数据结构与算法

Data Structures and Algorithms

谢昊

xiehao@cuz.edu.cn

非线性结构
Non-Linear Structures

大纲

1. 基本术语
2. 图的存储结构
3. 图的遍历
    - 深度优先搜索
    - 广度优先搜索
4. 最小生成树
5. 小结

引例：Königsberg 七桥问题

- 在 Königsberg 市有 7 座桥连通了 4 块区域
- 是否有算法实现
  - 从某处出发
  - 依次穿过所有桥仅 1 次
  - 回到原地

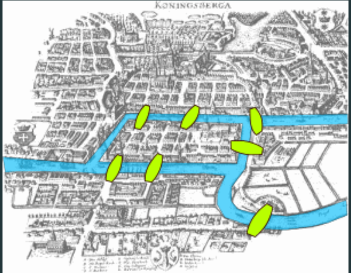


图 1: Königsberg 七桥问题

引例：Königsberg 七桥问题

- 在 Königsberg 市有 7 座桥连通了 4 块区域
- 是否有算法实现
  - 从某处出发
  - 依次穿过所有桥仅 1 次
  - 回到原地
- 可抽象为图的一笔画问题

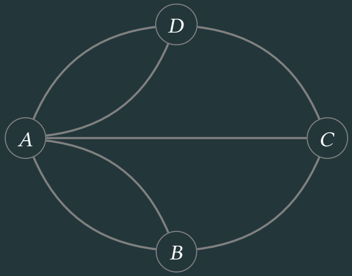


图 1: Königsberg 七桥问题

非线性结构

- 在半线性结构的基础上允许有环的存在
- 半线性结构的一种扩展
- 非线性结构主要指**图**结构
- 图与树之间可转换

基本术语

基本术语

图 (Graph)

- 可被定义为 $G = (V, E)$，其中[1]
    - 集合 V 中元素 v 为**顶点 (vertex)**
    - 集合 E 中元素 $e \in V \times V$ 为**边 (edge)**
- 只定义拓扑关系，与几何位置无关

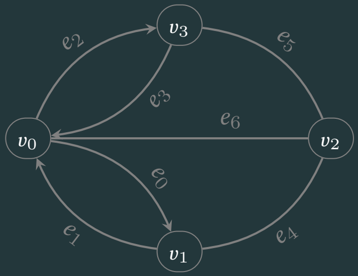
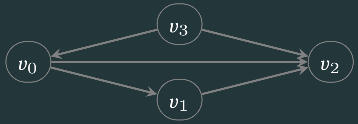


图 2: 图

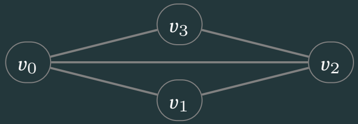
[1] 顶点又名**结点 (node)**，边又名**弧 (arc)**，且 V 与 E 均为有限集

无向图 (Undigraph) 与有向图 (Digraph)

- 按顶点是否有顺序可将边 e 分为
  - 无顺序的**无向边**，记作 (u,v)
  - 有顺序的**有向边**，记作 $\langle u,v\rangle$
- 只有无向边的图为**无向图**
- 只有有向边的图为**有向图**
- 二者均有的图为**混合图 (mixed graph)**
- 无向图与混合图均可转化为有向图
  - 将一条无向边拆成两条相反的有向边



(a) 有向图



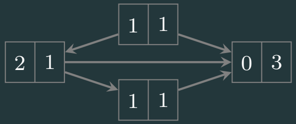
(b) 无向图

图 3: 有向图与无向图

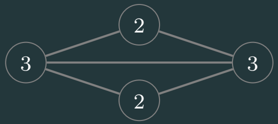
基本术语

顶点的度 (Degree)

- 若边 $e = \langle v_a, v_b \rangle$，则
  - 称 v_a 与 v_b **邻接 (adjacent)**
  - 称二者均与 e 彼此**关联 (incident)**
  - 称 e 为 v_a 的**出边 (outgoing edge)**
  - 称 e 为 v_b 的**入边 (incoming edge)**
- 在无向图中称与顶点 v 关联的边数为 v 的**度**
- 在有向图中称与顶点 v 关联的出入边数分别为 v 的**出度 (out-degree)** 与**入度 (in-degree)**



(a) 有向图的出度（左）与入度（右）

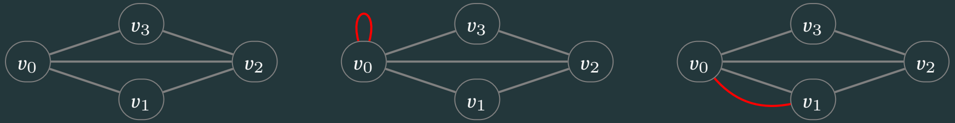


(b) 无向图的度

图 4: 顶点的度

简单图 (Simple Graph)

