北京邮电大学

计算机学院

贪心算法设计与分析实验报告

(分别使用 c++ 和 Rust 实现)

姓名: 吴清柳

学号: 2020211597

班级: 2020211323

指导老师: 邵蓥侠

课程名称: 算法设计与分析

December 3, 2022

Contents

1 实验题目

1.1 Huffman 算法

1.1.1 题目描述

对给出的字符设计 Huffman 编码, 计算期望:

$$W = \sum_{1}^{n} P_i \times L_i \tag{1}$$

1.1.2 输入格式

- 输入文件名为 huffman.in, 输入共两行.
- 第一行一个正整数 n, 代表字符个数.
- 第二行包含 n 个三位小数 P_i, 代表第 i 个字符的出现概率, 两个数字之间用空格隔开.

1.1.3 输出格式

- 输出文件名为 huffman.out, 输出共一行.
- 第一行包含一个三位小数 W, 为最后的期望.

1.1.4 输入输出样例

huffman.in	huffman.out
4 0.100 0.100 0.200 0.600	1.600

1.1.5 数据范围

$0 < n \leq 10^6, 0 < P_i \leq 1$.

1.1.6 解法

使用两种方法实现. C++ 版使用最小堆算法实现建树, Rust 版使用不稳定排序算法实现建树, 用以对比不同算法的速度, 以及验证正确性.

1.2 Dijkstra 算法

1.2.1 题目描述

求出给定顶点 1 与顶点 V 之间的最短路.

1.2.2 输入格式

- 输入文件名为 dijkstra.in.
- 第一行有两个正整数 V, E, 分别代表顶点数, 边数.
- 接下来 E 行包含三个正整数 u, v, w, 代表 u 和 v 之间存在一条权值为 w 的无向边.

1.2.3 输出格式

- 输出文件名为 dijkstra.out, 输出共一行.
- 第一行包含一个整数, 为最短路上权值之和, 若顶点 1 与顶点 V 不连通, 输出-1.

1.2.4 输入输出样例

dijkstra.in	dijkstra.out
3 3 1 2 5 2 3 5 3 1 2	2

1.2.5 数据范围

$0 < V \leq 5000, 0 < E \leq 2 \times 10^5, 0 < u, v \leq V, 0 < w \leq 2 \times 10^5$.

1.2.6 解法

使用两种数据结构实现图. C++ 版使用链表实现邻接表来存图, Rust 版使用二维可变数组实现邻接表来存图, 用以对比不同数据结构对速度的影响, 以及对比正确性.

1.3 Prim 算法

1.3.1 题目描述

使用 **Prim** 算法计算给定无向图上以顶点 1 为根的最小生成树的边权和.

1.3.2 输入格式

- 输入文件名为 prim.in.
- 第一行包含两个正整数 V, E, 分别代表顶点数和边数.

- 接下来 E 行包含三个正整数 u, v, w, 代表 u 和 v 之间存在一条权值为 w 的无向边.

1.3.3 输出格式

- 输出文件名为 prim.out, 输出共一行.
- 第一行包含一个整数代表最小生成树的权值和.

1.3.4 输入输出样例

prim.in	prim.out
4 5 1 2 2 1 3 2 1 4 3 2 3 4 3 4 3	7

1.3.5 数据范围

$0 < V \leq 5000$, $0 < E \leq 2 \times 10^5$, $0 < u, v \leq V$, $0 < w \leq 2 \times 10^5$.

1.3.6 解法

使用两种数据结构实现存图. C++ 版使用链表存储邻接表, Rust 版使用二维可变数组存储邻接表. 用以通过比较来验证算法的正确性.

1.4 Kruskal 算法

1.4.1 题目描述

使用 **Kruskal** 算法计算给定无向图上以顶点 1 为根的最小生成树的边权和.

1.4.2 输入格式

- 输入文件名为 kruskal.in.
- 第一行包含两个正整数 V, E, 分别代表顶点数和边数.
- 接下来 E 行包含三个正整数 u, v, w, 代表 u 和 v 之间存在一条权值为 w 的无向边.

1.4.3 输出格式

- 输出文件名为 kruskal.out, 输出共一行.
- 第一行包含一个整数代表最小生成树的权值和.

1.4.4 输入输出样例

kruskal.in	kruskal.out
4 5 1 2 2 1 3 2 1 4 3 2 3 4 3 4 3	7

1.4.5 数据范围

$0 < V \leq 5000, 0 < E \leq 2 \times 10^5, 0 < u, v \leq V, 0 < w \leq 2 \times 10^5$.

1.4.6 解法

使用两种数据结构实现联通分支 connected branch. C++ 版使用 HashSet 实现联通分支的创建, 合并和存储, Rust 版使用 disjoint set union 来模拟联通分支. 用以通过比较来验证算法的正确性, 并比较两种数据结构的效率.

2 实验过程

2.1 huffman

2.1.1 算法类别

此处介绍的 Huffman 编码是一种贪心算法.

2.1.2 算法思路

2.1.2.1 Huffman 算法的贪心选择性质 即证明 Huffman 算法中, 让出现频率越低的字符的编码越长可以得到最优的前缀码, 即让平均码长最短.

- 设字符集 C 的一个最优前缀码表示为二叉树 T. 采用一定的方法, 将 T 修改得到新树 T', 且互为兄弟.
- T' 还是 C 的最优前缀. 这样 x, y 在最优前缀码 T' 中只有最后一位不同.

假设: b, c 是 T 中最深的叶子且互为兄弟, $f(b) \leq f(c)$.

已知: C 中 2 个最小频率字符 $f(x) \leq f(y)$, 但是在 T 中, x, y 可能不是最深节点. 由于 x, y 具有最小频率, 所以 $f(x) \leq f(b), f(y) \leq f(c)$. 之后进行如下操作:

1. 在 T 中交换 b 和 x 的位置得到 T1.
2. 在 T1 中交换 c 和 y 的位置得到 T'.

易得, 由于 $B(T) - B(T1) \leq 0$, 故第一步交换不会增加平均码长; 由于 $B(T1) - B(T') \leq 0$, 故第二步交换也不会增加平均码长. 故 T' 的码长仍旧是最短的, 即 T' 是最优前缀码, 且其最小频率的 x, y 具有最深的深度, 对应的哈弗曼编码也是最长的, 且只有最后一位不同. 因此, Huffman 算法的贪心选择性质得证.

2.1.2.2 Huffman 算法的最优子结构性质

即证明给定字符集 C 和其对应的最优前缀码 T, 可以从中得到子问题 C'(C 的子集) 及其对应的最优前缀子树 T'.

构造性证明: 对 T 中 2 个互为兄弟的叶节点 x 和 y, z 为其父节点. 将 z 看做频率为 $f(z) = f(x) + f(y)$ 的字符, 则 $T' = T - \{x, y\}$ 是子问题 $C' = (C - \{x, y\} \cup \{z\})$ 的最优编码. 得到子问题的最优编码后, 原问题的最优编码也自然知道了.

T 的平均码长 $B(T)$ 可以用子树 T 的平均码长 $B(T')$ 来表示:

$$B(T) = B(T') + 1 \times f(x) + 1 \times f(y) \tag{2}$$

可以证明, T' 所表示的 C' 的前缀码的码长 $B(T')$ 是最优的. 反证法证明:

假设还有另外一个 T'', 是子问题 C' 的最优前缀码, 即 $B(T') > B(T'')$. 节点 z 在 T'' 还是一个叶节点, 在 T'' 中将其替换为其子节点 x, y, 得到 T'''. 则由式 huffman 最优子结构递推式得, $B(T''') < B(T)$, T''' 的码长比 T 更短, 与 T 是最优解矛盾. 故子问题是最优的.

2.1.3 关键函数及代码段的描述

其伪代码描述如下:

Algorithm 1 Huffman algorithm

Require: C
  $n = |C|$
  Q = C
  for i=1 to n-1 **do** allocate a new node z
    $z.left = x = Extract - Min(Q)$
    $z.right = y = Extract - Min(Q)$
    $z.freq = x.freq + y.freq$
    $Insert(Q, z)$
  end forreturn Extract-Min(Q) // Return the root of the tree

2.1.3.1 Huffman 算法的时间复杂性分析

使用二叉最小堆进行建树, 堆中最多需要存放 |C| 个顶点, 空间复杂度为 $O(C)$.

2.1.3.2 Huffman 算法的空间复杂性分析 对 |C| 个字符, 进行 |C|-1 次合并后可以得到 Huffman 树. 因此, 代码段 1中的循环进行 n-1 次. 在每次循环中, 需要执行两次 Extra-Min 操作, 由于使用了二叉最小堆, 每次操作的时间复杂度为 $O(lgC)$. 执行一次 Insert 操作, 时间复杂度也为 $O(lgC)$. 据此可得

$$T = O(C) \times O(lgC) = O(ClogC) \tag{3}$$

故总的时间复杂度为 $O(ClogC)$.

2.2 Dijkstra

2.2.1 算法类别

Dijkstra 单源最短路径算法属于贪心算法.

2.2.2 问题描述

给定带权有向图 $G = (V,E)$, 其中每条边的权是非负实数. 另外, 还给定 V 中的一个顶点, 称为源. 现在要计算从源到所有其他各顶点的最短路长度. 此处的最短路长度指路上各边权之和. 这个问题通常称为单源最短路径问题.

2.2.3 Dijkstra 算法的基本思路

求解的基本思路是, 设置顶点集合 S 并不断地进行贪心选择来扩充这个集合. 一个顶点属于集合 S 当且仅当从源到该顶点的最短路径长度已知.

初始的时候, S 中仅含有源. 设 u 是 G 的某一个顶点, 把从源到 u 且中间只经过 S 中顶点的路称为从源到 u 的特殊路径, 并记录当前每个顶点所对应的最短特殊路径长度. Dijkstra 算法每次从 V-S 中取出具有最短特殊路径长度的顶点 u, 将 u 添加到 S 中, 同时更新影响到的顶点所对应的最短特殊路径长度. 一旦 S 中包含了所有 V 中的顶点, 就已经完成了从源到其他顶点之间的最短路径长度.

2.2.4 Dijkstra 算法的贪心选择性质

贪心选择策略 在每步迭代的时候, 从 $V - S$ 中选择具有最短特殊路径 u.dist 的顶点 u, 加入 S.

贪心策略的正确性 需要证明对顶点 u, 从 v 开始, 经过 G 中任意顶点到达 u 的全局最短路径长度 $d(v,u) = u.dist$, 其中 u.dist 是从 v 开始, 只经过 S 中的顶点到达 u 的最短路径. 即不存在另一条 v 到 u 的最短路径, 使得其中某些节点在 $V - S$ 中, 且该路径的长度 $d(v,u) < u.dist$.

使用反证法证明.

假设:

1. 在迭代求解的过程中, 顶点 u 是 V-U 中 u.dist 最小的顶点, 且是第一个满足全局最短路径不完全在 S 集合中的顶点;

2. 从 v 到 u 的全局最短路径上, 第一个属于 $V-S$ 的顶点为 x.

则有:

首先, 因为 u 是第一个满足全局最短路径不完全在 S 集合中的顶点, 即

$$d(v,u) < u.dist \tag{4}$$

而 x 是在 u 之前遇到的顶点, x 的最短路径完全在 S 中, 因此

$$x.dist = d(v,x) \leq d(v,u) \tag{5}$$

由式 4和式 5得,

$$x.dist = d(v,x) \leq d(v,u) < u.dist \tag{6}$$

但是, 由假设得, 顶点 u 是 V-U 中 u.dist 最小的顶点, 否则不会是 dijkstra 算法中选择的下一个 V-U 中的顶点, 即有

$$u.dist \leq v.dist \tag{7}$$

这与式 6中的 $x.dist < u.dist$ 矛盾.

故贪心选择策略正确.

2.2.5 Dijkstra 的最优子结构性质

对图 $G(V,E)$, 源点 v, 问题从以下三方面描述:

1. 源点 v, 途中顶点集合 V 在算法的迭代过程中均保持不变;
2. 用于构造最短路径的当前顶点集合 S_i, 不断增加, 定义不同规模的子问题;
3. 指标: 相对于现有的 S_i, 对各顶点 u, 记录其到源点的距离 u.distance.

不同的 S_i, 定义了不同规模的子问题; 当 $S_i = V$ 的时候, 算法迭代结束, 此时 $u.distance, u \in V$ 表示途中任意顶点到源的距离.

对越小的 S_i, 问题越简单, 故自下而上求解.

对顶点 i($i|i \in V-U$), 考察将 u 加入到 S 前后, i.dist 的变化.

1. 假设添加 u 后, 出现了一条从 v 到 V-U 中另一个节点 i 的新路, 该路径从 v 经过 S_{old} 中的顶点到达 u, 再经过 u 的一条直接边到达 i, 该路径的最短长度为 $u.dist + w(u,i)$. 如果 $u.dist + w(u,i) < i.dist_{old}$, 则算法将更新 i.dist. 否则, 不更新.
2. 如果新路径先经过 u, 再回到 S 中的 x. 由于 x 比 u 先加入 u 中, 故 x.dist 已经是最短路径, $x.dist \leq u.dist + d(u,x)$. 此时, 从原 v 到 i 的最短路径 i.dist 小于路径 (v,u,x,i) 的长度, 保持算法更新 i.dist 的时候不需要考虑该路径, u 的加入对 dist[i] 无影响.

因此, 无论算法中 i.dist 的值是否变化, 它总是关于当前顶点集合 S 到顶点 u 的最短路径. 即对加入 u 之前和之后的 S_{old} 和 S_{new} 对应的两个子问题, 算法的执行过程保证了 $(i.dist|i)$ 始终是 i 的最优解.

2.2.6 关键函数及代码段的描述

其伪代码描述如下:

Algorithm 2 Dijkstra's algorithm

Require: (G, w, s)
  $S = \emptyset$
  $Q = G.V$
  while $Q \neq \emptyset$ **do**
    $u = Extract - Min(Q)$
    $S = S \cup \{u\}$
    for $eachvertexv \in G.Adj[u]$ **do**
      Relax(u, v, w)
    end for
  end while

Algorithm 3 Relax(x)

Require: (u, v, w)
  if $v.d > u.d + w(u, v)$ **then**
    $v.d = u.d + w(u, v)$
  end if

其核心代码实现如下:

2.2.7 算法时间及空间复杂性分析

2.2.7.1 空间复杂度分析 Dijkstra 算法使用了邻接表存储图, 对 V 个顶点, E 条边的情况, 空间复杂度为 $O(V+E)$. 在使用了二叉最小堆进行优化的情况下, 需要额外的 $O(V)$ 空间用于存储二叉堆数组.

2.2.7.2 时间复杂度分析 使用了二叉最小堆优化的 Dijkstra 算法对每一个顶点执行了一次 Insert 和 Extract-Min 操作. 因为每个顶点 $u \in V$ 都被加入 S 一次, 邻接表 Adj[u] 中的每条边在 for 循环中也被检查了 1 次. 由于所有邻接表中边的数量为 $|E|$, 故 for 循环重复了 $|E|$ 次, 因此算法更新 u.dist 的次数最多为 $|E|$ 次.

Dijkstra 算法的时间复杂度与二叉最小堆的时间复杂度密切相关. 在二叉最小堆中, Extract-Min 操作的开销为 $O(lgV)$, 共进行 $|V|$ 次. 建立二叉最小堆的时间为 $O(V)$. 每次修改 u.distance 操作的开销为 $O(lgV)$, 且最多有 $|E|$ 次这样的操作. 故总时间复杂度为 $O((V+E)lgV)$. 如果所有的顶点都能从源到达, 则总时间复杂度为 $O(ElgV)$. 但是, 从原本的 $O(V^2)$ 时间复杂度优化到 $O(ElgV)$ 的前提是图是稀疏的, $E = o(V^2/lgV)$.

2.3 Prim

Prim 算法和 Kruskal 算法都属于最小生成树算法.

2.3.1 最小生成树

生成树 设一个网络表示为无向连通带权图 $G=(V,E)$, E 中每条边 (u,v) 的权为 $w(u,v)$. 如果 G 的子图 G' 是一棵包含 G 的所有顶点的树, 则称 G' 为 G 的生成树.

生成树的成本 生成树上各边权的总和.

G 的最小生成树 在 G 的所有生成树中, 耗费最小的生成树.

2.3.1.1 最小生成树性质

基于贪心选择策策略, 构造最小生成树算法:

1. Kruskal 算法
2. Prim 算法

这 2 个算法的贪心选择方法不同, 但是都利用了最小生成树 (MST) 性质:

设 $G=(V,E)$ 是连通带权图, 顶点集 U 是 V 的真子集. 如果:

1. $(u,v)\in E$ 为横跨点集 U 和 V-U 的边, 即 $u\in U, v\in V-U$, 并且
2. 在所有这样的边中, (u,v) 的权 $w(u,v)$ 最小, 则一定存在 G 的一颗最小生成树, 它以 (u,v) 为其中一条边, 即 (u,v) 出现在最小生成树中.

2.3.1.2 最小生成树性质证明

使用反证法.

假设对 G 的任意一个最小生成树 T, 针对点集 U 和 V-U, $(u,v)\in E$ 为横跨这两个点集的最小权边, T 不包含该最小权边 (u,v), 但是 T 包含 u 和 v.

将 (u,v) 添加到树 T 中, 树 T 变为含回路的子图, 且该回路上有一条不同于 (u,v) 的边 $(u',v'), u'\in U, v'\in V-U$. 将 (u',v') 删去, 得到另一个树 T', 即树 T' 是通过将 T 中的边 (u', v') 替换为 (u,v) 得到的.

由于这两条边的耗费满足 $w(u,v)\leq w(u',v')$, 因此用较小耗费的边 (u,v) 替换后得到的树 T' 的耗费更小, 即

$$Cost(T')\leq Cost(T) \tag{8}$$

这与 T 是任意最小生成树的假设矛盾.

故 MST 的性质得证.

2.3.2 算法类别

Prim 算法属于贪心算法.

2.3.3 问题描述

设 $G=(V,E)$ 是连通带权图, $V=\{1,2,\ldots,n\}$. 求节 2.3.1所描述的最小生成树.

