作业 4

xwzeng

2023 年 5 月 27 日

1

按如下操作顺序构建一棵伸展树：

插入 3，插入 2，插入 1，查询 4，插入 10，插入 6，删除 3，插入 7，查询 2，插入 13，删除 6，插入 5。

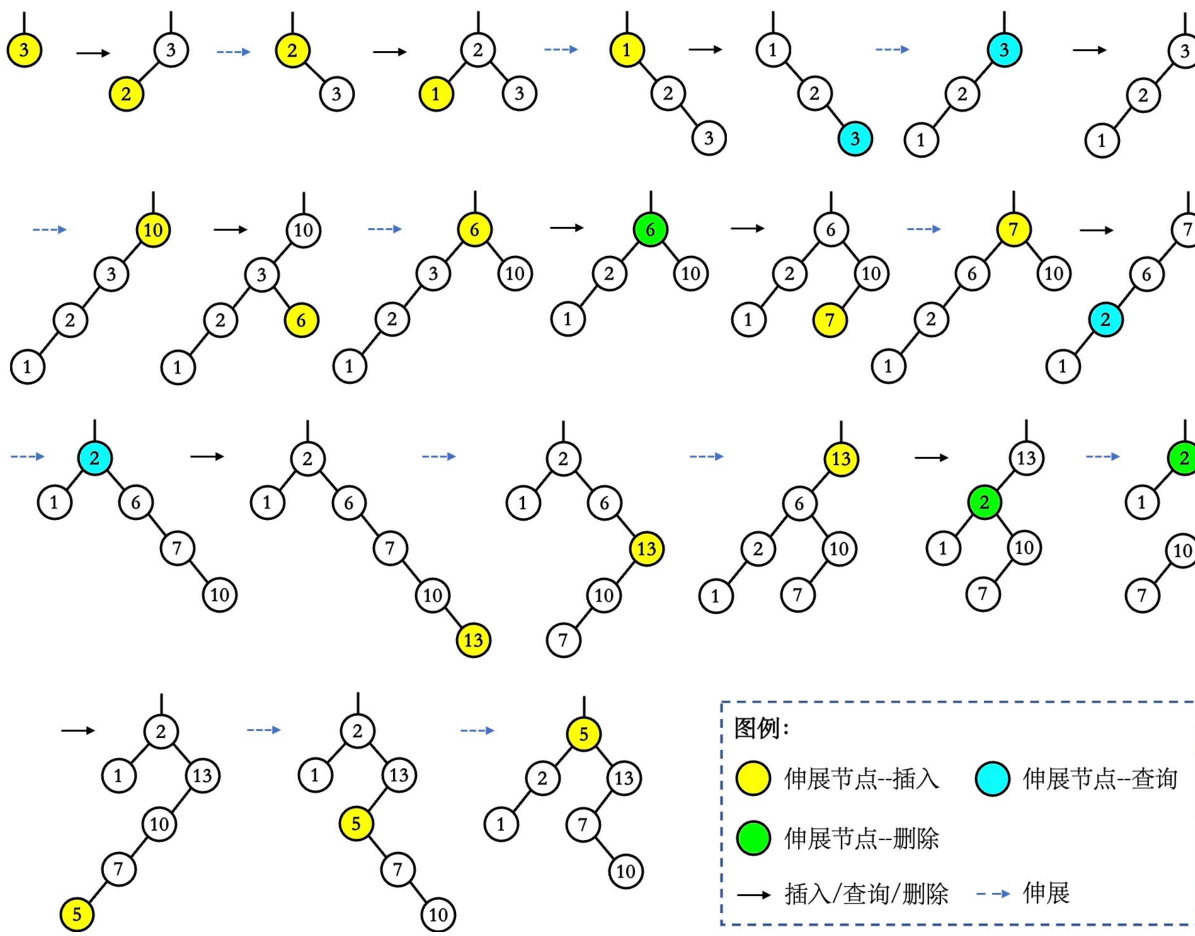

图 1: 伸展树构建示意图

2

按如下操作顺序构建一棵红黑树：

插入 3，插入 2，插入 1，插入 10，插入 6，删除 3，插入 7，插入 4，删除 6，插入 5。

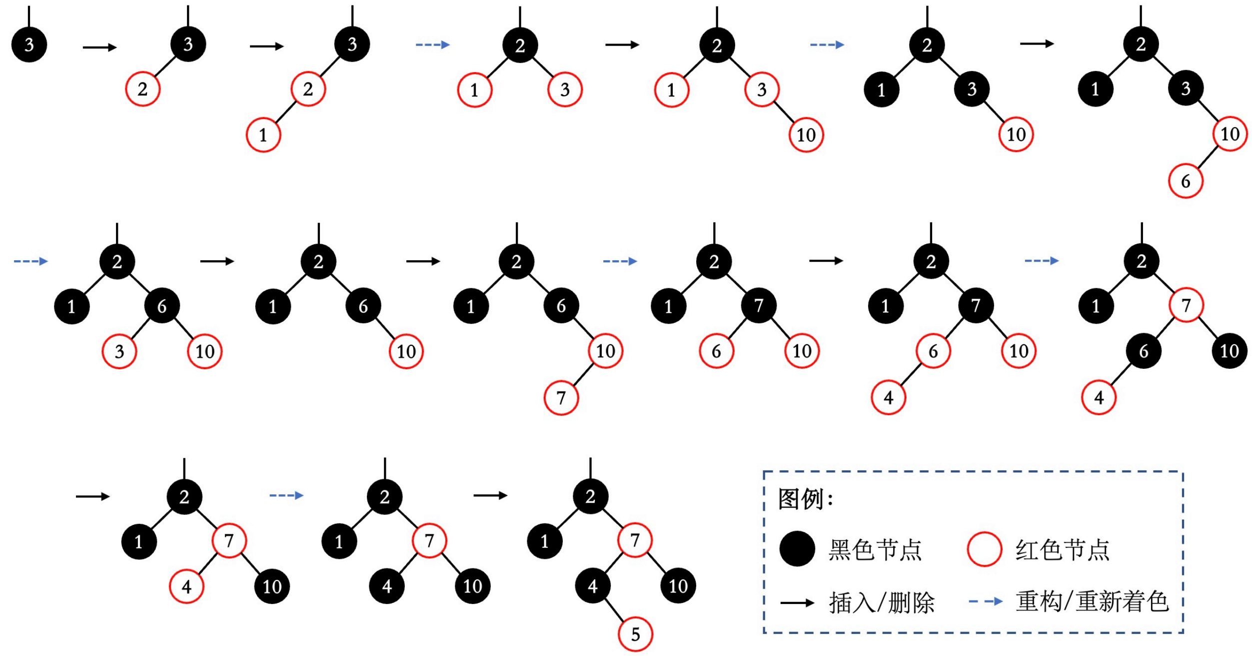

图 2: 红黑树构建示意图

3

使用归并排序，对序列 3, 1, 4, 7, 5, 2, 6, 8 进行排序。

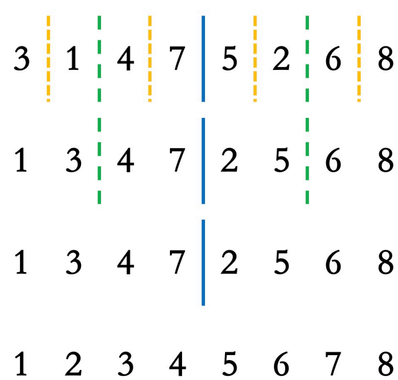

图 3: 归并排序示意图

4

a)

三位取中法实现就地快速排序：

1. 选取数组头部、正中间（头部索引与尾部索引的平均值，去尾得到）、尾部这 3 个数进行排序。
2. 若数组的长度为 2 或 3，排序终止；否则继续进行下列操作。
3. 将正中间的数与数组的倒数第二个数交换，以保证右边扫描过的数都大于等于基准值。
4. 令 right marker 为 $b-2$，扫描步骤与标准就地快速排序相同。
5. 扫描完成后，将 left marker 与基准值交换，对基准值的前后两部分数组递归调用就地快速排序函数。

