);
int isGoalState(state);
void printSolution(state);
state pop();
```

```c
    state pop_dfs();
    void push(state);

    int main()
    {
        printf("*** WELCOME TO THE HOOVI INTERFACE *** \n\n");

        printf("!!! HOOVI's battery level is critically low !!!\n");
        printf("Please return to the charger room immediately. \n\n");

        //Initial state - user input
        char initial_a;
        int initial_b;
        int initial_c = -1;

        while((initial_a < 65 || initial_a > 78) || (initial_b < 1 || initial_b > 3) || (initial_c < 0 || initial_c > 2) || (initial_a < 73 && initial_c == 0) || (initial_a > 74 && initial_c == 0) || (initial_a == 73 && initial_c != 0) || (initial_a == 74 && initial_c != 0))
        {
            printf("Please enter your location\n");
            printf("(Ib.0 = the lobby)\n");
            printf("(Jb.0 = the lift)\n");
            printf("(X1.0 = the charger room)\n");
            printf("(Where b = floor level (1, 2, 3))\n");
            printf("(All other room names are as the floor plan states).\n: ");
            scanf("%c%d.%d", &initial_a, &initial_b, &initial_c);
            if((initial_a < 65 || initial_a > 78) || (initial_b < 1 || initial_b > 3) || (initial_c < 0 || initial_c > 2) || (initial_a < 73 && initial_c == 0) || (initial_a > 74 && initial_c == 0) || (initial_a == 73 && initial_c != 0) || (initial_a == 74 && initial_c != 0))
            {
                printf("Invalid input.\n\n");
                fflush(stdin); //clearing invalid input from storage buffer
            }
        }
        state initialState = {initial_a, initial_b, initial_c, -1}; //Initial state set
        state s; //current state
        int parentIndex = 0;

        if(initialState.a == 'I')
        {
            printf("\nOur starting position is LOBBY %d.%d. \n", initialState.b, initialState.c);
        }
        else if(initialState.a == 'J')
        {
            printf("\nOur starting position is LIFT %d.%d. \n", initialState.b, initialState.c);
        }
        else
        {
            printf("\nOur starting position is %c%d.%d. \n", initialState.a, initialState.b, initialState.c);
        }

        printf("Our goal position is the CHARGER ROOM (X1.0). \n\n");
        getchar();
```

```
        // Detailed search messages?
        printf("Would you like to receive detailed status messages regarding HOOVI's search activity? Y/N \n");
        while((v != 'Y') && (v != 'y') && (v != 'N') && (v != 'n'))
        {
            scanf("%c", &v);
            if(v == 'Y' || v == 'y')
            {
                verbatim = 1;
            }
            else if(v == 'N' || v == 'n')
            {
                verbatim = 0;
            }
            else
            {\
                printf("Invalid input. Y/N\n");
                fflush(stdin);
            }
        }

        // Breadth First Search or Depth First Search?
        char search[20];
        while((strcmp(search, "BFS")) != 0 && (strcmp(search, "bfs")) != 0 && (strcmp(search, "DFS")) != 0 && (strcmp(search, "dfs")) != 0)
        {
            getchar();
            printf("\nWould you like to implement a Breadth First Search (BFS) or a Depth First Search (DFS)? BFS/DFS \n");
            scanf("%s", search);

            if((strcmp(search, "BFS")) == 0 || (strcmp(search, "bfs")) == 0)
            {
                printf("...... searching for the fastest route using BFS ...... \n\n");

               // Search - we are trying to explore states as long as there are any left in the queue

                push(initialState);             // add initial state to the "waiting" queue
                while(stateCount > 0)
                {
                    // GET NEXT STATE
                    s = pop();                      // get a state from the front of the queue
                    if(verbatim) printf("Retrieving %c%d.%d from the WAITING queue.\n",s.a, s.b, s.c);
                    parentIndex = addToVisited(s);  // add this state to the visited list and retrieve storage index. parentIndex = vfront index (position in array)
                    if(verbatim) getchar();

                    // GOAL TEST
                    if(isGoalState(s))
                    {
                       // if the current state is the goal, then print the solution
```

```
                    if(verbatim) printf("%c%d.%d is the goal state!\n", s.a, s.b, s.c);
                    printf("\nI found a solution (%d states explored): ", searchCost);
                    printSolution(s);
                    // Wait for key press
                    getchar();
                    printf("\n\nSearching remaining states...\n");
                }
                // if current state s is not the goal, then run successor function
                else
                {
                    if(verbatim) printf("%c%d.%d is not the goal - I need to run the successor function...\n\n",s.a , s.b, s.c);
                    generateSuccessors(s, parentIndex);   // generate the children of s, and make them remember s as their parent
                }
                // increment search iterations counter
                searchCost++;
            }
        }
        else if((strcmp(search, "DFS")) == 0 || (strcmp(search, "dfs")) == 0)
        {
            printf("...... searching for the fastest route using DFS ...... \n\n");

