

Санкт-Петербургский государственный политехнический
университет
Кафедра “Прикладная математика”

Отчет по лабораторной работе 3

“Алгоритмы и структуры данных”

Студент группы № 5030102/20002

ФИО Двас Павел Григорьевич

Выполнил (дата) 15.11.2023

Оглавление

Постановка задачи.....	2
Описание алгоритма	2
Текст программы	3
Описание тестирования	7

Постановка задачи

На шахматной доске размером $N \times N$ найти кратчайший путь ходами коня из поля A в поле B.

Описание алгоритма

Для описания алгоритма требуется пояснить как работает поиск в ширину и очереди

Алгоритм BFS (Breadth-First Search) представляет собой метод обхода графа или дерева, начиная с заданной вершины и пошагово расширяя поиск на ближайшие соседние вершины перед переходом к более отдаленным. Он использует очередь для хранения вершин, которые нужно посетить. На первом шаге начальная вершина добавляется в очередь, а затем извлекается. Затем все непосещенные соседние вершины добавляются в очередь. Этот процесс повторяется, пока не будут посещены все вершины графа.

Очередь - это структура данных, представляющая собой упорядоченную коллекцию элементов, в которой элементы добавляются в конец (enqueue) и извлекаются из начала (dequeue). Для реализации BFS используется очередь, чтобы следовать принципу "первым пришел, первым ушел" (First In, First Out, FIFO). Это означает, что вершины обрабатываются в порядке, в котором они были добавлены в очередь, что и обеспечивает пошаговое расширение поиска от начальной вершины к ближайшим соседям перед переходом к более отдаленным.

Представим очередь в качестве структуры

```
typedef struct
{
    int x;
    int y;
} Point;

typedef struct
{
    Point point;
    int dist;
} QueueNode;

typedef struct
{
    QueueNode *arr;
    int front;
    int rear;
    int size;
} Queue;
```

Где int x, y – координаты точки, int dist – пройденное расстояние, QueueNode *arr – упорядоченная коллекция элементов, int front, rear – «номера» первого и последнего элемента, int size – размер очереди

Добавление элемента

Для добавления нового узла необходимо:

1. Записать новое значение
2. Сдвинуть «указатели» на начало и конец очереди

Удаление элемента

Для удаления элемента из очереди необходимо:

1. Сдвинуть «указатели» на начало и конец очереди
2. Вернуть удаляемое значение

Текст программы

main.c:

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

#define N 1000

typedef struct
{
    int x;
    int y;
} Point;

typedef struct
{
    Point point;
    int dist;
} QueueNode;

typedef struct
{
    QueueNode *arr;
    int front;
    int rear;
    int size;
} Queue;

Queue * QueueStart( int size )
{
    Queue *queue = (Queue *)malloc(sizeof(Queue));

    queue->size = size;
    queue->front = queue->rear = -1;
    queue->arr = (QueueNode *)malloc(queue->size * sizeof(QueueNode));
```

```

    return queue;
}

void Enqueue( Queue *queue, QueueNode data )
{
    queue->arr[++queue->rear] = data;

    if (queue->front == -1)
        queue->front++;
}

QueueNode Dequeue( Queue *queue )
{
    QueueNode data = queue->arr[queue->front];

    if (queue->front == queue->rear)
        queue->front = queue->rear = -1;
    else
        queue->front++;
    return data;
}

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

int moves[8][2] = {{2, 1}, {1, 2}, {-1, 2}, {-2, 1}, {-2, -1}, {-1, -2}, {1, -2}, {2, -1}};

void ShortestPath( Point Start, Point End )
{
    int **visited;
    int **distance;
    int i, j;
    Point **prev;
    Queue *queue;
    QueueNode qn;

    if ((visited = malloc(N * sizeof(int *))) == NULL)
    {
        printf("Mem error");
        return;
    }

    for (i = 0; i < N; i++)
        if ((visited[i] = malloc(N * sizeof(int))) == NULL)
        {
            printf("Mem error");
            return;
        }

    if ((distance = malloc(N * sizeof(int *))) == NULL)
    {
        printf("Mem error");
        return;
    }

    for (i = 0; i < N; i++)
        if ((distance[i] = malloc(N * sizeof(int))) == NULL)
        {

```

```

    printf("Mem error");
    return;
}

if ((prev = malloc(N * sizeof(Point *))) == NULL)
{
    printf("Mem error");
    return;
}

for (i = 0; i < N; i++)
    if ((prev[i] = malloc(N * sizeof(Point))) == NULL)
    {
        printf("Mem error");
        return;
    }

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
    {
        visited[i][j] = 0;
        distance[i][j] = 0;
        prev[i][j].x = -1;
        prev[i][j].y = -1;
    }

queue = QueueStart(N * N);
qn.point = Start;
qn.dist = 0;
Enqueue(queue, qn);
visited[Start.x][Start.y] = 1;

while (queue->front != -1)
{
    QueueNode current = Dequeue(queue);

    for (i = 0; i < 8; i++)
    {
        int next_x = current.point.x + moves[i][0];
        int next_y = current.point.y + moves[i][1];

        if (IsValid(next_x, next_y) && !visited[next_x][next_y])
        {
            visited[next_x][next_y] = 1;
            distance[next_x][next_y] = current.dist + 1;
            prev[next_x][next_y] = current.point;

            qn.point.x = next_x;
            qn.point.y = next_y;
            qn.dist = current.dist + 1;
            Enqueue(queue, qn);

            if (next_x == End.x && next_y == End.y)
            {
                Point *path;
                int length = distance[next_x][next_y];
                int index = length;

                path = malloc(N * N * sizeof(Point));

                while (!(prev[next_x][next_y].x == -1 && prev[next_x][next_y].y == -1))

```

```

    {
        Point temp;

        path[--index] = prev[next_x][next_y];
        temp = prev[next_x][next_y];
        next_x = temp.x;
        next_y = temp.y;
    }

    printf("Shortest path:\n");
    path[length].x = End.x;
    path[length].y = End.y;
    for (i = 0; i <= length; i++)
        printf("(%d, %d) ", path[i].x, path[i].y);

    printf("\nPath Length: %d\n", length + 1);
    free(path);
    for (i = 0; i < N; i++)
    {
        free(prev[i]);
        free(visited[i]);
        free(distance[i]);
    }
    free(prev);
    free(visited);
    free(distance);
    free(queue->arr);
    return;
}
}
}

for (i = 0; i < N; i++)
{
    free(prev[i]);
    free(visited[i]);
    free(distance[i]);
}
free(prev);
free(visited);
free(queue->arr);
free(distance);
printf("No path\n");
}

int main()
{
    Point start = {0, 1};
    Point end = {3, 4};

    ShortestPath(start, end);

    _getch();
    return 0;
}

```

Описание тестирования

Для тестирования данной программы производился ее неоднократный запуск с вводом конкретных данных, при котором проверялось:

- 1) Стабильность работы программы при одинаковых входных данных;
- 2) Отсутствие “падений” и “зависаний”;
- 3) Корректное выполнение всех заявленных процедур;
- 4) Корректное завершение программы;

Пример тестирования:

```
Shortest path:  
(0, 1) (2, 2) (3, 4)  
Path Length: 3
```

При проведении тестирования такого рода никаких проблем обнаружено не было, что позволяет судить о корректности работы программы в целом.