算法基础project3实验报告

PB19071535徐昊天

一.实验内容

1. Bellman-Ford算法

   根据邻接矩阵实现求单源最短路径，统计算法所需运行时间 ， 画出时间曲线，并分析程序性能。

2. Johnson算法

   根据邻接矩阵实现求所有点最短路径，统计算法所需运行时间 ， 画出时间曲线，并分析程序性能。

二.实验设备和环境

- windows操作系统，VMware虚拟机
- vscode、dev-c++
- Excel

三.实验方法和步骤

1.Bellman-Ford算法

- 处理文件

  本次实验须通过相对路径读取 `input` 文件夹内的文件数据，经过算法处理后得到输出数据并通过相对路径写入 `output` 文件夹。由于存在八种数据，故可将文件指针存入二维数组，便于调用。涉及文件处理的代码如下：

```
FILE *in[4][2], *out[4][2], *fp;
// FILE *in_11,*in_12,*in_21,*in_22,*in_31,*in_32,*in_41,*in_42;
// FILE *out_11,*out_12,*out_21,*out_22,*out_31,*out_32,*out_41,*out_42,*fp;
// 打开输入输出文件
if((in[0][0] = fopen("../input/input11.txt","r"))==NULL)
```

```
    {
        printf("cannot open input11 file\n");
        exit(0);
    }
    if((in[0][1] = fopen("../input/input12.txt","r"))==NULL)
    {
        printf("cannot open input12 file\n");
        exit(0);
    }
    if((in[1][0] = fopen("../input/input21.txt","r"))==NULL)
    {
        printf("cannot open input21 file\n");
        exit(0);
    }
    if((in[1][1] = fopen("../input/input22.txt","r"))==NULL)
    {
        printf("cannot open input22 file\n");
        exit(0);
    }
    if((in[2][0] = fopen("../input/input31.txt","r"))==NULL)
    {
        printf("cannot open input31 file\n");
        exit(0);
    }
    if((in[2][1] = fopen("../input/input32.txt","r"))==NULL)
    {
        printf("cannot open input32 file\n");
        exit(0);
    }
    if((in[3][0] = fopen("../input/input41.txt","r"))==NULL)
    {
        printf("cannot open input41 file\n");
        exit(0);
    }
    if((in[3][1] = fopen("../input/input42.txt","r"))==NULL)
    {
        printf("cannot open input42 file\n");
        exit(0);
    }
    if((out[0][0] = fopen("../output/result11.txt","w"))==NULL)
    {
        printf("cannot open result11 file\n");
        exit(0);
    }
    if((out[0][1] = fopen("../output/result12.txt","w"))==NULL)
    {
        printf("cannot open result12 file\n");
        exit(0);
    }
    if((out[1][0] = fopen("../output/result21.txt","w"))==NULL)
    {
        printf("cannot open result21 file\n");
        exit(0);
    }
    if((out[1][1] = fopen("../output/result22.txt","w"))==NULL)
    {
        printf("cannot open result22 file\n");
        exit(0);
    }
    if((out[2][0] = fopen("../output/result31.txt","w"))==NULL)
    {
```

```
        printf("cannot open result31 file\n");
        exit(0);
    }
    if((out[2][1] = fopen("../output/result32.txt","w"))==NULL)
    {
        printf("cannot open result32 file\n");
        exit(0);
    }
    if((out[3][0] = fopen("../output/result41.txt","w"))==NULL)
    {
        printf("cannot open result41 file\n");
        exit(0);
    }
    if((out[3][1] = fopen("../output/result42.txt","w"))==NULL)
    {
        printf("cannot open result42 file\n");
        exit(0);
    }
    if((fp = fopen("../output/time.txt","w"))==NULL)
    {
        printf("cannot open time file\n");
        exit(0);
    }
    ......
    ......
    ......
    //关闭输入输出文件
    for (i = 0; i < 4; i++)
        for (j = 0; j < 2; j++)
        {
            fclose(in[i][j]);
            fclose(out[i][j]);
        }
    fclose(fp);
```

- 定义数据结构

  根据课本中图结点的成员和具体功能，定义结构体如下：

```
typedef struct Node
{
    struct Node *p;     //前驱结点
    int d;      //到指定结点的最短距离
    int front;      //前驱结点的id
}Node;
```

