

南 开 大 学

人 工 智 能 学 院

《算法设计与分析实验》课程大作业

专 业: 智能科学与技术姓 名: 石若川

学 号: 2111381

指导教师: 刘曜玮

课 号: 0422

完成日期: 2024 年 6 月 5 日

**目录**

1. [实验目的](#_bookmark0) 2
2. [实验内容](#_bookmark1) 2
3. [算法原理](#_bookmark3) 2
   1. [Kruskal 算法](#_bookmark4) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
   2. [Prim 算法](#_bookmark5) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
4. [代码实现](#_bookmark6) 3
5. [实验结果](#_bookmark7) 8
6. [实验总结](#_bookmark9) 8

# 实验目的

1. 了解并掌握 Kruskal 算法的基本原理和实现步骤。
2. 了解并掌握 Prim 算法的基本原理和实现步骤。
3. 通过编程实现 Kruskal 算法和 Prim 算法，比较它们的性能和适用场景。

# 实验内容

1. 实现 Kruskal 算法和 Prim 算法的 C++ 代码。
2. 利用 Kruskal 算法和 Prim 算法，对给定的无向图[1](#_bookmark2)，计算最小生成树。
3. 输出最小生成树的边及其对应的权重。

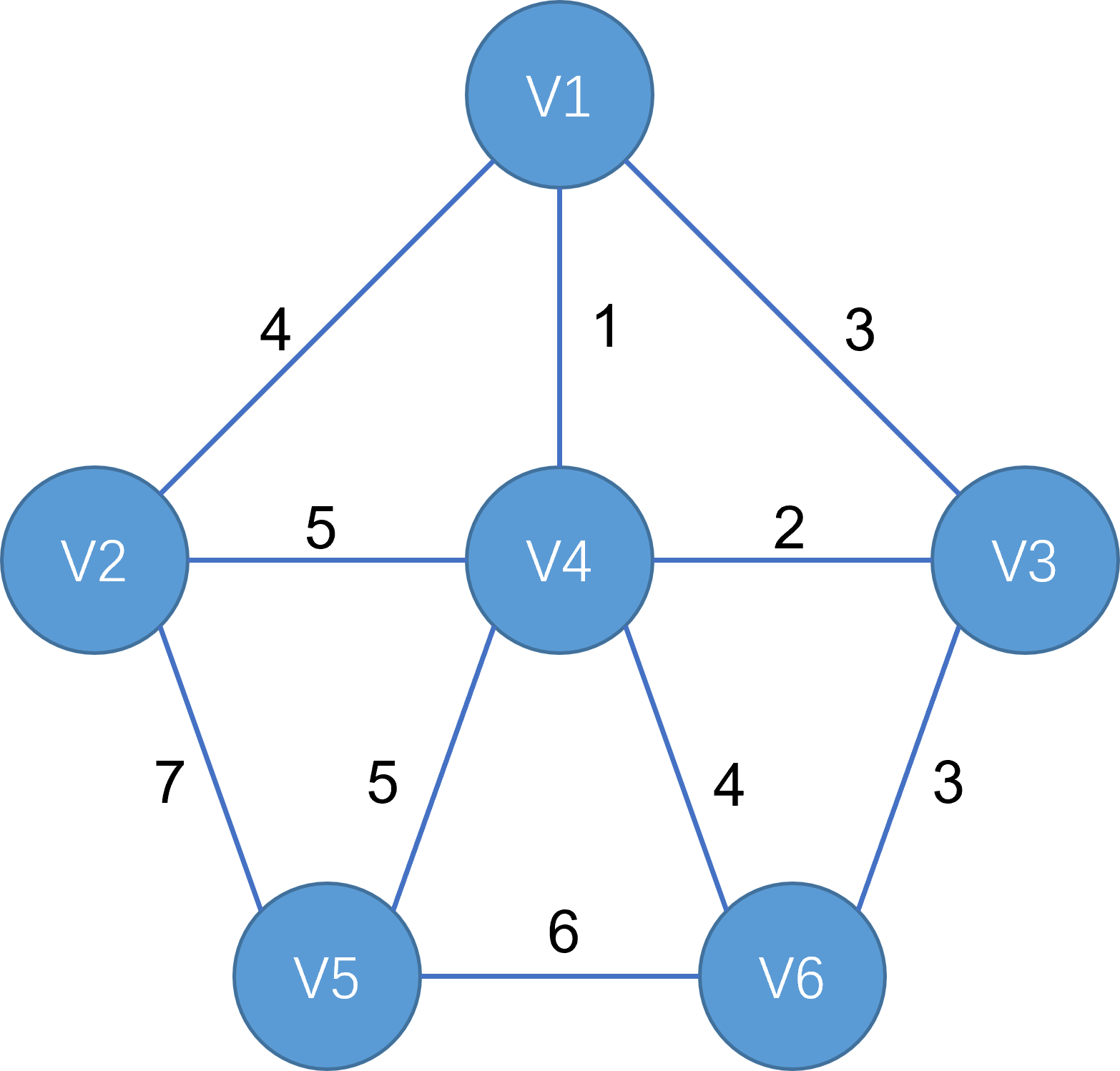


图 1: 实验使用的无向图

# 算法原理

## Kruskal 算法

Kruskal 算法是一种贪心算法，用于寻找加权连通无向图的最小生成树。其核心思想是：从小到大选择边，逐步构建生成树，确保不形成环。具体步骤如下：

* + 1. 将图中的所有边按权重从小到大排序。
    2. 初始化一个并查集用于检测环。
    3. 按权重顺序依次选择边，若加入该边不会形成环，则将其加入最小生成树中。
    4. 重复上述步骤直到生成树包含所有顶点。

Kruskal 算法的伪代码如下：

**Algorithm 1** Kruskal 算法

1: 将边集 *E* 按权重 *w*(*e*) 从小到大排序

2: 初始化一个空的最小生成树 *T* = ∅

3: 初始化并查集 *U* ，使每个顶点各自为一个独立的集合

