

导航软件

一、实验要求

要求在所给的数据集上建立图结构（邻接矩阵或者邻接表），并在建立的图结构上自行实现Dijkstra算法求解任意两点之间的最短路径。

- 输入输出要求：

Input : src(源点) Dst(目标点)

Output :

(1) 最短路径的长度： distance

(2) Src到Dsr的一条最短路径，例如： Src->p1->p2->p3->...->Dst(逆序输入也对)

二、实验目的，

熟悉并掌握图的建立算法和Dijkstra求图上最短路径算法，了解Dijkstra算法的改进方法，掌握时间复杂度的分析方法并且去分析对比验证不同时间复杂度的Dijkstra算法的时间开销，了解稀疏的图结构的压缩存储方法。

三、程序能实现的功能

- 对数据文件进行二进制处理，同时少量压缩二进制文件
- 根据数据文件建立图结构(邻接表)
- 使用朴素法，二叉堆，配对堆，fibonacci堆搜索最短路径，对于大数据测试样例，搜索时间能够进入10s

四、设计思路

- 预处理
 - 将原数据文件转换为二进制形式
 - 建立图的邻接表，将图按顺序输出，同一个出发点的边只记录以此源点，用度数来标记下一个源点
- dijkstra
 - 建立 `visited` 与 `distance` 数组
 - 初始 `visited` 数组置0， `distance` 数组置最大(由于 `memset()` 的限制，稍小于 `INT_MAX`)
 - 将源点距离设为0，访问设为1，遍历所有与源点相连的边，记录距离进入 `distance` 数组
 - 从 `distance` 数组中找出 `distance` 最短的点，记为 u ，遍历所有与 u 相连的边，相连的点记为 v_1, v_2, \dots, v_n ，如果 $visited[v_i] == 0 \& distance[v_i] > distance[u] + weight_{u \rightarrow v}$ ，更新 `distance[v_i]` 的值
 - 重复上述操作，知道 $visited[Dst] == 1$ 或者整个图已搜索完毕，仍然没有找到路径
 - 逆序输出最短路径

五、项目结构

```

|-- CMakeLists.txt
|-- binary
|   |-- pre.cpp
|-- include
|   |-- Heap.h
|   |-- dijkstra.h
|   |-- fibheap.h
|   |-- graph.h
|   |-- pair_heap.h
|-- src
|   |-- Heap.cpp
|   |-- dijkstra.cpp
|   |-- fibheap.cpp
|   |-- graph.cpp
|   |-- main.cpp
|   |-- pair_heap.cpp

```

六、文件的具体实现

- 预处理
 - 二进制

```

1 void input::begin_binary()
2 {
3     auto loc = data_info.find('.');
4     out_info = data_info.substr(0, loc) + "_pre.txt";
5     std::ifstream infile(data_info.c_str(), std::ios::in);
6     std::ofstream outfile(out_info.c_str(), std::ios::out |
std::ios::binary);
7     if (!infile || !outfile)
8     {
9         return;
10    }
11    int num1, num2, info;
12
13    printf("\033[?25l");
14    std::string line;
15    while (getline(infile, line))
16    {
17        std::cout << "\r" << line;
18        auto linstream = std::istringstream(line);
19        linstream >> num1 >> num2 >> info;
20        outfile.write((char *)&num1, sizeof(int));
21        outfile.write((char *)&num2, sizeof(int));
22        outfile.write((char *)&info, sizeof(int));
23    }
24    printf("\033[?25h");
25    infile.close();
26    outfile.close();
27 }

```

- 压缩

根据邻接表直接二进制输出图的数据

- dijkstra

- **main.cpp**

不同函数的调用，与用户交互

- **graph.h**和**graph.cpp**

类及函数的定义

```
1  typedef struct ArcNode
2  {
3      int adjvex;
4      int weight;
5      struct ArcNode *nextarc;
6  } ArcNode;
7  struct VNode
8  {
9      int vex_data;
10     ArcNode *firstarc;
11 };
12 class AGraph
13 {
14 private:
15     int vexnum;
16     struct ArcNode *last_ptr;
17 public:
18     std::vector<VNode *> data;
19     AGraph();
20     ~AGraph();
21     bool input(int x, int y, int info); //输入数据
22     bool input_(int x, int y, int info); //输入数据(用于压缩输入)
23     int get_vex(); //返回点数
24 };
```