  图的信息直接用输入文件提供的邻接矩阵储存，从输入文件中读取邻接矩阵的代码如下：

```
int G[730][730];    // 定义一个足够大的全局二维数组用于储存邻接矩阵
......
......
......
for (k = 0; k < N; k++)
{
    for (l = 0; l < N; l++)
    {
        scanf(in[i][j], "%d,", &G[k][l]);
    }
}
```

- 松弛操作

  首先对图中结点进行初始化操作，初始化内容包括最短路径和前驱结点，对应的算法代码如下：

```
void initialize_single_source(int G[730][730], Node *s, int N)
{
   int i;
   for (i = 1; i < N; i++)
   {
      vertex[i]->d = 9999;
      vertex[i]->p = NULL;
      vertex[i]->front = i;
   }
   s->d = 0;
}
```

  对指定边(u,v)进行松弛：首先测试是否可以对最短路径进行改善，若可以改善，则对结点的属性进行更新，对应算法代码如下：

```
void relax(int u, int v, int G[730][730])
{
   if(vertex[v]->d > vertex[u]->d + G[u][v])
   {
      vertex[v]->d = vertex[u]->d + G[u][v];
      vertex[v]->p = vertex[u];
      vertex[v]->front = u;
   }
}
```

- 实现Bellman-Ford算法

  Bellman-Ford算法通过对边进行松弛操作来渐近地降低从源节点s到每个结点v的最短路径估计值，直到该估计值与实际的最短路径权重相同时为止。该算法返回 `1` 时当且仅当输入图不包含负环路。对应算法代码如下：

```
int bellman_ford(int G[730][730], Node *s, int N)
{
   initialize_single_source(G, s, N);
   int i, j, k;
   for (i = 1; i < N; i++)
   {
```

```
        for (j = 0; j < N; j++)
        {
            for (k = 0; k < N; k++)
            {
                if(G[j][k] != 0)
                {
                    relax(j, k, G);
                }
            }
        }
    }
    for (j = 0; j < N; j++)
    {
        for (k = 0; k < N; k++)
        {
            if(G[j][k] != 0 && vertex[k]->d > vertex[j]->d + G[j][k])
            {
                return 0;
            }
        }
    }
    return 1;
}
```

- 输出信息

  单源最短路径权值存储于图结点结构体中的 `d` 成员；对于最短路径，可通过遍历终止结点的前驱直至源结点并将前驱路径存于数组中，并逆序输出即为单源最短路径。对应代码如下：

```
        for (k = 1; k < N; k++)
        {
            if (vertex[k]->d != 9999)
            {
                int length = 0;
                Node *node = vertex[k];
                int flag = 0;
                while (node != vertex[0])
                {
                    path[length++] = node->front;
                    node = vertex[node->front];
                    if (node->d == 9999)
                    {
                        flag = 1;
                        break;
                    }
                }
                if (flag)
                    continue;
                fprintf(out[i][j], "0,%d,%d;", k, vertex[k]->d);
                for (l = length - 1; l >= 0; l--)
                {
                    fprintf(out[i][j], "%d,", path[l]);
                }
                fprintf(out[i][j], "%d\n", k);
            }
        }
```

- 记录算法运行时间

  利用头文件 `#include<time.h>` 中 `clock()` 函数记录时间，结果以毫秒为单位，实现代码如下：

```
        clock_t start, end;
        double time;
        start = clock();
        int ford = bellman_ford(G, vertex[0], N);
        end = clock();
        time = (double)(end - start) / (CLOCKS_PER_SEC);
        fprintf(fp, "%f\n", time);
```

  得到时间单位为毫秒。

2.Johnson算法

- 处理文件

  本次实验须通过相对路径读取 `input` 文件夹内的文件数据，经过算法处理后得到输出数据并通过相对路径写入 `output` 文件夹。由于存在八种数据，故可将文件指针存入二维数组，便于调用。涉及文件处理的代码同 `Bellman-Ford算法` 。