- 称起点与终点相同的边为**自环 (self-loop)**
- 称不含自环且所有边均唯一的图为**简单图**[2]



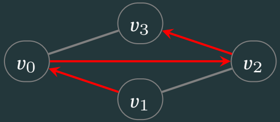
(a) 简单图　**(b)** 带自环的非简单图　**(c)** 有重复边的非简单图

图 5: 简单图与非简单图

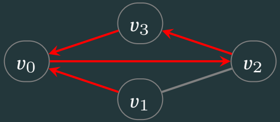
[2]本课程只讨论简单图

路径 (Path)

- 若序列 $\pi = \{v_k\}_{k=0}^{n}$ 满足 v_k 与 v_{k+1} 邻接[3]，则称之为自 v_0 至 v_n 的一条**路径**
  - 称经过的总边数 $|\pi| - 1$ 为路径**长度 (length)**
  - 称无重复顶点的路径为**简单 (simple) 路径**
- 两顶点间的路径一般不唯一



(a) 简单路径：(v_1, v_0, v_2, v_3)



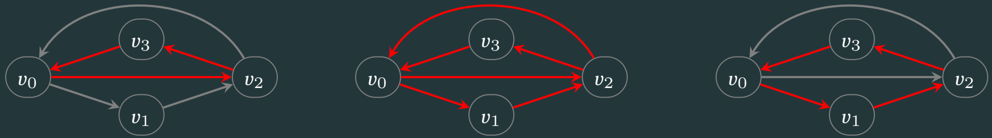
(b) 非简单路径：$(v_1, v_0, v_2, v_3, v_0)$

图 6: 图的路径

[3] 指存在边 e_k 满足 $e_k = (v_k, v_{k+1})$ 或 $e_k = \langle v_k, v_{k+1} \rangle$， $k \in \mathbb{Z} \cap [0, n)$

环路 (Cycle)

- 若路径 $\pi = \{v_k\}_{k=0}^{n}$ 中起止顶点相同，即 $v_0 = v_n$，则称其为**环路**
  - 若除起止顶点相同外无任何其他顶点两两相同，则称其为 **简单环路**
  - 称经过图中各边一次且仅一次的环路为**欧拉环路 (Eulerian tour)**
  - 称经过图中各顶点一次且仅一次的环路为 **哈米尔顿环路 (Hamiltonian tour)**

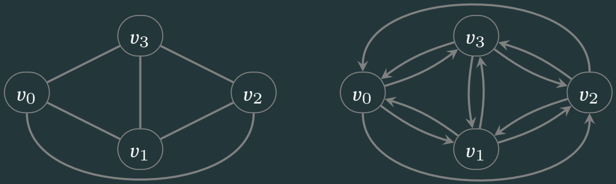


(a) 简单环路：(v_0, v_2, v_3) **(b)** 欧拉环路：$(v_0, v_1, v_2, v_0, v_2, v_3, v_0)$ **(c)** 哈米尔顿环路：$(v_0, v_1, v_2, v_3, v_0)$

图 7: 图的环路

完全图 (Complete Graph)

- 图中任意两顶点均邻接
- 若顶点数为 n 则无向与有向边数分别为 $\frac{n(n-1)}{2}$ 与 $n(n-1)$

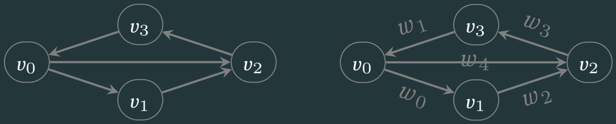


(a) 完全无向图 **(b)** 完全有向图

图 8: 完全图

带权图 (Weighted Graph)

- 为每条边指定权重，又名**带权网络 (network)**
- 用于表示顶点关系的细节，如长度、流量、成本等
- 普通图可看作所有边权重均为 1 的带权图



(a) 普通图　**(b)** 带权图

图 9: 普通图与带权图

图的存储结构

图的抽象数据类型

ADT Graph {

数据:

　　数据对象: $\mathcal{D} = \{v_k | v_k \in \text{顶点集合}, k \in \mathbb{Z} \cap [1, n]\}$

　　逻辑关系: $\mathcal{R} = \{\langle v_x, v_y \rangle | \exists e_x \in \text{边集合}, s.t.\ e_x = \langle v_x, v_y \rangle\}$

操作:

　　create_graph(), destroy_graph(g)

　　　　构造与销毁一个图g

　　get_vertex(g, v), get_first_neighbor(g, v), get_next_neighbor(g, v, w)

　　　　返回顶点v的信息，获取顶点v的第一个邻接顶点与相对于w的下一个邻接顶点

　　insert_vertex(g, v), remove_vertex(g, v)

