《算法设计与分析》

课程实验报告

专业：计算机科学与技术

班级：2021211306

姓名：杜抒泽

学号：2021211110

第五/六章：TSP 问题

2021211110 杜抒泽

2023 年 12 月 20 日

目录

1 书面作业

参照讲义 PPT 中（P26~28）给出的面向最大化问题（如 01 背包问题）的分支限界法算法框架，设计面向最小化问题，e.g. 旅行商问题，的分支限界法算法框架。

回答.

面向最小化问题的分支限界算法框架如下：

1. 选择初始解对应的根节点 v_0，根据限界函数 $bound$，估计根节点的目标函数上下界 $bound(v_0)$，确定目标最优解的上下界 $[\text{down}, \text{up}]$。
2. 将活结点表 ANT 初始化为空。
3. 按照宽度优先顺序生成根结点 v_0 的全部子结点。
   - 对每个子结点 v，执行以下操作：
     1. 估算 v 的目标函数值 $bound(v)$
     2. 若 $bound(v) \leq \text{up}$，将 v 加入 ANT 表。

     对最小化问题，要求沿 v 分支搜索到的完全解的目标值（最小价值）估计。显然其必须小于现有已知的最优目标函数的上界 up（目前已有的最大价值）。
4. 循环，直到某个叶结点的目标函数值在表 ANT 中最小（也即找到了一个完全解，其值比已知的最优解更优）：
   1. 从表 ANT 中选择 $bound(v_i)$ 值最小的结点 v_i，扩展其子结点（选择任意一个具有最小可能目标值的扩展结点 v_i 都有可能更新答案，贪心地选择最小者可以玄学地压缩搜索空间，不必然正确）。
   2. 对结点 v_i 的每个子结点 c，执行下列操作：
      1. 估算 c 的目标函数值 $bound(c)$，如果 $bound(c) \leq \text{up}$，将 c 加入 ANT 表。
         - 最优解上界是 up，$bound(c) \geq \text{up}$ 说明子结点 c 有可能产生更优的解，将其加入活结点表，以后考虑对其进行扩展。
      2. 如果 c 是叶结点并且 $bound(c)$ 在表 ANT 中最大，则将结点 c 对应的完全解输出，**算法结束**。
         - 结点 c 对应一个新找到的、具有最小目标值（e.g. 哈密顿回路长度）的完全解（最优解）

3. 如果 c 是叶结点并且 $bound(c)$（或者说，$value(c)$）在表 ANT 中不是最小（结点 c 对应了一个新找到的完全解，但该完全解的目标函数值与已经找到的或未来可能找到完全解相比，并非更优），则：
   1. 令 $\text{up} = \min(\text{up}, value(c))$。
      - 其中 $value(c)$ 是叶结点 c 对应的完全解的哈密顿回路长度。
      - 利用新找到的完全解的实际目标函数 $value(c)$，更新问题解的上界。
   2. 对表 ANT 中所有满足 $bound(v_j) \geq \text{up} = bound(c)$ 的结点 v_j，从表 ANT 中删除该结点。
      - 利用新找到的完全解的目标函数 $bound(c)$，进行剪枝：从 ANT 表中去掉那些目标函数上界值不可能小于结点 c 的 $bound(c)$ 的结点 v_j，即去掉那些目标函数上界值大于或等于当前新找到的完全解 c 的目标值 $bound(c)$ 的结点。

2 实验内容

2.1 实验要求

由于我是算法竞赛出身，在讲解算法时可能并不会使用和课上相同的术语，但我保证描述一定干练、正确且规范，表达我已经完全掌握了算法的思想，并且希望读者能够更好地理解算法的本质。

- 采用回溯法、分支限界法，编程求解不同规模的旅行商问题 TSP，并利用给定数据，验证算法正确性，对比算法的时间复杂性、空间复杂性。

2.2 题目需求分析

本次实验的主要内容是对于图论意义上的带权简单无向图求解 TSP 问题[1]

形式化地，给定带权简单无向图 $G = (V, E)$，对于 G 中的某个结点 S，求出 S 经过且只经过 G 中所有结点各一次，最后再回到点 S 的最短路径。图中的路径定义为一个结点序列 $v_0, v_1, ..., v_k$，使得存在边的序列 $e_1, e_2, ..., e_k$，满足对于每个 e_i 都存在 $e_i = (v_{i-1}, v_i, w_i)$，其中 w_i 是 e_i 的权值。若 $v_0 = S$，则称这个路径是 S 到 v_k 的一条路径。路径的长度定义为所有边的权值之和 $W = \sum_{i=1}^{k} w_i$。

2.3 具体实验内容

[1]实际上 TSP 问题限定在完全图上，听闻 301~304 的课程群里已经讨论了这个问题。作业给出的问题只能说是任意图上的 Hamilton 回路问题。不过无所谓，Hamilton 问题也是 NP-Hard 的。

具体地，我在本次实验中实现了以下内容：

- 编写数据预处理脚本，从给出的 .xls 文件中提取出基站经纬度数据并写入文本文件供后续使用。
- 将图抽象为类型，使底层数据结构对使用者透明，同时包含从文本文件中读入图的接口。
- 在实现的类型上编写回溯算法和分支限界算法，求解本实验定义的 TSP 问题。
- 在不同规模的数据上运行程序，统计运行运行时间和使用的内存空间。
- 编写构建与运行脚本，提供易于使用的接口用于运行程序的各个部分。脚本在类 Unix 环境下能够工作，**但没有在 Windows 环境下测试，不保证能够运行。**若有需要请联系我或自行适配。
- 同时，代码中包含的空间统计模块只提供了 macOS 和 Linux 上的实现，没有提供 Windows 的版本（若编译会输出友好的报错信息）。若需要在 Windows 上编译运行，请自行删除相关代码。

3 算法设计

3.1 题目背景与数学抽象

给定一个不含重边和自环的无向图 $G=(V,E)$ 和源点 S。边带正权，也即命题 $\forall e_i \in E(e_i=(u_i,v_i,w_i),w_i>0)$ 为真。以 S 为起点，求出一条经过 G 中所有结点且最后回到 S 的最短路径。

在以下的讨论中，我们记 $n=\Theta(|V|)$，$m=\Theta(|E|)$。

3.2 回溯算法与剪枝

3.2.1 算法介绍

其实回溯法没啥好说的啊，就是 DFS。

从 S 出发，每次选择一个未访问过的结点，将其加入路径中，继续向下搜索。当所有结点都被访问过时，检查当前路径是否是最短路径，如果是则更新答案。最后回到 S，算法结束。

具体地，我们可以把回溯算法的过程抽象为一个树。树的每个结点表示当前的搜索状态，树的每条路径表示一条搜索路径。树的根结点表示初始状态，树的叶结点表示搜索结束。树的每个结点对应一个路径 $path$，维护一个 dis 值，代表当前路径的长度。每次搜索时，我们从当前结点出发，选择一个未访问过的结点，将其加入路径中，建立当前状态的一个儿子，

继续向下搜索。当所有结点都被访问过时，检查当前路径是否是最短路径，如果是则更新答案。最后回到 S，算法结束。

这个算法有一个显然的剪枝策略：如果当前路径的长度已经超过了已知的最短路径的长度，那么就不必再继续搜索了。这个剪枝策略可以有效减少搜索空间，提高算法的性能。

此外，还有一些玄学想法，比如对于每个结点，枚举递归的顺序，贪心地选择具有最小距离的未被访问的相邻结点，可以进一步压缩搜索空间，但是这个太人类智慧了，并且对于可以构造的数据是不具备正确性的，退一万步讲它也不能降低理论复杂度，因此我没有实现。

该算法的时间复杂度是 $\mathcal{O}(n \times n!)$，因为每个深度为 d 的结点有 $n-d$ 个儿子，总共有 $n!$ 个叶结点。需要的时间会随着图的不同而不同，同时剪枝也会影响算法的性能。算法只需要 $\mathcal{O}(n)$ 的额外空间，与其递归深度相同，和其高昂的时间复杂度相比不值一提。