■ bool input(int x,int y,int info)

```
1  bool AGraph::input(int x, int y, int info)
2  {
3      std::cout << "\r" << x << " " << y << " " << info;
4      if (data[x] == nullptr)
5      {
6          vexnum++;
7          struct VNode *first_init = new VNode;
8          first_init->vex_data = x;
9          first_init->firstarc = nullptr;
10         data[x] = first_init;
11     }
12     struct ArcNode *p = new ArcNode;
13     p->nextarc = nullptr;
14     p->weight = info;
15     p->adjvex = y;
16
17     struct ArcNode *q = data[x]->firstarc;
18     if (q == nullptr)
19     {
```

```

20     data[x]->firstarc = p;
21     last_ptr = p;
22     return true;
23 }
24 while (q->nextarc != nullptr)
25     q = q->nextarc;
26 q->nextarc = p;
27
28 last_ptr = p;
29 return true;
30 }

```

◦ dijkstra.h和dijkstra.cpp

函数定义

```

1
2 struct node
3 {
4     int begin;
5     int distance;
6     node()
7     {
8         begin = 0;
9         distance = 0;
10    };
11    node(int a, int b)
12    {
13        begin = a;
14        distance = b;
15    };
16    friend bool operator>(struct node n1, struct node n2)
17    {
18        return n1.distance < n2.distance;
19    }
20    friend bool operator<(struct node n1, struct node n2)
21    {
22        return n1.distance > n2.distance;
23    }
24 };
25 struct cmp
26 {
27     bool operator()(struct node x, struct node y)
28     {
29         return x.distance > y.distance;
30     }
31 };
32 int find_min(std::unordered_map<int, int> &dis_map, int vis[]);
33 clock_t dijkstra(AGraph &G, int x, int y); //朴素算法
34 clock_t dijkstra_heap(AGraph &G, int x, int y); //二叉堆
35 clock_t dijkstra_fibheap(AGraph &G, int x, int y);
36 //Fibonacci堆
37 clock_t dijkstra_pair_heap(AGraph &G, int x, int y); //配对堆（递归）
38 clock_t dijkstra_pair_heap_0(AGraph &G, int x, int y); //配对堆（队列）

```

```

38 clock_t dijkstra_heap_gnu(AGraph &G, int x, int y); // pbds库
    优先队列
39 clock_t dijkstra_pair_heap_gnu(AGraph &G, int x, int y); // pbds库
    配对堆
40 clock_t dijkstra_thin_heap_gnu(AGraph &G, int x, int y); // pbds库
    Fibonacci堆

```

- **clock_t dijkstra_heap(AGraph &G, int x, int y)** (所有搜索除了使用的数据结构不同, 剩下实现部分基本相同, 只举一个介绍)

```

1  clock_t dijkstra_heap(AGraph &G, int x, int y)
2  {
3      int *visited = new int[100000000];
4      memset(visited, 0, sizeof(int) * 100000000);
5      int *dis = new int[100000000];
6      memset(dis, 127, sizeof(int) * 100000000);
7      int *pre = new int[100000000];
8      memset(pre, -1, sizeof(int) * 100000000);
9      Heap<struct node> q;
10     dis[x] = 0;
11     q.push(node(x, dis[x]));
12
13     while (!q.empty())
14     {
15         struct node now = q.top();
16         q.pop();
17         if (visited[now.begin] == 1)
18             continue;
19         visited[now.begin] = 1;
20         if (now.begin == y)
21             break;
22         struct ArcNode *p = G.data[now.begin]->firstarc;
23         while (p != nullptr)
24         {
25             if (visited[p->adjvex] != 1 && dis[p->adjvex] >
dis[now.begin] + p->weight)
26             {
27                 dis[p->adjvex] = dis[now.begin] + p->weight;
28                 q.push(node(p->adjvex, dis[p->adjvex]));
29                 pre[p->adjvex] = now.begin;
30             }
31             p = p->nextarc;
32         }
33     }
34
35     clock_t endtime = clock();
36
37     if (visited[y] != 1)
38     {
39         std::cout << "不能到达!" << std::endl;
40         delete[] visited;
41         delete[] dis;
42         delete[] pre;
43         return endtime;
44     }
45     int road = y;
46     while (road != -1)