　　　　插入与删除顶点v

　　insert_edge(g, va, vb), remove_edge(g, va, vb)

　　　　在顶点v_a与v_b间插入与删除一条边

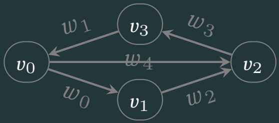
　　depth_first_search(g, x), breadth_first_search(g, x)

　　　　对图g进行深度与广度优先搜索值x

}

邻接矩阵 (Adjacency Matrix)

- 图抽象数据类型的一种基本实现
- 用稠密方阵表示，其元素描述一对顶点间可能的邻接关系
    - 若有边相连，则元素为该边权重；否则可为 ∞ 或 0



(a) 带权图

$$
\begin{array}{c c}
 & \begin{array}{cccc} v_0 & v_1 & v_2 & v_3 \end{array} \\
\begin{array}{c} v_0 \\ v_1 \\ v_2 \\ v_3 \end{array} &
\begin{pmatrix}
\infty & w_0 & w_4 & \infty \\
\infty & \infty & w_2 & \infty \\
\infty & \infty & \infty & w_3 \\
w_1 & \infty & \infty & \infty
\end{pmatrix}
\end{array}
$$

(b) 带权图的邻接矩阵

图 10: 邻接矩阵

图的存储结构

邻接矩阵特点

- 无向图邻接矩阵必对称，故可只存储上三角部分
- 有向图邻接矩阵第 k行/列非零元素个数为对应顶点的 出/入度
- 增删边只需修改矩阵特定元素值
- 增删顶点需调整矩阵维度，导致大量元素移动

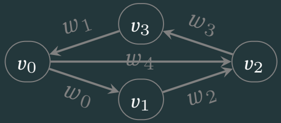
复杂度

- 空间复杂度：$O(n^2)$，不利于表达稀疏图 (sparse graph)[4]
- 查找特定边的时间复杂度：$O(1)$

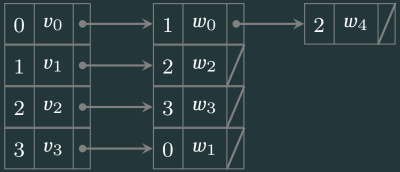
[4]指边数远小于完全图边数的图

邻接表 (Adjacency List)

- 每个顶点以链表存储其邻接顶点集合
  - 链表结点存储信息包括：顶点序号、边权重、下一个邻接顶点
- 相当于只存储邻接矩阵中的有效元素



(a) 带权图



(b) 带权图的邻接表

图 11: 邻接矩阵

邻接表特点

- 只存储邻接顶点信息，可大幅节省空间
- 增删边或顶点只需修改极少量数据

复杂度

- 空间复杂度：$O(n+e)$[5]
- 查找特定边的时间复杂度：$O(n)$

[5] n 与 e 分别为顶点数与边数

图的存储结构

基本结构实现

- 顶点结构

```c
typedef struct {
    DataType data; // 数据
    int index; // 序号
    Vector *neighbors; // 邻边
    bool visited; // 被访问状态
} Vertex;
```

- 边结构

```c
typedef struct {
    Vertex *from; // 起始顶点
    Vertex *to; // 目的顶点
    unsigned weight; // 权重
    bool visited; // 被访问状态
} Edge;
```

- 邻接矩阵表示

```c
typedef struct {
    Vertex *vertices; // 顶点列表（用数组）
    unsigned **weights; // 权重矩阵
    int size; // 顶点个数
} AdjacencyMatrix;
```

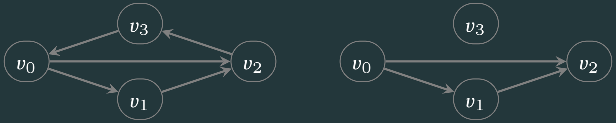
- 邻接表表示

```c
typedef struct {
    Vector *vertices; // 顶点列表（用向量）
    Vector *edges; // 边列表（用向量）
} AdjacencyList;
```

图的遍历

图的连通性

- 图中任意两顶点间均有路径相连[6]，则称该图为**连通图 (connected graph)**
- 若图 $G=(V,E)$ 与 $G'=(V',E')$ 满足 $V'\subseteq V$ 且 $E'\subseteq E$，则称 G' 为 G 的**子图 (subgraph)**
- 称非连通图的极大连通子图为**连通域 (connected component)**



(a) 连通图　　**(b)** 非连通图

图 12: 连通图与非连通图

[6]满足单向相连即可，亦称为**可达 (reachable)**；约定只有单个顶点的图为连通图

遍历 (Traversal)[7]

- 按某种约定顺序访问非线性结构中的所有顶点与边
- 每个顶点与边均被且仅被访问 1 次
- 意义：使非线性结构转化为半线性结构
- 遍历的产物[8]
  - 称对连通图遍历生成的树为**生成树 (spanning tree)**
  - 称对非连通图遍历生成的森林为**生成森林 (spanning forest)**

---

[7] 在图中亦称**搜索 (search)**

[8] 此处生成亦可替换为**支撑 (support)** 或**遍历**

图的遍历

几个关键问题

- 遍历过程从何处出发?
- 如何避免遗漏顶点或边?
- 如何避免重复访问顶点?
- 如何选择下一个顶点?