- 定义数据结构

  本实验所定义图的数据结构与图结点数据结构同 `Bellman-Ford算法` 。

- dijkstra算法

  dijkstra算法在运行过程中维持一组节点集合作为关键信息，解决带权重的单源最短路径问题，对应代码如下：

```
void dijkstra(int G[730][730], int s, int N)
{
    initialize_single_source(G, vertex[s], N);
    int i, j, min, min_s;
    int S[N + 1], T[N + 1];
    for (i = 0; i < N; i++)
    {
        S[i] = G_john[s][i];
        D[s][i] = G[s][i];
        T[i] = 0;
    }
    T[s] = 1;
    for (i = 0; i < N - 1; i++)
    {
        min = 9999;
        for (j = 0; j < N; j++)
        {
            if (T[j] == 0 && S[j] < min)
            {
                min = S[j];
```

```
            min_s = j;
        }
    }
    T[min_s] = 1;
    for (j = 0; j < N; j++)
    {
        if (G[min_s][j] < 9999 && S[j] > S[min_s] + G_john[min_s][j])
        {
            S[j] = S[min_s] + G_john[min_s][j];
            D[s][j] = D[s][min_s] + G[min_s][j];
        }
    }
  }
}
```

- 实现Johnson算法

  Johnson算法调用 `Bellman-Ford算法` 和 `dijkstra算法` 作为子程序来计算所有结点对之间的最短路径，对应代码如下：

```
void johnson(int G[730][730], int N)
{
  int i, j;
  for (i = 0; i < N; i++)
  {
    G[N][i] = 0;
    G[i][N] = 9999;
  }
  G[N][N] = 9999;
  if (bellman_ford(G, vertex[N], N + 1) == 0)
  {
    printf("the input graph contains a negative-weight cycle\n");
    exit(0);
  }
  int h[N + 1];
  for (i = 0; i <= N; i++)
    h[i] = vertex[i]->d;
  for (i = 0; i <= N; i++)
  {
    for (j = 0; j <= N; j++)
    {
      if (G[i][j] != 9999)
      {
        G_john[i][j] = G[i][j] + h[i] - h[j];
      }
      else
      {
        G_john[i][j] = G[i][j];
      }
    }
  }
  for (i = 0; i < N; i++)
  {
    dijkstra(G, i, N);
  }
}

```

调用Johnson算法得到修改后的D矩阵便是所求的最短路径矩阵。

- 记录算法运行时间

  利用头文件 `#include<time.h>` 中 `clock()` 函数记录时间，结果以毫秒为单位，实现代码如下：

```
clock_t start, end;
double time;
start = clock();
int ford = bellman_ford(G, vertex[0], N);
end = clock();
time = (double)(end - start) / (CLOCKS_PER_SEC);
fprintf(fp, "%f\n", time);
```