```

```

47     {
48         printf("\033[0;30;47m%d\033[0m ", road);
49         road = pre[road];
50         if (road != -1)
51             std::cout << "<- ";
52         else
53             printf("\n");
54     }
55     std::cout << "\nOutput:\n" << dis[y] << std::endl;
56
57     delete[] visited;
58     delete[] dis;
59     delete[] pre;
60
61     return endtime;
62 }

```

◦ heap.h和heap.cpp

类的定义及重要函数内容

```

1  template <class T>
2  class Heap
3  {
4  private:
5      std::vector<T> data;
6      int length;
7  public:
8      Heap();
9      ~Heap();
10     inline void swim(int k);      //上浮
11     inline void sink(int k);     //下沉
12     inline void push(T e);       //入堆
13     inline void pop();           //出堆
14     inline T top();              //返回堆顶元素
15     inline bool empty();         //判断是否为空
16     inline int size();           //返回大小
17     inline void swap(T &a, T &b); //交换元素
18 };

```

■ 实现思路

使用vector存储数据，下标为0的点不用，这样下表为 n 的数据，双亲为 $\frac{n}{2}$ ，孩子为 $2n$ 和 $2n + 1$ ，每次入堆，在结围加入元素，然后最后一个元素上浮，出堆，将最后一个元素和第一个元素交换，删除最后一个元素，将第一个元素下沉，此时push和pop都是 $\log n$ 的

■ void push(T e)

```

1  template <class T>
2  inline void Heap<T>::push(T e)
3  {
4      data.push_back(e);
5      length++;
6      swim(length);
7  }

```

■ void pop()

```

1 inline void Heap<T>::pop()
2 {
3     swap(data[1], data[length--]);
4     data.pop_back();
5     sink(1);
6 }

```

■ void swim(int k)

```

1 template <class T>
2 inline void Heap<T>::swim(int k) //上浮
3 {
4     while (k > 1 && data[k] > data[k / 2])
5     {
6         swap(data[k], data[k / 2]);
7         k /= 2;
8     }
9 }

```

■ void sink(int k)

```

1 template <class T>
2 inline void Heap<T>::sink(int k) //下沉
3 {
4     while (k * 2 <= length)
5     {
6         int j = 2 * k;
7         if (j < length && (data[j] < data[j + 1])) //找到左右子树
            中更小的
8             j++;
9         if (data[k] > data[j])
10            break;
11        swap(data[k], data[j]);
12        k = j;
13    }
14 }

```

○ pair_heap.h和pair_heap.cpp

类的定义及重要函数实现

```

1 template <class T>
2 class pair_node
3 {
4 public:
5     T val;
6     int left;
7     int right;
8     pair_node()
9     {
10         left = 0;
11         right = 0;
12     }
13     pair_node(T e)
14     {
15         val = e;

```

```

16         left = 0;
17         right = 0;
18     }
19 };
20 template <class T>
21 class pair_heap
22 {
23 private:
24     std::vector<pair_node<T>> data;
25     int length; //已经到达的vector地址
26     int _size; //实际存储的大小
27     inline void merge(int x, int y);
28     inline int merges(int x, int y);
29     int root;
30     inline int pop_();
31
32 public:
33     pair_heap();
34     ~pair_heap();
35     inline void push(T e);
36     inline void pop();
37     inline void _pop();
38     inline T top();
39     inline bool empty();
40     inline int size();
41 };

```

■ 实现思路

使用vector储存节点，节点的元素包含左右孩子的下标，以及自身的值，配对堆的核心操作是merge操作，即将两棵树合并，有递归和队列两种方法，push操作即将新节点与原来的root节点进行一次merge操作，复杂度是 $O(1)$ 的，pop操作是将根节点的去除，将他的每一个孩子merge在一起，其中merge操作分为两种，递归merge和队列merge