4: **for** 每条边 *e* = (*u, v*) 按权重顺序 **do** 5: **if** *U.find*(*u*) ̸= *U.find*(*v*) **then** 6: *T* = *T* ∪ {*e*}

7: *U.union*(*u, v*)

8: **end if**

9: **end for**

10: **return** *T*

## Prim 算法

Prim 算法同样是一种贪心算法，用于寻找加权连通无向图的最小生成树。其核心思想是：从一个起始顶点开始，逐步扩展最小生成树，每次选择连接生成树的权重最小的边。具体步骤如下：

* + 1. 初始化最小生成树包含一个起始顶点，其余顶点标记为未加入。
    2. 从当前生成树中选择一条权重最小的边，将其连接的顶点加入生成树。
    3. 重复上述步骤直到生成树包含所有顶点。

Prim 算法的伪代码如下：

**Algorithm 2** Prim 算法

1: 选择任意起始顶点 *s*，初始化最小生成树 *T* = {*s*}

2: 初始化优先队列 *Q*，将 *s* 的所有边加入 *Q*

3: **while** *T* 中的顶点数 *<* |*V* | **do**

4: 从 *Q* 中取出权重最小的边 (*u, v*)

5: **if** *v* ∈/ *T* **then**

6: 将顶点 *v* 和边 (*u, v*) 加入 *T*

7: 将 *v* 的所有边加入 *Q*

8: **end if**

9: **end while**

10: **return** *T*

# 代码实现

实验中，Kruskal 算法的代码实现包括：

* 边类（Edge）：定义边的结构，包含起点 u、终点 v 和权重 weight。重载了小于操作符以便边可以按权重排序。
* 并查集类（DisjointSet）：实现并查集，用于检测图中是否形成环。包括 find 和 union 操作。
* kruskal 函数：先对边按权重排序，然后逐个检查每条边，如果添加这条边不会形成环，则将其加入最小生成树。

Prim 算法的代码实现包括：

* graph 表示方法：使用邻接表，graph[u] 存储了顶点 u 的所有邻接边，每条边用一个 pair<int, int> 表示，其中第一个元素是权重，第二个元素是终点顶点。
* key 数组：存储每个顶点加入最小生成树的最小权重。
* parent 数组：存储每个顶点在生成树中的父节点。
* inMST 数组：标记顶点是否已包含在最小生成树中。
* 优先队列 pq：用于选择当前权重最小的边，使用最小堆实现。
* prim 函数：从起始顶点开始，逐步扩展最小生成树，直到包含所有顶点。