3.2.2 算法实现

我在 `dfs.h` 中实现了回溯算法。具体地，我实现了一个 `Dfs` 类，其中包含了算法记录的所有信息：

成员	说明
`const Graph &g`	算法运行所在的图的引用
`std::vector<int> path`	当前递归到的路径
`std::vector<bool> visited`	当前路径中结点的访问状态
`T cost`	当前递归到的路径的长度
`std::vector<int> min_path`	最短路径
`T min_cost`	最短路径的长度
`std::int64_t node_count`	搜索过的结点数
`void run()`	运行算法

以下是具体代码：

```
template<typename Weight>
class Dfs {
  const Graph<Weight> &g;
  std::vector<int> path{};
  std::vector<bool> visited{};
  Weight cost;
```

```
public:
  std::vector<int> min_path{};
  Weight min_cost{Graph<Weight>::INF};
  std::int64_t node_count{};

  Dfs(const Graph<Weight> &g_): g(g_) {
    path.emplace_back(g.src);
    visited.resize(g.n + 1);
    visited[g.src] = true;
  }

  void run() {
    ++node_count;
    int u = path.back();

    if (static_cast<int>(path.size()) == g.n) {
      if (g.adj[u][g.src] == g.INF)
        return;

      path.emplace_back(g.src);
      cost += g.adj[u][g.src];

      if (cost < min_cost) {
        min_cost = cost;
        min_path = path;
      }

      cost -= g.adj[u][g.src];
      path.pop_back();
      return;
    }

    for (int v = 1; v <= g.n; ++v) {
      if (u == v || g.adj[u][v] == g.INF || visited[v] ||
          cost + g.adj[u][v] >= min_cost)
        continue;

      path.emplace_back(v);
      cost += g.adj[u][v];
      visited[v] = true;
      run();
      visited[v] = false;
      cost -= g.adj[u][v];
      path.pop_back();
    }
```

```
    }
};
```

3.3 优先队列维护分支限界算法

3.3.1 算法介绍

分支限界算法本质也是搜索。其维护一个解空间的上界（当前已经求得的最小开销，下述为 min cost），将每个状态都抽象成搜索树上的一个结点，并为每个结点维护一个 lower bound 值（下界）。通过这种手段，算法可以丢弃所有 lower bound 值大于 min cost 的结点，从而有效地减小搜索空间。

其中，假设某个状态 s 对应经过结点 $(u_1, u_2, ..., u_k)$ 的路径。定义集合 $U = \{u_1, u_2, ..., u_k\}$，邻接矩阵 G 中 $G_{u,v}$ 表示结点 u 和结点 v 之间的无向距离。那么，状态 s 的 lower bound l_s 定义为

$$l_s = \frac{1}{2}\left(2\sum_{i=1}^{k-1} G_{u_i,u_{i+1}} + \min_{v\notin U} G_{u_1,v} + \min_{v\notin U} G_{u_k,v} + \sum_{v\notin U}\left(\min_{1\le i<j\le n} G(v,V_i) + G(v,V_j)\right)\right)$$

记号 $V_i\ (1 \le i \le n)$ 表示图 $G = (V, E)$ 中点集 V 的第 i 个点。

特别地，若状态 s 对应的路径长度为 n（包含全部点的一条链），此时上述式子中中间的两项都没有良定义，l_s 特别定义为这条链的长度加之链的两个端点之间连边的长度（也即无良定义的两项直接看作 0，外加最后一项退化为两端点的直接连边）。

上式的意义比较容易理解。其中第一项表示已经走过的路径的长度（已经确定）。把所有点分成两个集合已经走过的点 U 和还没有到达的点 $V - U$。第二、三项表示从已经走过的路径向 $V - U$ 中延伸的最小开销，第四项表示集合 $V - U$ 中所有点随意连边的总开销的一个下界（很明显这不是下确界。维护下确界的时空开销极大，无法承担，这是在界的松紧与时间开销进行 trade off 后的 walkaround）。

这个式子的意义清晰地展示了这个算法的思想。我们不需要也无法得到一个精确的下界（也就是答案本身，如果直接就能轻松得到答案本身那还要分支限界法做什么呢？），但是我们可以在相对短的时间内得到一个相对紧的下界，保证该状态的答案一定大于等于这个下界。将所有已知状态的下界都求出后，我们就可以根据求得的少量叶结点的确定答案迅速剪枝掉连非紧下界都已经比不上确定答案的子树，从而大大减少搜索空间。

每个结点保存自己对应的状态，包括路径、开销和 lower bound 值。这带来了一定的空间 shock。每次从当前状态出发，选择一个未访问过的结点，将其加入路径中，建立当前状态的一个儿子，继续向下搜索。

剪枝采取以下的策略：初始时将上界置为无穷大，每次搜索到一个叶结点时，如果该叶结点的开销小于上界，则更新（缩紧）上界。每当遍历到一个结点时，如果当前状态的 lower bound 值大于上界，则不必再继续搜索了。也即在实现时实现为懒删除：并不在上界改变时改变整个 ANT 表，而是在后续遍历 ANT 表时发现某个结点的 lower bound 值大于上界时将其跳过。这个剪枝策略可以有效减少搜索空间，提高算法的性能。

还有一个极其重要的问题：在最初，上界 up 是不能像课件中那样贪心求得的。只有真正的完全图上 TSP 才能这么做。在一般图上，贪心（或者其他的什么算法）求出一个 Hamilton 回路的可行解的开销并不比求出一个最优解低多少。难道要为了求出一个不一定紧的上界，在分支限界法的 BFS 前先 DFS 一遍实现贪心吗？显然不现实。因此我选择在开始时直接将上界置为无穷大，等待后续可行解的更新。

3.3.2 复杂度特别说明

我注意到，课件上声称这个算法的复杂度是 $\mathcal{O}(n^2 \times 2^n)$ 的。显然这个说法不是很正确。推测搞错成这个复杂度了可能是因为分支限界法在其他的很多场景下的搜索树是二叉树，结点数是 $\mathcal{O}(2^n)$ 的。

这个算法的理论复杂度（很可能）是 $\mathcal{O}(n^2 \times n!)$ 的。总状态数是 $\mathcal{O}(n!)$ 的，而对于每个结点计算 lower bound 值是 $\mathcal{O}(n^2)$ 的，这两者成为了总时间复杂度的瓶颈。在和算法竞赛现役选手讨论了数百条消息后，我们仍然无法确定这个剪枝策略能带来理论复杂度上界的下降，只能认为其是玄学优化。

然而理论复杂度高昂并不能说明它的实际表现不好。实际上，分支限界法的时间和空间甚至都要低过正牌 $\mathcal{O}(2^n \times n^2)$ 的状态压缩动态规划 !这说明了搜索算法与玄学剪枝进行结合后，算法会依据数据的不同，有不同的时间表现，理论复杂度不一定可靠。

此外，这个算法跑不满复杂度还有一个极其重要的原因：正牌的 TSP 问题是跑在完全图上的，而这个图的稠密程度**远远不能达到**完全图的水平。这个问题只能被称作求一般图上的最小 Hamilton 回路。而拥有正牌 $\mathcal{O}(2^n \times n^2)$ 复杂度的状压 DP 的表现是不会随着图的稠密性而变化的（总会跑满）。

对于真正的完全图上的 TSP 问题（包含欧几里得平面 TSP 问题），可以通过玄学贪心和其他手段进一步提升算法的实际表现，有可能能证明出理论复杂度低于 $\mathcal{O}(2^n \times n!)$[2]。

3.3.3 算法实现

[2]纯猜测，不能保证。

我在 bfs.h 中实现了分支限界算法。具体地，我实现了一个 Bfs 类，其中包含了算法记录的所有信息：

成员	说明
`const Graph &g`	算法运行所在的图的引用
`struct Node`	搜索树的结点类型，其中包含了结点的路径、开销和 lower bound 值
`void update_lower_bound(Node &node)`	更新结点的 lower bound 值
`std::vector<int> min_path`	最短路径
`Weight min_cost`	最短路径的长度
`std::int64_t node_count`	搜索过的结点数
`void run()`	运行算法