图的遍历

几个关键问题

- 遍历过程从何处出发?
  - 原则上约定从编号最小的顶点出发
- 如何避免遗漏顶点或边?
- 如何避免重复访问顶点?
- 如何选择下一个顶点?

几个关键问题

- 遍历过程从何处出发?
  - 原则上约定从编号最小的顶点出发
- 如何避免遗漏顶点或边?
  - 多发生于非连通图中，可对不同连通子图分别执行
- 如何避免重复访问顶点?

- 如何选择下一个顶点?

图的遍历

几个关键问题

- 遍历过程从何处出发?
  - 原则上约定从编号最小的顶点出发
- 如何避免遗漏顶点或边?
  - 多发生于非连通图中，可对不同连通子图分别执行
- 如何避免重复访问顶点?
  - 在每个顶点上设置被访问状态，并在访问后更新
- 如何选择下一个顶点?

几个关键问题

- 遍历过程从何处出发?
  - 原则上约定从编号最小的顶点出发
- 如何避免遗漏顶点或边?
  - 多发生于非连通图中，可对不同连通子图分别执行
- 如何避免重复访问顶点?
  - 在每个顶点上设置被访问状态，并在访问后更新
- 如何选择下一个顶点?
  - 常见思路：深度优先搜索与广度优先搜索

深度优先搜索

深度优先搜索 (Depth-First Search, DFS)

- 基本原则：不撞南墙不回头
- 类似树的先序遍历或后序遍历
- 结果不唯一，与遍历邻接顶点的顺序有关

算法步骤

1. 标记所有顶点为未被访问状态
2. 按序号依次取出未被访问顶点并执行
   a. 访问该顶点并将其标记为已被访问
   b. 对其所有未被访问的邻接顶点递归执行深度优先搜索
   c. 返回步骤 2
3. 结束