在定义图的方式上两种方式有所不同，Kruskal 算法使用边集的方式，Prim 算法使用邻接表的方式。这是因为：

* Kruskal 算法：该算法的核心在于对边进行排序，并逐步选取边构建最小生成树。因此，边集是最自然的表示方法。边集包含了所有边的信息，包括两个顶点和边的权重，方便排序和边的选择。
* Prim 算法：该算法从一个起始顶点开始，逐步扩展到整个图，重点在于维护一个顶点集和其邻接边。使用邻接表表示图可以高效地找到与当前生成树相连的最小权重边。邻接表可以快速访问一个顶点的所有邻接边，适合 Prim 算法的逐步扩展过程。

实验中的具体代码如下：

#include <iostream> #include <vector> #include <algorithm> #include <queue> #include <climits>

using namespace std;

// 边类，用于Kruskal算法

class Edge { public:

int u, v, weight;

Edge(int u, int v, int weight) : u(u), v(v), weight(weight) {} bool operator<(const Edge& other) const {

return weight < other.weight;

}

};

// 并查集类，用于Kruskal算法 class DisjointSet { private:

vector<int> parent, rank;

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23 public:

24 DisjointSet(int n) : parent(n), rank(n, 0) {

25 for (int i = 0; i < n; ++i)

26 parent[i] = i;

27 }

28 int find(int u) {

29 if (u != parent[u])

30 parent[u] = find(parent[u]);

31 return parent[u];

32 }

33 void unite(int u, int v) {

34 int rootU = find(u);

35 int rootV = find(v);

36 if (rootU != rootV) {

37 if (rank[rootU] > rank[rootV])

38 parent[rootV] = rootU;

39 else if (rank[rootU] < rank[rootV])

40 parent[rootU] = rootV;

41 else {

42 parent[rootV] = rootU;

43 rank[rootU]++;

44 }

45 }

46 }

47 };

48

49 // Kruskal算法函数

50 vector<Edge> kruskal(int n, vector<Edge>& edges) {

51 sort(edges.begin(), edges.end()); // 按权重排序所有边

52 DisjointSet ds(n);

53 vector<Edge> mst; // 最小生成树

54 for (size\_t i = 0; i < edges.size(); ++i) {

55 if (ds.find(edges[i].u) != ds.find(edges[i].v)) {

56 ds.unite(edges[i].u, edges[i].v); // 合并两个顶点的集合

57 mst.push\_back(edges[i]);

58 }

59 }

60 return mst;

61 }

62

63 // Prim算法函数

64 typedef pair<int, int> PII; // 一对（权重，顶点）

65 vector<Edge> prim(int n, vector<vector<PII> >& graph) {

66 vector<int> key(n, INT\_MAX); // 存储最小权重值

67 vector<int> parent(n, -1); // 存储构造树的父节点

68 vector<bool> inMST(n, false); // 标记顶点是否在最小生成树中

69 priority\_queue<PII, vector<PII>, greater<PII> > pq; // 最小堆优先队列

70

71 int start = 0; // 从顶点0开始

72 pq.push(make\_pair(0, start));

73 key[start] = 0;

74

75 while (!pq.empty()) {

76 int u = pq.top().second;

77 pq.pop();

78 inMST[u] = true;

79

80 // 更新相邻顶点的键值

81 for (size\_t i = 0; i < graph[u].size(); ++i) {

82 int weight = graph[u][i].first;

83 int v = graph[u][i].second;