■ void merge(int x,int y)

```

1  template <class T>
2  inline void pair_heap<T>::merge(int x, int y)
3  {
4      if (!x || !y)
5          root = x + y;
6      else if (x == y)
7          root = x;
8      else
9      {
10         if (data[x].val < data[y].val)
11         {
12             int temp = x;
13             x = y;
14             y = temp;
15         }
16         data[y].right = data[x].left;
17         data[x].left = y;
18         data[x].right = 0;
19         root = x;
20     }
21 }

```


■ **int merges(int x,int y)** (递归用merge, 除了返回root外和上述merge完全相同)

■ **void push(T e)**

```
1  template <class T>
2  inline void pair_heap<T>::push(T e)
3  {
4      pair_node<T> temp(e);
5      data.push_back(temp);
6      length++;
7      merge(root, length);
8      _size++;
9  }
```

■ **void pop()** (队列 pop)

```
1  template <class T>
2  inline void pair_heap<T>::pop()
3  {
4      if (_size == 0)
5      {
6          std::cout << "为空!" << std::endl;
7          return;
8      }
9      if (data[root].left == 0)
10         root = 0;
11     else if (data[data[root].left].right == 0)
12         root = data[root].left;
13     else
14     {
15         std::queue<int> que;
16         int now = root;
17         que.push(data[now].left);
18         now = data[now].left;
19         while (data[now].right != 0)
20         {
21             que.push(data[now].right);
22             now = data[now].right;
23         }
24         int _que_size = que.size();
25         while (_que_size >= 2)
26         {
27             int a = que.front();
28             que.pop();
29             int b = que.front();
30             que.pop();
31             merge(a, b);
32             que.push(root);
33             _que_size--;
34         }
35     }
36     _size--;
37 }
```

■ **void _pop()** (递归pop)

```

1  template <class T>
2  inline void pair_heap<T>::_pop()
3  {
4      root = pop_();
5      _size--;
6  }
7  template <class T>
8  inline int pair_heap<T>::pop_()
9  {
10     if (_size == 0)
11     {
12         std::cout << "为空!" << std::endl;
13         return 0;
14     }
15     if (data[root].left == 0)
16         return 0;
17     else if (data[data[root].left].right == 0)
18         return data[root].left;
19     else
20     {
21         int son1 = data[root].left;
22         int son2 = data[data[root].left].right;
23         data[root].left =
24         data[data[data[root].left].right].right;
25         return merges(merges(son1, son2), pop_());
26     }
27 }

```

◦ fibheap.h和fibheap.cpp

类的定义及重要函数内容

```

1  template <typename T>
2  struct fib_node
3  {
4      struct fib_node<T> *parent;
5      struct fib_node<T> *child;
6      struct fib_node<T> *left;
7      struct fib_node<T> *right;
8      T key;
9      int degree;
10     fib_node() : parent(nullptr), child(nullptr), left(this),
11     right(this), degree(0), mark(false) {}
12 };
13 template <class T>
14 class FibHeap
15 {
16 private:
17     int keyNum;
18     int maxDegree;
19     struct fib_node<T> *min;
20     struct fib_node<T> **cons;
21     void removeNode(struct fib_node<T> *node); // 剔除该节点
22     void addNode(struct fib_node<T> *node, struct fib_node<T>
23     *root);
24     void consolidate(); // 合并
25     struct fib_node<T> *extractMin(); // 抽出最小节点对应的树

```

```

24     void makeCons(); //开辟空间
25     void link(struct fib_node<T> *node, struct fib_node<T>
    *root); //合并节点时, 相同度数节点的合并
26     void theEnd(struct fib_node<T> *node); //用于释放空间
27 public:
28     FibHeap();
29     ~FibHeap();
30     void push(T e);
31     void pop();
32     bool empty();
33     int size();
34     T top();
35 };

```

■ 实现思路

由于只需要使用入堆和出堆, 所以对fibonacci堆的内容进行了一定的简化, fibonacci堆是一颗树, 兄弟节点之间可以当作双向循环链表, push操作直接在根节点所在的双向循环链表中加入该节点, 根据大小更新根节点, pop操作同配对堆类似, 但合并的是根节点的孩子以及根节点的兄弟, 此时需要引入辅助节点