2.3.4 Prim 算法的基本思路

1. 首先置顶点集合 $S = \{1\}$
2. 当 S 是 V 的真子集的时候, 作如下贪心选择: 选取满足条件 $i \in S, j \in V - S$, 且 $w(i, j)$ 最小的边 (i,j), 将顶点 j 添加到 S 中, 边 (i,j) 添加到边集 T 中.
3. 重复上述过程, 直到 S=V 为止. 此时边集 T 就是一棵最小生成树.

该算法的时间复杂度为 $O(N^2)$.

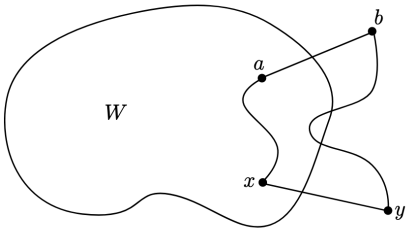
2.3.5 Prim 算法的正确性证明

即证明: 如果 S 是 Prim 算法从图 $G = (V, E)$ 中选择的生成树, 则 S 是 G 的最小生成树.

证明: 用反证法. 假设 S 不是权重和最小的. 设 $ES = (e_1, e_2, \cdots, e_{n-1})$ 是由 Prim 算法按顺序选择的边序列, U 为 ES 生成的最小生成树, 其中的边与 ES 有尽可能长的重合前缀.

设 $e_i = (x, y)$ 为 Prim 算法选出来的边序列中第一条不在 U 中的边, W 为在 (x, y) 选择之前的点集合. 易知 U 包含边 $e_1, e_2, \cdots, e_{i-1}$ 但不包含边 e_i.

在 U 中一定有一条路径 $x|y \in U$, 使得 (a,b) 是这条路径上的第一条边, 且一个顶点在 W 中, 另一个顶点在 W 外.



设边集 $T = U + \{(x, y)\} - \{(a, b)\}$, 其中 (x,y) 是前面提到的 ES 中的边. 注意到 T 是图 G 的一个生成树. 考虑边 (x,y) 和 (a,b) 的权重的三种可能情况:

Case 1 $w(a, b) > w(x, y)$. 在这种情况下, 创建 T 的时候树的权重总和减小, 因此 $w(T) < w(U)$. 这与 U 是最小生成树矛盾.

Case 2 $w(a, b) = w(x, y)$. 在这种情况下 $w(T) = w(U)$, 因此 T 也是最小生成树. 此外, 由于 Prim 算法还没有选择边 (a,b), 这条边不可能在 $e_1, e_2, \cdots, e_{i-1}$ 中. 这说明 T 包含边 $e_1, e_2, \cdots, e_i$, 即 T 中含有 ES 的前缀比 U 的更长, 这与 U 的定义矛盾.

Case 3 $w(a, b) < w(x, y)$. 在这种情况下, 因为 w(a,b) 更小, Prim 算法在这一步会选择边 (a, b), 矛盾.

综上, Prim 选出的是最小生成树. 至此, 完成了 Prim 算法的正确性证明.

2.3.6 关键函数及代码段的描述

Require: (G, w, r)
  for $each u \in G.V$ **do**
    $u.distance = \infty$
  end for
  r.distance = 0
  Q = G.V
  while $Q \neq \emptyset$ **do**
    $u = Extract - Min(Q)$
    for $each v \in G.Adj[u]$ **do**
      if $v \in Q and w(u,v) < v.distance$ **then**
        v.distance = w(u,v)
      end if‘
    end for
  end while

其核心代码实现如下:

2.3.7 算法时间及空间复杂性分析

2.3.7.1 空间复杂度分析 使用邻接表存图, 空间复杂度为 $O(V+E)$. 使用二叉最小堆维护顶点集合, 空间复杂度为 $O(V)$.

2.3.7.2 时间复杂度分析 由于使用了二叉最小堆, 堆的初始化开销为 $O(|V|lg|V|)$. 对源进行 Decrease-Key 开销为 $O(lg|V|)$. while 循环中, |V| 的 Extract-Min 操作开销 $O(|V|lg|V|)$. $\leq |E|$ Decrease-Key 开销为 $O(|E|lg|E|$. 总计为

$$\begin{aligned} T =& O(|V|\lg|V|) + O(\lg|V|) + O(|V|\lg|V|) + O(|E|\lg|E|) \\ =& O(E\lg V) \end{aligned}$$

由于图 G 是联通的, 故 $\lg|E| = \Theta(\lg|V|)$, 时间复杂度为 $O(E\lg V)$.

2.4 Kruskal

2.4.1 算法类别

Kruskal 算法属于贪心算法.

2.4.2 问题描述

设 $G = (V,E)$ 是连通带权图, $V = \{1,2,\ldots,n\}$. 求节 2.3.1所描述的最小生成树.

2.4.3 Kruskal 算法的基本思路

1. 将 G 的 n 个顶点看做 n 个孤立的联通分支.
2. 将所有的边按权从小到大排序.
3. 从权最小的第一条边开始, 依边权递增的顺序查看每一条边 (v, w), 并按下述方法链接 2 个不同的联通分支:

   当查看到第 k 条边 (v,w) 的时候,

   (a) 如果端点 v 和 w 分别是当前 2 个不同的联通分支 T_1 和 T_2 中的顶点的时候, 用边 (v,w) 将 T_1 和 T_2 合并成一个联通分支, 然后继续查看后续第 $k+1$ 条边.

   (b) 如果端点 v 和 w 已经属于当前的同一个联通分支中, 不允许将 (v,w) 加入, 否则会产生回路. 此时, 直接查看后续第 k$k+1$ 条边.

   (c) 重复上述过程, 直到只剩下一个联通分支, 即最小生成树.

2.4.4 Kruskal 算法的正确性证明

即证明: 如果 T 是 Kruskal 算法从图 $G=(V,E)$ 中选择的生成树, 则 T 是 G 的最小生成树.

首先, T 是一棵生成树.

T 是森林 由联通分支的连接方式可知, 在 T 中没有环生成.

T 是生成的 假设有一个顶点 $v \in G$ 不在 T 中的边中. 则 v 作为顶点的边一定在算法的某一步被考虑过. 有且仅有以 v 为顶点的最小的边应该已经被包括了, 否则有环, 与 T 的定义冲突.

T 是连通的 假设 T 不是联通的, 则 T 由两个或更多的联通分支. 由于 G 是联通的, 过这些联通分支一定会被 G 中不在 T 中的边连接. 这些边中有且仅有最小的一个应该已经被 T 包含了, 因为 T 中不能有环, 否则与 T 的定义矛盾.

其次, T 是最小生成树. 使用归纳法证明. 让 T^* 是一棵最小生成树, 如果 $T=T^*$, 则 T 是最小生成树. 如果 $T \neq T^*$, 则 $\exists e \in T^*$ **有最小权重**且不在 T 中. 因此, $T \cup e$ 有一个环 C 使得

- 由 Kruskal 算法建立 T 的过程知, C 中的每条边的全都小于 $w(e)$, 否则 e 将在 C 的建立过程中的某一步被考虑.
- $\exists f \in C$ 使得 f 不在 T^* 中. (因为 T^* 没有环 C).

考虑树 $T_2 = T - f + e$:

1. T_2 是生成树.
2. 相比 T, T_2 与 T^* 共有的边更多.

3. $w(T_2) \geq w(T)$. (我们将 T 中的边 f 与 T^* 中的边 e 进行了交换, 而 $w(f) < w(e)$).

重复上面的过程, 每次用相同方法替换一条边, 直到达到 T*. 有

$$w(T) \leq w(T_2) \leq w(T_3) \leq \cdots \leq w(T^*) \tag{9}$$

由于 T^* 是最小生成树, 这些不等号必须相等. 因此, T 也是最小生成树. Kruskal 算法的正确性得证.

2.4.5 关键函数及代码段的描述

使用 **disjoint-union set** 数据结构来模拟联通分支对 Kruskal 算法进行优化. Make-Set(), Find-Set() 和 Union() 为 disjoint-union set 数据结构的三个操作.

Algorithm 4 Kruskal 算法

Require: (G, w)
  $A = \emptyset$
  for each vertex $v \in G.V$ **do**
    $Make - Set((v))$ sort the edges of G.E into non-decreasing order by weight w
  end for
  for each edge $(u, v \in G.E$, taken in non-deecreasing order by weight **do**
    if $Find - Set(u) \neq Find - Set(v)$ **then**
      $A = A \cup \{(u, v)\}$
      $Union(u, v)$
    end if
  end forreturn A

Algorithm 5 Disjoint-set union 的 Make-Set 算法

Require: x
  x.p = x
  x.rank = 0

Algorithm 6 Disjoint-set union 的 Union 算法

Require: (x, y)
  $Link(Find - Set(x), Find - Set(y))$

其核心代码实现如下:

2.4.6 算法时间及空间复杂性分析

2.4.6.1 空间复杂度分析 使用邻接表存图, 空间复杂度为 $O(V + E)$. 借用二叉最小堆的初始化过程进行排序, 空间复杂度为 $O(E)$. 使用 Disjoint-set union 来维护联通分支, 空间复杂度为 $O(E)$. 综上, 空间复杂度为 $O(E)$.

Algorithm 7 Disjoint-set union 的 Link 算法

Require: (x, y)
  if $x.rank > y.rank$ **then**
    y.p = x
  else$x.p = y$
    if $x.rank == y.rank$ **then** y.rank = y.rank+1
    end if
  end if

Algorithm 8 Disjoint-set union 的 Find-Set 算法

Require: x
  if $x \neq x.p$ **then**
    x.p = Find-Set(x.p)
  end ifreturn x.p

2.4.6.2 时间复杂度分析 由于使用了二叉最小堆对边进行排序, 堆的初始化开销为 $O(|E|lg|E|)$. 在 for 循环中, 在 disjoint-set 森林中执行了 $O(E)$ 次 Find-Set 和 Union 操作, 还有 $|V|$ 次 Make-Set 操作, 共计花费了 $O((V+E)\alpha(V))$ 时间, 其中 α 是 disjoint-set union 数据结构中一个增长非常缓慢的函数. 由于假设 G 是联通的, 有 $|E| \geq |V|-1$, 因此 disjoint-set 操作花费 $O(E\alpha(V))$ 时间. 此外, 由于 $\alpha(|V|) = O(\lg V) = O(\lg E)$, Kruskal 算法的总时间复杂度为 $O(E \lg E)$. 注意到 $|E| \leq |V|^2$, 有 $\lg|E| = O(\lg V)$, 因此可以将 Kruskal 算法的时间复杂度表述为 $O(E \lg V)$.

3 实验结果

3.1 huffman

Huffman 编码算法分别使用 C++ 与 Rust 进行编写, 在具体的实现方式上, C++ 方法使用了二叉最小堆, 而 Rust 版在每次插入操作结束后都对数组重新进行排序. 以通过对比输出结果来验证算法正确性.

3.2 程序执行环境

在以下环境下进行过测试.

操作系统 macOS 13.1 22C5044e arm64;

C++ 编译器 g++ (Homebrew GCC 12.2.0) 12.2.0;

Rust 编译器 rustc 1.65.0 (897e37553 2022-11-02)

构建和测试工具 GNU Make 3.81;

文档编译工具 XeTeX 3.141592653-2.6-0.999994 (TeX Live 2022)

文本编辑器 nvim v0.8.1

3.3 程序运行方式

3.3.1 Huffman 编码程序的编译, 运行和测试方法

C++ 版本和 Rust 版本均使用 make 进行统一构建和测试, 并使用 diff 命令将结果进行比较, 以确保排序结果正确.

程序 (及文档) 均使用 make 进行构建, 对于 Huffman 编码程序的构建, 在 Huffman/文件夹下的 shell 中执行以下命令:

```
make
```

二进制文件 Huffman 和 HuffmanRS 将被编译在../build/目录下, 其中, Huffman 为 C++ 版的编译输出, 而 HuffmanRS 为 Rust 版的编译输出. 如果要运行样例测试, 可以在 shell 中执行这条命令:

```
make huffman_default_test
```

程序输出将存储在 data/huffman/huffman[_rs].out 中.

如果要自定义生成数据进行测试, 如下

```
make test
```

默认情况下会使用数据生成器 data_gen 生成范围在 0 到 1000 范围内的 5000 个数字进行测试.

程序同样支持 **debug** 模式编译. 在 debug 模式下, 会输出更多的运行信息. 指定 DEBUG 变量如下:

```
make test DEBUG=1
```

其中, huffman_data_gen 生成的输入存储在 data/huffman_yasample.in 中, 程序的输出存储在 data/huffman_yasample.out 和 data/huffman_rs_yasample.out 中.

如果要使用数据生成器 data_gen 生成指定范围内的均匀分布的随机数进行测试, 可以用这条命令:

```
make test [from=<range start>] [to=<range end>] [amount=<number of random
    numbers>]
# e.g.
# make test from=0 to=12345 amount=50
```

由于分别使用 c++ 和 rust 实现了命令行参数处理工具, 因此如果要使用自定义文件进行测试, 可以用这条命令:

```
# c++版
../build/Huffman --in ${infilename} --out [${outfilename}]
# Rust版
../build/HuffmanRS --in ${infilename} --out [${outfilename}]
```

其中, ${infilename} 是按第一章中的输入格式组织的输入数据. 默认情况下将同时输出到标准输出和 data/huffman_yasample.out, 可选自定义输出到 ${outfilename} 路径下. 输出格式为第一章中的输出格式.

3.3.2 文档编译

除排序程序使用 make 管理之外, 本实验的课程报告使用 latex 进行编写, 因此同样可以使用 make 进行管理. 如果要编译文档, 在 Greedy/目录下, 运行这条命令:

```
# cwd = Greddy/
make docs
```

make 将调用 xelatex 进行编译; 为了确保 cross-referencing 的正确工作, make 将执行两遍 xelatex 编译.

编译生成的中间文件及文档都保存在 docs/build 下, docs/report.pdf 和 docs/build/ report.pdf 均是生成的实验报告.

3.4 程序执行示例

此处演示程序从构建到测试的过程.

```
# in Huffman/ directory
# user input
make clean

# console output
rm -rf ../build/Huffman*

# user input
make # Compile Rust and C++ programs

# console output
g++ -c -Wall -std=c++17 -O3 -DNDEBUG src/Huffman.cpp -o ../build/Huffman.o
g++ -Wall -std=c++17 -O3 -DNDEBUG ../build/CLIParser.o ../build/Huffman.o -o
    ../build/Huffman
rustc -O --edition 2021 --cfg 'feature=""' -o ../build/HuffmanRS src/main.rs

# user input
# Run default test with sample given in *DEBUG* mode
make clean huffman_default_test DEBUG=1
# or use `make clean huffman_default_test` to run in *RELEASE* mode

# data/huffman/huffman.in:
4
0.100 0.100 0.200 0.600

# console output
rm -rf ../build/Huffman*
g++ -c -Wall -Werror -Wextra -std=c++17 -O0 -DDEBUG src/Huffman.cpp -o
    ../build/Huffman.o
g++ -Wall -Werror -Wextra -std=c++17 -O0 -DDEBUG ../build/CLIParser.o
    ../build/Huffman.o -o ../build/Huffman
rustc -g --edition 2021 --cfg 'feature="debug"' -o ../build/HuffmanRS
    src/main.rs
../build/Huffman --in data/huffman/huffman.in --out data/huffman/huffman.out
data/huffman/huffman.in
data/huffman/huffman.out
[Huffman] Time measured: 5e-06 seconds.
Expectation:
```

```
1.600
 (2, 0.2): 00
 (0, 0.1): 010
 (1, 0.1): 011
 (3, 0.6): 1
--------------------END OF C++ OUTPUT----------------------
------------------------------------------------------------
-------------------BEGIN OF RUST OUTPUT--------------------
../build/HuffmanRS --in data/huffman/huffman.in --out
    data/huffman/huffman_rs.out
Load from file: data/huffman/huffman.in
(1, 0.1): 000
(0, 0.1): 001
(2, 0.2): 01
(3, 0.6): 1

[Huffman RUST] Time measured: 0.00018425 seconds.
Expectation:
1.600
Write to file: data/huffman/huffman_rs.out
--------------------END OF RUST OUTPUT----------------------
diff data/huffman/huffman.out data/huffman/huffman_rs.out

# user input
# test with random number generator
make clean test # Compile data_gen and generate sample to test

# console output
rm -rf ../build/Huffman*
g++ -c -Wall -std=c++17 -O3 -DNDEBUG src/Huffman.cpp -o ../build/Huffman.o
g++ -Wall -std=c++17 -O3 -DNDEBUG ../build/CLIParser.o ../build/Huffman.o -o
    ../build/Huffman
rustc -O --edition 2021 --cfg 'feature=""' -o ../build/HuffmanRS src/main.rs
[INFO]: Generate 5000 random frequences sum to 1.
../build/huffman_data_gen 0 1000 5000
../build/Huffman --in data/huffman_yasample.in --out
    data/huffman_yasample.out
data/huffman_yasample.in
data/huffman_yasample.out
[Huffman] Time measured: 0.00162 seconds.
Expectation:
12.045
--------------------END OF C++ OUTPUT----------------------
------------------------------------------------------------
-------------------BEGIN OF RUST OUTPUT--------------------
../build/HuffmanRS --in data/huffman_yasample.in --out
    data/huffman_rs_yasample.out
Load from file: data/huffman_yasample.in
[Huffman RUST] Time measured: 0.036416875 seconds.
Expectation:
12.045
Write to file: data/huffman_rs_yasample.out
--------------------END OF RUST OUTPUT----------------------
diff data/huffman_yasample.out data/huffman_rs_yasample.out

# user input
make test from=0 to=1000 amount=500
# It means generate 500 random numbers ranging from 0 to 1000.
```

```
# console output
[INFO]: Generate 500 random frequences sum to 1.
../build/huffman_data_gen 0 1000 500
../build/Huffman --in data/huffman_yasample.in --out
    data/huffman_yasample.out
data/huffman_yasample.in
data/huffman_yasample.out
[Huffman] Time measured: 0.00019 seconds.
Expectation:
8.729
--------------------END OF C++ OUTPUT----------------------
------------------------------------------------------------
-------------------BEGIN OF RUST OUTPUT--------------------
../build/HuffmanRS --in data/huffman_yasample.in --out
    data/huffman_rs_yasample.out
Load from file: data/huffman_yasample.in
[Huffman RUST] Time measured: 0.000185541 seconds.
Expectation:
8.729
Write to file: data/huffman_rs_yasample.out
--------------------END OF RUST OUTPUT----------------------
diff data/huffman_yasample.out data/huffman_rs_yasample.out
```

3.5 Dijkstra

Dijkstra 算法分别使用 C++ 与 Rust 进行编写, 在邻接表具体的实现方式上, C++ 方法使用了链表, 而 Rust 版使用了二维可变长数组 Vec<>. 以通过对比输出结果来验证算法正确性.