[1]:
```python
def my_inplace_quick_sort(S, a, b):
    if a >= b:
        return
    # 三位取中法
    mid = int(a + (b - a) / 2)
    left = a
    right = b
    if S[left] < S[mid]:
        if S[mid] < S[right]:
            pass
        else:  # S[mid] >= S[right]
            if S[left] > S[right]:
                S[left], S[mid], S[right] = S[right], S[left], S[mid]
            else:  # S[left] <= S[right]
                S[mid], S[right] = S[right], S[mid]
    else:  # S[left] >= S[mid]
        if S[mid] > S[right]:
            S[left], S[right] = S[right], S[left]
        else:  # S[mid] <= S[right]
            if S[left] > S[right]:
                S[left], S[mid], S[right] = S[mid], S[right], S[left]
            else:  # S[left] <= S[right]:
                S[left], S[mid] = S[mid], S[left]
```

```
    # if length of the list is 2 or 3, already sorted
    if b - a + 1 <= 3:
        return

    # if length of the list larger than 3
    pivot = S[mid]
    # swap the pivot with penultimate place
    S[mid], S[right - 1] = S[right - 1], S[mid]
    right = b - 2

    while left <= right:
        # scan until reaching value equal or larger than pivot (or right marker)
        while left <= right and S[left] < pivot:
            left += 1
        # scan until reaching value equal or smaller than pivot (or left marker)
        while left <= right and pivot < S[right]:
            right -= 1
        if left <= right:
            S[left], S[right] = S[right], S[left]
            left, right = left + 1, right - 1

    # put pivot into its final place (currently marked by left index)
    S[left], S[b - 1] = S[b - 1], S[left]

    # make recursive calls
    my_inplace_quick_sort(S, a, left - 1)
    my_inplace_quick_sort(S, left + 1, b)
```