在 run 函数中，包含使用到的其他信息：

成员	说明
`std::priority_queue<Node> q`	ANT 表，用于维护搜索树的活结点，采用优先队列实现
`std::multiset<Weight> lower_bounds`	用于维护搜索树的结点的 lower bound 值的多重集，可以 $\mathcal{O}(\log n)$ 地求最小值和插入、删除元素

```
template<typename Weight>
class Bfs {
  const Graph<Weight> &g;

  struct Node {
    std::vector<int> path{};
    Weight cost{}, lower_bound{};

    Node(int src_): path{src_} {}

    bool operator<(const Node &rhs) const {
```

```
    return lower_bound > rhs.lower_bound;
  }
};

void update_lower_bound(Node &node) {
  if (static_cast<int>(node.path.size()) == g.n) {
    node.lower_bound = node.cost + g.adj[node.path.back()][g.src];
    return;
  }

  node.lower_bound = 2 * node.cost;

  std::vector<bool> visited(g.n + 1);
  for (int u : node.path)
    visited[u] = true;

  auto get_min_elem = [&](int u) -> Weight {
    Weight min_elem = Graph<Weight>::INF;
    for (int v = 1; v <= g.n; ++v)
      if (!visited[v] && g.adj[u][v] < min_elem)
        min_elem = g.adj[u][v];
    return min_elem;
  };
  auto get_two_min_elem = [&](int u) -> std::pair<Weight, Weight> {
    Weight min_elem = Graph<Weight>::INF,
           second_min_elem = Graph<Weight>::INF;
    for (int v = 1; v <= g.n; ++v)
      if (v != u) {
        if (g.adj[u][v] < min_elem) {
          second_min_elem = min_elem;
          min_elem = g.adj[u][v];
        } else if (g.adj[u][v] < second_min_elem)
          second_min_elem = g.adj[u][v];
      }
    return {min_elem, second_min_elem};
  };

  if (node.path.size() != 1) {
    node.lower_bound += get_min_elem(node.path.front());
    node.lower_bound += get_min_elem(node.path.back());
  } else {
    auto [x, y] = get_two_min_elem(node.path.front());
    node.lower_bound += x + y;
  }

  for (int u = 1; u <= g.n; ++u)
    if (!visited[u]) {
```

```
        auto [x, y] = get_two_min_elem(u);
        node.lower_bound += x + y;
      }

    node.lower_bound /= 2;
  }

public:
  std::vector<int> min_path;
  Weight min_cost{Graph<Weight>::INF}; // global upper bound
  std::int64_t node_count{};

  Bfs(const Graph<Weight> &g_): g(g_) {}

  void run() {
    std::priority_queue<Node> q;
    std::multiset<Weight> lower_bounds;

    Node node(g.src);
    update_lower_bound(node);
    lower_bounds.emplace(node.lower_bound);
    q.push(node);

    while (!q.empty()) {
      Node node = q.top();
      q.pop();

      lower_bounds.erase(lower_bounds.find(node.lower_bound));
      if (node.lower_bound > min_cost)
        continue;
      ++node_count;

      if (static_cast<int>(node.path.size()) == g.n) {
        if (g.adj[node.path.back()][g.src] == g.INF)
          continue;

        node.cost += g.adj[node.path.back()][g.src];
        node.path.emplace_back(g.src);

        if (node.cost == *lower_bounds.begin()) {
          min_cost = node.cost;
          min_path = node.path;
          break;
        }

        if (node.cost < min_cost) {
          min_cost = node.cost;
```

```
          min_path = node.path;
        }
        continue;
      }

      int u = node.path.back();
      std::vector<bool> visited(g.n + 1);
      for (int v : node.path)
        visited[v] = true;

      for (int v = 1; v <= g.n; ++v) {
        if (g.adj[u][v] == g.INF || visited[v])
          continue;

        Node new_node = node;
        new_node.path.emplace_back(v);
        new_node.cost += g.adj[u][v];
        update_lower_bound(new_node);

        if (new_node.lower_bound < min_cost) {
          lower_bounds.emplace(new_node.lower_bound);
          q.push(new_node);
        }
      }
    }
  }
};
```

4 程序设计说明

项目名称为 SRH（搜索 SeaRcH 的缩写）。程序在类 Unix 系统下开发与调试，Python 代码用于数据预处理，大规模数据的生成和控制程序的编译、运行，在 Python 3.10.6 版本下编写；C++ 代码用于编写算法，遵守 C++20 语言标准，确保可移植性。

- 但是需要注意测量空间占用的代码不跨平台，我只实现了 macOS 和 Linux 可用的版本。若需要在 Windows 上编译运行，请自行删除相关代码。

4.1 如何运行

我提供了构建/运行脚本 `run.py`，包含安装依赖、数据预处理、编译、运行功能，帮助信息如下图：

```
$ ./run.py -h
usage: run.py [-h] [-d DFS] [-b BFS] [-c] [-s]

options:
  -h, --help         show this help message and exit
  -d DFS, --dfs DFS  run dfs with the given node amount
  -b BFS, --bfs BFS  run bfs with the given node amount
  -c, --clean        clean the target dir
  -s, --silent       do not print execution info
```

图 1　构建脚本帮助信息

其会从环境变量中读取 `CXX`，`CXXFLAGS` 和 `LDFLAGS` 的值，分别作为 C++ 编译器、编译选项和链接选项。如果环境变量中不存在这些变量，则使用默认值 `CXX=g++`，`CXXFLAGS="-W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc"` 和 `LDFLAGS=""`。

以下是构建脚本的使用示例：

```
chmod +x run.py

./run.py -d 15 # 运行回溯法，数据为 15 个结点的图
./run.py -b 30 # 运行分支限界法，数据为 30 个结点的图
./run.py --clean # 清理编译产物

CXX=g++-13 ./run.py -b 30 # 我的机器上的编译器名字是 g++-13，指定环境变量
```

4.2 算法接口

`algo_base.h` 中定义了算法的输出类型 `Result<Weight>`：

```
template<typename Weight>
using Result = std::tuple<std::vector<int>, Weight, std::int64_t>;

// 等价于如下的聚合类型定义。将上面的定义修改成下面的，不改变任何其他代码，代码仍然能通过编译并正确运行。

template<typename Weight>
struct Result {
  std::vector<int> path{};
  Weight cost{};
  std::int64_t node_count{};
};
```

该类型包含三个成员，代表算法接口的输出，分别是最短 Hamilton 回路、最短 Hamilton 回路的长度和搜索算法搜索过的结点数。

dfs.h 和 bfs.h 中分别据此实现了以下的算法接口：

```
template<typename Weight>
Result<Weight> dfs(const Graph<Weight> &g) {
  Dfs<Weight> dfs(g);
  dfs.run();
  return {dfs.min_path, dfs.min_cost, dfs.node_count};
}
```

```
template<typename Weight>
Result<Weight> bfs(const Graph<Weight> &g) {
  Bfs<Weight> bfs(g);
  bfs.run();
  return {bfs.min_path, bfs.min_cost, bfs.node_count};
}
```

main.cpp 中按照如下的方式调用算法接口：

```
  auto [path, cost, node_count] =
#ifdef DFS
      dfs
#else // including #if defined(BFS)
      bfs
#endif
      (g);
```

可以看到，接口的一致性非常好，使得只需要修改定义的宏就可以切换算法，大大提高了代码的可读性和易用性。

4.3 输入数据的格式

输入数据为邻接矩阵格式，其定义如下：

```
n src // 结点个数和源点编号
id_1 id_2 ... id_n1 // 结点编号序列
a_11 a_12 ... a_1n1 // 邻接矩阵
a_21 a_22 ... a_2n1
...
a_n11 a_n12 ... a_n1n1
```

4.4 时空统计模块

统计时空采用以下的代码：

```
```

```cpp
auto start = std::chrono::high_resolution_clock::now(); // 开始计时
std::atomic<bool> recording_mem(true);
std::future<unsigned long> mem_usage_monitor = std::async(
    std::launch::async, monitor_mem_usage, std::ref(recording_mem)
); // 开始统计空间

// 运行算法

auto end = std::chrono::high_resolution_clock::now(); // 结束计时
recording_mem = false;                                // 结束统计空间

double time_elapsed =
    static_cast<double>(
        std::chrono::duration_cast<std::chrono::microseconds>(end - start)
            .count()
    ) /
    1000;                                             // 单位：微秒
unsigned long mem_usage = mem_usage_monitor.get(); // 单位：字节
```