3.6 程序执行环境

在以下环境下进行过测试.

操作系统 macOS 13.1 22C5044e arm64;

C++ 编译器 g++ (Homebrew GCC 12.2.0) 12.2.0;

Rust 编译器 rustc 1.65.0 (897e37553 2022-11-02)

构建和测试工具 GNU Make 3.81;

文档编译工具 XeTeX 3.141592653-2.6-0.999994 (TeX Live 2022)

文本编辑器 nvim v0.8.1

3.7 程序运行方式

3.7.1 Dijkstra 编码程序的编译, 运行和测试方法

C++ 版本和 Rust 版本均使用 make 进行统一构建和测试, 并使用 diff 命令将结果进行比较, 以确保排序结果正确.

程序 (及文档) 均使用 make 进行构建, 对于 Huffman 编码程序的构建, 在 Huffman/文件夹下的 shell 中执行以下命令:

```
make
```

二进制文件 Dijkstra 和 DijkstraRS 将被编译在../build/目录下, 其中, Dijkstra 为 C++ 版的编译输出, 而 DijkstraRS 为 Rust 版的编译输出. 如果要运行样例测试, 可以在 shell 中执行这条命令:

```
make default_test
```

程序输出将存储在 data/dijkstra[_rs].out 中.

程序同样支持 **debug** 模式编译. 在 debug 模式下, 会输出更多的运行信息. 例如, 在默认测试中, 指定 DEBUG 变量如下:

```
make default_test DEBUG=1
```

如果要自定义生成数据进行测试, 如下

```
make test
```

默认情况下会使用数据生成器 data_gen 生成范围在 0 到 1000 范围内的 5000 个数字进行测试.

自定义数据同样支持 **debug** 模式编译. 在 debug 模式下, 会输出更多的运行信息. 指定 DEBUG 变量如下:

```
make test DEBUG=1
```

其中, dijkstra_data_gen 生成的输入存储在 data/dijkstra_yasample.in 中, 程序的输出存储在 data/dijkstra_yasample.out 和 data/dijkstra_rs_yasample.out 中.

如果要使用数据生成器 dijkstra_data_gen 生成指定范围内的均匀分布的随机数进行测试, 可以用这条命令:

```
make test [from=<range start>] [to=<range end>] [amount=<number of random
    numbers>]
# e.g.
# make test from=0 to=12345 amount=50 DEBUG=[01]
```

由于分别使用 c++ 和 rust 实现了命令行参数处理工具, 因此如果要使用自定义文件进行测试, 可以用这条命令:

```
# c++版
../build/Dijkstra --in ${infilename} --out [${outfilename}]
# Rust版
../build/DijkstraRS --in ${infilename} --out [${outfilename}]
```

其中, ${infilename} 是按第一章中的输入格式组织的输入数据. 默认情况下将同时输出到标准输出和 data/dijkstra_yasample.out, 可选自定义输出到 ${outfilename} 路径下. 输出格式为第一章中的输出格式.

3.8 程序执行示例

此处演示程序从构建到测试的过程.

```
# cwd = Dijkstra/ directory
# user input
make clean

# console output
rm -rf ../build/Dijkstra*

# user input
make # Compile Rust and C++ programs

# console output
g++ -c -Wall -std=c++17 -O3 -DNDEBUG src//Dijkstra.cpp -o ../build/dijkstra.o
g++ -c -Wall -std=c++17 -O3 -DNDEBUG src/utils/CLIParser.cpp -o
    ../build/CLIParser.o
g++ -Wall -std=c++17 -O3 -DNDEBUG ../build/CLIParser.o ../build/Dijkstra.o
    -o ../build/Dijkstra
rustc -O --edition 2021 --cfg 'feature=""' -o ../build/DijkstraRS
    src//main.rs

# user input
# Run default test with sample given in *DEBUG* mode
make clean default_test DEBUG=1
# or use `make clean huffman_default_test` to run in *RELEASE* mode

# data/dijkstra.in:
3 3
1 2 5
2 3 5
3 1 2

# console output
rm -rf ../build/Dijkstra*
rm -rf ../build/CLIParser.o
g++ -c -Wall -Werror -Wextra -std=c++17 -O0 -DDEBUG src/utils/CLIParser.cpp
    -o ../build/CLIParser.o
g++ -Wall -Werror -Wextra -std=c++17 -O0 -DDEBUG ../build/CLIParser.o
    ../build/Dijkstra.o -o ../build/Dijkstra
rustc -g --edition 2021 --cfg 'feature="debug"' -o ../build/DijkstraRS
    src//main.rs
../build/Dijkstra --in data/dijkstra.in --out data/dijkstra.out
data/dijkstra.in
data/dijkstra.out
[dijkstra] Time measured: 1e-06 seconds.
2
--------------------END OF C++ OUTPUT----------------------
------------------------------------------------------------
-------------------BEGIN OF RUST OUTPUT--------------------
../build/DijkstraRS --in data/dijkstra.in --out data/dijkstra_rs.out
Load from file: data/dijkstra.in
[Dijkstra Rust] Time measured: 0.000010375 seconds.
2
Write to file: data/dijkstra_rs.out
--------------------END OF RUST OUTPUT----------------------
diff data/dijkstra.out data/dijkstra_rs.out

# user input
# test with random number generator
make clean test # Compile data_gen and generate sample to test
```

```
# console output
rm -rf ../build/Dijkstra*
rm -rf ../build/CLIParser.o
g++ -Wall -std=c++17 -O3 -DNDEBUG src/data_gen/dijkstra_data_gen.cpp -o
    ../build/dijkstra_data_gen
g++ -c -Wall -std=c++17 -O3 -DNDEBUG src/utils/CLIParser.cpp -o
    ../build/CLIParser.o
g++ -Wall -std=c++17 -O3 -DNDEBUG ../build/CLIParser.o ../build/Dijkstra.o
    -o ../build/Dijkstra
rustc -O --edition 2021 --cfg 'feature=""' -o ../build/DijkstraRS
    src//main.rs
[INFO]: Generate 5000 nodes.
../build/dijkstra_data_gen 0 1000 5000
../build/Dijkstra --in data/dijkstra_yasample.in --out
    data/dijkstra_yasample.out
data/dijkstra_yasample.in
data/dijkstra_yasample.out
[dijkstra] Time measured: 0.000709 seconds.
59066
--------------------END OF C++ OUTPUT----------------------
------------------------------------------------------------
-------------------BEGIN OF RUST OUTPUT--------------------
../build/DijkstraRS --in data/dijkstra_yasample.in --out
    data/dijkstra_rs_yasample.out
Load from file: data/dijkstra_yasample.in
[Dijkstra Rust] Time measured: 0.000879208 seconds.
59066
Write to file: data/dijkstra_rs_yasample.out
--------------------END OF RUST OUTPUT----------------------
diff data/dijkstra_yasample.out data/dijkstra_rs_yasample.out

# user input
make test from=0 to=1000 amount=500
# It means generate 500 random numbers ranging from 0 to 1000.

# console output
rm -rf ../build/Dijkstra*
rm -rf ../build/CLIParser.o
g++ -c -Wall -std=c++17 -O3 -DNDEBUG src/utils/CLIParser.cpp -o
    ../build/CLIParser.o
g++ -Wall -std=c++17 -O3 -DNDEBUG ../build/CLIParser.o ../build/Dijkstra.o
    -o ../build/Dijkstra
rustc -O --edition 2021 --cfg 'feature=""' -o ../build/DijkstraRS
    src//main.rs
[INFO]: Generate 500 nodes.
../build/dijkstra_data_gen 0 1000 500
../build/Dijkstra --in data/dijkstra_yasample.in --out
    data/dijkstra_yasample.out
data/dijkstra_yasample.in
data/dijkstra_yasample.out
[dijkstra] Time measured: 4.1e-05 seconds.
4077
--------------------END OF C++ OUTPUT----------------------
------------------------------------------------------------
-------------------BEGIN OF RUST OUTPUT--------------------
../build/DijkstraRS --in data/dijkstra_yasample.in --out
    data/dijkstra_rs_yasample.out
```

```
Load from file: data/dijkstra_yasample.in
[Dijkstra Rust] Time measured: 0.000005208 seconds.
4077
Write to file: data/dijkstra_rs_yasample.out
--------------------END OF RUST OUTPUT----------------------
diff data/dijkstra_yasample.out data/dijkstra_rs_yasample.out
```

3.9 Prim

Prim 算法分别使用 C++ 与 Rust 进行编写, 在邻接表具体的实现方式上, C++ 方法使用了链表, 而 Rust 版使用了二维可变长数组 Vec<>. 以通过对比输出结果来验证算法正确性.

3.10 程序执行环境

在以下环境下进行过测试.

操作系统 macOS 13.1 22C5044e arm64;

C++ 编译器 g++ (Homebrew GCC 12.2.0) 12.2.0;

Rust 编译器 rustc 1.65.0 (897e37553 2022-11-02)

构建和测试工具 GNU Make 3.81;

文档编译工具 XeTeX 3.141592653-2.6-0.999994 (TeX Live 2022)

文本编辑器 nvim v0.8.1

3.11 程序运行方式

3.11.1 Prim 编码程序的编译, 运行和测试方法

C++ 版本和 Rust 版本均使用 make 进行统一构建和测试, 并使用 diff 命令将结果进行比较, 以确保排序结果正确.

程序 (及文档) 均使用 make 进行构建, 对于 Huffman 编码程序的构建, 在 Huffman/文件夹下的 shell 中执行以下命令:

```
make
```

二进制文件 prim 和 primRS 将被编译在../build/目录下, 其中, prim 为 C++ 版的编译输出, 而 primRS 为 Rust 版的编译输出. 如果要运行样例测试, 可以在 shell 中执行这条命令:

```
make default_test
```

程序输出将存储在 data/prim[_rs].out 中.

程序同样支持 **debug** 模式编译. 在 debug 模式下, 会输出更多的运行信息. 例如, 在默认测试中, 指定 DEBUG 变量如下:

```
make default_test DEBUG=1
```

如果要自定义生成数据进行测试, 如下

```
make test
```

默认情况下会使用数据生成器 data_gen 生成范围在 0 到 1000 范围内的 5000 个数字进行测试.

自定义数据同样支持 **debug** 模式编译. 在 debug 模式下, 会输出更多的运行信息. 指定 DEBUG 变量如下:

```
make test DEBUG=1
```

其中, prim_data_gen 生成的输入存储在 data/prim_yasample.in 中, 程序的输出存储在 data/prim_yasample.out 和 data/prim_rs_yasample.out 中.

如果要使用数据生成器 prim_data_gen 生成指定范围内的均匀分布的随机数进行测试, 可以用这条命令:

```
make test [from=<range start>] [to=<range end>] [amount=<number of random
    numbers>]
# e.g.
# make test from=0 to=12345 amount=50 DEBUG=[01]
```

由于分别使用 c++ 和 rust 实现了命令行参数处理工具, 因此如果要使用自定义文件进行测试, 可以用这条命令:

```
# c++版
../build/Prim --in ${infilename} --out [${outfilename}]
# Rust版
../build/PrimRS --in ${infilename} --out [${outfilename}]
```

其中, ${infilename} 是按第一章中的输入格式组织的输入数据. 默认情况下将同时输出到标准输出和 data/prim_yasample.out, 可选自定义输出到 ${outfilename} 路径下. 输出格式为第一章中的输出格式.

3.12 程序执行示例

此处演示程序从构建到测试的过程.

```
# cwd = Prim/ directory
# user input
make clean

# console output
rm -rf ../build/prim*
rm -rf ../build/CLIParser.o

# user input
make # Compile Rust and C++ programs

# console output
g++ -c -Wall -std=c++17 -O3 -DNDEBUG src//prim.cpp -o ../build/prim.o
g++ -c -Wall -std=c++17 -O3 -DNDEBUG src/utils/CLIParser.cpp -o
    ../build/CLIParser.o
g++ -Wall -std=c++17 -O3 -DNDEBUG ../build/CLIParser.o ../build/prim.o -o
    ../build/prim
```

```
rustc -O --edition 2021 --cfg 'feature=""' -o ../build/primRS src//main.rs

# user input
# Run default test with sample given in *DEBUG* mode
make clean default_test DEBUG=1
# or use `make clean huffman_default_test` to run in *RELEASE* mode

# data/prim.in:
4 5
1 2 2
1 3 2
1 4 3
2 3 4
3 4 3

# console output
rm -rf ../build/prim*
rm -rf ../build/CLIParser.o
g++ -c -Wall -Werror -Wextra -std=c++17 -O0 -DDEBUG src//prim.cpp -o
    ../build/prim.o
g++ -c -Wall -Werror -Wextra -std=c++17 -O0 -DDEBUG src/utils/CLIParser.cpp
    -o ../build/CLIParser.o
g++ -Wall -Werror -Wextra -std=c++17 -O0 -DDEBUG ../build/CLIParser.o
    ../build/prim.o -o ../build/prim
rustc -g --edition 2021 --cfg 'feature="debug"' -o ../build/primRS
    src//main.rs
../build/prim --in data/prim.in --out data/prim.out
Load data from: data/prim.in
Write data to: data/prim.out
Current edge: from[1], to[2], weight[2]
Neighbor vertex: 2: distance[2], parent[-1], inMST[0]
Current edge: from[1], to[3], weight[2]
Neighbor vertex: 3: distance[2], parent[-1], inMST[0]
Current edge: from[1], to[4], weight[3]
Neighbor vertex: 4: distance[3], parent[-1], inMST[0]
Current edge: from[2], to[3], weight[4]
Neighbor vertex: 3: distance[2], parent[-1], inMST[0]
Current edge: from[3], to[4], weight[3]
Neighbor vertex: 4: distance[3], parent[-1], inMST[0]
[Prim] Time measured: 2.1e-05 seconds.
3 edges in MST.
from[2], to[1], weight[2]
from[3], to[1], weight[2]
from[4], to[1], weight[3]
7
../build/primRS --in data/prim.in --out data/prim_rs.out
Load from file: data/prim.in
1: edges:
to[2], weight[2]
to[3], weight[2]
to[4], weight[3]
2: edges:
to[1], weight[2]
to[3], weight[4]
3: edges:
to[1], weight[2]
to[2], weight[4]
to[4], weight[3]
```

```
4: edges:
to[1], weight[3]
to[3], weight[3]
[Prim RUST] Time measured: 5.3333e-5 seconds.
From[2], To[1], Weight[2]
From[3], To[1], Weight[2]
From[4], To[1], Weight[3]
7
Write to file: data/prim_rs.out
diff data/prim.out data/prim_rs.out

# user input
# test with random number generator
make clean test # Compile data_gen and generate sample to test

# console output
rm -rf ../build/prim*
rm -rf ../build/CLIParser.o
g++ -Wall -std=c++17 -O3 -DNDEBUG src/data_gen/prim_data_gen.cpp -o
    ../build/prim_data_gen
g++ -c -Wall -std=c++17 -O3 -DNDEBUG src//prim.cpp -o ../build/prim.o
g++ -c -Wall -std=c++17 -O3 -DNDEBUG src/utils/CLIParser.cpp -o
    ../build/CLIParser.o
g++ -Wall -std=c++17 -O3 -DNDEBUG ../build/CLIParser.o ../build/prim.o -o
    ../build/prim
rustc -O --edition 2021 --cfg 'feature=""' -o ../build/primRS src//main.rs
[INFO]: Generate 5000 nodes.
../build/prim_data_gen 0 1000 5000
../build/prim --in data/prim_yasample.in --out data/prim_yasample.out
Load data from: data/prim_yasample.in
Write data to: data/prim_yasample.out
[Prim] Time measured: 0.000427 seconds.
2067227
../build/primRS --in data/prim_yasample.in --out data/prim_rs_yasample.out
Load from file: data/prim_yasample.in
[Prim RUST] Time measured: 0.0025155 seconds.
2067227
Write to file: data/prim_rs_yasample.out
diff data/prim_yasample.out data/prim_rs_yasample.out

# user input
make test from=0 to=1000 amount=500
# It means generate 500 random numbers ranging from 0 to 1000.

# console output
[INFO]: Generate 500 nodes.
../build/prim_data_gen 0 1000 500
../build/prim --in data/prim_yasample.in --out data/prim_yasample.out
Load data from: data/prim_yasample.in
Write data to: data/prim_yasample.out
[Prim] Time measured: 2.9e-05 seconds.
218509
../build/primRS --in data/prim_yasample.in --out data/prim_rs_yasample.out
Load from file: data/prim_yasample.in
[Prim RUST] Time measured: 9.3833e-5 seconds.
218509
Write to file: data/prim_rs_yasample.out
diff data/prim_yasample.out data/prim_rs_yasample.out
```

3.13 Kruskal

Kruskal 算法分别使用 C++ 与 Rust 进行编写, 连通分支的模拟上, C++ 版使用了 unordered_set 进行操作, 而 Rust 版本使用 **disjoint-set union** 进行优化. 以通过对比输出结果来验证算法正确性, 以及比较 disjoint-set union 带来的提升.

3.14 程序执行环境

在以下环境下进行过测试.

操作系统 macOS 13.1 22C5044e arm64;

C++ 编译器 g++ (Homebrew GCC 12.2.0) 12.2.0;

Rust 编译器 rustc 1.65.0 (897e37553 2022-11-02)

构建和测试工具 GNU Make 3.81;

文档编译工具 XeTeX 3.141592653-2.6-0.999994 (TeX Live 2022)

文本编辑器 nvim v0.8.1

3.15 程序运行方式

3.15.1 Kruskal 编码程序的编译, 运行和测试方法

C++ 版本和 Rust 版本均使用 make 进行统一构建和测试, 并使用 diff 命令将结果进行比较, 以确保排序结果正确.