■ void push(T e)

```

1  template <class T>
2  void FibHeap<T>::push(T e)
3  {
4      struct fib_node<T> *temp = new struct fib_node<T>;
5      temp->key = e;
6      temp->left = temp;
7      temp->right = temp;
8      if (keyNum == 0)
9          min = temp;
10     else
11     {
12         addNode(temp, min);
13         if (min->key < temp->key)
14             min = temp;
15     }
16     keyNum++;
17 }

```

■ void pop()

```

1  template <class T>
2  void FibHeap<T>::pop()
3  {
4      if (min == nullptr)
5          return;
6      keyNum--;
7      struct fib_node<T> *p = min;
8      struct fib_node<T> *child = nullptr;
9      while (p->child != nullptr)
10     {
11         child = p->child;
12         removeNode(child);
13         if (child->right == child)
14             p->child = nullptr;

```

```

15         else
16             p->child = child->right;
17             addNode(child, min);
18             child->parent = nullptr;
19     }
20     removeNode(p);
21     if (p->right == p)
22         min = nullptr;
23     else
24     {
25         min = p->right;
26         consolidate();
27     }
28     delete p;
29 }

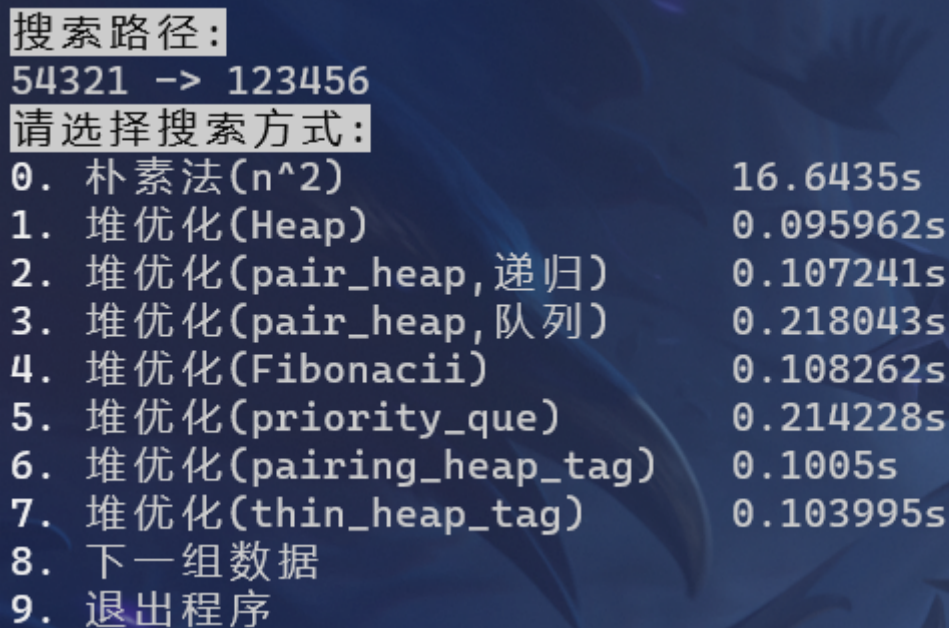
```

pop操作主要为将最小节点的孩子上升到根节点所在的循环链表内，接着从中不断抽取最小节点及其对应的树，将度数相同的节点合并(利用辅助空间)，更新根节点(具体实现参照源码)

七、测试程序的正确性及性能

引入pbds库中各种最小堆进行比较

- 对于小数据测试样例



```

搜索路径:
54321 -> 123456
请选择搜索方式:
0. 朴素法(n^2)                16.6435s
1. 堆优化(Heap)               0.095962s
2. 堆优化(pair_heap, 递归)    0.107241s
3. 堆优化(pair_heap, 队列)    0.218043s
4. 堆优化(Fibonacci)         0.108262s
5. 堆优化(priority_que)      0.214228s
6. 堆优化(pairing_heap_tag)   0.1005s
7. 堆优化(thin_heap_tag)     0.103995s
8. 下一组数据
9. 退出程序

```

可以看到，对于小数据测试样例，除了朴素法时间过长以外，手写的堆以及pbds库中的堆都可以在1s之内完成搜索，由于电脑性能不稳定，时间的差距可以忽略

- 对于两组大数据测试样例
 - 3141592->2718281

搜索路径:

3141592 -> 2718281

请选择搜索方式:

0. 朴素法(n^2)	0s
1. 堆优化(Heap)	6.72801s
2. 堆优化(pair_heap, 递归)	9.37885s
3. 堆优化(pair_heap, 队列)	11.1983s
4. 堆优化(Fibonacii)	11.061s
5. 堆优化(priority_que)	9.7215s
6. 堆优化(pairing_heap_tag)	8.91119s
7. 堆优化(thin_heap_tag)	9.51418s
8. 下一组数据	
9. 退出程序	

o 1000000->2000000

搜索路径:

1000000 -> 2000000

请选择搜索方式:

0. 朴素法(n^2)	0s
1. 堆优化(Heap)	0.295858s
2. 堆优化(pair_heap, 递归)	0.367577s
3. 堆优化(pair_heap, 队列)	0.424836s
4. 堆优化(Fibonacii)	0.483372s
5. 堆优化(priority_que)	0.359514s
6. 堆优化(pairing_heap_tag)	0.356479s
7. 堆优化(thin_heap_tag)	0.385739s
8. 下一组数据	
9. 退出程序	

- o 对于大数据测试, 朴素法在一个小时内都无法搜索出答案
- o 所有堆优化的算法都可以在25s之内完成搜索
- o 手动实现的二叉堆拥有最好的性能, 较之系统的优先队列, 也有更好的表现
- o 递归实现pop的配对堆与系统配对堆性能相仿, 队列实现pop则稍有差距(如果在编译时开启O2优化, 队列的性能会强于递归, 但是O2优化不够稳定)
- o Fibonacci堆较之pbds库有着不小的差距

• 一些思考

配对堆和Fibonacci堆在理论上的时间复杂度都应该优于二叉堆, 但是手写的二叉堆反而性能最佳, pbds库配对堆和Fibonacci堆较二叉堆也没有极其突出的表现, 我认为原因可能有:

- o 测试数据点特殊, 具有偶然性
- o 在入堆的操作上, 二叉堆是 $O(\log(n))$ 的, 配对堆和Fibonacci堆是 $O(1)$ 的, 此时二叉堆的性能较差, 但由于数据特殊性, 在每次入堆时, 所需要经历的上浮操作较少
- o 在出堆的操作上, 三种堆结构均是 $O(\log(n))$ 的, 但是配对堆和Fibonacci堆是均摊复杂度, 单次操作极限情况下: $\lim_{n \rightarrow +\infty} \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots = n$, 由于出堆几乎不是连续的, 虽然上次出堆使根节点的孩子节点变成了原来的一半, 由于不能立刻弹出, 经历了多次入堆操作后, 出堆的复杂度仍然是较高的, 浪费了过多的时间

八、编译过程

- 编译环境
 - Ubuntu-20.04 (wsl2)
 - gcc 9.3.0
 - cmake version 3.16.3
- CMakeLists.txt (注意修改Debug模式和Release模式)

```
1  cmake_minimum_required(VERSION 3.5)
2
3  project(navigation)
4
5  set (CMAKE_CXX_STANDARD 17)
6
7  set(SOURCES
8      src/main.cpp
9      src/dijkstra.cpp
10     src/graph.cpp
11     src/Heap.cpp
12     src/pair_heap.cpp
13     src/fibheap.cpp)
14
15  add_executable(navigation ${SOURCES})
16
17  SET(CMAKE_BUILD_TYPE "Release")
18  # SET(CMAKE_BUILD_TYPE "Debug")
19
20  target_include_directories(navigation
21  PRIVATE
22      ${CMAKE_CURRENT_SOURCE_DIR}/include)
```

- 编译操作(Release版本为例，首先进入项目目录)

```
1  mkdir Release
2  cd Release
3  cmake ..
4  make
5  ulimit -s unlimited # linux下打开内存限制
6  ./navigation
7  # 运行程序
```

九、实验收获

- 进一步加深了对图这种数据结构的认识
- 熟悉了Dijkstra算法及其堆优化
- 探索了多种堆的数据结构
- 了解了文件存储的机制