统计时间的部分比较简单，不过多介绍。

统计空间的部分启动了一个异步线程。该线程会调用 monitor_mem_usage 函数，其实现如下：

```cpp
inline unsigned long monitor_mem_usage(std::atomic<bool> &keep_running) {
  unsigned long mem_usage = get_mem_usage();
  while (keep_running.load()) {
    std::this_thread::sleep_for(std::chrono::milliseconds(5));
    mem_usage = std::max(mem_usage, get_mem_usage());
  }
  return mem_usage;
}
```

该函数使用一个 std::atomic<bool> 变量进行同步。当主线程将其置为 false 时，该线程退出。该线程每隔 5 毫秒调用一次 get_mem_usage 函数，获取当前进程的内存占用。get_mem_usage 函数是对系统 API 的封装，该函数生成的代码会随着平台的不同而不同，在此不深入介绍。

5 测试、运行结果与分析说明

5.1 运行答案：正确性

拿到代码运行所有 8 个运行组合，输出分别如下。

图片较多，点此跳转到图片后。

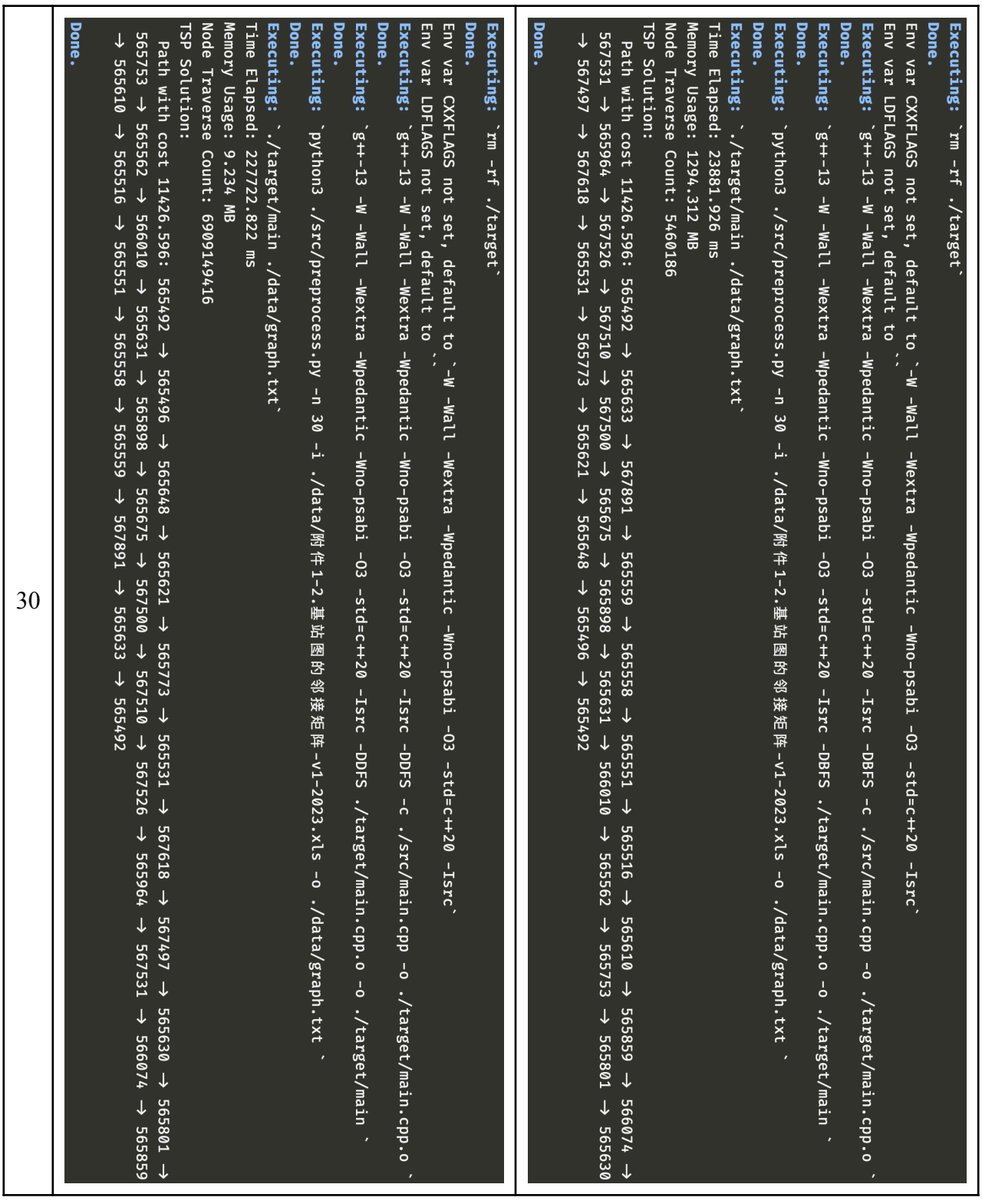
n	回溯法	分支限界法
15	Executing: `rm -rf ./target` Done. Env var CXXFLAGS not set, default to `-W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc` Env var LDFLAGS not set, default to `` Executing: `g++-13 -W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc -DDFS -c ./src/main.cpp -o ./target/main.cpp.o ` Done. Executing: `g++-13 -W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc -DBFS ./target/main.cpp.o -o ./target/main ` Done. Executing: `python3 ./src/preprocess.py -n 15 -i ./data/附件1-2.基站图的邻接矩阵-v1-2023.xls -o ./data/graph.txt ` Done. Executing: `./target/main ./data/graph.txt` Time Elapsed: 6.126 ms Memory Usage: 9.219 MB Node Traverse Count: 256955 TSP Solution: Path with cost 5506.880: 567443 → 567260 → 566742 → 568098 → 566802 → 567439 → 566751 → 566798 → 567547 → 566967 → 566993 → 566631 → 567238 → 566750 → 566783 → 567443 Done.	Executing: `rm -rf ./target` Done. Env var CXXFLAGS not set, default to `-W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc` Env var LDFLAGS not set, default to `` Executing: `g++-13 -W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc -DBFS -c ./src/main.cpp -o ./target/main.cpp.o ` Done. Executing: `g++-13 -W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc -DBFS ./target/main.cpp.o -o ./target/main ` Done. Executing: `python3 ./src/preprocess.py -n 15 -i ./data/附件1-2.基站图的邻接矩阵-v1-2023.xls -o ./data/graph.txt ` Done. Executing: `./target/main ./data/graph.txt` Time Elapsed: 3.133 ms Memory Usage: 9.766 MB Node Traverse Count: 2281 TSP Solution: Path with cost 5506.880: 567443 → 567260 → 566742 → 568098 → 566802 → 567439 → 566751 → 566798 → 567547 → 566967 → 566993 → 566631 → 567238 → 566750 → 566783 → 567443 Done.