84

|  |  |  |  |
| --- | --- | --- | --- |
| 85 | if | (!inMST[v] && weight < key[v]) { | |
| 86 |  | key[v] = weight; | |
| 87 |  | pq.push(make\_pair(key[v], v)); | |
| 88 |  | parent[v] = u; | |
| 89 | } |  | |
| 90 | } | |  |
| 91 | } | |  |
| 92 |  | |  |
| 93 | vector<Edge> mst; // 最小生成树 | |  |
| 94 | for (int v = 1; v < n; ++v) { | |  |
| 95 | if (parent[v] != -1) { | |  |
| 96 | mst.push\_back(Edge(parent[v], | | v, key[v])); |
| 97 | } | |  |
| 98 | } | |  |
| 99  100 } | return mst; | |  |
| 101 |  | |  |
| 102 int  103 | main() {  int n = 6; // 图的顶点数 | |  |
| 104 |  | |  |
| 105 | // Kruskal算法的边定义 | |  |
| 106 | vector<Edge> edges; | |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 107 | edges.push\_back(Edge(0, | 1, | 4)); | | |
| 108 | edges.push\_back(Edge(0, | 2, | 3)); | | |
| 109 | edges.push\_back(Edge(0, | 3, | 1)); | | |
| 110 | edges.push\_back(Edge(1, | 3, | 5)); | | |
| 111 | edges.push\_back(Edge(1, | 4, | 7)); | | |
| 112 | edges.push\_back(Edge(2, | 3, | 2)); | | |
| 113 | edges.push\_back(Edge(2, | 5, | 3)); | | |
| 114 | edges.push\_back(Edge(3, | 4, | 5)); | | |
| 115 | edges.push\_back(Edge(3, | 5, | 4)); | | |
| 116 | edges.push\_back(Edge(4, | 5, | 6)); | | |
| 117 |  |  |  | | |
| 118 | // Prim算法的图定义 | | |  | |
| 119 | vector<vector<PII> > graph(n); | | |  | |
| 120 | graph[0].push\_back(make\_pair(4, | | | 1)); | |
| 121 | graph[0].push\_back(make\_pair(3, | | | 2)); | |
| 122 | graph[0].push\_back(make\_pair(1, | | | 3)); | |
| 123 | graph[1].push\_back(make\_pair(4, | | | 0)); | |
| 124 | graph[1].push\_back(make\_pair(5, | | | 3)); | |
| 125 | graph[1].push\_back(make\_pair(7, | | | 4)); | |
| 126 | graph[2].push\_back(make\_pair(3, | | | 0)); | |
| 127 | graph[2].push\_back(make\_pair(2, | | | 3)); | |
| 128 | graph[2].push\_back(make\_pair(3, | | | 5)); | |
| 129 | graph[3].push\_back(make\_pair(1, | | | 0)); | |
| 130 | graph[3].push\_back(make\_pair(5, | | | 1)); | |
| 131 | graph[3].push\_back(make\_pair(2, | | | 2)); | |
| 132 | graph[3].push\_back(make\_pair(5, | | | 4)); | |
| 133 | graph[3].push\_back(make\_pair(4, | | | 5)); | |
| 134 | graph[4].push\_back(make\_pair(7, | | | 1)); | |
| 135 | graph[4].push\_back(make\_pair(5, | | | 3)); | |
| 136 | graph[4].push\_back(make\_pair(6, | | | 5)); | |
| 137 | graph[5].push\_back(make\_pair(3, | | | 2)); | |
| 138 | graph[5].push\_back(make\_pair(4, | | | 3)); | |
| 139 | graph[5].push\_back(make\_pair(6, | | | 4)); | |
| 140 |  | | | |  |
| 141 | // Kruskal算法 | | | |  |
| 142 | vector<Edge> mst\_kruskal = kruskal(n, | | | | edges); |
| 143  144  145 | cout << "最小生成树中的边 (Kruskal):" << endl; for (size\_t i = 0; i < mst\_kruskal.size(); ++i)  cout << "(" << mst\_kruskal[i].u << ", " << mst\_kruskal[i].v << ") -> " <<  mst\_kruskal[i].weight << endl; | | | | |
| 146 |  | | | | |
| 147 | // Prim算法 | | | | |

148

vector<Edge> mst\_prim = prim(n, graph); cout << "最小生成树中的边 (Prim):" << endl;

for (size\_t i = 0; i < mst\_prim.size(); ++i)

cout << "(" << mst\_prim[i].u << ", " << mst\_prim[i].v << ") -> " << mst\_prim[i].weight << endl;

return 0;