[2]:
```
s = [3, 1, 4, 7, 5, 2, 6, 8]
my_inplace_quick_sort(s, 0, 7)
print(s)
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

b)

使用该排序算法，对序列 3, 1, 4, 7, 5, 2, 6, 8 进行排序。

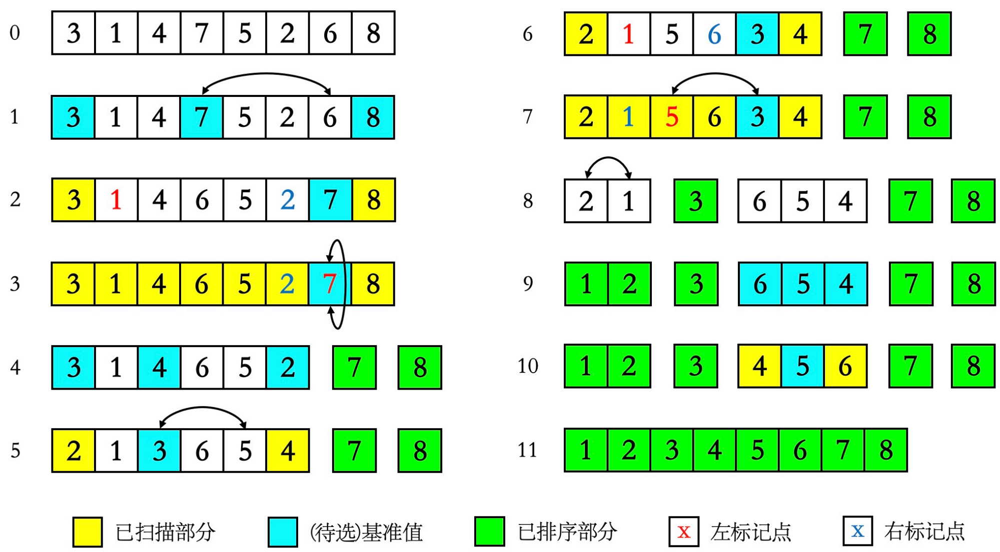

图 4: 三位取中法就地快速排序示意图

5

对于一个有 V 个顶点和 E 条边的简单无向图:

1. 连通分支下界为:

$$\begin{cases} 1, & E \geq V-1 \\ V-E, & E < V-1 \end{cases} \tag{1}$$

证明: 连通图的边数一定满足 $E \geq V-1$, 故:

- 当 $E \geq V-1$ 时, 该图本身就可能为连通图, 连通分支下界为 1。
- 当 $E < V-1$ 时, 该图一定不为连通图, 有多个连通分支。

  假设该图共有 k 个连通分支, 每个连通分支有 V_i 个顶点, E_i 条边, 且 $\sum_{i=1}^{k} V_i = V$, $\sum_{i=1}^{k} E_i = E$。由连通图的性质, 有:

$$V_i - 1 \leq E_i \tag{2}$$

  累加后得到:

$$\sum_{i}^{k}(V_i - 1) \leq \sum_{i}^{k} E_i \tag{3}$$

  即:

$$k \geq V - E \tag{4}$$

  故连通分支下界为 $V-E$。此时图结构为 E 个 2 顶点的连通分支和 $(V-2E)$ 个单顶点的连通分支。

2. 连通分支上界为:

$$1 + V - \left\lceil \frac{1+\sqrt{1+8E}}{2} \right\rceil \tag{5}$$

证明：要让连通分支最多，就要让单顶点的连通分支最多，即让边与尽可能少的顶点组合。

在顶点数目相同的图中，可组合边数最多的图是连通图 (证明见 [1])，因此我们应该让所有边与部分顶点组合成为一个连通分支，然后让剩余的顶点作为单顶点的连通分支。

假设含有全部边的大连通分支顶点数量为 $\tilde{V}$，由简单无向图的性质，有:

$$E \le \frac{\tilde{V}(\tilde{V}-1)}{2}, \quad \tilde{V} > 0 \tag{6}$$

求解不等式，得到:

$$\min \tilde{V} = \left\lceil \frac{1+\sqrt{1+8E}}{2} \right\rceil \tag{7}$$

其中 $\lceil x \rceil$ 表示对 x 向上取整。故连通分支最多有 1 个大连通分支 + 剩余顶点数 $(V - \min \tilde{V})$ 个单顶点连通分支，结论得证。

[1] 假设图有 V 个顶点，k 个连通分支，每个连通分支有 V_i 个顶点，$\sum_{i=1}^{k} V_i = V$，则有:

$$\max E = \begin{cases} \sum_{i=1}^{k} V_i(V_i-1)/2, & k \ge 2 \\ V(V-1)/2, & k = 1 \end{cases} \tag{8}$$

又有:

$$\sum_{i=1}^{k} V_i(V_i-1) = \left(\sum_{i=1}^{k} V_i^2 - V\right) \le (V^2 - V) = V(V-1) \tag{9}$$

故当 $k=1$，即图为连通图时，V 个顶点可组合的边数最多。

6

补全 Graph 类，实现 remove_vertex 方法和 remove_edge 方法。每删除一条边后，为了防止这条被删除的边被意外找到，我将该边的起点和终点都修改为 None，以此说明这条边已经被删除。

[]:
```
def remove_vertex(self, v):
    """Remove the Vertex v."""
    self._validate_vertex(v)
    # 删除所有与顶点 v 相连的边
    edges = self._outgoing.pop(v, None)
    if edges:
        for e in edges.values():
            e._origin = None
            e._destination = None
```

```
    for _, v_dict in self._outgoing.items():
        e = v_dict.pop(v, None)
        if e:
            e._origin = None
            e._destination = None
    edges = self._incoming.pop(v, None)
    if edges:
        for e in edges.values():
            e._origin = None
            e._destination = None
    for _, u_dict in self._incoming.items():
        e = u_dict.pop(v, None)
        if e:
            e._origin = None
            e._destination = None

def remove_edge(self, e):
    """Remove the Edge e."""
    u, v = e.endpoints()
    if (u is not None) & (v is not None):
        del self._outgoing[u][v]
        del self._incoming[v][u]
        e._origin = None
        e._destination = None
    else: # 该边已被删除
        print('This edge has already been removed.')
```

[3]:
```
%run week10_6.py
```

```
[无向图] 节点数: 6 边数: 7
C F B A E D | C<->D | D<->E | E<->F | D<->F | B<->C | A<->C | A<->B |

[无向图] 删除的节点为: C , 剩余边数: 4
F B A E D | A<->B | E<->F | D<->E | D<->F |

[无向图] 删除的边为: A<->B, 剩余边数: 3
F B A E D | E<->F | D<->E | D<->F |
```

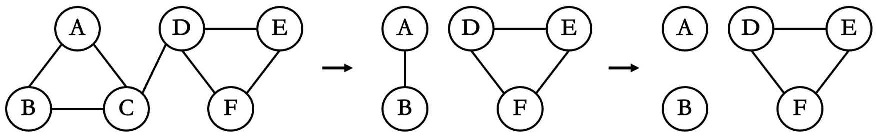

图 5: 无向图示意