20	Executing: `rm -rf ./target` Done. Env var CXXFLAGS not set, default to `-W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc` Env var LDFLAGS not set, default to `` Executing: `g++-13 -W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc -DBFS -c ./src/main.cpp -o ./target/main.cpp.o ` Done. Executing: `g++-13 -W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc -DBFS ./target/main.cpp.o -o ./target/main ` Done. Executing: `python3 ./src/preprocess.py -n 20 -i ./data/附件1-2.基站图的邻接矩阵-v1-2023.xls -o ./data/graph.txt ` Done. Executing: `./target/main ./data/graph.txt` Time Elapsed: 23.244 ms Memory Usage: 13.609 MB Node Traverse Count: 11705 TSP Solution: Path with cost 6987.509: 567443 → 567260 → 566742 → 568098 → 566802 → 567439 → 567322 → 566751 → 33109 → 566798 → 567547 → 566999 → 566967 → 567203 → 565696 → 566993 → 566631 → 567238 → 566750 → 566783 → 567443 Done.
	Executing: `rm -rf ./target` Done. Env var CXXFLAGS not set, default to `-W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc` Env var LDFLAGS not set, default to `` Executing: `g++-13 -W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc -DDFS -c ./src/main.cpp -o ./target/main.cpp.o ` Done. Executing: `g++-13 -W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc -DDFS ./target/main.cpp.o -o ./target/main ` Done. Executing: `python3 ./src/preprocess.py -n 20 -i ./data/附件1-2.基站图的邻接矩阵-v1-2023.xls -o ./data/graph.txt ` Done. Executing: `./target/main ./data/graph.txt` Time Elapsed: 2289.802 ms Memory Usage: 9.297 MB Node Traverse Count: 76329668 TSP Solution: Path with cost 6987.509: 567443 → 567260 → 566742 → 568098 → 566802 → 567439 → 567322 → 566751 → 33109 → 566798 → 567547 → 566999 → 566967 → 567203 → 565696 → 566993 → 566631 → 567238 → 566750 → 566783 → 567443 Done.

```
Executing: `rm -rf ./target`
Done.
Env var CXXFLAGS not set, default to `-W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc`
Env var LDFLAGS not set, default to ``
Executing: `g++-13 -W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc -DBFS -c ./src/main.cpp -o ./target/main.cpp.o `
Done.
Executing: `g++-13 -W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc -DBFS ./target/main.cpp.o -o ./target/main `
Done.
Executing: `python3 ./src/preprocess.py -n 22 -i ./data/附件1-2.基站图的邻接矩阵-v1-2023.xls -o ./data/graph.txt `
Done.
Executing: `./target/main ./data/graph.txt`
Time Elapsed: 10.368 ms
Memory Usage: 11.391 MB
Node Traverse Count: 5109
TSP Solution:
  Path with cost 7690.802: 567443 → 567260 → 566742 → 568098 → 566802 → 567439 → 566751 → 567322 → 566747 → 566720 → 33109 → 566798 → 567547 → 566999 → 566967 → 567203 → 565696 → 566993 → 566631 → 567238 → 566750 → 566783 → 567443
Done.
```

```
Executing: `rm -rf ./target`
Done.
Env var CXXFLAGS not set, default to `-W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc`
Env var LDFLAGS not set, default to ``
Executing: `g++-13 -W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc -DDFS -c ./src/main.cpp -o ./target/main.cpp.o `
Done.
Executing: `g++-13 -W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20 -Isrc -DDFS ./target/main.cpp.o -o ./target/main `
Done.
Executing: `python3 ./src/preprocess.py -n 22 -i ./data/附件1-2.基站图的邻接矩阵-v1-2023.xls -o ./data/graph.txt `
Done.
Executing: `./target/main ./data/graph.txt`
Time Elapsed: 16170.850 ms
Memory Usage: 9.234 MB
Node Traverse Count: 487370492
TSP Solution:
  Path with cost 7690.802: 567443 → 567260 → 566742 → 568098 → 566802 → 567439 → 566751 → 567322 → 566747 → 566720 → 33109 → 566798 → 567547 → 566999 → 566967 → 567203 → 565696 → 566993 → 566631 → 567238 → 566750 → 566783 → 567443
Done.
```

22



点此回到图片前。

除去以上截图展示的输出，我还另外使用

```
for x in 15 20 22 30; do
    echo "x = $x"
    CXX=g++-13 ./run.py -sd $x > ./ans/dfs_$x.txt
```

```
    CXX=g++-13 ./run.py -sb $x > ./ans/bfs_$x.txt
done
```

将另外一次的输出保存到了 ans 目录下，以便读者查看。

前述所有图片中包含的和算法运行结果相关的信息如下：

基站数	算法	最短 Hamilton 回路长度	最短 Hamilton 回路
15	回溯法	5506.880	567443 → 567260 → 566742 → 568098 → 566802 → 567439 → 566751 → 566798 → 567547 → 566967 → 566993 → 566631 → 567238 → 566750 → 566783 → 567443
	分支限界法	5506.880	567443 → 567260 → 566742 → 568098 → 566802 → 567439 → 566751 → 566798 → 567547 → 566967 → 566993 → 566631 → 567238 → 566750 → 566783 → 567443
20	回溯法	6987.509	567443 → 567260 → 566742 → 568098 → 566802 → 567439 → 567322 → 566751 → 33109 → 566798 → 567547 → 566999 → 566967 → 567203 → 565696 → 566993 → 566631 → 567238 → 566750 → 566783 → 567443
	分支限界法	6987.509	567443 → 567260 → 566742 → 568098 → 566802 → 567439 → 567322 → 566751 → 33109 → 566798 → 567547 → 566999 → 566967 → 567203 → 565696 → 566993 → 566631 → 567238 → 566750 → 566783 → 567443
22	回溯法	7690.802	567443 → 567260 → 566742 → 568098 → 566802 → 567439 → 566751 → 567322 → 566747 → 566720 → 33109 → 566798 → 567547 → 566999 → 566967 → 567203 → 565696 → 566993 → 566631 → 567238 → 566750 → 566783 → 567443
	分支限界法	7690.802	567443 → 567260 → 566742 → 568098 → 566802 → 567439 → 566751 → 567322 → 566747 → 566720 → 33109 → 566798 → 567547 → 566999 → 566967 → 567203 → 565696 → 566993 → 566631 → 567238 → 566750 → 566783 → 567443

30	回溯法	11426.596	565492 → 565496 → 565648 → 565621 → 565773 → 565531 → 567618 → 567497 → 565630 → 565801 → 565753 → 565562 → 566010 → 565631 → 565898 → 565675 → 567500 → 567510 → 567526 → 565964 → 567531 → 566074 → 565859 → 565610 → 565516 → 565551 → 565558 → 565559 → 567891 → 565633 → 565492
	分支限界法	11426.596	565492 → 565633 → 567891 → 565559 → 565558 → 565551 → 565516 → 565610 → 565859 → 566074 → 567531 → 565964 → 567526 → 567510 → 567500 → 565675 → 565898 → 565631 → 566010 → 565562 → 565753 → 565801 → 565630 → 567497 → 567618 → 565531 → 565773 → 565621 → 565648 → 565496 → 565492

根据表格简单比对可知：

- 对于每组输入数据，两个算法输出的最短 Hamilton 回路长度相同。
- 对于前三组输入数据，两个算法输出的最短 Hamilton 回路相同。
- 对于最后一组数据，两个算法输出的最短 Hamilton 回路虽然看似不同，但是观察发现实际上它们是同一条回路朝着不同的方向走了一圈。

可以断言，两个算法都正确解决了这个问题。

5.2 算法性能：时间与空间 [题给表格]

基站数	算法	最短 Hamilton 回路长度	搜索的状态数	运行时间 / s	消耗空间 / MB
15	回溯法	5506.880	256955	0.0061	9.219
	分支限界法		2281	0.0031	9.766
20	回溯法	6987.509	76329668	2.2898	9.297
	分支限界法		11705	0.0232	13.609
22	回溯法	7690.802	487370492	16.1708	9.234
	分支限界法		5109	0.0103	11.391

30	回溯法	11426.596	6909149416	227.7228	9.234
	分支限界法		5460186	23.8819	1294.312

这个表只作展示用。我们还是将它按照算法分成两部分来分析比较好：

基站数	**算法**	**最短 Hamilton 回路长度**	**搜索的状态数**	**运行时间 /** s	**消耗空间 /** MB
15	回溯法	5506.880	256955	0.0061	9.219
20		6987.509	76329668	2.2898	9.297
22		7690.802	487370492	16.1708	9.234
30		11426.596	6909149416	227.7228	9.234
15	分支限界法	5506.880	2281	0.0031	9.766
20		6987.509	11705	0.0232	13.609
22		7690.802	5109	0.0103	11.391
30		11426.596	5460186	23.8819	1294.312

可以得出以下结论：

- 对于回溯法：
  - 搜索的状态数、消耗的时间快速增长，但消耗的空间变化不大。
  - 搜索的搜索的状态数与基站数（结点数）并不能用时间复杂度 $\mathcal{O}(n \times n!)$ 拟合，但是大致可以拟合成 $y = 18363.6x \times 1.36966^x$，相关系数 $R^2 = 0.9994$。这是玄学剪枝导致的。
  - 同样地，时间也无法用时间复杂度 $\mathcal{O}(n \times n!)$ 拟合，但是大致可以拟合成 $y = 0.00000601886x \times 1.36992^x$，相关系数 $R^2 = 0.9993$。
  - 消耗的空间与基站数（结点数）关系非常小，可以看作正常抖动。这符合 $\mathcal{O}(n)$ 的空间复杂度理论预期。
- 对于分支限界法：
  - 搜索的状态数、消耗的时间和空间随着基站数（结点数）的增加都有增长趋势，但对于中间的两组数据反而逆转了。这也是由剪枝的不确定性导致的。三份数据都不具有拟合的价值。
- 将两种算法进行比较：