程序 (及文档) 均使用 make 进行构建, 对于 Huffman 编码程序的构建, 在 Huffman/文件夹下的 shell 中执行以下命令:

```
make
```

二进制文件 kruskal 和 kruskalRS 将被编译在../build/目录下, 其中, kruskal 为 C++ 版的编译输出, 而 kruskalRS 为 Rust 版的编译输出. 如果要运行样例测试, 可以在 shell 中执行这条命令:

```
make default_test
```

程序输出将存储在 data/kruskal[_rs].out 中.

程序同样支持 **debug** 模式编译. 在 debug 模式下, 会输出更多的运行信息. 例如, 在默认测试中, 指定 DEBUG 变量如下:

```
make default_test DEBUG=1
```

如果要自定义生成数据进行测试, 如下

```
make test
```

默认情况下会使用数据生成器 data_gen 生成范围在 0 到 1000 范围内的 5000 个数字进行测试.

自定义数据同样支持 **debug** 模式编译. 在 debug 模式下, 会输出更多的运行信息. 指定 DEBUG 变量如下:

```
make test DEBUG=1
```

其中, kruskal_data_gen 生成的输入存储在 data/kruskal_yasample.in 中, 程序的输出存储在 data/kruskal_yasample.out 和 data/kruskal_rs_yasample.out 中.

如果要使用数据生成器 kruskal_data_gen 生成指定范围内的均匀分布的随机数进行测试, 可以用这条命令:

```
make test [from=<range start>] [to=<range end>] [amount=<number of random
    numbers>]
# e.g.
# make test from=0 to=12345 amount=50 DEBUG=[01]
```

由于分别使用 c++ 和 rust 实现了命令行参数处理工具, 因此如果要使用自定义文件进行测试, 可以用这条命令:

```
# c++版
../build/Kruskal --in ${infilename} --out [${outfilename}]
# Rust版
../build/KruskalRS --in ${infilename} --out [${outfilename}]
```

其中, ${infilename} 是按第一章中的输入格式组织的输入数据. 默认情况下将同时输出到标准输出和 data/kruskal_yasample.out, 可选自定义输出到 ${outfilename} 路径下. 输出格式为第一章中的输出格式.

3.16 程序执行示例

此处演示程序从构建到测试的过程.

```
# cwd = Kruskal/ directory
# user input
make clean

# console output
rm -rf ../build/kruskal*
rm -rf ../build/CLIParser.o

# user input
make # Compile Rust and C++ programs

# console output
g++ -c -Wall -std=c++17 -O3 -DNDEBUG src//kruskal.cpp -o ../build/kruskal.o
g++ -c -Wall -std=c++17 -O3 -DNDEBUG src/utils/CLIParser.cpp -o
    ../build/CLIParser.o
g++ -Wall -std=c++17 -O3 -DNDEBUG ../build/CLIParser.o ../build/kruskal.o -o
    ../build/kruskal
rustc -O --edition 2021 --cfg 'feature=""' -o ../build/kruskalRS src//main.rs

# user input
# Run default test with sample given in *DEBUG* mode
make clean default_test DEBUG=1
```

```
# or use `make clean huffman_default_test` to run in *RELEASE* mode

# data/kruskal.in:
4 5
1 2 2
1 3 2
1 4 3
2 3 4
3 4 3

# console output
rm -rf ../build/kruskal*
rm -rf ../build/CLIParser.o
g++ -c -Wall -Werror -Wextra -std=c++17 -O0 -DDEBUG src//kruskal.cpp -o
    ../build/kruskal.o
g++ -c -Wall -Werror -Wextra -std=c++17 -O0 -DDEBUG src/utils/CLIParser.cpp
    -o ../build/CLIParser.o
g++ -Wall -Werror -Wextra -std=c++17 -O0 -DDEBUG ../build/CLIParser.o
    ../build/kruskal.o -o ../build/kruskal
rustc -g --edition 2021 --cfg 'feature="debug"' -o ../build/kruskalRS
    src//main.rs
../build/kruskal --in data/kruskal.in --out data/kruskal.out
Load data from: data/kruskal.in
Write data to: data/kruskal.out

Edges:
1:
to[2], weight[2]
to[3], weight[2]
to[4], weight[3]
2:
to[1], weight[2]
to[3], weight[4]
3:
to[1], weight[2]
to[2], weight[4]
to[4], weight[3]
4:
to[1], weight[3]
to[3], weight[3]
(1, 2): 2
(1, 3): 2
(1, 4): 3
(2, 3): 4
(3, 4): 3

edges[5], heap[5]
[Kruskal] Time measured: 2.9e-05 seconds.
4 Vertices in MST:
4 3 2 1
3 Edges in MST:
(1, 4): 3
(1, 3): 2
(1, 2): 2
Weight sum of MST:
7
--------------------END OF C++ OUTPUT----------------------
------------------------------------------------------------
```

```
------------------BEGIN OF RUST OUTPUT---------------------
../build/kruskalRS --in data/kruskal.in --out data/kruskal_rs.out
Load edges from file: data/kruskal.in
[Kruskal RUST] Time measured: 1.4333e-5 seconds.
3 edges in Kruskal MST:
(1, 2): 2
(1, 3): 2
(1, 4): 3

Weight sum of MST:
7
Write to file: data/kruskal_rs.out
-------------------END OF RUST OUTPUT----------------------
diff data/kruskal.out data/kruskal_rs.out

# user input
# test with random number generator
make clean test # Compile data_gen and generate sample to test

# console output
rm -rf ../build/kruskal*
rm -rf ../build/CLIParser.o
g++ -Wall -std=c++17 -O3 -DNDEBUG src/data_gen/kruskal_data_gen.cpp -o
    ../build/kruskal_data_gen
g++ -c -Wall -std=c++17 -O3 -DNDEBUG src//kruskal.cpp -o ../build/kruskal.o
g++ -c -Wall -std=c++17 -O3 -DNDEBUG src/utils/CLIParser.cpp -o
    ../build/CLIParser.o
g++ -Wall -std=c++17 -O3 -DNDEBUG ../build/CLIParser.o ../build/kruskal.o -o
    ../build/kruskal
rustc -O --edition 2021 --cfg 'feature=""' -o ../build/kruskalRS src//main.rs
[INFO]: Generate 5000 nodes.
../build/kruskal_data_gen 0 1000 5000
../build/kruskal --in data/kruskal_yasample.in --out
    data/kruskal_yasample.out
Load data from: data/kruskal_yasample.in
Write data to: data/kruskal_yasample.out
[Kruskal] Time measured: 0.027958 seconds.
Weight sum of MST:
2510558
../build/kruskalRS --in data/kruskal_yasample.in --out
    data/kruskal_rs_yasample.out
Load edges from file: data/kruskal_yasample.in
[Kruskal RUST] Time measured: 0.001215166 seconds.
Weight sum of MST:
2510558
Write to file: data/kruskal_rs_yasample.out
diff data/kruskal_yasample.out data/kruskal_rs_yasample.out

# user input
make test from=0 to=1000 amount=500
# It means generate 500 random numbers ranging from 0 to 1000.

# console output
[INFO]: Generate 500 nodes.
../build/kruskal_data_gen 0 1000 500
../build/kruskal --in data/kruskal_yasample.in --out
    data/kruskal_yasample.out
Load data from: data/kruskal_yasample.in
```

```
Write data to: data/kruskal_yasample.out
[Kruskal] Time measured: 0.000755 seconds.
Weight sum of MST:
251140
../build/kruskalRS --in data/kruskal_yasample.in --out
    data/kruskal_rs_yasample.out
Load edges from file: data/kruskal_yasample.in
[Kruskal RUST] Time measured: 5.9375e-5 seconds.
Weight sum of MST:
251140
Write to file: data/kruskal_rs_yasample.out
diff data/kruskal_yasample.out data/kruskal_rs_yasample.out
```

4 实验总结

本次作业共用时约 50 小时. 其中 20 小时 45 分钟在 C++ 程序代码编写和测试上, 17 小时 39 分钟在 Rust 程序代码编写和测试上, 3 小时 21 分钟在 makefile 的编写和修改上, 7 小时 18 分钟在 tex 实验报告的编写上.

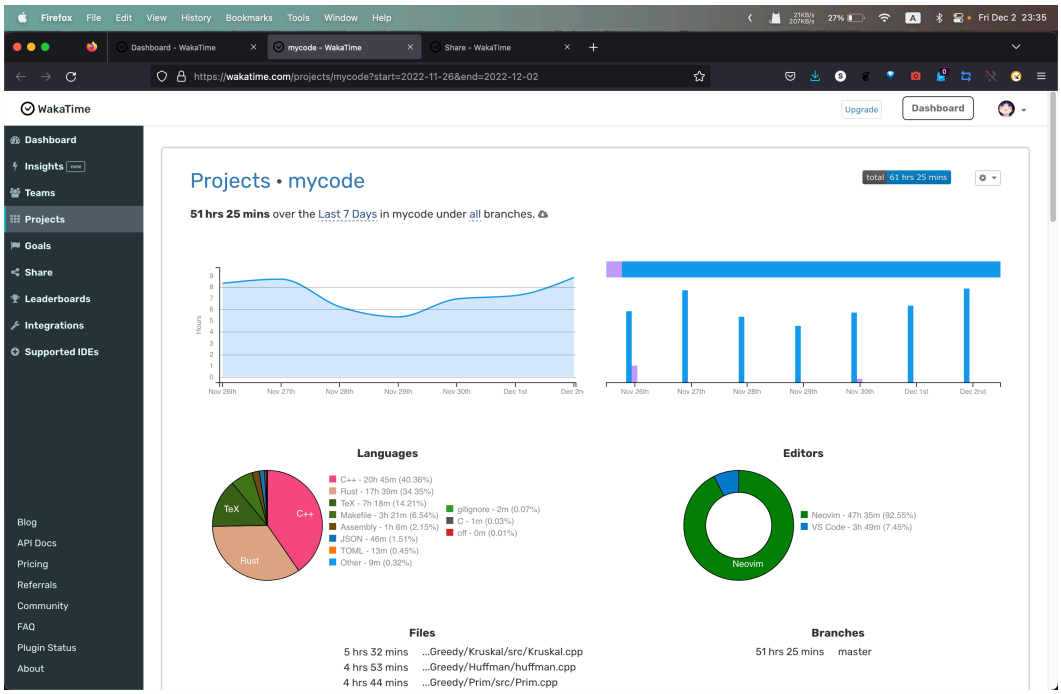


Figure 1: 编写时间统计

统计链接: 编码时间统计

通过这次实验, 我动手实现了 Huffman, Dijkstra, Prim, Kruskal 算法及其用到的 MinHeap 和 Disjoint-set union 算法. 动手实践了各算法的时间复杂度和空间复杂度分析. 此外, 为了综合评价各算法的正确性, 以及衡量各算法的性能, 还动手实现了数据生成器, 用来方便的生成指定数量的均匀分布在给定范围的随机排序的数据作为排序程序的输入.

借助这次实验, 我还对 Rust 语言进行了学习, 并与 C++ 进行了比较. 因为 Rust

语言还比较生疏, 所以 Rust 编写程序花费的时间要比 C++ 要长一些. 但是 Debug 的时间却短得多, 说明 Rust 严格的编译检查除了能确保内存严格安全之外, 实际上反而能帮助程序员节省时间在内存相关的 debug 上.

在构建和管理项目的工具方面, 使用了 make 工具, 方便的实现了对程序的构建和测试; 此外, 对于文档, 也使用了 make 工具作为管理, 很大程度上方便了实验报告修改后的再次编译. 在实验报告编写方面, 练习了 latex 的使用, 对 latex 语法和使用更加熟悉.

但是, 由于时间有限, 没有实现更多的优化内容, 比如 Fibonacci Heap 代替 BinaryHeap 等.

总的来说, 这次实验在动态规划算法之内及之外都学到了许多知识, 收获颇丰.

5 算法源代码

5.1 Huffman 算法

5.1.1 c++ 版本 Huffman 算法

```cpp
#include "utils/CLIParser.hpp"
#include <cassert>
#include <chrono>
#include <climits>
#include <cstdlib>
#include <fstream>
#include <iomanip>
#include <ios>
#include <iostream>
#include <queue>
#include <sstream>
#include <string>
#include <vector>

const int MAXN = 1 << 20;
using namespace std;

template <typename T> void swap(T *a, T *b) {
    T tmp = *a;
    *a = *b;
    *b = tmp;
}

string arrayToString(int *arr, int len) {
    stringstream ss;
    for (int i = 0; i < len; i++) {
        ss << arr[i];
    }
    ss << std::endl;
    return ss.str();
}

class Node {
```

```
  public:
    int identifier;
    double freq;
    Node *left, *right;
    Node(int identifier = -10, double freq = 0.0, Node *left = nullptr,
         Node *right = nullptr)
        : identifier(identifier), freq(freq), left(left), right(right) {}
};

ostream &operator<<(ostream &os, Node node) {
    os << setw(2) << "(" << node.identifier << ", " << setw(2) << node.freq
       << ")";
    return os;
}

class MinHeap {
  private:
    Node **harr;
    int capacity;
    int heapSize;

    double comparator(Node *a, Node *b) { return (a->freq - b->freq); }
    int parent(int idx) { return (idx - 1) / 2; }
    int left(int idx) { return (idx * 2) + 1; }
    int right(int idx) { return (idx * 2) + 2; }

  public:
    // Constructor
    MinHeap(int capacity);

    MinHeap(int identifiers[], double freq[], int size);
    MinHeap(double freq[], int size);

    MinHeap(MinHeap &minHeap)
        : harr(minHeap.harr), capacity(minHeap.capacity),
          heapSize(minHeap.heapSize) {}

    Node *newNode(int identifier, double freq);

    void minHeapify(int rootIdx);

    Node *extractMin();

    void insertNode(Node *minHeapNode);

    void buildMinHeap();

    bool isLeaf(Node *node) { return !(node->left) && !(node->right); }

    int getHeapSize() { return heapSize; }

    string toString();
};

MinHeap::MinHeap(int capacity) {
    heapSize = 0;
    this->capacity = capacity;
    this->harr = new Node *[capacity];
```

```
}

MinHeap::MinHeap(double freq[], int size) {
    int *identifiers = new int[size];
    for (int i = 0; i < size; i++) {
        identifiers[i] = i;
    }
    heapSize = 0;
    capacity = size;
    harr = new Node *[capacity];

    for (int i = 0; i < capacity; i++) {
        harr[i] = newNode(identifiers[i], freq[i]);
        heapSize++;
    }
    buildMinHeap();
}

MinHeap::MinHeap(int identifiers[], double freq[], int size) {
    heapSize = 0;
    capacity = size;
    harr = new Node *[capacity];

    for (int i = 0; i < capacity; i++) {
        harr[i] = newNode(identifiers[i], freq[i]);
        heapSize++;
    }

    buildMinHeap();
}

Node *MinHeap::newNode(int identifier, double freq) {
    Node *tmp = new Node;
    tmp->left = tmp->right = NULL;
    tmp->identifier = identifier;
    tmp->freq = freq;

    return tmp;
};

void MinHeap::minHeapify(int rootIdx) {
    int smallest = rootIdx;
    int left = rootIdx * 2 + 1;
    int right = rootIdx * 2 + 2;
    if (left < heapSize && (comparator(harr[left], harr[smallest]) < 0)) {
        smallest = left;
    }
    if (right < heapSize && (comparator(harr[right], harr[smallest]) < 0)) {
        smallest = right;
    }
    if (smallest != rootIdx) {
        swap(harr + smallest, harr + rootIdx);
        minHeapify(smallest);
    }
}

void MinHeap::insertNode(Node *newNode) {
    int i = heapSize;
```

```
    harr[heapSize++] = newNode;
    while (i != 0 && (comparator(newNode, harr[parent(i)]) < 0)) {
        swap(harr[i], harr[parent(i)]);
        i = parent(i);
    }
}

Node *MinHeap::extractMin() {
    Node *minNode = harr[0];
    harr[0] = harr[--heapSize];
    minHeapify(0);
    return minNode;
}

// Build min heap by minHeapifying from bottom to top.
void MinHeap::buildMinHeap() {
    int n = heapSize - 1;
    for (int i = parent(n); i >= 0; i--) {
        minHeapify(i);
    }
}

string MinHeap::toString() {
    stringstream ss;
    for (int i = 0; i < heapSize; i++) {
        ss << *harr[i] << " ";
    }
    ss << std::endl;
    return ss.str();
}

class CmpNodePtrs {
  public:
    bool operator()(const Node *lhs, const Node *rhs) const {
        return lhs->freq > rhs->freq;
    }
};

class Huffman {
  private:
    MinHeap minHeap;
    Node *huffmanTree;

  public:
    Huffman(MinHeap minHeap);
    Huffman(int identifiers[], double freq[], int size);
    Huffman(double freq[], int size);

    // Separate build huffman tree alone to test how much time it costs.
    void buildHuffmanTree();
    void priority_queue_huild_huffman_tree();

    double getExpectation(double &exp, int top);
    double getExpectation(Node *root, double &exp, int top);

    string codesToString(Node *root, int arr[], int top);
    string codesToString(int arr[], int top);
};
```

```
Huffman::Huffman(MinHeap minHeap) : minHeap(minHeap) {}

Huffman::Huffman(int identifiers[], double freq[], int size)
    : minHeap(identifiers, freq, size) {}
Huffman::Huffman(double freq[], int size) : minHeap(freq, size) {}

void Huffman::buildHuffmanTree() {
    Node *left, *right, *top;

    while (!(minHeap.getHeapSize() == 1)) {
        // Extract two minimum freq items from min heap.
        left = minHeap.extractMin();
        right = minHeap.extractMin();
        // cout << *left << ", " << *right << endl;

        // Create a new internal node with frequency equal to the sum of the
            two
        // nodes' frequencies. Make the two extracted nodes as left and right
        // child of this new node.
        // Add the new internal node to the min heap. -10 is its identifier
        // which is never used.
        top = minHeap.newNode(-10, left->freq + right->freq);
        top->left = left;
        top->right = right;
        minHeap.insertNode(top);
    }

    // The remaining node is the root node. Now the huffman tree is complete.
    huffmanTree = minHeap.extractMin();
}

void Huffman::priority_queue_huild_huffman_tree() {
    priority_queue<Node *, vector<Node *>, CmpNodePtrs> pqHeap;
    while (minHeap.getHeapSize()) {
        pqHeap.push(minHeap.extractMin());
    }

    while (pqHeap.size() > 1) {
        auto left = pqHeap.top();
        pqHeap.pop();
        auto right = pqHeap.top();
        pqHeap.pop();

        auto new_node = new Node(-10, left->freq + right->freq, left, right);
        pqHeap.push(new_node);
    }

    huffmanTree = pqHeap.top();
}

string Huffman::codesToString(Node *root, int *arr, int top) {
    stringstream ss;

    if (root->left) {
        arr[top] = 0;
        ss << codesToString(root->left, arr, top + 1);
    }
```

```
    if (root->right) {
        arr[top] = 1;
        ss << codesToString(root->right, arr, top + 1);
    }

    if (minHeap.isLeaf(root)) {
        ss << *root << ": ";
        ss << arrayToString(arr, top);
    }

    return ss.str();
}

string Huffman::codesToString(int arr[], int top) {
    return codesToString(huffmanTree, arr, top);
}

double Huffman::getExpectation(double &exp, int top) {
    return getExpectation(huffmanTree, exp, top);
}

// Get expectation of Huffman code length by adding each leaf's code length *
// appearance frequnce recursively in DFS way.
double Huffman::getExpectation(Node *root, double &exp, int top) {
    if (root->left) {
        exp += getExpectation(root->left, exp, top + 1);
    }
    if (root->right) {
        exp += getExpectation(root->right, exp, top + 1);
    }
    if (minHeap.isLeaf(root)) {
        return top * root->freq;
    }
    return 0.0;
}

int fstream_main(int argc, char **argv) {
    CLIParser cliParser(argc, argv);
    cliParser.args["--in"] = "../data/huffman/huffman.in";
    cliParser.args["--out"] = "../data/huffman/huffman.out";
    cliParser.argParse();

    fstream fs;
    try {
        fs.open(cliParser.args["--in"], ios_base::in);
    } catch (std::system_error &e) {
        cerr << e.what();
        exit(-1);
    }

    cout << cliParser.args["--in"] << endl;
    cout << cliParser.args["--out"] << endl;

    int size;
    fs >> size;

    double freq[size];
    for (int i = 0; i < size; i++) {
```

```
        fs >> freq[i];
    }

    Huffman huffman(freq, size);
    fs.close();

    double exp = 0;
    auto begin = std::chrono::high_resolution_clock::now();

    // BUG: Bug in self-implemented Min Heap.
    // huffman.buildHuffmanTree();
    huffman.priority_queue_huild_huffman_tree();
    huffman.getExpectation(exp, 0);

    auto end = std::chrono::high_resolution_clock::now();
    auto elapsed =
        std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin);
    std::cout.precision(6);
    std::cout << "[Huffman] Time measured: " << elapsed.count() * 1e-9
              << " seconds.\n";

    // Max precision for double.
    cout << "Expectation:\n" << fixed << setprecision(3) << exp << endl;

#ifdef DEBUG
    int huffmanArr[MAXN], top = 0;
    cout << huffman.codesToString(huffmanArr, top);

#endif // DEBUG

    fs.open(cliParser.args["--out"], std::ios_base::out);
    // Print sorted numbers to output_file_name
    fs << fixed << setprecision(3) << exp << std::endl;
    fs.close();

    return 0;
}

int main(int argc, char **argv) {
    fstream_main(argc, argv);
    return 0;
}
```