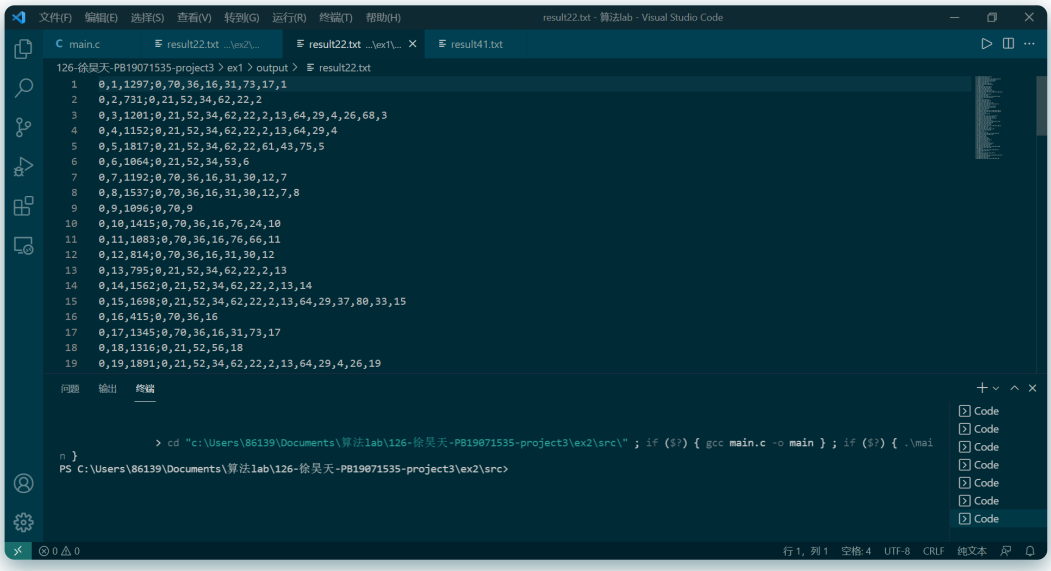
  得到时间单位为毫秒。

四.实验结果和分析

1.Bellman-Ford算法

- result文件结果

  以result22.txt为例，打印结果如下图所示：



- 时间复杂度分析

  将代码放入虚拟机中通过以下指令编译运行：

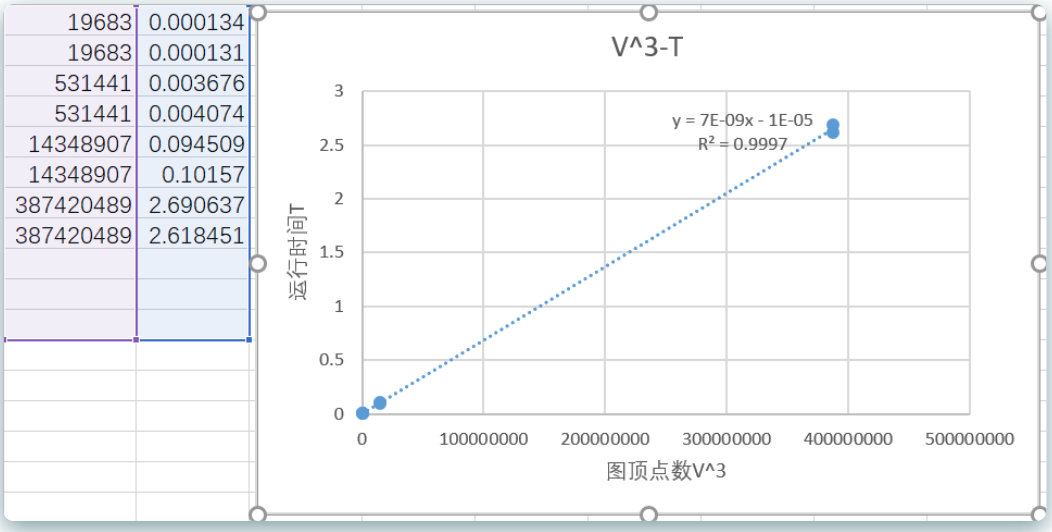
```
gcc -o main.c main
./main
```

  得到 `time.txt` 中时间如下图所示：

八组数据图的信息如下表所示：

	V	E	V•E	V•V•V
1	27	3	81	19683
2	27	2	54	19683
3	81	3	243	531441
4	81	3	243	531441
5	243	4	972	14348907
6	243	3	729	14348907
7	729	5	3645	387420489
8	729	4	2916	387420489

