

YILDIZ TEKNİK ÜNİVERSİTESİ
ELEKTRİK ELEKTRONİK FAKÜLTESİ / BİLGİSAYAR
MÜHENDİSLİĞİ BÖLÜMÜ

2021-2022 Bahar Yarıyılı



BLM2512 Veri Yapıları ve Algoritmalar
Proje Ödevi

HAZIRLAYAN:
ÖMER TALHA BAYSAN
18011103

5 Haziran 2022

Doç. Dr. Mehmet Amaç GÜVENSAN
Veri Yapıları ve Algoritmalar

Verilen labirent üzerinde başlangıç hücresinden bitiş hücresine yolu bulan algoritmayı tasarlayınız ve yapılan gezintiyi ekranda animasyon şeklinde gösteren programı yazınız. Gezintide yaptığınız tüm hareketleri (yanlış girdiğiniz, geri döndüğünüz) ve anlık puanınızı göstermelisiniz. İşlem adımlarını gösterirken yavaşlatmak için araya makul miktar bekleme süresi (100ms gibi) koyunuz. Çıkışa ulaştığınızda sonuç puanını gösteriniz.

NxN hücreden oluşan labirenti temsil için bir matris kullanınız. Labirent matrisinde duvarları 0, açık yolları 1 ile gösteriniz. Örneğin matrisin [1][2] değeri 0 ise 1. satır 2. sütundaki hücre duvardır. Her hücrenin kuzey, güney, doğu ve batı yönünde toplam 4 komşusu vardır. Matrisin kenar hücrelerinin de 4 komşusu olabilmesi için (N+1)x(N+1)'lik bir matris tanımlayıp kenar hücreler için işlem yapmayınız.

Gerçekleştirdiğim Çözüm:

Labirentteki yapıları bir enum yapısında belirtiyoruz.

```
enum terrain {  
    empty,  
    wall,  
    goal,  
    apple,  
    crumb  
};
```

empty	labirentteki boş yerleri temsil ediyoruz.
wall	labirentteki duvarları temsil ediyoruz.
goal	labirentteki hedefi(çıkışı) temsil ediyoruz.
apple	labirentteki elmaları temsil ediyoruz.
crumb	labirentte ilerlerken ki bıraktığımız kırıntıları temsil ediyoruz.

Labirenti iki boyutlu bir matriste tutacağız. Bu yüzden iki boyutlu bir matris tahsis ediyoruz.

```
char** alloc_maze(int rows, int cols)  
{  
    char** maze;  
    maze = malloc(rows * sizeof(char*));  
  
    int i;  
    for (i = 0; i < rows; ++i){  
        maze[i] = malloc(cols * sizeof(char*));  
    }  
    return maze;  
}
```

Labirentte gezdiğimiz yerleri tutmak için ayrı bir matris tahsis ediyoruz.

```
int** alloc_visited(int rows, int cols)  
{  
    int** visited;  
    visited = malloc(rows * sizeof(int*));  
    int i;  
    for (i = 0; i < rows; ++i){  
        visited[i] = malloc(cols * sizeof(int*));  
    }  
    return visited;  
}
```

Adı, satır ve sütun değerleri fonksiyondan alınan dosyayı, labirentimize tahsis edilen maze matrisine yazıyoruz. En son matrisi geri döndürüyoruz.

```
char** get_maze(char* file_name, int rows, int cols)
{
    char c;

    FILE* maze_file = fopen(file_name, "r");

    char** maze = alloc_maze(rows, cols);

    int i,j;

    for (i = 0; i < rows; ++i) {
        for (j = 0; j < cols; ++j) {
            if ((c = getc(maze_file)) == '\n') {
                c = getc(maze_file);
            }

            maze[i][j] = c;
        }
    }

    fclose(maze_file);
    return maze;
}
```

Labirentin başlangıç noktasını buluyoruz ve bulunan noktayı bir dizi olarak döndürüyoruz.

```
int* find_start(char** maze, int rows, int cols){
    int i,j;
    int* start;
    start = malloc(2 * sizeof(int*));

    for (i = 0; i < rows; ++i) {
        for (j = 0; j < cols; ++j) {
            if (maze[i][j] == 'b') {
                start[0] = i;
                start[1] = j;
            }
        }
    }

    return start;
}
```

Labirentte gezilen yerleri visited matrisinde işaretliyoruz ve işaretlenen matrisi döndürüyoruz.

```
int** init_visited(char** maze, int rows, int cols)
{
    int** visited = alloc_visited(rows, cols);

    int i, j;
    for (i = 0; i < rows; ++i) {
        for (j = 0; j < cols; ++j) {
            if (maze[i][j] == '|' || maze[i][j] == '-' || maze[i][j] == '+') {
                visited[i][j] = wall;           //duvarlar işaretleniyor.
            } else if (maze[i][j] == 'c') {
                visited[i][j] = goal;           //hedef işaretleniyor.
            } else if (maze[i][j] == 'o') {
                visited[i][j] = apple;          //elmalar işaretleniyor.
            } else {
                visited[i][j] = empty;          //geri kalan yerler boşluk olarak işaretleniyor.
            }
        }
    }
    return visited;
}
```

Labirenti ekrana yazdırıyoruz.

```
void print_maze(char** maze, int rows, int cols)
{
    int i, j;
    printf("-- Maze --\n\n");
    for (i = 0; i < rows; ++i) {
        for (j = 0; j < cols; ++j) {
            printf("%c", maze[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}
```