5.1.2 Rust 版本 Huffman 算法

```
pub mod utils;
use std::{
    collections::HashMap,
    fs::File,
    io::{self, Read, Write},
    path::Path,
    time::Instant,
};
use utils::cli_parser;

/**
```

```rust
 * Marcros written for debug log.
 */
#[cfg(feature = "debug")]
macro_rules! debug {
    ($($args: expr), *) => {
        println!($($args), *);
    };
}

#[cfg(not(feature = "debug"))]
macro_rules! debug {
    ($($args: expr), *) => {};
}

struct Node {
    identifier: Option<i32>,
    freq: f64,
    // Use option to represent null node equal to NULL in C++.
    left: Option<Box<Node>>,
    right: Option<Box<Node>>,
}

impl Node {
    #[allow(unused)]
    pub fn to_string(&self) -> String {
        let mut sstream = "".to_string();
        match self.identifier {
            Some(_) => {
                sstream += &("(".to_string()
                    + &self.identifier.unwrap().to_string()
                    + ", "
                    + &self.freq.to_string()
                    + ")");
            }
            None => {
                sstream += "<NULL>";
            }
        }

        sstream
    }
}

fn new_node(f: f64, iden: Option<i32>) -> Node {
    Node {
        identifier: iden,
        freq: f,
        left: None,
        right: None,
    }
}

fn new_box(n: Node) -> Box<Node> {
    Box::new(n)
}

// Assign huffman code to leaves in huffman tree.
#[cfg(feature = "debug")]
```

```rust
fn assign_codes(p: &Box<Node>, s: String) -> String {
    let mut sstream = "".to_string();
    // If the char field of a node is not None, then it is a leaf node.
    // Insert it and its huffman code to the HashMap.
    if let Some(ref l) = p.left {
        sstream += &(assign_codes(l, s.clone() + "0"));
    }
    if let Some(ref r) = p.right {
        sstream += &(assign_codes(r, s.clone() + "1"));
    }
    if let Some(_identifier) = p.identifier {
        sstream.push_str(&p.to_string());
        sstream += ": ";
        sstream.push_str(&s);
        sstream += "\n";
    }

    sstream
}

#[cfg(not(feature = "debug"))]
fn assign_codes(_p: &Box<Node>, _s: String) -> String {
    _s
}

// Convert huffman codes to string.
#[cfg(feature = "debug")]
#[allow(unused)]
fn codes_to_string(
    freq_map: &HashMap<i32, f64>,
    code_map: &mut HashMap<i32, String>,
) -> String {
    let mut s = String::new();
    for (key, value) in code_map.into_iter() {
        s.push_str("(");
        s.push_str(&(key.to_string()));
        s.push_str(", ");
        s.push_str(&((freq_map.get(key)).unwrap().to_string()));
        s.push_str("): ");
        s.push_str(&(value.to_string()));
        s.push_str("\n");
    }

    s
}

// Calculate huffman code length expectation
#[allow(unused)]
#[cfg(feature = "debug")]
fn get_expectation_from_maps(
    freq_map: &mut HashMap<i32, f64>,
    code_map: &mut HashMap<i32, String>,
) -> f64 {
    let mut exp: f64 = 0.0;
    for (key, value) in code_map.into_iter() {
        let freq = freq_map.get(key).unwrap();
        exp += freq * (String::len(value) as f64);
    }
```

```
    exp
}

fn get_expectation(
    freq_map: &HashMap<i32, f64>,
    root: &Box<Node>,
    depth: i32,
    exp: &mut f64,
) -> f64 {
    // If the char field of a node is node None, then it is a leaf node.
    // Accumulate the value calculated by multiplying its frequency and code
        length.
    if let Some(_) = root.identifier {
        *exp += depth as f64 * root.freq;
    } else {
        if let Some(ref l) = root.left {
            get_expectation(freq_map, l, depth + 1, exp);
        }
        if let Some(ref r) = root.right {
            get_expectation(freq_map, r, depth + 1, exp);
        }
    }

    *exp
}

fn load_freq_map_from_file(filename: String) -> HashMap<i32, f64> {
    // The HashMap used to store identifiers and corresponding frequencies.
    let mut freq_map: HashMap<i32, f64> = HashMap::new();

    // Create a path to the desired file.
    let path = Path::new(&filename);
    let display = path.display();

    println!("Load from file: {}", display);

    // Load lines of file to String.
    let mut file = File::open(filename).unwrap();
    let mut contents = String::new();
    file.read_to_string(&mut contents).unwrap();
    let mut lines = contents.lines();

    let size: usize = lines.next().unwrap().parse::<usize>().unwrap();

    // Read the file contents into freq_map.
    let freqs: Vec<&str> = lines.next().unwrap().split(" ").collect();
    for i in 0..size {
        let freq: String = String::from(freqs[i]);
        freq_map.insert(i.try_into().unwrap(), freq.parse::<f64>().unwrap());
    }

    assert!(freq_map.len() == size.try_into().unwrap());

    // println!("Freq_map loaded: {:?}", freq_map);

    freq_map
}
```

```rust
fn write_expectation_to_file(
    filename: String,
    expectation: f64,
) -> io::Result<()> {
    let path = Path::new(&filename);
    let display = path.display();

    println!("Write to file: {}", display);

    let mut file = File::create(filename)?;
    write!(file, "{:.3}\n", expectation)
}

fn main() -> io::Result<()> {
    let mut in_file: String = "data/huffman/huffman.in".to_owned();
    let mut out_file: String = "data/huffman/huffman_rs.out".to_owned();
    (in_file, out_file) = cli_parser(in_file, out_file);

    let freq_map = load_freq_map_from_file(in_file);

    // Put <k, v> from hashmap to vector.
    let mut p: Vec<Box<Node>> = freq_map
        .iter()
        .map(|x| new_box(new_node(*(x.1), Some(*(x.0)))))
        .collect();

    // Measure the elapsed time of Huffman algorithm.
    let begin = Instant::now();

    // Build the huffman tree.
    while p.len() > 1 {
        p.sort_unstable_by(|a, b| (b.freq).partial_cmp(&(a.freq)).unwrap());
        let a = p.pop().unwrap();
        let b = p.pop().unwrap();
        let mut c = new_box(new_node(a.freq + b.freq, None));
        c.left = Some(a);
        c.right = Some(b);
        p.push(c);
    }

    // Get root of huffman tree.
    let root = p
        .pop()
        .unwrap_or_else(|| panic!("Failed to get huffman root."));

    // Create the huffman code.
    let _code_string = assign_codes(&root, "".to_string());
    debug!("{}", _code_string);

    let mut exp: f64 = 0.0;
    exp = get_expectation(&freq_map, &root, 0, &mut exp);
    let elapsed = begin.elapsed().as_secs_f64();

    // Print the elapsed time of Huffman algorithm.
    println!("[Huffman RUST] Time measured: {} seconds.", elapsed);

    // println!("{}", codes_to_string(&mut freq_map, &mut code_map));
```

```
    println!("Expectation:\n{:.3}", exp);

    // Write expectation to file.
    write_expectation_to_file(out_file, exp)
}
```

5.2 数据生成器 data_gen

```
#include <fstream>
#include <iostream>
#include <random>
#include <sstream>
#include <stdexcept>
#include <string>
#include <vector>

using namespace std;
const int default_amount_of_random_numbers = 50;

class Parser {
  public:
    Parser(int argc, char **argv);
    int get_range_from() const { return range_from; }
    int get_range_to() const { return range_to; }
    int get_amount_of_random_numbers() const {
        return amount_of_random_numbers;
    }

  private:
    int parse_unit_arg(char *single_arg);
    void parse_arg_count_4();
    void parse_arg_count_3();
    void parse_no_arg();
    int argc;
    char **argv;
    int range_from;
    int range_to;
    int amount_of_random_numbers;
};
int Parser::parse_unit_arg(char *single_arg) {
    int x = 0;
    string arg = single_arg;
    try {
        size_t pos;
        x = stoi(arg, &pos);
        if (pos < arg.size()) {
            cerr << "Trailing characters after number: " << arg << "\n";
        }
    } catch (invalid_argument const &e) {
        cerr << "Invalid number: " << arg << "\n";
    } catch (out_of_range const &e) {
        cerr << "Number out of range: " << arg << "\n";
    }
    return x;
}
```

```
/**
 * Generate an amount of argv[3] random numbers distributed
 * from argv[1] to argv[2].
 */
void Parser::parse_arg_count_4() {
    range_from = parse_unit_arg(argv[1]);
    range_to = parse_unit_arg(argv[2]);
    amount_of_random_numbers = parse_unit_arg(argv[3]);
}

void Parser::parse_arg_count_3() {
    range_from = parse_unit_arg(argv[1]);
    range_to = parse_unit_arg(argv[2]);
}

Parser::Parser(int argc, char **argv) : argc(argc), argv(argv) {
    if (argc == 4) {
        parse_arg_count_4();
    } else if (argc == 3) {
        parse_arg_count_3();
        amount_of_random_numbers = default_amount_of_random_numbers;
    } else {
        range_from = 0;
        range_to = 1000;
        amount_of_random_numbers = default_amount_of_random_numbers;
    }
}

int main(int argc, char **argv) {
    Parser parser(argc, argv);
    int range_from = parser.get_range_from();
    int range_to = parser.get_range_to();
    int amount_of_random_numbers = parser.get_amount_of_random_numbers();

    random_device rand_dev;
    mt19937 generator(rand_dev());
    uniform_int_distribution<int> distr(range_from, range_to);

    fstream fs;
    string filename = "data/huffman_yasample.in";
    fs.open(filename, ios_base::out);

    vector<int> rawNumbers;
    long long sum = 0;
    // Generate frequencies.
    for (int i = 0; i < amount_of_random_numbers; i++) {
        int tmp = distr(generator);
        rawNumbers.push_back(tmp);
        sum += (long long)tmp;
    }

    if (fs.is_open()) {
        fs << amount_of_random_numbers << std::endl;
        for (int i = 0; i < amount_of_random_numbers; i++) {
            fs << (double)rawNumbers[i] / (double)sum << " ";
        }
        fs << std::endl;
    }
```

```
    fs.close();
    return 0;
}
```

5.3 Dijkstra 算法

5.3.1 c++ 版本 Dijkstra 算法

```
#include "utils/CLIParser.hpp"
#include <cassert>
#include <chrono>
#include <climits>
#include <cstdlib>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <queue>
#include <sstream>
#include <string>
#include <vector>

const int MAXN = 1 << 20;
using namespace std;

// MinHeap
template <typename T> void swap(T *a, T *b) {
    T tmp = *a;
    *a = *b;
    *b = tmp;
}

string arrayToString(int *arr, int len) {
    stringstream ss;
    for (int i = 0; i < len; i++) {
        ss << arr[i] << " ";
    }
    ss << std::endl;
    return ss.str();
}

// Dijkstra
class Edge {
  public:
    int len;
    int to;
    Edge *next;
    Edge() : len(0), to(0), next(nullptr){};
    Edge(int to, int len = 0x7fffffff, Edge *next = nullptr)
        : len(len), to(to), next(next){};
};

ostream &operator<<(ostream &os, Edge &edge) {
    os << "To[" << edge.to << "], Len[" << edge.len << "]";
    return os;
}
```

```
class Node {
  public:
    int idx;
    int dis;
    bool visited;
    Edge *head;
    Node() : idx(-1), dis(0x7fffffff), visited(false), head(nullptr){};
    Node(int node, int dis, bool visited = false)
        : idx(node), dis(dis), visited(visited){};
    bool operator<(const Node &y) const { return y.dis < dis; }
    void linkEdge(Edge *&edge) {
        if (this->head == nullptr) {
            this->head = edge;
        } else {
            auto pEdge = this->head;
            while (pEdge->next != nullptr) {
                // cout << "[" << *pEdge << "]";
                pEdge = pEdge->next;
                // cout << ".nextEdge[" << *pEdge << "] ";
            }
            // now pEdge points to the last edge in current node's adjancent
            // list.
            pEdge->next = edge;
        }
    }
};

ostream &operator<<(ostream &os, Node &node) {
    os << node.idx << ": "
       << "dis[" << node.dis << "], vis[" << node.visited << "], edges:\n";
    Edge *edge = node.head;
    do {
        os << *edge << endl;
    } while ((edge = edge->next) != nullptr);

    return os;
}

class Dijkstra {
  private:
    int begin;
    int end;
    vector<Node *> nodes;

    class CmpNodePtrs {
      public:
        bool operator()(const Node *lhs, const Node *rhs) const {
            return lhs->dis > rhs->dis;
        }
    };

  public:
    Dijkstra(int begin, int end, vector<Node *> &nodes);
    int dijkstra();
};

Dijkstra::Dijkstra(int begin, int end, vector<Node *> &nodes)
    : begin(begin), end(end), nodes(nodes){};
```

```cpp
// Do dijkstra computation. Return distance from begin node to end node.
int Dijkstra::dijkstra() {
    priority_queue<Node *, vector<Node *>, CmpNodePtrs> pq;
    nodes.at(begin)->dis = 0;
    pq.push(nodes.at(begin));

    while (!pq.empty()) {
        Node *node = pq.top();
        pq.pop();
        // Skip nodes that has already visited.
        if (node->visited) {
            continue;
        }

        // Return immediately once we reach the end node.
        // if (node->idx == end) {
        // return node->dis;
        // }

        node->visited = true;
        Edge *edge = node->head;
        // Iter through current node's edges.
        do {
            Node *nextNode = nodes.at(edge->to);
            // Renew the distance from start point to next node.
            if (nextNode->dis > node->dis + edge->len) {
                nextNode->dis = node->dis + edge->len;
                // If next node has not been visited, push it to priority
                    queue.
                if (!nextNode->visited) {
                    pq.push(nextNode);
                }
            }
        } while ((edge = edge->next) != nullptr);
    }

    return (nodes.at(end)->dis == 0x7fffffff) ? -1 : nodes.at(end)->dis;
}

int fstream_main(int argc, char **argv) {
    CLIParser cliParser(argc, argv);
    cliParser.args["--in"] = "data/dijkstra.in";
    cliParser.args["--out"] = "data/dijkstra.out";
    cliParser.argParse();

    fstream fs;
    try {
        fs.open(cliParser.args["--in"], ios_base::in);
    } catch (std::system_error &e) {
        cerr << e.what();
        exit(-1);
    }

    cout << cliParser.args["--in"] << endl;
    cout << cliParser.args["--out"] << endl;
    // assert(fs.is_open() == true);
```

```
    int V, E;
    fs >> V >> E;
    // cout << V << " " << E << endl;

    vector<Node *> nodes;
    for (int i = 0; i <= V; i++) {
        Node *node = new Node();
        nodes.emplace_back(node);
    }

    for (int i = 1; i <= E; i++) {
        int nodeA, nodeB, dis;
        fs >> nodeA >> nodeB >> dis;

        // Add edge to nodeA
        Edge *tmp = new Edge(nodeB, dis);
        Node *node = nodes.at(nodeA);
        node->idx = nodeA;
        // Link new edge to nodeA's adjacent list.
        node->linkEdge(tmp);

        // Add edge to nodeB
        tmp = new Edge(nodeA, dis);
        node = nodes.at(nodeB);
        node->idx = nodeB;
        // Link new edge to nodeB's adjacant list.
        node->linkEdge(tmp);
    }

    // for (int i = 1; i <= V; i++) {
    // cout << *nodes.at(i) << endl;
    // }

    Dijkstra dij(1, V, nodes);

    fs.close();

    auto begin = std::chrono::high_resolution_clock::now();

    int result = dij.dijkstra();

    auto end = std::chrono::high_resolution_clock::now();
    auto elapsed =
        std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin);
    std::cout << "[dijkstra] Time measured: " << elapsed.count() * 1e-9
              << " seconds.\n";

    cout << result << endl;

    fs.open(cliParser.args["--out"], std::ios_base::out);
    // Print sorted numbers to output_file_name
    fs << result << std::endl;
    fs.close();

    return 0;
}

int stdio_main() {
```

```
    int V, E;
    cin >> V >> E;
    vector<Node *> nodes;
    for (int i = 0; i <= V; i++) {
        Node *node = new Node();
        nodes.emplace_back(node);
    }

    for (int i = 1; i <= E; i++) {
        int nodeA, nodeB, dis;
        cin >> nodeA >> nodeB >> dis;

        // Add edge to nodeA
        Edge *tmp = new Edge(nodeB, dis);
        Node *node = nodes.at(nodeA);
        node->idx = nodeA;
        node->head = tmp;

        // Add edge to nodeB
        tmp = new Edge(nodeA, dis);
        node = nodes.at(nodeB);
        node->idx = nodeB;
        node->head = tmp;
    }

    Dijkstra dij(1, V, nodes);
    cout << dij.dijkstra();
    return 0;
}

int main(int argc, char **argv) {
    // stdio_main();
    fstream_main(argc, argv);
    return 0;
}
```