将时间T与顶点数V^3数据在excel中拟合图如下图所示：

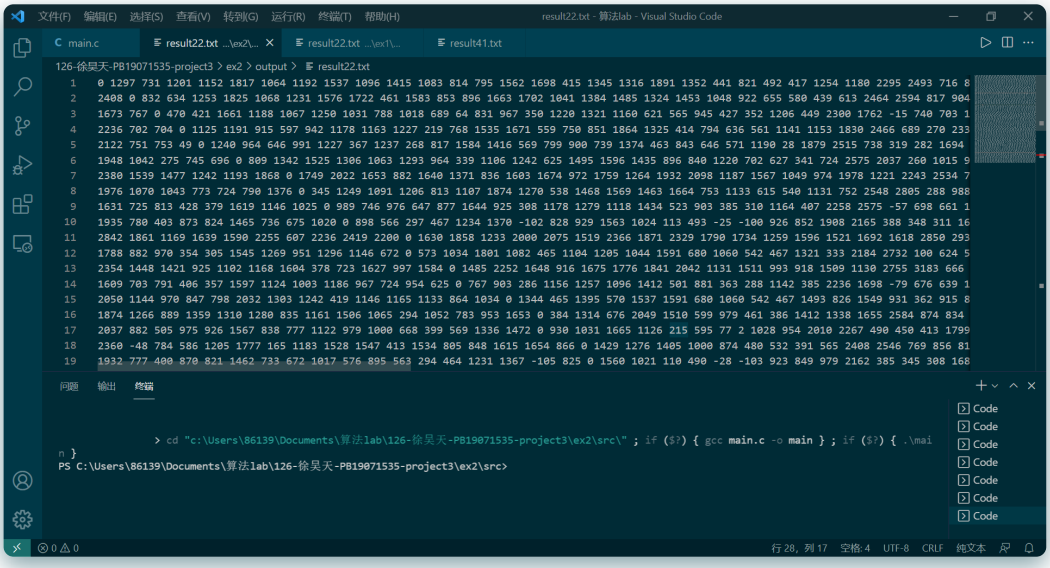


- 理论时间复杂度为O(VE)，由于算法初始化操作所需时间为O(V)，循环运行时间为O(E)，一共要进行V-1次循环并对每次遍历的E进行松弛操作。
- 根据拟合得到的分布图可知，实际时间复杂度为O(V^3)，由于使用的图数据结构为邻接矩阵，故遍历所有边时运行时间为O(V^2)，初始化操作运行时间为O(V)，实际时间复杂度即为O(V^3)。

2.Johnson算法

- result文件结果

  以result22.txt为例，打印结果如下图所示：



- 时间复杂度分析

  将代码放入虚拟机中通过以下指令编译运行：

```
gcc -o main.c main
./main
```

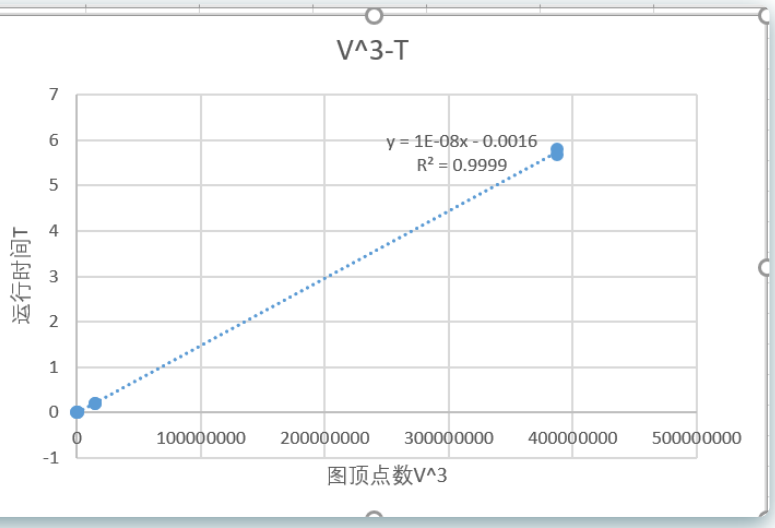
  得到 time.txt 中时间如下图所示：

八组数据图的信息如下表所示：

	V	E	V•V•V
1	27	3	19683
2	27	2	19683
3	81	3	531441
4	81	3	531441
5	243	4	14348907
6	243	3	14348907
7	729	5	387420489
8	729	4	387420489

将时间T与顶点数V^3数据在excel中拟合图如下图所示：

19683	0.000499
19683	0.000444
531441	0.008868
531441	0.008885
14348907	0.204652
14348907	0.207991
387420489	5.793321
387420489	5.678725



- 若使用斐波那契堆实现dijkstra算法里面的最小优先队列，则理论时间复杂度为O(VE+V^2·lgV)；若使用二叉最小堆实现则理论运行时间为O(VElgV)。
- 根据拟合得到的分布图可知，实际时间复杂度为O(V^3)，由于实现dijkstra算法时未使用优先队列，故实际时间复杂度即为O(V^3)。

五.实验收获与感想

1. 强化了对于图数据结构的处理能力，并通过实现经典算法完成求图中最短路径问题，对以上算法有了更深刻的理解和更灵活的运用。
2. 通过处理较为复杂的图数据结构及其相关算法，提升了对C语言的运用和调试能力。