深度优先搜索

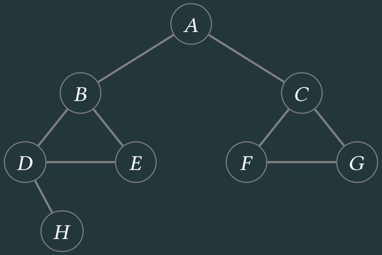


图 13: 图的深度优先遍历过程：A, B, D, E, H, C, F, G

深度优先搜索

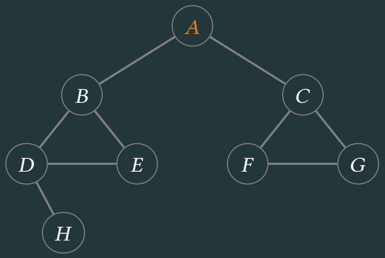


图 13: 图的深度优先遍历过程：A, B, D, E, H, C, F, G

深度优先搜索

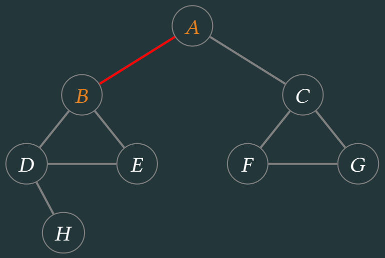


图 13: 图的深度优先遍历过程： A, B, D, E, H, C, F, G

深度优先搜索

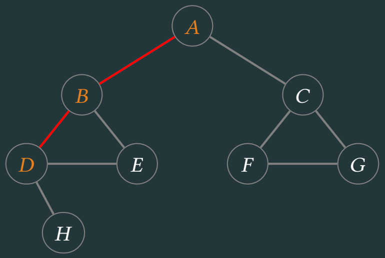


图 13: 图的深度优先遍历过程：A, B, D, E, H, C, F, G

深度优先搜索

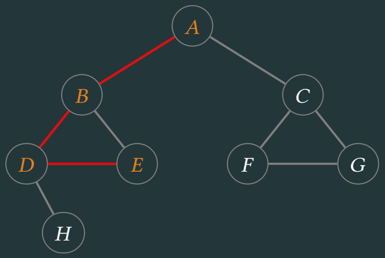


图 13: 图的深度优先遍历过程：A, B, D, E, H, C, F, G

深度优先搜索

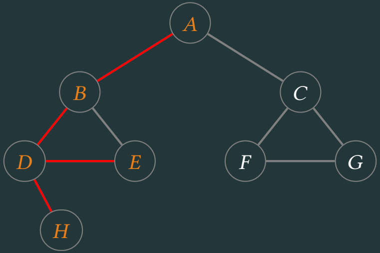


图 13: 图的深度优先遍历过程：A, B, D, E, H, C, F, G

深度优先搜索

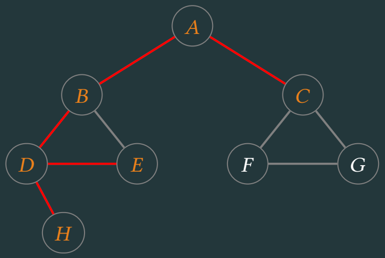


图 13: 图的深度优先遍历过程：A, B, D, E, H, C, F, G

深度优先搜索

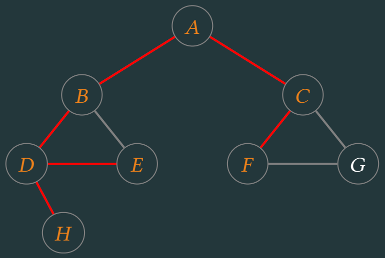


图 13: 图的深度优先遍历过程：A, B, D, E, H, C, F, G

深度优先搜索

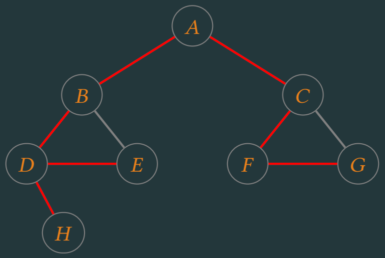


图 13: 图的深度优先遍历过程：A, B, D, E, H, C, F, G

练习：从 A 开始的深度优先搜索可能结果为

I. A, B, E, C, D, F

II. A, C, F, E, B, D

III. A, E, B, C, F, D

IV. A, E, D, F, C, B

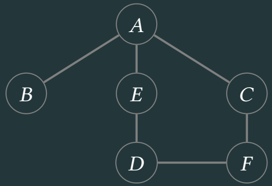


图 14: 图的深度优先遍历测试

练习：从 A 开始的深度优先搜索可能结果为

I. A, B, E, C, D, F
II. A, C, F, E, B, D
III. A, E, B, C, F, D
IV. A, E, D, F, C, B

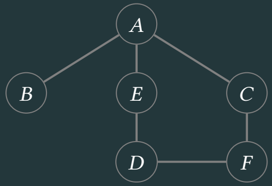


图 14: 图的深度优先遍历测试

深度优先搜索

深度优先搜索的一种邻接矩阵结构递归实现

- 多次调用以便找出可能的多个连通域

```
void depth_first_search_matrix(
        AdjacencyMatrix *m, VisitType v) {
    assert(m);
    clear_matrix_states(m); // 清除顶点状态
    for (int i = 0; i < m->size; ++i) {
        if (!vertex_at(m, i)->visited) {
            depth_first_search_matrix_core(
                    m, i, v);
        }
    }
}
```

- 对特定顶点递归执行深度优先搜索

```
static void depth_first_search_matrix_core(
        AdjacencyMatrix *m, int k, VisitType v) {
    Vertex *vertex = vertex_at(m, k);
    v(vertex->data); // 访问顶点数据
    vertex->visited = true; // 标记已被访问
    for (int i = 0; i < m->size; ++i) {
        if (is_adjacent(m, k, i) &&
            !vertex_at(m, i)->visited) {
            depth_first_search_matrix_core(
                    m, i, v);
        }
    }
}
```

深度优先搜索的一种邻接表结构递归实现

- 多次调用以便找出可能的多个连通域

```c
void depth_first_search_list(
        AdjacencyList *l, VisitType v) {
    assert(l);
    clear_list_states(l); // 清除顶点状态
    int n_vertices =
        size_of_vector(l->vertices);
    for (int i = 0; i < n_vertices; ++i) {
        Vertex *x =
            get_vertex_at(l->vertices, i);
        if (x && !x->visited) {
            depth_first_search_list_core(
                    l, x, v);
        }
    }
}
```

- 对特定顶点递归执行深度优先搜索

```c
static void depth_first_search_list_core(
        AdjacencyList *l, Vertex *x, VisitType v) {
    v(x->data); // 访问顶点数据
    x->visited = true; // 标记已被访问
    int n_neighbors =
        size_of_vector(x->neighbors);
    for (int i = 0; i < n_neighbors; ++i) {
        Vertex *to =
            get_edge_at(x->neighbors, i)->to;
        if (!to->visited) {
            depth_first_search_list_core(
                    l, to, v);
        }
    }
}
```

广度优先搜索 (Breadth-First Search, BFS)