5.3.2 Rust 版本 Dijkstra 算法

```
pub mod utils;
use std::{
    cmp::Ordering,
    collections::BinaryHeap,
    fs::File,
    io::{self, Read, Write},
    path::Path,
    time::Instant,
};

use crate::utils::cli_parser;

#[derive(Clone, Copy)]
struct Edge {
    node: i32,
    dis: i32,
}
```

```rust
// Derive Copy, Clone, PartialEq and Eq for Ord and PartialOrd of State.
#[derive(Clone, Copy, PartialEq, Eq)]
struct State {
    dis: i32,
    position: i32,
}

// Implement Ord for priority queue.
// To let it be a min-heap.
impl Ord for State {
    fn cmp(&self, other: &Self) -> Ordering {
        other
            .dis
            .cmp(&self.dis)
            .then_with(|| self.position.cmp(&other.position))
    }
}

// Implement PartialOrd for Orf of State
impl PartialOrd for State {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}

fn dijkstra(adj_list: &Vec<Vec<Edge>>, begin: i32, end: i32) -> Option<i32> {
    // dist[node] = current shortest distance from begin to node.
    let mut dist: Vec<_> = (0..adj_list.len()).map(|_| i32::MAX).collect();

    let mut heap = BinaryHeap::new();

    dist[begin as usize] = 0;
    heap.push(State {
        dis: 0,
        position: begin,
    });

    while let Some(State { dis, position }) = heap.pop() {
        // If reaches end node, return. Else can stay here and find shortest
            way to all
        // nodes.
        if position == end {
            return Some(dis);
        }

        // If an shorter way already exists, skip this state.
        if dis > dist[position as usize] {
            continue;
        }

        for edge in &adj_list[position as usize] {
            let next_node = State {
                dis: dis + edge.dis,
                position: edge.node,
            };

            if next_node.dis < dist[next_node.position as usize] {
                heap.push(next_node);
```

```
                dist[next_node.position as usize] = next_node.dis;
            }
        }
    }

    // Return none if end node not reached.
    None
}

fn read_graph_from_file(filename: String) -> (i32, i32, Vec<Vec<Edge>>) {
    let path = Path::new(&filename);
    let display = path.display();

    println!("Load from file: {}", display);

    let mut file = File::open(filename).unwrap();
    let mut contents = String::new();
    file.read_to_string(&mut contents).unwrap();
    let mut lines = contents.lines();

    let v: i32;
    let e: i32;
    let first_line: Vec<&str> = lines.next().unwrap().split(" ").collect();
    v = first_line[0].parse::<i32>().unwrap();
    e = first_line[1].parse::<i32>().unwrap();

    // adj_list[i] collects edges starting from node i.
    let mut adj_list: Vec<Vec<Edge>> = vec![vec![]; (v + 1) as usize];
    // println!("adj_list[{}]", adj_list.capacity());

    for line in lines {
        // println!("{}", line);

        let vec: Vec<_> = line.split(" ").collect();
        if vec.len() < 3 {
            break;
        }

        let node_a: i32 = vec[0].parse::<i32>().unwrap();
        let node_b: i32 = vec[1].parse::<i32>().unwrap();
        let dist: i32 = vec[2].parse::<i32>().unwrap();

        let edges_a: &mut Vec<Edge> = &mut adj_list[node_a as usize];
        edges_a.push(Edge {
            node: node_b,
            dis: dist,
        });

        adj_list[node_b as usize].push(Edge {
            node: node_a,
            dis: dist,
        })
    }

    // for i in 1..e {
    // println!("{}: edges:", i);
    // for &mut edge in &mut adj_list[i as usize] {
    // println!("To[{}], Len[{}]", edge.node, edge.dis);
```

```
    // }
    // println!("");
    // }

    (v, e, adj_list)
}

fn write_shortest_distance_to_file(
    filename: String,
    distance: i32,
) -> io::Result<()> {
    let path = Path::new(&filename);
    let display = path.display();

    println!("Write to file: {}", display);

    let mut file = File::create(filename)?;
    write!(file, "{}\n", distance)
}

fn main() {
    let mut in_file: String = "data/dijkstra.in".to_owned();
    let mut out_file: String = "data/dijkstra.out".to_owned();
    (in_file, out_file) = cli_parser(in_file, out_file);
    let (v, _e, adj_list) = read_graph_from_file(in_file);
    // TODO: Add timer.
    let begin = Instant::now();
    let result = dijkstra(&adj_list, 1, v);
    let elapsed = begin.elapsed().as_secs_f64();
    println!("[Dijkstra Rust] Time measured: {} seconds.", elapsed);
    let distance: i32;
    match result {
        // If node[e] cannot be reached from node[1], assign 1 to distance.
        None => distance = -1,
        Some(result) => {
            distance = result;
        }
    }
    println!("{}", distance);

    let ok = write_shortest_distance_to_file(out_file.clone(), distance);
    if ok.is_err() {
        panic!(
            "Panicked to write result[{}] to file[{}]",
            distance,
            out_file.clone()
        );
    }
}
```

5.4 数据生成器 data_gen

```
#include <fstream>
#include <iostream>
#include <random>
#include <sstream>
```

```cpp
#include <stdexcept>
#include <string>
#include <unordered_set>
#include <vector>

using namespace std;
const int default_amount_of_random_numbers = 50;

class Parser {
  public:
    Parser(int argc, char **argv);
    int get_range_from() const { return range_from; }
    int get_range_to() const { return range_to; }
    int get_amount_of_random_numbers() const {
        return amount_of_random_numbers;
    }

  private:
    int parse_unit_arg(char *single_arg);
    void parse_arg_count_4();
    void parse_arg_count_3();
    void parse_no_arg();
    int argc;
    char **argv;
    int range_from;
    int range_to;
    int amount_of_random_numbers;
};
int Parser::parse_unit_arg(char *single_arg) {
    int x = 0;
    string arg = single_arg;
    try {
        size_t pos;
        x = stoi(arg, &pos);
        if (pos < arg.size()) {
            cerr << "Trailing characters after number: " << arg << "\n";
        }
    } catch (invalid_argument const &e) {
        cerr << "Invalid number: " << arg << "\n";
    } catch (out_of_range const &e) {
        cerr << "Number out of range: " << arg << "\n";
    }
    return x;
}

/**
 * Generate an amount of argv[3] random numbers distributed
 * from argv[1] to argv[2].
 */
void Parser::parse_arg_count_4() {
    range_from = parse_unit_arg(argv[1]);
    range_to = parse_unit_arg(argv[2]);
    amount_of_random_numbers = parse_unit_arg(argv[3]);
}

void Parser::parse_arg_count_3() {
    range_from = parse_unit_arg(argv[1]);
    range_to = parse_unit_arg(argv[2]);
```

```
}

Parser::Parser(int argc, char **argv) : argc(argc), argv(argv) {
    if (argc == 4) {
        parse_arg_count_4();
    } else if (argc == 3) {
        parse_arg_count_3();
        amount_of_random_numbers = default_amount_of_random_numbers;
    } else {
        range_from = 0;
        range_to = 1000;
        amount_of_random_numbers = default_amount_of_random_numbers;
    }
}

class Edge {
  public:
    int nodeA;
    int nodeB;

    Edge(int a, int b) : nodeA(a), nodeB(b){};
};

bool operator==(const Edge &a, const Edge &b) {
    return ((a.nodeA == b.nodeB) && (a.nodeB == b.nodeA)) ||
           ((a.nodeB == b.nodeA) && (a.nodeA == b.nodeB));
}

class EdgeEq {
  public:
    bool operator()(const Edge &a, const Edge &b) const { return a == b; }
};

class EdgeHash {
  public:
    size_t operator()(const Edge &edge) const {
        return std::hash<int>()(edge.nodeA + edge.nodeB);
    }
};

int main(int argc, char **argv) {
    Parser parser(argc, argv);
    int range_from = parser.get_range_from();
    int range_to = parser.get_range_to();
    int amount_of_random_numbers = parser.get_amount_of_random_numbers();

    random_device rand_dev;
    mt19937 generator(rand_dev());

    // Distance random selector
    uniform_int_distribution<int> distr(range_from, range_to);

    // Node index random selector
    uniform_int_distribution<int> node_distr(1, amount_of_random_numbers);

    fstream fs;
    string filename = "data/dijkstra_yasample.in";
    fs.open(filename, ios_base::out);
```

```
    unordered_set<Edge, EdgeHash, EdgeEq> edges;

    // Number of nodes
    int num_nodes = amount_of_random_numbers;

    if (fs.is_open()) {
        // Generate edges.
        for (int i = 1; i <= num_nodes; i++) {
            int node_a = i;
            int node_b = node_distr(generator);

            // If node_a == node_b, regenerate node_b.
            while (node_a == node_b) {
                node_b = node_distr(generator);
            }

            Edge edge(node_a, node_b);
            // Edge already exists. Regenerate one.
            while (edges.find(edge) != edges.end()) {
                edge.nodeB = node_distr(generator);
            }

            edges.insert(edge);
        }

        int num_edges = edges.size();
        fs << num_nodes << " " << num_edges << endl;

        // Generate distance and write edges to file.
        for (auto &a : edges) {
            int dis = distr(generator);
            fs << a.nodeA << " " << a.nodeB << " " << dis << endl;
        }
    }
    fs.close();
    return 0;
}
```

5.5 Prim 算法

5.5.1 c++ 版本 Prim 算法

```
#include "utils/CLIParser.hpp"
#include <cassert>
#include <chrono>
#include <climits>
#include <exception>
#include <fstream>
#include <ios>
#include <iostream>
#include <queue>
#include <sstream>
#include <stdexcept>
#include <string>
#include <utility>
```

```cpp
#include <vector>

using namespace std;

const int INF = 0x7fffffff;

class Edge {
  public:
    int from;
    int to;
    int weight;
    Edge *next;

    Edge(int from, int to, int weight)
        : from(from), to(to), weight(weight), next(nullptr){};
};

ostream &operator<<(ostream &os, const Edge &e) {
    auto edge = e;
    // Print first edge.
    os << "from[" << edge.from << "], to[" << edge.to << "], "
       << "weight[" << edge.weight << "]" << endl;

    return os;
}

class Vertex {
  public:
    int idx;
    int distance;
    int parent;
    Edge *head;
    bool inMST;

    Vertex(int idx = -1, int weight = INF, Edge *head = nullptr)
        : idx(idx), distance(weight), parent(-1), head(head), inMST(false){};
    void addEdge(Edge *&edge);
};

ostream &operator<<(ostream &os, const Vertex &ver) {
    os << ver.idx << ": "
       << "distance[" << ver.distance << "], parent[" << ver.parent
       << "], inMST[" << ver.inMST << "]";
    return os;
}

// Only insert new edge with new following edges.
void Vertex::addEdge(Edge *&edge) {
    assert(edge->next == nullptr);
    // Allocate and push new edge to vertex a's adjacent list.
    if (this->head == nullptr) {
        this->head = edge;
    } else {
        auto last_edge = this->head;
        // Find last edge in thistex a's adjacent list.
        while (last_edge->next != nullptr) {
            last_edge = last_edge->next;
        }
```

```
        // Link new edge next to last_edge's next.
        last_edge->next = edge;
    }
}

class CmpVertexPtr {
  public:
    /**
     * First, compare distance of vertices. Then, compare their index.
     */
    bool operator()(const Vertex *a, const Vertex *b) const {
        if (a->distance == b->distance) {
            return a->idx > b->idx;
        }
        return a->distance > b->distance;
    }
};

class Prim {
  public:
    vector<Vertex *> vertices;
    int begin;
    int target;
    int weight_sum;

    vector<Edge *> edges_in_MST;
    const string vertices_to_string() const;
    const string edges_in_mst_to_string() const;
    int prim_MST();
    Prim(vector<Vertex *> &vers, int begin)
        : vertices(vers), begin(begin), target(begin), weight_sum(0){};
};

const string Prim::edges_in_mst_to_string() const {
    stringstream ss;
    for (auto a : edges_in_MST) {
        ss << *a;
    }
    return ss.str();
}

const string Prim::vertices_to_string() const {
    stringstream ss;
    // vertices.size() = V + 1
    for (size_t i = 1; i < vertices.size(); i++) {
        ss << *(vertices[i]);
    }
    return ss.str();
}

int Prim::prim_MST() {
    priority_queue<Vertex *, vector<Vertex *>, CmpVertexPtr> pq;
    // Initialize the distance between begin and begin to 0.
    auto ver = vertices.at(begin);
    ver->distance = 0;

    // Initialize pq by pushing begin vertex into pq.
    pq.push(ver);
```

```
    // Find edge whose two endpoints contains u and the other endpoint
        already
    // in MST.
    auto find_edge = [&](const Vertex *u) {
        auto edge = u->head;
        // Iterate through u's edges until edge.to are already in MST.
        while (edge != nullptr && !(vertices.at(edge->to)->inMST)) {
            edge = edge->next;
        }

        // If edge found, return pointer pointing to the edge. Else return
        // nullptr.
        return edge;
    };

    // Build MST using MinHeap.
    while (!pq.empty()) {
        // Pick the neighbor vertex with minimal distance, add it to set S.
        ver = pq.top();
        pq.pop();

        // Skip current vertex if it is already in MST.
        if (ver->inMST) {
            continue;
        }

        // Find the edge corresponding to current vertex. Skip source vertex.
        if (ver->idx != begin) {
            if (auto last_edge = find_edge(ver)) {
                edges_in_MST.push_back(last_edge);
            } else {
                // Corresponding edge not found. Throw exception.
                stringstream ss;
                ss << "Failed to fild the edge that are just inserted to MST.
                    "
                      "Current Vertex["
                   << ver->idx << "]";
                throw std::runtime_error(ss.str());
            }
        }

        // Target vertex is the last vertex to be included in the MST.
        // By using this, we can find targe vertex. Thus can print the MST
        // with the help of Vertex.parent. Compute the sum of weights when
        // building MST.
        ver->inMST = true;
        target = ver->idx;
        weight_sum += ver->distance;

        // Iterate through ver's neighbor vertices.
        auto edge = ver->head;
        while (edge != nullptr) {
            auto neighbor_ver = vertices[edge->to];
            // Skip neighbor vertices that are already in MST.
            if (neighbor_ver->inMST) {
                edge = edge->next;
                continue;
```

```
            }
            // Update neighbor_ver's distance to begin vertex.
            if (neighbor_ver->distance > edge->weight) {
                neighbor_ver->distance = edge->weight;
            }

#ifdef DEBUG
            cout << "Current edge: " << *edge
                 << "Neighbor vertex: " << *neighbor_ver << endl;

#endif // DEBUG
            pq.push(neighbor_ver);

            // Next edge in vertex's adjacent list.
            edge = edge->next;
        }
    }

    return weight_sum;
}

int fs_main(int argc, char **argv) {
    CLIParser cliParser(argc, argv);
    cliParser.args["--in"] = "data/prim.in";
    cliParser.args["--out"] = "data/prim.out";
    cliParser.argParse();

    fstream fs;
    try {
        fs.open(cliParser.args["--in"], ios_base::in);
    } catch (std::exception &e) {
        cerr << e.what();
        cerr << "Failed to load data from " << cliParser.args["--in"]
             << ". Exit now.";
        exit(-1);
    }

    cout << "Load data from: " << cliParser.args["--in"] << endl;
    cout << "Write data to: " << cliParser.args["--out"] << endl;

    int V, E;
    fs >> V >> E;

    // Initialize vertices.
    vector<Vertex *> vertices;
    for (int i = 0; i <= V; i++) {
        vertices.emplace_back(new Vertex(i));
    }

    for (int i = 0; i < E; i++) {
        int idx_a, idx_b, weight;
        fs >> idx_a >> idx_b >> weight;

        // Initialize vertex a.
        auto ver = vertices.at(idx_a);
        assert(ver->idx == idx_a);
        Edge *edge = new Edge(idx_a, idx_b, weight);
        ver->addEdge(edge);
```

```
        // Initialize vertex b.
        ver = vertices.at(idx_b);
        assert(ver->idx == idx_b);
        edge = new Edge(idx_b, idx_a, weight);
        ver->addEdge(edge);
    }

    // Initialize Prim which treats vertex 1 as source of MST.
    Prim prim(vertices, 1);
    // cout << prim.vertices_to_string() << endl;

    // End of read and initialize prim.
    fs.close();

    // Use timer to record how much time the Prim algorithm with MinHeap
    // optimization consumes.

    auto begin = chrono::high_resolution_clock::now();

    // Prim algorithm build MST.
    auto result = prim.prim_MST();

    auto end = chrono::high_resolution_clock::now();
    auto elapsed = chrono::duration_cast<chrono::nanoseconds>(end - begin);
    cout.precision(6);
    cout << "[Prim] Time measured: " << elapsed.count() * 1e-9 << "
        seconds.\n";

#ifdef DEBUG
    // print vertces for debug purpose.
    cout << prim.edges_in_MST.size() << " edges in MST.\n";
    cout << prim.edges_in_mst_to_string();

#endif // DEBUG

    // Print result for debug purpose.
    cout << result << endl;

    // Write result to file.
    fs.open(cliParser.args["--out"], ios_base::out);
    fs << result << endl;
    fs.close();

    return 0;
}

int main(int argc, char **argv) { return fs_main(argc, argv); }
```