}

149

150

151

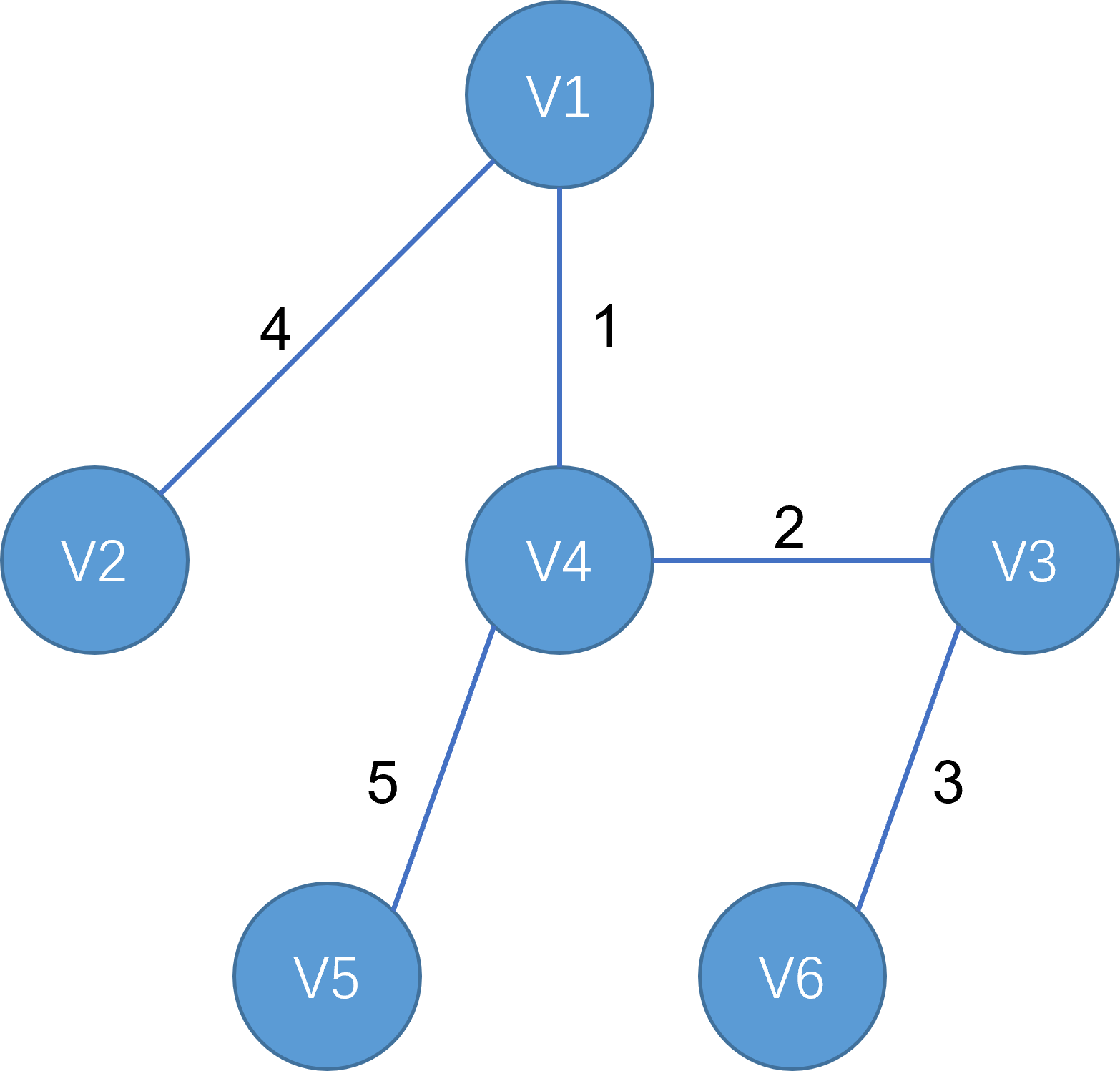
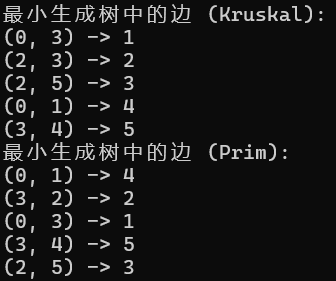
152

153

154

# 实验结果

代码运行结果如图[2](#_bookmark8)所示，两种算法得到的最小生成树一致，但是生成的顺序不一样。



(a) 输出结果 (b) 最小生成树

图 2: 实验结果

# 实验总结

经过实验对比，总结 Kruskal 和 Prim 的特点：

* Kruskal 算法：算法按边的权值排序，依次选择，所以与点的数量无关，适用于点多边少的稀疏图。时间复杂度为 *O*(*mlogn*)。
* Prim 算法：每次都要遍历一遍所有点，找出与已确定区间相连的边，无论两点之间有无边，我们都需要每轮遍历一次。所以 Prim 算法的时间复杂度是 *O*(*n*2)，适用于点少边多的稠密图。当使用优先队列时，时间复杂度为 *O*(*mlogn*)。

通过本次实验，掌握了 Kruskal 算法和 Prim 算法的基本原理和实现方法，并能在实际应用中根据图的特点选择合适的算法求解最小生成树。