```
[有向图] 节点数: 6 边数: 7
C F B A E D | C->D | B->C | A->B | A->C | E->F | D->E | D->F |

[有向图] 删除的节点为: F , 剩余边数: 5
C B A E D | C->D | B->C | A->B | A->C | D->E |

[有向图] 删除的边为: A->C, 剩余边数: 4
C B A E D | C->D | B->C | A->B | D->E |
```

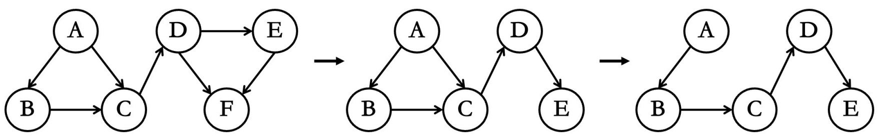

图 6: 有向图示意

附加题

a) Boruvka 算法实现

[ ]:
```
def MST_Boruvka(g):
    """Compute a minimum spanning tree of weighted graph g with Boruvka.
    Return a list of edges that comprise the MST."""
    C = [{v} for v in g.vertices()]
    V = set(g.vertices())
    tree = []
    n = g.vertex_count()
    while len(tree) < n - 1:
        for idx1, Ci in enumerate(C):
            min_weight = float('inf')
            min_edge = None
            for u in Ci:
                for v in (V - Ci):
                    if g.get_edge(u, v) is not None:
```

```
                        if g.get_edge(u, v).element() < min_weight:
                            min_weight = g.get_edge(u, v).element()
                            min_edge = g.get_edge(u, v)
                            min_opposite = v
            if min_edge:
                tree.append(min_edge)
                for idx2, other in enumerate(C):
                    if min_opposite in other:
                        C[idx1] |= other
                        C[idx2] = {}
                        break
        while {} in C:
            C.remove({})
    return tree
```

测试代码。

[4]:
```
%run week10_ 附加题.py
```

```
| B<->C | E<->F | A<->B | D<->F | A<->E |
```

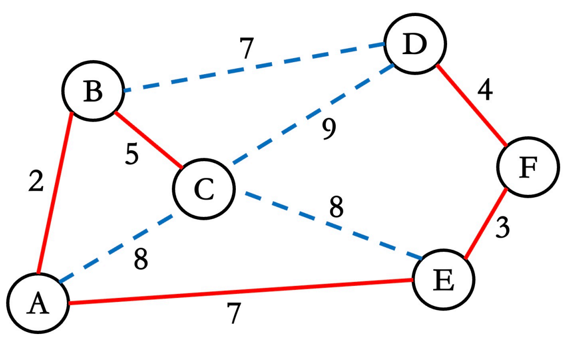

图 7: 最小生成树示意图

b) 正确性证明

Boruvka 算法的核心就是使用最小权重边不断地两两合并 components，直到全部合并。算法最初有 n 个 component（每个顶点作为 1 个 component），每进行一次两两合并，component 的总个数减少 1，获得 1 条边。将 components 全部合并需要进行 $n-1$ 次合并，相应地得到 $n-1$ 条边，因此该算法的输出一定有且仅有 $n-1$ 条边。

在合并 components 时，使用的边一定是与被合并的两部分相连的，故这 $n-1$ 条边与原图 n 个顶

点构成的图 G 一定是连通图。由连通图的性质，当连通图的边数比顶点数少 1 时，该图为一棵树，因此图 G 是树。

每个 component 内部连通，我们可以把 component 和除 component 以外的顶点看作图的分割，由最小生成树的分割性质，连接这两部分的权重最小的边一定存在于图的一棵最小生成树中，因此图 G 就是原图的一棵最小生成树，Boruvka 算法的正确性得证。

c) 性能分析

在外层循环中，每进行 1 轮合并，components 的个数减半，剩余 1 个 component 时停止循环，故最多循环 $O(\log n)$ 次；在内层循环中，每次需要遍历 component 中顶点与其他顶点相连的所有边，时间复杂度为 $O(m)$。因此 Boruvka 算法的时间复杂度为 $O(m \log n)$。