5.5.2 Rust 版本 Prim 算法

```
pub mod utils;
use core::fmt;
use std::{
    collections::{BinaryHeap, HashSet},
    fs::File,
```

```
    io::{self, Read, Write},
    path::Path,
    time::Instant,
};

use crate::utils::cli_parser;

/**
 * Macros written for debug purpose.
 *
 * If debug output is needed, add --features debug flag in compilation.
 */
#[cfg(feature = "debug")]
macro_rules! debug {
    ($($args: expr), *) => {
        println!($($args), *);
    };
}

#[cfg(not(feature = "debug"))]
macro_rules! debug {
    ($($args: expr), *) => {};
}

#[derive(Clone, Copy, PartialEq, Eq)]
struct Edge {
    from: i32,
    to: i32,
    weight: i32,
}

impl PartialOrd for Edge {
    // Partial Comparator for Edge. First compare weight, then compare from
        vertex, finally compare to vertex.
    fn partial_cmp(&self, other: &Self) -> Option<std::cmp::Ordering> {
        if self.weight == other.weight {
            if self.from == other.from {
                return self.to.partial_cmp(&other.to);
            }
            return self.from.partial_cmp(&other.from);
        }

        self.weight.partial_cmp(&other.weight)
    }
}

impl fmt::Display for Edge {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(
            f,
            "From[{}], To[{}], Weight[{}]",
            self.from, self.to, self.weight
        )
    }
}

fn read_graph_from_file(filename: String) -> (i32, i32, Vec<Vec<Edge>>) {
    let path = Path::new(&filename);
```

```
    let display = path.display();
    println!("Load from file: {}", display);
    let mut file = File::open(filename).expect("Error opening file.");

    // Get lines from file.
    let mut contents = String::new();
    file.read_to_string(&mut contents)
        .expect("Error reading file.");
    let mut lines = contents.lines();

    let v: i32;
    let e: i32;
    let new_line = lines.next();
    // Get first line and handle error.
    let first_line: Vec<&str> = match new_line {
        Some(new_line) => new_line.split(" ").collect(),
        None => {
            println!("Error reading V and E.");
            Vec::new()
        }
    };
    // Read V and E from file.
    v = first_line[0].parse::<i32>().expect("Error reading V.");
    e = first_line[1].parse::<i32>().expect("Error reading E.");

    // adj_list[i] collects edges starting from vertex i.
    let mut adj_list: Vec<Vec<Edge>> = vec![vec![]; (v + 1) as usize];
    // println!("adj_list[{}]", adj_list.capacity());

    for line in lines {
        // println!("{}", line);

        let vec: Vec<&str> = line.split(" ").collect();
        // Stop reading when line doesn't match the input requirement.
        if vec.len() < 3 {
            break;
        }

        let ver_a: i32 = vec[0]
            .parse::<i32>()
            .expect("Error reading vertex {vec[0]}");
        let ver_b: i32 = vec[1]
            .parse::<i32>()
            .expect("Error reading vertex {vec[1]}");
        let wei: i32 = vec[2]
            .parse::<i32>()
            .expect("Error reading weight {vec[2]}");

        // Create edge and append to vertex a's adjacent list.
        let edges_a = &mut adj_list[ver_a as usize];
        edges_a.push(Edge {
            from: ver_a,
            to: ver_b,
            weight: wei,
        });

        // Create edge and append to vertex b's adjacent list.
        adj_list[ver_b as usize].push(Edge {
```

```
            from: ver_b,
            to: ver_a,
            weight: wei,
        });
    }

    // Print adjacent lists to debug.
    for i in 1..e {
        debug!("{}: edges:", i);
        // Borrow edge from adj_list.
        for &_edge in &adj_list[i as usize] {
            debug!("to[{}], weight[{}]", _edge.to, _edge.weight);
        }
    }

    (v, e, adj_list)
}

#[derive(Copy, Clone, PartialEq, Eq, Ord)]
struct State {
    distance: i32,
    vertex: i32,
}

// Implement ParitialOrd for State for MinHeap.
impl PartialOrd for State {
    // Reverse from greater fist to lesser first.
    fn partial_cmp(&self, other: &Self) -> Option<std::cmp::Ordering> {
        self.distance
            .partial_cmp(&other.distance)
            .map(|ord| ord.reverse())
    }
}

// Return end vertex ,weight sum and edges in Prim MST.
fn prim(adj: &Vec<Vec<Edge>>, source: i32, v: i32) -> (i32, i32, Vec<Edge>) {
    type Vertex = i32;
    type Distance = i32;
    // Memorize end vertex in MST.
    let mut end: Vertex = source;
    // Accumulate MST weight sum.
    let mut weight_sum: i32 = 0;

    // Edges in Prim MST for debug.
    let mut edges_in_mst: Vec<Edge> = Vec::new();

    // Initialize states of all vertices.
    let mut dists: Vec<Distance> = Vec::new();
    dists.extend((0..v + 1).into_iter().map(|_| i32::MAX));

    // Initialize state of source's distance to 0 and push source into
        Minheap.
    let mut heap: BinaryHeap<State> = BinaryHeap::new();
    // Get ownership of distance of vertex source.
    let dis = &mut dists[source as usize];
    *dis = 0;
    heap.push(State {
        distance: *dis,
```

```
        vertex: source,
    });

    // Collect vertices that are already in MST to avoid adding a point
        twice.
    let mut mst: HashSet<Vertex> = HashSet::new();

    // Build MST.
    while !heap.is_empty() {
        // stat gets top vertex's ownership.
        let stat = heap.pop().expect("Error picking top state from heap.");

        // Skip vertices already in MST.
        if mst.contains(&(stat.vertex)) {
            continue;
        }

        // Insert current vertex into MST.
        mst.insert(stat.vertex);

        // Function used to find edge that one endpoint is u, the other
            endpoint has already been
        // in MST.
        // stat.distance == edge.weight and edge.to in MST.
        let find_edge = |s: State| -> Option<Edge> {
            for e in &adj[s.vertex as usize] {
                // println!("Current edge: {}", e);
                if mst.contains(&e.to) {
                    return Some(*e);
                }
            }

            // Return None if edge not fount.
            None
        };

        // Add current edge to edges_in_mst for debug purpose.
        // Skip if current vertex is source.
        if stat.vertex != source {
            let edge_curr = find_edge(stat).unwrap_or_else(|| {
                panic!(
                    "Failed to find edge from {} that are already in MST.",
                    stat.vertex
                )
            });
            edges_in_mst.push(edge_curr);
        }

        // Memorize the end vertex.
        end = stat.vertex;
        weight_sum += stat.distance as i32;

        // Iterate through current state vertex's edges.
        let edges = &adj[stat.vertex as usize];
        for edge in edges {
            let neighbor_ver = edge.to;
            let neighbor_dist = &mut dists[edge.to as usize];
            if *neighbor_dist > edge.weight {
```

```
                *neighbor_dist = edge.weight;
            }
            heap.push(State {
                distance: *neighbor_dist,
                vertex: neighbor_ver,
            });
        }
    }

    (end, weight_sum, edges_in_mst)
}

fn write_distance_to_file(filename: &String, distence: &i32) ->
    io::Result<()> {
    let path = Path::new(&filename);
    let display = path.display();

    println!("Write to file: {}", &display);

    // Try to create new file {filename}.
    let mut file = File::create(&filename)?;
    write!(file, "{}\n", distence)
}

fn main() {
    let mut in_file: String = "data/prim.in".to_string();
    let mut out_file: String = "data/prim_rs.out".to_string();
    (in_file, out_file) = cli_parser(in_file, out_file);
    let (v, _e, adj_list) = read_graph_from_file(in_file);
    // Compute Prim MST cost whose source is vertex 1.
    let source: i32 = 1;

    // Measure the time elapsed of prim algorithm.
    let begin = Instant::now();
    let (_end, distance, edges_in_mst) = prim(&adj_list, source, v);
    // Record the elapsed time in seconds.
    let elapsed = begin.elapsed().as_secs_f64();
    println!("[Prim RUST] Time measured: {:?} seconds.", elapsed);

    for _e in edges_in_mst {
        debug!("{}", _e);
    }
    // Error handler
    println!("{}", distance);

    // Write result to out file.
    let ok = write_distance_to_file(&out_file, &distance);
    if ok.is_err() {
        panic!(
            "Panicked to write result[{}] to file [{}]",
            distance, &out_file
        );
    }
}
```

5.6 数据生成器 data_gen

```cpp
#include <fstream>
#include <iostream>
#include <random>
#include <sstream>
#include <stdexcept>
#include <string>
#include <unordered_set>
#include <vector>

using namespace std;
const int default_amount_of_random_numbers = 50;

class Parser {
  public:
    Parser(int argc, char **argv);
    int get_range_from() const { return range_from; }
    int get_range_to() const { return range_to; }
    int get_amount_of_random_numbers() const {
        return amount_of_random_numbers;
    }

  private:
    int parse_unit_arg(char *single_arg);
    void parse_arg_count_4();
    void parse_arg_count_3();
    void parse_no_arg();
    int argc;
    char **argv;
    int range_from;
    int range_to;
    int amount_of_random_numbers;
};
int Parser::parse_unit_arg(char *single_arg) {
    int x = 0;
    string arg = single_arg;
    try {
        size_t pos;
        x = stoi(arg, &pos);
        if (pos < arg.size()) {
            cerr << "Trailing characters after number: " << arg << "\n";
        }
    } catch (invalid_argument const &e) {
        cerr << "Invalid number: " << arg << "\n";
    } catch (out_of_range const &e) {
        cerr << "Number out of range: " << arg << "\n";
    }
    return x;
}

/**
 * Generate an amount of argv[3] random numbers distributed
 * from argv[1] to argv[2].
 */
void Parser::parse_arg_count_4() {
    range_from = parse_unit_arg(argv[1]);
    range_to = parse_unit_arg(argv[2]);
```

```
    amount_of_random_numbers = parse_unit_arg(argv[3]);
}

void Parser::parse_arg_count_3() {
    range_from = parse_unit_arg(argv[1]);
    range_to = parse_unit_arg(argv[2]);
}

Parser::Parser(int argc, char **argv) : argc(argc), argv(argv) {
    if (argc == 4) {
        parse_arg_count_4();
    } else if (argc == 3) {
        parse_arg_count_3();
        amount_of_random_numbers = default_amount_of_random_numbers;
    } else {
        range_from = 0;
        range_to = 1000;
        amount_of_random_numbers = default_amount_of_random_numbers;
    }
}

class Edge {
  public:
    int nodeA;
    int nodeB;

    Edge(int a, int b) : nodeA(a), nodeB(b){};
};

bool operator==(const Edge &a, const Edge &b) {
    return ((a.nodeA == b.nodeB) && (a.nodeB == b.nodeA)) ||
           ((a.nodeB == b.nodeA) && (a.nodeA == b.nodeB));
}

class EdgeEq {
  public:
    bool operator()(const Edge &a, const Edge &b) const { return a == b; }
};

class EdgeHash {
  public:
    size_t operator()(const Edge &edge) const {
        return std::hash<int>()(edge.nodeA + edge.nodeB);
    }
};

int main(int argc, char **argv) {
    Parser parser(argc, argv);
    int range_from = parser.get_range_from();
    int range_to = parser.get_range_to();
    int amount_of_random_numbers = parser.get_amount_of_random_numbers();

    random_device rand_dev;
    mt19937 generator(rand_dev());

    // Distance random selector
    uniform_int_distribution<int> distr(range_from, range_to);
```

```
    // Node index random selector
    uniform_int_distribution<int> node_distr(1, amount_of_random_numbers);

    fstream fs;
    string filename = "data/prim_yasample.in";
    fs.open(filename, ios_base::out);

    unordered_set<Edge, EdgeHash, EdgeEq> edges;

    // Number of nodes
    int num_nodes = amount_of_random_numbers;

    if (fs.is_open()) {
        // Generate edges.
        for (int i = 1; i <= num_nodes; i++) {

            // Generate new node that has no conflict with node_a.
            auto new_node = [&](const int node_a) {
                int new_node = node_distr(generator);
                // If node_a == node_b, regenerate node_b.
                while (node_a == new_node) {
                    new_node = node_distr(generator);
                }

                return new_node;
            };

            int node_a = i;
            int node_b = new_node(node_a);

            Edge edge(node_a, node_b);
            // If edge already exists. Regenerate one.
            while (edges.find(edge) != edges.end()) {
                edge.nodeB = new_node(node_a);
            }

            edges.insert(edge);
        }

        int num_edges = edges.size();
        fs << num_nodes << " " << num_edges << endl;

        // Generate distance and write edges to file.
        for (auto &a : edges) {
            int dis = distr(generator);
            fs << a.nodeA << " " << a.nodeB << " " << dis << endl;
        }
    }
    fs.close();
    return 0;
}
```