- 回溯法的搜索状态数和消耗的时间都远远超过分支限界法，其中状态数差了 3 个数量级以上，时间差了至少 1 个数量级。可以说是被分支限界法完爆了。
- 空间消耗上回溯法扳回一城。然而内存条并没有时间那么宝贵，何况分支限界法的空间消耗只有 1.3GB，也不算特别大，所以这个优势并没有那么大的用处。

我们可以大致通过算法的状态数、时间和空间消耗，推测出它们的时间复杂度都是非多项式的，但难以精确地确定时间复杂度。而且，由于剪枝的存在，算法的运行效率会和数据产生较大的关系。总之，我实现的算法的时间和空间复杂性还是相对比较优秀的。

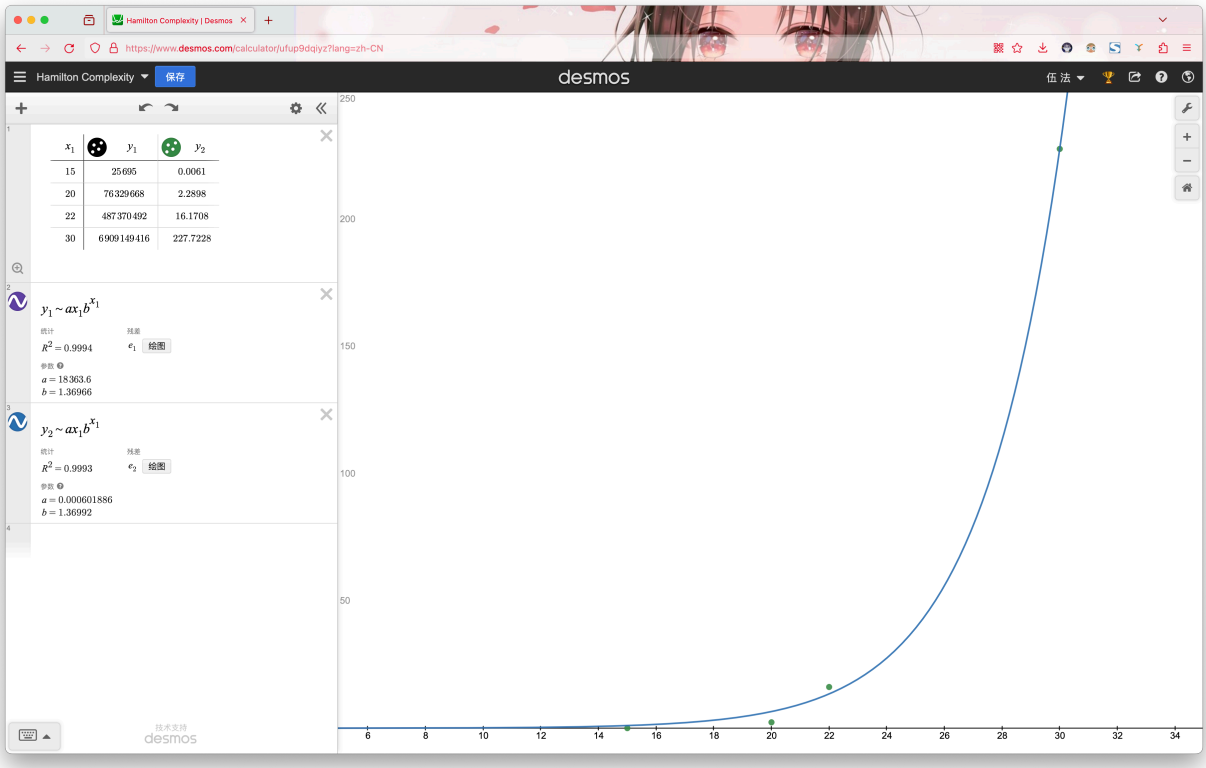


图 2　用工具进行拟合

6 总结

本次实验到此完全结束。我在实验中实现了回溯法和分支限界法，用于求解 TSP 问题[3]。由于我是算法竞赛退役选手，对算法有深入理解的要求，因此我和两个 XCPC 校集训队现役队员讨论了很久关于分支限界算法的细节，包括但不限于它的原理，复杂度分析，以及它与其他算法（例如 A* 算法）的异同。这些讨论对我理解算法的原理和实现有很大的帮助。

实验还有以下的改进方向：

[3]实际上是一般带权简单无向图上的最短 Hamilton 回路问题

- 实现解决 Hamilton 回路 / TSP 问题的正牌算法——状态压缩动态规划，与回溯法和分支限界法进行比较。
- 厘清分支限界法解决 Hamilton 回路问题的理论复杂度到底是不是 $\mathcal{O}(n^2 \times n!)$。感觉很难。
- 为运行脚本和统计内存占用的模块实现 Windows 平台的版本，使得程序能够在 Windows 上运行。

附录 A - 附件清单

```
SRH
├── ans
│   ├── .DS_Store
│   ├── bfs_15.txt
│   ├── bfs_20.txt
│   ├── bfs_22.txt
│   ├── bfs_30.txt
│   ├── dfs_15.txt
│   ├── dfs_20.txt
│   ├── dfs_22.txt
│   └── dfs_30.txt
├── run.py
└── src
    ├── algo_base.h
    ├── bfs.h
    ├── dfs.h
    ├── graph.h
    ├── main.cpp
    ├── mem.h
    ├── preprocess.py
    └── test.py
```

附录 B - 源代码

B.1 src/algo_base.h

```cpp
#pragma once

#ifndef ALGO_BASE
#  define ALGO_BASE

#  include <cstdint>
#  include <tuple>
#  include <vector>

template<typename Weight>
using Result = std::tuple<std::vector<int>, Weight, std::int64_t>;

// 等价于如下的聚合类型定义。将上面的定义修改成下面的，不改变任何其他代码，代码仍然能通过编译并正确运行。

// template<typename Weight>
// struct Result {
```

```
//   std::vector<int> path{};
//   Weight cost{};
//   std::int64_t node_count{};
// };

#endif // !ALGO_BASE
```