- 基本原则：远交近攻
- 类似树的层次遍历
- 结果不唯一，与遍历邻接顶点的顺序有关

算法步骤

1. 标记所有顶点为未被访问状态
2. 按序号依次取出未被访问顶点并执行
    a. 初始化顶点队列
    b. 访问该顶点、将其标记为已被访问并入队
    c. 若队列不空，则执行
        i. 将队首顶点出队
        ii. 对该顶点的所有未被访问邻接顶点执行：访问、标记为已被访问并入队
    d. 销毁队列并返回步骤 2
3. 结束

广度优先搜索

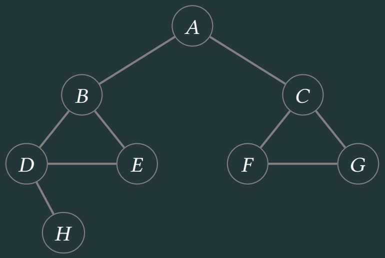


图 15: 图的一种广度优先遍历过程：A, B, C, D, E, F, G, H

广度优先搜索

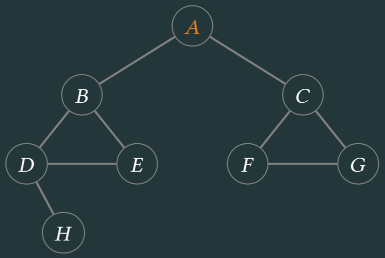


图 15: 图的一种广度优先遍历过程：A, B, C, D, E, F, G, H

广度优先搜索

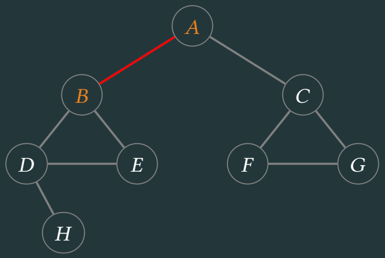


图 15: 图的一种广度优先遍历过程：A, B, C, D, E, F, G, H

广度优先搜索

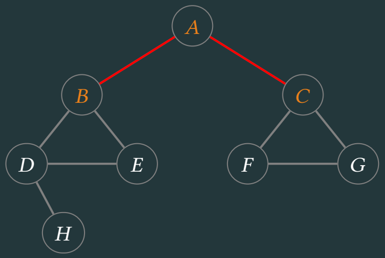


图 15: 图的一种广度优先遍历过程：A, B, C, D, E, F, G, H

广度优先搜索

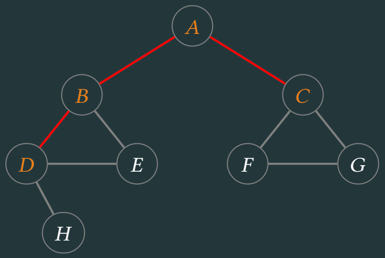


图 15: 图的一种广度优先遍历过程：A, B, C, D, E, F, G, H

广度优先搜索

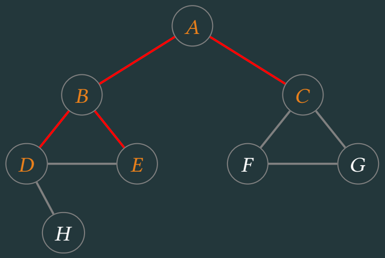


图 15: 图的一种广度优先遍历过程：A, B, C, D, E, F, G, H

广度优先搜索

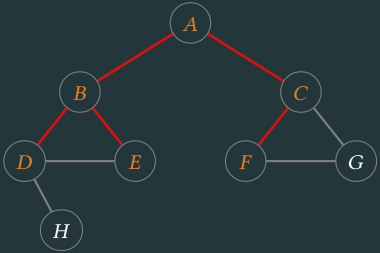


图 15: 图的一种广度优先遍历过程：A, B, C, D, E, F, G, H

广度优先搜索

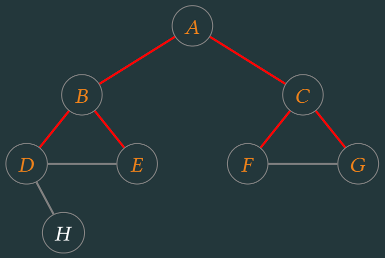


图 15: 图的一种广度优先遍历过程：A, B, C, D, E, F, G, H

广度优先搜索

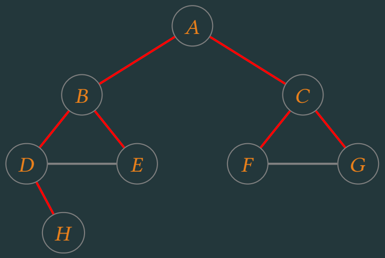


图 15: 图的一种广度优先遍历过程：A, B, C, D, E, F, G, H

广度优先搜索的一种邻接矩阵结构非递归实现

- 多次调用以便找出可能的多个连通域

```
void breadth_first_search_matrix(AdjacencyMatrix *m, VisitType v) {
    assert(m);
    clear_matrix_states(m);
    for (int i = 0; i < m->size; ++i) {
        if (!vertex_at(m, i)->visited) {
            breadth_first_search_matrix_core(m, i, v);
        }
    }
}
```