           // Search - we are trying to explore states as long as there are any left in the queue

            push(initialState);             // add initial state to the "waiting" queue
            while(stateCount > 0)
            {
                // GET NEXT STATE
                s = pop_dfs();                          // get a state from the rear of the queue
                      if(verbatim) printf("\nRetrieving %c%d.%d from the WAITING queue.\n",s.a, s.b, s.c);
                parentIndex = addToVisited(s);  // add this state to the visited list and retrieve storage index. parentIndex = vfront index (position in array)
                if(verbatim) getchar();

                // GOAL TEST
                if(isGoalState(s))
                {
                    // if the current state is the goal, then print the solution
                    if(verbatim) printf("%c%d.%d is the goal state!\n", s.a, s.b, s.c);
                    printf("\nI found a solution (%d states explored): ", searchCost);
                    printSolution(s);
                    // Wait for key press
                    getchar();
                    printf("\n\nSearching remaining states...\n");
                }
                // if current state s is not the goal, then run successor function
```

```
                else
                {
                    if(verbatim) printf("%c%d.%d is not the goal - I need to run the successor function...\n\n",s.a , s.b, s.c);
                    generateSuccessors(s, parentIndex);   // generate the children of s, and make them remember s as their parent
                }
                // increment search iterations counter
                searchCost++;
            }
        }
        else
        {
            printf("Invalid input. \n");
        }
    }
    printf("\n\nNo further solutions found. - (There should always only ever be one with this floor plan).");
    getchar();
    return 0;
}


//SEARCH FUNCTIONS

// push adds a state to the rear of the waiting queue
void push(state s)
{
     rear++;                        // increase rear index
     queue[rear] = s;               // store s - storing the next possible states in the queue.
     stateCount++;                  // increase the count of states in the queue
}

// pop retrieves a state from the front of the waiting queue
state pop()
{
    if(stateCount > 0)
    {           // check if there are items in the queue
        state s = queue[front];      // get state at front index
        front++;                    // increase front index to point at the next state
        stateCount--;                // decrement state counter
        return s;                    // pass state back to the point of call
    }
}

// pop_dfs retrieves a state from the rear of the waiting queue
state pop_dfs()
{
    if(stateCount > 0)
    {           // check if there are items in the queue
        state s = queue[rear];       // get state at rear index
        rear--;                     // decrease rear index to point at the next state
        stateCount--;                // decrement state counter
        return s;                    // pass state back to the point of call
    }
```

```
    }

        //Successor function
        void generateSuccessors(state s, int p) // p = parentIndex = vfront index = pos in visited queue of parent state
        {
            if(s.c == 2)
            {
                if(verbatim) printf("Hoovi could travel from office %c%d.%d to reception %c%d.%d.\n", s.a, s.b, s.c, s.a, s.b, s.c - 1);
                addToFringe(s.a, s.b, s.c - 1, p);    // move from office to reception
            }

            if(s.c == 1)
            {
                if(verbatim) printf("Hoovi could travel from reception %c%d.%d to office %c%d.%d.\n", s.a, s.b, s.c, s.a, s.b, s.c + 1);
                addToFringe(s.a, s.b, s.c + 1, p); // move from reception to office
                if(verbatim) printf("Hoovi could travel from reception %c%d.%d to LOBBY %d.%d.\n", s.a, s.b, s.c, s.b, s.c - 1);
                addToFringe(s.a = 'I', s.b, s.c - 1, p); //move from reception to lobby (I = lobby)
            }

            if((s.a == 'I') && (s.c == 0))
            {
                for(i = 1; i < 9; i++)
                {
                    if(verbatim) printf("Hoovi could travel from LOBBY %d.%d to reception %c%d.%d.\n", s.b, s.c, s.a - i, s.b, s.c + 1);
                    addToFringe(s.a - i, s.b, s.c + 1, p); //move 'left' from lobby to reception -> rooms A, B, C, D, E, F, G, H.
                }

                for(i = 2; i < 6; i++)
                {
                    if(verbatim) printf("Hoovi could travel from LOBBY %d.%d to reception %c%d.%d.\n", s.b, s.c, s.a + i, s.b, s.c + 1);
                    addToFringe(s.a + i, s.b, s.c + 1, p); //move 'right' from lobby to reception (skipping lift) -> rooms K, L, M, N.
                }

                if(verbatim) printf("Hoovi could travel from LOBBY %d.%d to LIFT %d.%d.\n", s.b, s.c, s.b, s.c);
                addToFringe(s.a + 1, s.b, s.c, p); //move from lobby to lift   (J = lift)