B.2 src/bfs.h

```
#pragma once

#include <cstddef>
#ifndef BFS_H
#  define BFS_H

#  include "algo_base.h"
#  include "graph.h"

#  include <cstdint>
#  include <queue>
#  include <set>
#  include <vector>

template<typename Weight>
class Bfs {
  const Graph<Weight> &g;

  struct Node {
    std::vector<int> path{};
    Weight cost{}, lower_bound{};

    Node(int src_): path{src_} {}

    bool operator<(const Node &rhs) const {
      return lower_bound > rhs.lower_bound;
    }
  };

  void update_lower_bound(Node &node) {
    if (static_cast<int>(node.path.size()) == g.n) {
      node.lower_bound = node.cost + g.adj[node.path.back()][g.src];
      return;
    }
```

```
    node.lower_bound = 2 * node.cost;

    std::vector<bool> visited(g.n + 1);
    for (int u : node.path)
      visited[u] = true;

    auto get_min_elem = [&](int u) -> Weight {
      Weight min_elem = Graph<Weight>::INF;
      for (int v = 1; v <= g.n; ++v)
        if (!visited[v] && g.adj[u][v] < min_elem)
          min_elem = g.adj[u][v];
      return min_elem;
    };
    auto get_sum_two_min_elem = [&](int u) -> Weight {
      Weight min_elem = Graph<Weight>::INF,
             second_min_elem = Graph<Weight>::INF;
      for (int v = 1; v <= g.n; ++v)
        if (v != u) {
          if (g.adj[u][v] < min_elem) {
            second_min_elem = min_elem;
            min_elem = g.adj[u][v];
          } else if (g.adj[u][v] < second_min_elem)
            second_min_elem = g.adj[u][v];
        }
      return min_elem + second_min_elem;
    };

    if (node.path.size() != 1) {
      node.lower_bound += get_min_elem(node.path.front());
      node.lower_bound += get_min_elem(node.path.back());
    } else
      node.lower_bound += get_sum_two_min_elem(node.path.front());

    for (int u = 1; u <= g.n; ++u)
      if (!visited[u])
        node.lower_bound += get_sum_two_min_elem(u);

    node.lower_bound /= 2;
  }

public:
  std::vector<int> min_path;
  Weight min_cost{Graph<Weight>::INF}; // global upper bound
```

```
  std::int64_t node_count{};

  Bfs(const Graph<Weight> &g_): g(g_) {}

  void run() {
    std::priority_queue<Node> q;
    std::multiset<Weight> lower_bounds;

    Node node(g.src);
    update_lower_bound(node);
    lower_bounds.emplace(node.lower_bound);
    q.push(node);

    while (!q.empty()) {
      Node node = q.top();
      q.pop();

      lower_bounds.erase(lower_bounds.find(node.lower_bound));
      if (node.lower_bound > min_cost)
        continue;
      ++node_count;

      if (static_cast<int>(node.path.size()) == g.n) {
        if (g.adj[node.path.back()][g.src] == g.INF)
          continue;

        node.cost += g.adj[node.path.back()][g.src];
        node.path.emplace_back(g.src);

        if (node.cost < min_cost) {
          min_cost = node.cost;
          min_path = node.path;
          if (node.cost == *lower_bounds.begin())
            break;
        }
        continue;
      }

      int u = node.path.back();
      std::vector<bool> visited(g.n + 1);
      for (int v : node.path)
        visited[v] = true;

      for (int v = 1; v <= g.n; ++v) {
```

```
        if (g.adj[u][v] == g.INF || visited[v])
          continue;

        Node new_node = node;
        new_node.path.emplace_back(v);
        new_node.cost += g.adj[u][v];
        update_lower_bound(new_node);

        if (new_node.lower_bound < min_cost) {
          lower_bounds.emplace(new_node.lower_bound);
          q.push(new_node);
        }
      }
    }
  }
};

template<typename Weight>
Result<Weight> bfs(const Graph<Weight> &g) {
  Bfs<Weight> bfs(g);
  bfs.run();
  return {bfs.min_path, bfs.min_cost, bfs.node_count};
}

#endif // !BFS_H
```

B.3 src/dfs.h

```
#pragma once

#ifndef DFS_H
#  define DFS_H

#  include "algo_base.h"
#  include "graph.h"

#  include <cstdint>
#  include <numeric>

template<typename Weight>
class Dfs {
  const Graph<Weight> &g;
  std::vector<int> path{};
```

```
  std::vector<bool> visited{};
  Weight cost;

public:
  std::vector<int> min_path{};
  Weight min_cost{Graph<Weight>::INF};
  std::int64_t node_count{};

  Dfs(const Graph<Weight> &g_): g(g_) {
    path.emplace_back(g.src);
    visited.resize(g.n + 1);
    visited[g.src] = true;
  }

  void run() {
    ++node_count;
    int u = path.back();

    if (static_cast<int>(path.size()) == g.n) {
      if (g.adj[u][g.src] == g.INF)
        return;

      path.emplace_back(g.src);
      cost += g.adj[u][g.src];

      if (cost < min_cost) {
        min_cost = cost;
        min_path = path;
      }

      cost -= g.adj[u][g.src];
      path.pop_back();
      return;
    }

    for (int v = 1; v <= g.n; ++v) {
      if (u == v || g.adj[u][v] == g.INF || visited[v] ||
          cost + g.adj[u][v] >= min_cost)
        continue;

      path.emplace_back(v);
      cost += g.adj[u][v];
      visited[v] = true;
      run();
```

```
      visited[v] = false;
      cost -= g.adj[u][v];
      path.pop_back();
    }
  }
};

template<typename Weight>
Result<Weight> dfs(const Graph<Weight> &g) {
  Dfs<Weight> dfs(g);
  dfs.run();
  return {dfs.min_path, dfs.min_cost, dfs.node_count};
}

#endif // !DFS_H
```

B.4 src/graph.h

```
#pragma once

#ifndef GRAPH_H
#  define GRAPH_H

#  include <algorithm>
#  include <exception>
#  include <format>
#  include <iostream>
#  include <type_traits>
#  include <vector>

// clang-format off
template<typename T>
requires std::is_arithmetic_v<T>
struct Graph {
  // clang-format on

  constexpr const static inline T INF = std::numeric_limits<T>::max() / 2;

  int n;
  int src;
  std::vector<int> id;
  std::vector<std::vector<T>> adj;
```

```
  Graph(int n_, int src_): n(n_), src(src_) {
    id.resize(n + 1);
    adj.resize(n + 1);
    for (int u = 1; u ⩽ n; ++u)
      adj[u].resize(n + 1);
  }

  static Graph<T> read_from(std::istream &is) {
    int n, src;
    is >> n >> src;
    Graph<T> g(n, src);
    for (int u = 1; u ⩽ n; ++u)
      is >> g.id[u];

    auto get_index = [&](int id) → int {
      if (auto it = std::find(g.id.begin(), g.id.end(), id);
          it ≠ g.id.end() && *it == id)
        return std::distance(g.id.begin(), it);
      throw std::invalid_argument(std::format("Invalid vertex id {}", id));
    };
    g.src = get_index(g.src);

    for (int u = 1; u ⩽ n; ++u)
      for (int v = 1; v ⩽ n; ++v) {
        is >> g.adj[u][v];

        if (std::abs(g.adj[u][v] - T(99999)) <
            std::numeric_limits<T>::epsilon())
          g.adj[u][v] = g.INF;
      }

    for (int u = 1; u ⩽ n; ++u)
      g.adj[u][u] = 0;
    return g;
  }
};

#endif // !GRAPH_H
```

B.5 src/main.cpp

```
#if __cplusplus < 202002L
#  error "请使用 C++20 或更新的语言标准编译此程序"
```

```
#endif

#include "algo_base.h"
#include "bfs.h"
#include "dfs.h"
#include "graph.h"
#include "mem.h"

// 所有包含的头文件都用到了，
#include <atomic>
#include <chrono>
#include <cstdlib>
#include <format>
#include <fstream>
#include <functional>
#include <future>
#include <iostream>
#include <string>
#include <string_view>
#include <thread>

using namespace std::string_view_literals;

constexpr const std::string_view PATH = "data/graph.txt"sv;

template<typename Weight>
void algo_main(const Graph<Weight> &g) {
  auto start = std::chrono::high_resolution_clock::now();
  std::atomic<bool> recording_mem(true);
  std::future<unsigned long> mem_usage_monitor = std::async(
      std::launch::async, monitor_mem_usage, std::ref(recording_mem)
  );

  auto [path, cost, node_count] =
#ifdef DFS
      dfs
#else // including #if defined(BFS)
      bfs
#endif
      (g);

  auto path_to_string = [&](const std::vector<int> &path) {
    std::string buf;
    for (int u : path)
```

```
      buf += std::format("{} → ", g.id[u]);
    if (!buf.empty())
      buf.pop_back(), buf.pop_back(), buf.pop_back();
    return buf;
  };

  auto end = std::chrono::high_resolution_clock::now();
  recording_mem = false;
  auto mem_usage = mem_usage_monitor.get();

  std::cout << std::format(
      "Time Elapsed: {:.3f} ms\nMemory Usage: {:.3f} MB\nNode Traverse Count: "
      "{}\nTSP Solution:\n  Path with cost {:.3f}: {}\n",
      static_cast<double>(
          std::chrono::duration_cast<std::chrono::microseconds>(end - start)
              .count()
      ) / 1000,
      static_cast<double>(mem_usage) / 1024 / 1024, node_count, cost,
      path_to_string(path)
  );
  std::cout << std::endl;
}

int main() {
  auto fin = std::ifstream(PATH.data());
  if (!fin.is_open()) {
    std::cerr << "Failed to open file: " << PATH << std::endl;
    return 1;
  }

  algo_main(Graph<double>::read_from(fin));
}
```