广度优先搜索的一种邻接矩阵结构非递归实现

- 对特定顶点执行广度优先搜索

```c
static void breadth_first_search_matrix_core(AdjacencyMatrix *m, int k, VisitType v) {
    LinkedQueue *buffer = create_linked_queue(); // 准备辅助队列
    if (!buffer) { return; }
    Vertex *from = vertex_at(m, k); // 取出当前顶点
    visit_vertex_and_push_queue(v, buffer, from); // 访问之并入队
    while (!empty_linked_queue(buffer)) { // 若队列非空，则循环执行
        pop_linked_queue(buffer, (DataType *)&from); // 出队
        for (int i = 0; i < m->size; ++i) { // 依次处理出队顶点的未被访问邻接顶点
            Vertex *to = vertex_at(m, i);
            if (is_adjacent(m, from->index, i) && !to->visited) {
                visit_vertex_and_push_queue(v, buffer, to); // 访问之并入队
            }
        }
    }
    destroy_linked_queue(buffer); // 销毁队列
}
```

广度优先搜索

广度优先搜索的一种邻接表结构非递归实现

- 多次调用以便找出可能的多个连通域

```c
void breadth_first_search_list(AdjacencyList *l, VisitType v) {
    assert(l);
    clear_list_states(l);
    int n_vertices = size_of_vector(l->vertices);
    for (int i = 0; i < n_vertices; ++i) {
        Vertex *x = get_vertex_at(l->vertices, i);
        if (!x->visited) {
            breadth_first_search_list_core(l, x, v);
        }
    }
}
```

广度优先搜索的一种邻接表结构非递归实现

- 对特定顶点执行广度优先搜索

```c
static void breadth_first_search_list_core(AdjacencyList *l, Vertex *x, VisitType v) {
    LinkedQueue *buffer = create_linked_queue();
    if (!buffer) { return; }
    visit_vertex_and_push_queue(v, buffer, x);
    while (!empty_linked_queue(buffer)) {
        pop_linked_queue(buffer, (DataType *)&x);
        int n_neighbors = size_of_vector(x->neighbors);
        for (int i = 0; i < n_neighbors; ++i) {
            Vertex *to = get_edge_at(x->neighbors, i)->to;
            if (!to->visited) {
                visit_vertex_and_push_queue(v, buffer, to);
            }
        }
    }
    destroy_linked_queue(buffer);
}
```

图的遍历

表 1: 深度优先搜索与广度优先搜索的比较

	深度优先搜索	**广度优先搜索**
其余邻接顶点	后遍历	先遍历
辅助结构	栈	队列
典型用途	识别连通域	求无权图最短路径

最小生成树

引例：网线铺设问题

- 假设 6 个城市之间路线连接关系如图
- 图中边的权值可表示路线的距离与代价
- 如何铺设网线连接所有城市且代价最低

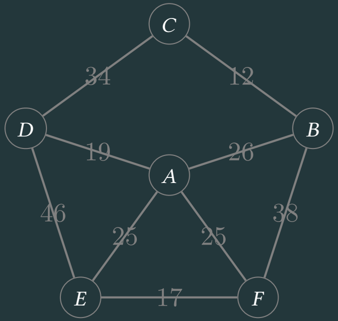


图 16: 引例：网线铺设问题

引例：网线铺设问题

- 假设 6 个城市之间路线连接关系如图
- 图中边的权值可表示路线的距离与代价
- 如何铺设网线连接所有城市且代价最低
- 可转化为连通图的最小生成树问题

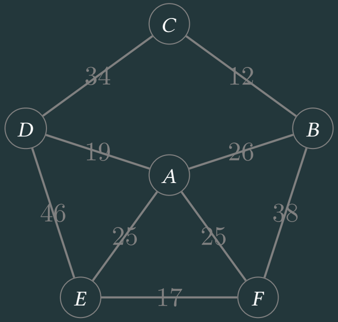


图 16: 引例：网线铺设问题

最小生成树 (Minimum Spanning Tree, MST)

- 在连通图 G 的所有生成树中，称边权重之和最小的为**最小生成树**

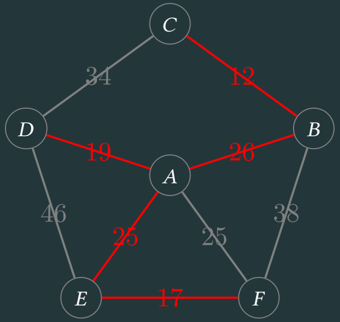


图 17: 最小生成树示例

连通图的分割与桥梁[9]