Labirette gezilen yerlerde kırıntı bırakıyoruz. Labirette kırıntılar '*' ile gösteriliyor.

```
void add_crumbs(char** maze, int** visited, int rows, int cols)
{
    int i, j;
    for (i = 0; i < rows; ++i) {
        for (j = 0; j < cols; ++j) {
            if (maze[i][j] != 'b' && visited[i][j] == crumb) {
                maze[i][j] = '*';
            }
        }
    }
}
```

Depth First Search algoritmasını burda gerçekleştiriyoruz.

```
int dfs(int row, int col, char** maze, int** visited, int rows, int cols)
{
    int* current = &visited[row][col];    // gezdiğimiz yerleri 'current' işaretçisi ile geziyoruz.
    int m=0;                               //çıkmaza gelip gelmediğimizi anlamak için bir m değişkeni tanımlıyoruz.
    if (*current == goal) {                 //hedefe geldiysek algoritma sonlanıyor.
        return 1;
    }

    if (*current == apple){                 //elmalar ile karşılaşmış isek puanı güncelliyoruz.
        point = point + 10;
    }

    if (*current == empty || *current == apple) {
        add_crumb(maze, visited, rows, cols);    //gezdiğimiz yerlere kırıntı bırakıyoruz.
        print_maze(maze, rows, cols);           //anlık değişimleri görmek için labirenti ve
        print_point(point);                     //puanı ekrana yazdırıyoruz.
        Sleep(100);                             // değişimleri yakalayabilmek için 100 milisaniye bekliyoruz ve
        system("cls");                           // ekranı her seferinde temizliyoruz.
        *current = crumb;
        if (visited[row][col-1] != wall && visited[row][col-1] != crumb){    // dört farklı yönde özyinelemeli olarak
            m=1;                                                                // bulunduğumuz konumu kontrol ediyoruz.
            *current = crumb;                                                  // eğer girdiysek kırıntı bırakıyoruz ve
            if (dfs(row, col-1, maze, visited, rows, cols))                  // m değişkenine 1 değerini atıyoruz.
                return 1;
        }
        if (visited[row+1][col] != wall && visited[row+1][col] != crumb){
            m=1;
            *current = crumb;
            if (dfs(row + 1, col, maze, visited, rows, cols))
                return 1;
        }

        if (visited[row][col+1] != wall && visited[row][col+1] != crumb){
            m=1;
            *current = crumb;
            if (dfs(row, col+1, maze, visited, rows, cols))
                return 1;
        }
        if (visited[row-1][col] != wall && visited[row-1][col] != crumb){
            m=1;
            *current = crumb;
            if (dfs(row - 1, col, maze, visited, rows, cols))
                return 1;
        }
        if(m==0){                                                                //çıkmaza girip girmediğimizi m değerini kontrol ederek anlıyoruz.
            point = point - 5;                                                  //ve duruma göre puanımızı güncelliyoruz.
        }
    }

    return 0;
}
```

Tüm olay burada gerçekleşiyor. Yani çözümün gerçekleştiği asıl nokta burasıdır. Ben burada DFS algoritmasını özyinelemeli bir şekilde kullandım. Her adımda odayı boş gördükçe özyinelemeli bir şekilde ilerleyecek ancak çıkmaza girdiği zaman girdiği en içteki dfs fonksiyonundan çıkarak en son ki boş gördüğü odaya yöneliyor. Bu labirentte geri adım işlemi olarak görünüyor.

Problemde bizden elma gördükçe puanı 10 puan artırmamızı ve çıkmaza girdikçe de 5 puan azaltmamızı isteniyor. Fonksiyonun başlarında elmaya rastlayıp rastlamadığını kontrol ediyoruz. Rastlamış ise puanı 10 puan artırıyoruz. Çıkmaza girme durumunu ise bir m değişkeni ile kontrol ediyoruz. İlk başta m değişkenine 0 değerini atıyoruz. Eğer herhangi bir yol bulup girmişse m değişkenine 1 değerini atıyoruz. Yollardan herhangi birine girememişse m değeri 0 olarak kalıyor ve sondaki m kontrolüne göre puanı 5 azaltıyoruz.

Main fonksiyonumuz.

```
int main() {
    char** maze;
    int** visited;
    int* start;
    char fileName[100];
    int rows;
    int cols;

    printf("--Welcome to the maze--\n\n"); // kullanıcıdan dosya adını
    printf("Enter the filename: ");
    scanf("%s", fileName);
    printf("\nEnter the count of rows: "); // labirentin satır
    scanf("%d", &rows);
    printf("\nEnter the count of columns: "); //ve sütun değerlerini alıyoruz.
    scanf("%d", &cols);

    maze = get_maze(fileName, rows, cols); //okunan dosyadan labirenti alıyoruz.
    visited = init_visited(maze, rows, cols); // gezilen yerlerin işaretlendiği matris
    start = find_start(maze, rows, cols); // ve labirentin başlangıç noktasını buluyoruz.
    print_maze(maze, rows, cols); // labirenti boş bir şekilde yazıyoruz

    if (!dfs(start[0], start[1], maze, visited, rows, cols)) { //dfs algoritmamızı çalıştırıp çıkışı
        printf("No path to the goal could be found.\n"); //bulan yolu arıyoruz.
    } else {
        print_maze(maze, rows, cols); //son olarak yolu bulunan labirenti
        print_point(point); //ve puanı yazdırıyoruz.
        printf("\nThe End of the Maze");
    }
    return 0;
}
```

Video bağlantı linki:

<https://youtu.be/gRkTRhW23Jw>

Bu projeyi yaparken öğrendiğim birçok konuyu pekiştirme fırsatım oldu. Bu yüzden size teşekkür ederim hocam.