            }

            if((s.a == 'J') && (s.c == 0))
            {
                if(verbatim) printf("Hoovi could travel from LIFT %d.%d to LOBBY %d.%d.\n", s.b, s.c, s.b, s.c);
                addToFringe(s.a - 1, s.b, s.c, p); //move from lift to lobby
            }

            if((s.a == 'J') && (s.b < 3) && (s.c == 0))
            {
```

```
            if(verbatim) printf("Hoovi could travel up from LIFT %d.%d to LIFT %d.%d.\n", s.b, s.c, s.b + 1, s.c);
            addToFringe(s.a = 'J', s.b + 1, s.c = 0, p); // move up the lift
        }

        if((s.a == 'J') && (s.b > 1) && (s.c == 0))
        {
            if(verbatim) printf("Hoovi could travel down from LIFT %d.%d to LIFT %d.%d.\n", s.b, s.c, s.b - 1, s.c);
            addToFringe(s.a = 'J', s.b - 1, s.c = 0, p); // move down the lift
        }

        if(s.a == 'J' && s.b == 1 && s.c == 0)
        {
            if(verbatim) printf("Hoovi could travel from LIFT %d.%d to the CHARGER ROOM.\n", s.b, s.c);
            addToFringe(s.a + 14 , s.b = 1, s.c = 0, p); // move from lift J1.0 to charger room X1.0
        }

        if(s.a == 'X' && s.b == 1 && s.c == 0)
        {
            if(verbatim) printf("Hoovi could travel from the CHARGER ROOM to LIFT %d.%d.\n", s.b, s.c);
            addToFringe(s.a - 14, s.b = 1, s.c = 0, p); //move from charger room X1.0 to lift J1.0
        }

    }


// Takes a state as input and checks if this state is the goal state
// Returns 1 if this is so, and 0 if the state is not the goal
int isGoalState(state s)
{
    if(s.a == 'X' && s.b == 1 && s.c == 0)  // We are looking for charger room on the ground floor. (X1.0 = charger room)
        return 1;
    else
        return 0;
}

// OUTPUT FUNCTION
void printSolution(state s)
{
    if(s.p != -1)           // Check if we reached the root state
        printSolution(visited[s.p]);    // Recursive call to the parent of s. s.p = pos in visited queue of parent state.
    if(s.a == 'X')
    {
        printf(" CHARGER ROOM ");
    }
    else if(s.a == 'I')
    {
        printf(" LOBBY %d.%d ", s.b, s.c);
    }
    else if(s.a == 'J')
    {
```

```
            printf(" LIFT %d.%d ", s.b, s.c);
        }
        else
        {
        printf(" %c%d.%d ", s.a, s.b, s.c);
        }
        if(!isGoalState(s))     // print arrows if not the goal (i.e., last) state
            printf("->");
         return;
    }


    // UTILITY FUNCTIONS

    // addToVisited takes a state as an argument and adds it to the visited list
    // returns the position in the list  at which the state was stored
    // (we need this information for maintaining parent links)
    int addToVisited(state s)
    {
        vfront++;               // increase list index - creating a 'free' index. All indexes before are assigned to ther states. All indexes after are unassigned.
        visited[vfront] = s;    // add state at 'free' index
            if(verbatim)
            {
                printf("Adding %c%d.%d to the VISITED queue at index %d.\n", s.a, s.b, s.c, vfront);
                printf("Returning %d as the parent index for this state.\n",vfront);
            }
        return vfront;          // return storage index for s
    }

    // equalStates takes two states as input and compares their internal variables a, b and c.
    // if both a, both b and both c values are equal, this function will return 1, otherwise 0;
    int equalStates(state s1, state s2)
    {
        if(s1.a == s2.a && s1.b == s2.b && s1.c == s2.c)
            return 1;
        else
            return 0;
    }

    // hasBeenVisited takes a state as input and compares it to all states stored in the "visited" list
    // returns 1 if the state has been visited before
    // returns 0 if the state has not been visited before
    int hasBeenVisited(state s)
    {
        int i;
        for(i=0; i<vfront; i++) // loops until the vfront index is reached - checking every state in the visited queue.
        {
            if(equalStates(visited[i],s)) // checking visted states against current state.
            {
```

```
                if(verbatim) printf("But we have already visited %c%d.%d!\n\n", s.a, s.b, s.c);
                return 1; // has been visted
            }

        }
        return 0; // has not been visted
    }


    //addToFringe takes a state as input and checks if this state has not been explored yet
    // If the state was not previously visited, then we recognise the state to be "at the fringe" of its parent and add it to the waiting queue
    // otherwise, the function returns to the point of call without doing anything

    void addToFringe(char a, int b, int c, int p)
    {
        state s = {a, b, c, p};
        if(!hasBeenVisited(s))      // check if s was visited before
        {
            push(s);            // if not, then add to waiting queue
            if(verbatim) printf("Adding %c%d.%d to the WAITING queue. It's parent is at VISITED index %d.\n\n", s.a, s.b, s.c, s.p);
        }
        return;
    }

```