5.7 Kruskal 算法

5.7.1 c++ 版本 Kruskal 算法

```cpp
#include "utils/CLIParser.hpp"
#include <algorithm>
#include <chrono>
#include <exception>
#include <fstream>
#include <functional>
#include <iomanip>
#include <ios>
#include <iostream>
#include <iterator>
#include <queue>
#include <sstream>
#include <unordered_set>
#include <vector>

using namespace std;

// Kruskal is Edge - oriented, so there's no need to implement adjacent list
    for
// vertices.
typedef int Vertex;
typedef int Distance;
typedef int Weight;

template <typename T> string set_to_string(unordered_set<T *> set) {
    stringstream ss;
    for (auto &a : set) {
        ss << *a << " ";
    }
    return ss.str();
}

class ConnectedBranch {
  public:
    // Vertices in the connected branch.
    unordered_set<Vertex, vector<Vertex>, equal_to<Vertex>> vertices;

    ConnectedBranch(){};
};

class Edge {
  public:
    Vertex a;
    Vertex b;
    Weight weight;
    Edge(const Vertex &a, const Vertex &b, int weight)
        : a(a), b(b), weight(weight){};
    Edge(const Edge &e) : a(e.a), b(e.b), weight(e.weight){};
};

ostream &operator<<(ostream &os, const Edge &edge) {
    if (edge.a <= edge.b) {
        os << "(" << edge.a << ", " << edge.b << "): " << edge.weight;
    } else {
        os << "(" << edge.b << ", " << edge.a << "): " << edge.weight;
    }
    return os;
}
```

```
bool operator==(const Edge &edge_a, const Edge &edge_b) {
    return ((edge_a.weight == edge_b.weight) &&
            (((edge_a.a == edge_b.a) && (edge_a.b == edge_b.b)) ||
             ((edge_a.b == edge_b.a) && (edge_a.a == edge_b.b))));
}

/**
 * Edge hasher for unordered_set.
 */
class EdgeHash {
  public:
    size_t operator()(const Edge &edge) const {
        return hash<int>()(edge.a + edge.b + edge.weight);
    }
};

/**
 * Edge equal for unordered_set.
 */
class EdgeEq {
  public:
    bool operator()(const Edge &a, const Edge &b) const { return a == b; }
};

/**
 * Edge pointer comparator for priority_queue.
 */
class CmpEdgePtr {
  public:
    bool operator()(const Edge *edge_a, const Edge *edge_b) const {
        auto greater = [&](auto a, auto b) { return a > b; };
        auto eq = [&](auto a, auto b) { return a == b; };

        if (eq(edge_a->weight, edge_b->weight)) {
            if (eq(edge_a->a, edge_b->a)) {
                return greater(edge_a->b, edge_b->b);
            } else {
                return greater(edge_a->a, edge_b->a);
            }
        }
        return edge_a->weight > edge_b->weight;
    }
};

/**
 * Edge comparator for priority_queue
 */
class CmpEdge {
  public:
    bool operator()(const Edge eda, const Edge edb) const {
        if (eda.weight == edb.weight) {
            if (eda == edb) {
                return false;
            } else {
                if (eda.a == edb.a) {
                    return eda.b > edb.b;
                } else {
```

```
                    return eda.a > edb.a;
                }
            }
        } else {
            return eda.weight > edb.weight;
        }
    }
};

class Kruskal {
  public:
    Vertex source;
    // Number of edges.
    int e;
    // Number of vertices.
    int v;
    // Sum of weights of edges in MST.
    Weight weight_sum;

    unordered_set<Edge, EdgeHash, EdgeEq> edges_in_mst;
    unordered_set<Vertex, hash<Vertex>, equal_to<Vertex>> vertices_in_mst;

    vector<Edge> edges;

    Kruskal(Vertex source, int e, int v, vector<Edge> &edges)
        : source(source), e(e), v(v), weight_sum(0), edges(edges){};
    Distance kruskal_MST();

    // Debug strings.
    string ver_in_mst_to_string() const;
    string edges_in_mst_to_string() const;
};

string Kruskal::ver_in_mst_to_string() const {
    stringstream ss;
    ss << vertices_in_mst.size() << " Vertices in MST: \n";
    for (auto &a : vertices_in_mst) {
        ss << a << " ";
    }
    return ss.str();
}

string Kruskal::edges_in_mst_to_string() const {
    stringstream ss;
    ss << edges_in_mst.size() << " Edges in MST:\n";
    for (auto &a : edges_in_mst) {
        ss << a << endl;
    }
    return ss.str();
}

Distance Kruskal::kruskal_MST() {
    // Use MinHeap to optimize.
    priority_queue<Edge, vector<Edge>, CmpEdge> heap;

    /** Initialize all connected branches by treating all vertices as
        indepedent
     * connected branch.
```

```
     *
     * To merge two connected branches i and j, let
     * branches[i] and branches[j] point to the same ConnectedBranch after
     * unordered_set::merge.
     * To check whether Vertex a and Vertex b are in the same
         ConnectedBranch,
     * Test if branches[a].contains(b);
     */
    typedef unordered_set<Vertex> ConnectedBranch;
    vector<ConnectedBranch *> branches;

    // Detemine whether two vertices are in the same branch.
    auto in_same_branch = [&](Vertex a, Vertex b) {
        return (branches[a]->find(b) != branches[a]->end());
    };

    // Merge two ConnectedBranched.
    auto merge_branches = [&](ConnectedBranch *&p_branch_a,
                              ConnectedBranch *&p_branch_b) {
        auto new_branch = new ConnectedBranch;
        merge(p_branch_a->begin(), p_branch_a->end(), p_branch_b->begin(),
              p_branch_b->end(), inserter(*new_branch, new_branch->begin()));

        // Modify branches in new_branch to point to new_branch and free
        // p_branch_a and p_branch_b' memory. Assuming that the two branches
            are
        // in different branches.
        delete p_branch_a;
        delete p_branch_b;
        for (auto &ver : *new_branch) {
            branches[ver] = new_branch;
        }
    };

    for (Vertex i = 0; i <= v; i++) {
        auto new_branch = new ConnectedBranch;
        new_branch->insert(i);
        branches.push_back(new_branch);
    }

    // Inithalize MinHeap by pushing all edges in.
    for (auto &e : edges) {
        // Copy construct when push.
        heap.push(e);
    }
#ifdef DEBUG
    cout << "edges[" << edges.size() << "], heap[" << heap.size() << "]"
         << endl;

#endif // DEBUG

    // Count merge times.
    int merge_cnt = 0;

    // Pick edges to build Minimal Spanning Tree until all vertices exist in
    // MST.
    while ((merge_cnt < v - 1) && !(heap.empty())) {
        // Pick edge with minimal weight from MinHeap.
```

```
        auto e = heap.top();
        heap.pop();
        // cout << "Current edge: " << e << endl;

        /**
         * TODO: Considering the requirement of starting from source, skip the
         * edges with source vertex as one of its endpoints when source vertex
         * is already in the MST.
         *
         * WARNING: However, by doing so, the Kruskal algorithm can be proven
            to
         * be wrong.
         *
         * TODO: Prove that it is illegal to provide Kruskal a specific
            source.
         */
        /**
         * if (in_edge(source) && in_mst(source)) {
         * cout << "source[" << source << "] already in MST, "
         * << "skip edge: " << e << endl;
         * continue;
         * }
         */

        // If neither of its two endpoints is already in MST,
        // append this edge and its two endpoints to MST.
        if (!in_same_branch(e.a, e.b)) {
            vertices_in_mst.insert(e.a);
            vertices_in_mst.insert(e.b);
            // Collect edges in MST for debug purpose.
            edges_in_mst.insert(e);

            // Merge two endpoints' connected branches.
            merge_branches(branches[e.a], branches[e.b]);
            merge_cnt++;

            // Add current edge's weight to weight sum of MST.
            weight_sum += e.weight;
        }
    }

    // Return sum of weights of edges in MST.
    return weight_sum;
}

int main(int argc, char **argv) {
    CLIParser cliParser(argc, argv);
    cliParser.args["--in"] = "data/kruskal.in";
    cliParser.args["--out"] = "data/kruskal.out";
    cliParser.argParse();
    auto in_file = cliParser.args["--in"];
    auto out_file = cliParser.args["--out"];

    fstream fs;
    try {
        fs.open(cliParser.args["--in"], ios_base::in);
    } catch (exception &e) {
        cerr << e.what() << endl;
```

```
        cerr << "Failed to load data from " << cliParser.args["--in"]
             << ".\nExiting..." << endl;
        exit(-1);
    }

    cout << "Load data from: " << in_file << endl;
    cout << "Write data to: " << out_file << endl;

    // Read data from file with exception handler.
    int v, e;

    try {
        fs >> v >> e;
    } catch (exception &e) {
        cerr << "Error occurred during reading V and E:\n" << e.what();
        exit(-1);
    }

    // Initialize edges.
    vector<Edge> edges;
    // Considering that vertex index in sample starts from 1.
    for (int i = 0; i < e; i++) {
        Vertex a, b;
        Weight w;
        try {
            fs >> a >> b >> w;
        } catch (exception &e) {
            cerr << "Error occurred during read edge:\n" << e.what();
            exit(-1);
        }

        Edge edge = Edge(a, b, w);
        edges.emplace_back(edge);
    }

#ifdef DEBUG
    auto edges_to_string = [&](vector<Edge> edges) { // Print edges.
        auto in_edge = [&](Vertex v, Edge edge) {
            return ((edge.a == v) || (edge.b == v));
        };
        stringstream ss;

        // First, print edges in vertices' view.
        for (int i = 1; i <= v; i++) {
            ss << i << ":\n";
            for (auto &e : edges) {
                if (in_edge(i, e)) {
                    ss << "to[" << ((e.a == i) ? e.b : e.a) << "], weight["
                       << e.weight << "]\n";
                }
            }
        }
        cout << endl << "Edges:" << endl;

        // Second, print edges in edges' view.
        for (auto &e : edges) {
            ss << e << endl;
        }
```

```
        return ss.str();
    };

    cout << edges_to_string(edges) << endl;

#endif // DEBUG

    // End of data loading, close filestream.
    fs.close();

    // Initialize Kruskal.
    // Build Kruskal MST from vertex 1.
    Vertex source = 1;
    Kruskal kruskal(source, e, v, edges);

    // Initialize timer.
    auto begin = chrono::high_resolution_clock::now();

    // Kruskal build MST.
    auto distance = kruskal.kruskal_MST();

    auto end = chrono::high_resolution_clock::now();
    auto elapsed = chrono::duration_cast<chrono::nanoseconds>(end - begin);
    cout.precision(6);
    cout << "[Kruskal] Time measured: " << elapsed.count() * 1e-9
         << " seconds.\n";

#ifdef DEBUG
    // Print vertices in MST to debug.
    cout << kruskal.ver_in_mst_to_string() << endl;
    // Print edges in MST to debug.
    cout << kruskal.edges_in_mst_to_string();

#endif // DEBUG

    // Print weight sum of MST.
    cout << "Weight sum of MST:\n" << distance << endl;
    // Write weight sum of MST to out file.
    try {
        fs.open(out_file, ios_base::out);
    } catch (exception &e) {
        cerr << "Error occurred during writing result to file:\n" <<
            e.what();
    }
    fs << distance << endl;
    fs.close();

    return 0;
}
```

5.7.2 Rust 版本 Kruskal 算法

```
pub mod disjoint_set_union;
pub mod utils;

use disjoint_set_union::DisjointSetUnion;
```

```rust
use std::{
    cmp::Ordering,
    collections::BinaryHeap,
    fmt::Display,
    fs::File,
    io::{self, Read, Write as IoWrite},
    path::Path,
    time::Instant,
};

#[cfg(feature = "debug")]
use std::fmt::Write as fmtWrite;

#[allow(unused)]
use utils::cli_parser;

/**
 * Macros written for debug purpose.
 *
 * If debug output is needed, add --features debug flag in rustc compilation.
 */
#[cfg(feature = "debug")]
macro_rules! debug {
    ($($args: expr), *) => {
        println!($($args), *);
    };
}

#[cfg(not(feature = "debug"))]
macro_rules! debug {
    ($($args: expr), *) => {};
}

type Weight = i32;
type Vertex = i32;

#[derive(PartialEq, Eq, Ord, Clone)]
struct Edge {
    a: Vertex,
    b: Vertex,
    weight: Weight,
}

// Implement PartialEq for MinHeap.
impl PartialOrd for Edge {
    // Reverse from greater first to lesser first.
    fn partial_cmp(&self, other: &Self) -> Option<std::cmp::Ordering> {
        // First compare a and b's weight.
        let mut result = self
            .weight
            .partial_cmp(&other.weight)
            .map(|ord| ord.reverse());
        // If they have the same weight, compare their vertex a.
        if result == Some(Ordering::Equal) {
            result = self.a.partial_cmp(&other.a).map(|ord| ord.reverse());
            // If they have the same vertex a, compare vertex b.
            if result == Some(Ordering::Equal) {
                return self.b.partial_cmp(&other.b).map(|ord| ord.reverse());
```

```
            }

            return result;
        }

        result
    }
}

impl Edge {
    // Implement constructor for Edge.
    fn new(a: Vertex, b: Vertex, weight: Weight) -> Self {
        Self { a, b, weight }
    }
}

impl Display for Edge {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "({}, {}): {}", self.a, self.b, self.weight)
    }
}

impl Default for Edge {
    fn default() -> Self {
        Self {
            a: 0,
            b: 0,
            weight: 0,
        }
    }
}

struct Kruskal {
    v: i32,
    _e: i32,
    weight_sum: Weight,
    edges: Vec<Edge>,
    edges_in_mst: Vec<Edge>,
}

impl Kruskal {
    // Kruskal constructor
    pub fn new(v: i32, e: i32, edges: Vec<Edge>) -> Self {
        Self {
            v,
            _e: e,
            weight_sum: 0,
            edges,
            edges_in_mst: Vec::new(),
        }
    }

    pub fn kruskal_mst(&mut self) -> i32 {
        let mut heap: BinaryHeap<Edge> = BinaryHeap::new();

        // Push all edges into MinHeap.
        for e in &self.edges {
            heap.push(e.clone());
```

```
        }

        let mut connected_branches = DisjointSetUnion::new(self.v as usize);

        // Merge time counter.
        let mut merge_cnt = 0;
        while !heap.is_empty() && merge_cnt < self.v - 1 {
            let e = heap.pop().expect("Failed to pop from min heap.");

            // Two endpoints of e doesn't in the same connected branches.
            if connected_branches.merge(e.a, e.b) < Vertex::MAX {
                // Increase merge count, Add weight of edge to MST's weight
                    sum.
                merge_cnt += 1;
                self.weight_sum += e.weight;

                // Append newly inserted edge to edges_in_mst for Debug.
                self.edges_in_mst.push(e);
            }
        }

        self.weight_sum
    }

    #[cfg(feature = "debug")]
    pub fn edges_in_mst_to_string(&self) -> String {
        let mut s: String = String::new();
        for e in &self.edges_in_mst {
            write!(s, "({}, {}): {}\n", e.a, e.b, e.weight)
                .expect("Error writing edges in mst to string.");
        }

        s
    }
}

fn read_edges_from_file(filename: String) -> Option<(i32, i32, Vec<Edge>)> {
    let v: i32;
    let e: i32;
    let mut edges: Vec<Edge> = Vec::new();

    let path = Path::new(&filename);
    let display = path.display();
    println!("Load edges from file: {}", display);
    // Load lines of file to String.
    let mut file = File::open(filename).expect("Error opening file.");
    let mut contents = String::new();
    file.read_to_string(&mut contents)
        .expect("Error reading file.");
    let mut lines = contents.lines();

    let new_line = lines.next();
    // Read first line.
    let v_and_e: Vec<&str> = match new_line {
        Some(new_line) => new_line.split(" ").collect(),
        None => {
            println!("Error reading V and E.");
            Vec::new()
```

```
        }
    };
    v = v_and_e[0].parse::<i32>().expect("Error reading V.");
    e = v_and_e[1].parse::<i32>().expect("Error reading e.");

    // Read edges line by line.
    for line in lines {
        // Print line for debug.
        // println!("{}", line);
        let vec: Vec<&str> = line.split(" ").collect();

        // Stop reading lines when there's not enough values.
        if vec.len() < 3 {
            break;
        }

        let str2num =
            |x: &str| (x.parse::<i32>().expect("Error reading value."));
        let ver_a: i32 = str2num(vec[0]);
        let ver_b: i32 = str2num(vec[1]);
        let wgt: i32 = str2num(vec[2]);

        // Construct and push new edge to edges.
        let edge = Edge::new(ver_a, ver_b, wgt);
        edges.push(edge);

        // Print edges for debugging.
        // for edge in &edges {
        // println!("{}", edge);
        // }
    }

    Some((v, e, edges))
}

fn write_weight_sum_to_file(
    filename: String,
    kruskal: &Kruskal,
) -> io::Result<()> {
    let path = Path::new(&filename);
    let display = path.display();

    println!("Write to file: {}", display);

    // Write to file, then return write result.
    let mut file = File::create(&filename)?;

    write!(file, "{}\n", kruskal.weight_sum)
}

fn main() {
    let mut in_file: String = "data/kruskal.in".to_string();
    let mut out_file: String = "data/kruskal_rs.out".to_string();
    (in_file, out_file) = cli_parser(in_file, out_file);
    let (e, v, edges) = read_edges_from_file(in_file)
        .expect("Error occurred during unpacking e, v and edges from file.");

    // It's illegal to specify source vertex in Kruskal.
```

```
    #[allow(unused)]
    let source = 1;

    let mut kruskal = Kruskal::new(v, e, edges);
    // Measure the elapsed time of kruskal algorithm.
    let begin = Instant::now();
    let distance = kruskal.kruskal_mst();
    let duration = begin.elapsed().as_secs_f64();
    println!("[Kruskal RUST] Time measured: {:?} seconds.", duration);

    // Print the edges in MST.
    debug!("{} edges in Kruskal MST:", &kruskal.edges_in_mst.len());
    debug!("{}", kruskal.edges_in_mst_to_string());

    // Print the weight sum of Kruskal MST.
    println!("Weight sum of MST:\n{}", distance);

    // Write the weight sum to out file.
    write_weight_sum_to_file(out_file, &kruskal)
        .expect("Failed to write weight sum to file.");
}
```

5.8 数据生成器 data_gen

```
#include <fstream>
#include <iostream>
#include <random>
#include <sstream>
#include <stdexcept>
#include <string>
#include <unordered_set>
#include <vector>

using namespace std;
const int default_amount_of_random_numbers = 50;

class Parser {
  public:
    Parser(int argc, char **argv);
    int get_range_from() const { return range_from; }
    int get_range_to() const { return range_to; }
    int get_amount_of_random_numbers() const {
        return amount_of_random_numbers;
    }

  private:
    int parse_unit_arg(char *single_arg);
    void parse_arg_count_4();
    void parse_arg_count_3();
    void parse_no_arg();
    int argc;
    char **argv;
    int range_from;
    int range_to;
    int amount_of_random_numbers;
};
```

```
int Parser::parse_unit_arg(char *single_arg) {
    int x = 0;
    string arg = single_arg;
    try {
        size_t pos;
        x = stoi(arg, &pos);
        if (pos < arg.size()) {
            cerr << "Trailing characters after number: " << arg << "\n";
        }
    } catch (invalid_argument const &e) {
        cerr << "Invalid number: " << arg << "\n";
    } catch (out_of_range const &e) {
        cerr << "Number out of range: " << arg << "\n";
    }
    return x;
}

/**
 * Generate an amount of argv[3] random numbers distributed
 * from argv[1] to argv[2].
 */
void Parser::parse_arg_count_4() {
    range_from = parse_unit_arg(argv[1]);
    range_to = parse_unit_arg(argv[2]);
    amount_of_random_numbers = parse_unit_arg(argv[3]);
}

void Parser::parse_arg_count_3() {
    range_from = parse_unit_arg(argv[1]);
    range_to = parse_unit_arg(argv[2]);
}

Parser::Parser(int argc, char **argv) : argc(argc), argv(argv) {
    if (argc == 4) {
        parse_arg_count_4();
    } else if (argc == 3) {
        parse_arg_count_3();
        amount_of_random_numbers = default_amount_of_random_numbers;
    } else {
        range_from = 0;
        range_to = 1000;
        amount_of_random_numbers = default_amount_of_random_numbers;
    }
}

class Edge {
  public:
    int nodeA;
    int nodeB;

    Edge(int a, int b) : nodeA(a), nodeB(b){};
};

bool operator==(const Edge &a, const Edge &b) {
    return ((a.nodeA == b.nodeB) && (a.nodeB == b.nodeA)) ||
           ((a.nodeB == b.nodeA) && (a.nodeA == b.nodeB));
}
```

```cpp
class EdgeEq {
  public:
    bool operator()(const Edge &a, const Edge &b) const { return a == b; }
};

class EdgeHash {
  public:
    size_t operator()(const Edge &edge) const {
        return std::hash<int>()(edge.nodeA + edge.nodeB);
    }
};

int main(int argc, char **argv) {
    Parser parser(argc, argv);
    int range_from = parser.get_range_from();
    int range_to = parser.get_range_to();
    int amount_of_random_numbers = parser.get_amount_of_random_numbers();

    random_device rand_dev;
    mt19937 generator(rand_dev());

    // Distance random selector
    uniform_int_distribution<int> distr(range_from, range_to);

    // Node index random selector
    uniform_int_distribution<int> node_distr(1, amount_of_random_numbers);

    fstream fs;
    string filename = "data/kruskal_yasample.in";
    fs.open(filename, ios_base::out);

    unordered_set<Edge, EdgeHash, EdgeEq> edges;

    // Number of nodes
    int num_nodes = amount_of_random_numbers;

    if (fs.is_open()) {
        // Generate edges.
        for (int i = 1; i <= num_nodes; i++) {
            int node_a = i;

            // Generate new node that differs from node_a.
            auto new_node = [&](const int node_a) {
                auto new_node = node_distr(generator);
                while (node_a == new_node) {
                    new_node = node_distr(generator);
                }

                return new_node;
            };

            int node_b = new_node(node_a);

            Edge edge(node_a, node_b);
            // Edge already exists. Regenerate one.
            while (edges.find(edge) != edges.end()) {
                edge.nodeB = new_node(node_a);
            }
```

```
            edges.insert(edge);
        }

        int num_edges = edges.size();
        fs << num_nodes << " " << num_edges << endl;

        // Generate distance and write edges to file.
        for (auto &a : edges) {
            int dis = distr(generator);
            fs << a.nodeA << " " << a.nodeB << " " << dis << endl;
        }
    }
    fs.close();
    return 0;
}
```

6 附录

代码仓库: Greedy