B.6 src/mem.h

```
#pragma once

#ifndef MEM
#  define MEM

#  include <algorithm>
#  include <atomic>
#  include <future>
```

```
#  include <thread>

#  if defined(__APPLE__)
#    include <mach/mach.h>
#  elif defined(__linux__)
#    include <sys/resource.h>
#  else
#    error "Unsupported platform, only macOS and Linux are supported."
#  endif

inline unsigned long get_mem_usage() {
#  ifdef __APPLE__
  struct task_basic_info t_info;
  mach_msg_type_number_t t_info_count = TASK_BASIC_INFO_COUNT;
  return task_info(
             mach_task_self(), TASK_BASIC_INFO, (task_info_t)&t_info,
             &t_info_count
         ) == KERN_SUCCESS
             ? t_info.resident_size
             : 0;
#  else
  struct rusage r_usage;
  getrusage(RUSAGE_SELF, &r_usage);
  return r_usage.ru_maxrss * 1024;
#  endif
}

inline unsigned long monitor_mem_usage(std::atomic<bool> &keep_running) {
  unsigned long mem_usage = get_mem_usage();
  while (keep_running.load()) {
    std::this_thread::sleep_for(std::chrono::milliseconds(5));
    mem_usage = std::max(mem_usage, get_mem_usage());
  }
  return mem_usage;
}

#endif // !MEM
```

B.7 src/preprocess.py

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-
```

```python
from typing import Dict, List

import argparse
import dataclasses
import sys
import xlrd


@dataclasses.dataclass
class AdjMat:
    n: int
    src: int
    id: List[int]
    weight: List[List[float]]


def read_adj_mat(path: str, n: int, sheet_index: int) -> AdjMat:
    book = xlrd.open_workbook(path)
    sheet = book.sheet_by_index(sheet_index)

    first_row = sheet.row_values(0)
    # n = int(max(first_row[2:]))

    second_row = sheet.row_values(1)
    id = list(map(int, second_row[2:]))

    weight = []
    for i in range(2, n + 2):
        row = sheet.row_values(i)
        weight.append(list(row[2:]))

    # inject
    src = 567443
    if src not in id:
        src = id[0]

    return AdjMat(n, src, id, weight)


def write_adj_mat(path: str, graph: AdjMat) -> None:
    buf = []

    buf.append(f'{graph.n} {graph.src}\n')
```

```
    buf.append(' '.join(map(str, graph.id)))
    buf.append('\n')
    for row in graph.weight:
        buf.append(' '.join(map(str, row)))
        buf.append('\n')

    with open(path, 'w') as f:
        f.writelines(buf)


def main() -> None:
    parser = argparse.ArgumentParser()
    parser.add_argument('-n', '--amount', dest='amount', type=int)
    parser.add_argument('-i', '--input', dest='file_path', type=str)
    parser.add_argument('-o', '--output', dest='output_path', type=str)
    args = parser.parse_args()

    VALID_AMOUNT: Dict[int, int] = {
        15: 0,
        20: 1,
        22: 2,
        30: 3,
        5: 4,
    }

    if not args.amount or not args.file_path or not args.output_path:
        parser.print_help()
        exit(1)

    if not args.amount or args.amount not in VALID_AMOUNT:
        sys.stderr.write(
            'Must specify amount with one of the following values: '
            f'{", ".join(map(str, VALID_AMOUNT.keys()))}\n'
        )
        exit(1)

    write_adj_mat(
        args.output_path,
        read_adj_mat(args.file_path, args.amount, VALID_AMOUNT[args.amount])
    )


if __name__ == '__main__':
    main()
```

B.8 run.py

```python
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

from dataclasses import dataclass
from typing import List, Literal

import argparse
import os
import subprocess
import sys


SRC_DIR: str = './src'
TARGET_DIR: str = './target'
DATA_DIR: str = './data'

SRC_FILES: List[str] = [
    'main.cpp',
]

silent: bool = False


def print_info(s: str) -> None:
    global silent
    if not silent:
        print(s)


class Argument:
    @dataclass
    class Default:
        CXX: str = 'g++'
        CXXFLAGS: str = '-W -Wall -Wextra -Wpedantic -Wno-psabi -O3 -std=c++20
-Isrc'
        # CXXFLAGS: str = '-W -Wall -Wextra -Wpedantic -Wno-psabi -Og -g -std=c+
+20 -Isrc'
        LDFLAGS: str = ''

    def __init__(self):
        default = Argument.Default()
        for key in ('CXX', 'CXXFLAGS', 'LDFLAGS'):
```

```
            self.__setattr__(key, os.getenv(key))
            if self.__getattribute__(key) is None:
                print_info(
                    f'Env var {key} not set, '
                    f'default to `{default.__getattribute__(key)}`'
                )
                self.__setattr__(key, default.__getattribute__(key))


def run_cmd(*kargs) -> int:
    cmd = ' '.join(kargs)
    print_info(f'\033[1;34mExecuting:\033[0m `{cmd}`')
    sys.stdout.flush()
    ret = subprocess.run(cmd.split()).returncode
    if ret != 0:
        print_info(f'\033[1;31mFailed.\033[0m')
    else:
        print_info(f'\033[1;34mDone.\033[0m')
    return ret


def install_deps() -> None:
    try:
        import xlrd as _
    except ImportError:
        print_info('xlrd not installed, installing...')
        run_cmd(f'{sys.executable} -m pip install xlrd~=2.0.1')


def clean() -> None:
    run_cmd(f'rm -rf {TARGET_DIR}')


def preprocess(amount: int) -> None:
    install_deps()
    run_cmd(
        f'python3 {SRC_DIR}/preprocess.py '
        f'-n {amount} '
        f'-i {DATA_DIR}/附件 1-2.基站图的邻接矩阵-v1-2023.xls '
        f'-o {DATA_DIR}/graph.txt '
    )


def compile(type: Literal['dfs', 'bfs']) -> None:
```

```
    clean()

    if not os.path.exists(TARGET_DIR):
        os.mkdir(TARGET_DIR)

    arg = Argument()
    CXX: str = arg.CXX  # type: ignore
    CXXFLAGS: str = arg.CXXFLAGS  # type: ignore
    LDFLAGS: str = arg.LDFLAGS  # type: ignore

    if type == 'dfs':
        CXXFLAGS += ' -DDFS'
    else:
        CXXFLAGS += ' -DBFS'

    for src_file in SRC_FILES:
        if run_cmd(
            f'{CXX} {CXXFLAGS} -c {SRC_DIR}/{src_file} '
            f'-o {TARGET_DIR}/{src_file}.o {LDFLAGS}'
        ) != 0:
            exit(1)

    if run_cmd(
        f'{CXX} {CXXFLAGS} '
        f'{" ".join([f"{TARGET_DIR}/{src_file}.o" for src_file in SRC_FILES])}'
        f' -o {TARGET_DIR}/main {LDFLAGS}'
    ) != 0:
        exit(1)


def run() -> None:
    run_cmd(f'{TARGET_DIR}/main {DATA_DIR}/graph.txt')


def main() -> None:
    parser = argparse.ArgumentParser()

    parser.add_argument(
        '-d', '--dfs', type=int, dest='dfs',
        help='run dfs with the given node amount'
    )
    parser.add_argument(
        '-b', '--bfs', type=int, dest='bfs',
        help='run bfs with the given node amount'
```

```
    )
    parser.add_argument(
        '-c', '--clean', action='store_true', help='clean the target dir'
    )
    parser.add_argument(
         '-s', '--silent', action='store_true', help='do not print execution
info', default=False
    )
    args = parser.parse_args()

    global silent
    silent = args.silent

    if args.dfs and args.bfs:
        print_info('Please specify only one of dfs and bfs')
        exit(1)
    if not any((args.dfs, args.bfs, args.clean)):
        parser.print_help()
        exit(1)

    if args.clean:
        clean()
    elif args.dfs:
        compile('dfs')
        preprocess(args.dfs)
        run()
    elif args.bfs:
        compile('bfs')
        preprocess(args.bfs)
        run()


if __name__ == '__main__':
    main()
```

点此回到附录以前。