- 已知图 $G=(V,E)$ 与其子图 $G_a=(V_a,E_a)$ 与 $G_b=(V_b,E_b)$
- 若 G_a 与 G_b 满足

$$V=V_a\cup V_b,\quad \emptyset=V_a\cap V_b,$$

则称 G_a 与 G_b 构成图 G 的一个**分割 (cut)**，记作 $(G_a:G_b)$

- 若 G_a 与 G_b 构成图 G 的一个分割，且

$$e\in E-(E_a\cup E_b),$$

则称 e 为 G_a 与 G_b 间的一个**桥梁 (bridge)**

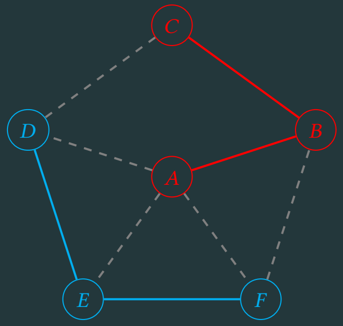


图 18: 图的分割与桥梁

[9] 以下图与子图均指连通图

Prim 算法[10]核心思想

- 不断将原图分割为逐渐膨胀与逐渐收缩的两个子图
- 在每次分割时始终选择权重最小的桥梁作为生成树的边

算法步骤

1. 初始化
   - 在图 G 中任选一顶点 v_0 作为起始点
   - 对 G 做分割 (G_a, G_b)，使得 $G_a = \{v_0\}$ 且 $G_b = G - G_a$
   - 令生成树边的集合 $E_{mst} = \emptyset$
2. 将权重最小的桥梁 e 加入 E_{mst}
3. 若 $G_b = \emptyset$ 则结束；否则在 G_b 中去除 e 的端点并将其加入 G_a 中，执行 2

[10]作者：R. C. Prim，1956 年

最小生成树

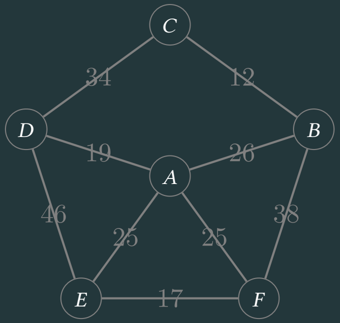


图 19: Prim 算法示例过程

最小生成树

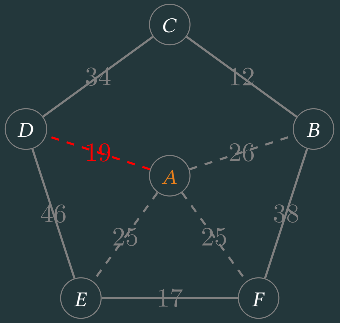


图 19: Prim 算法示例过程

最小生成树

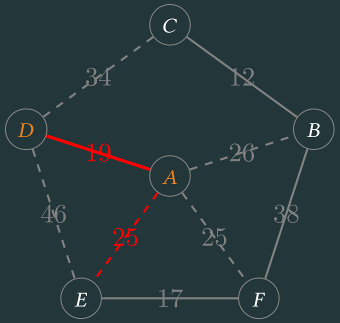


图 19: Prim 算法示例过程

最小生成树

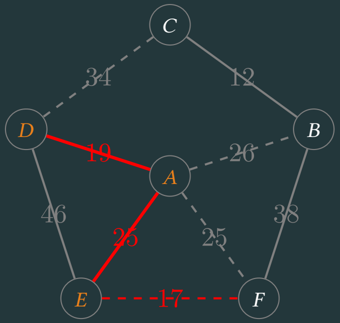


图 19: Prim 算法示例过程

最小生成树

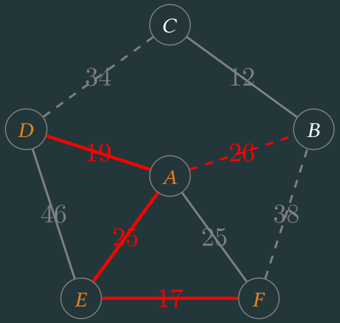


图 19: Prim 算法示例过程

最小生成树

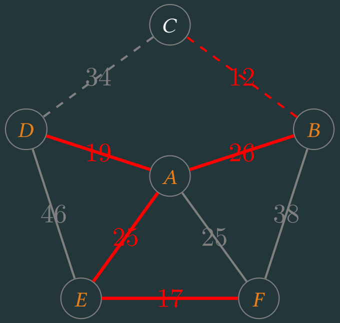


图 19: Prim 算法示例过程

最小生成树

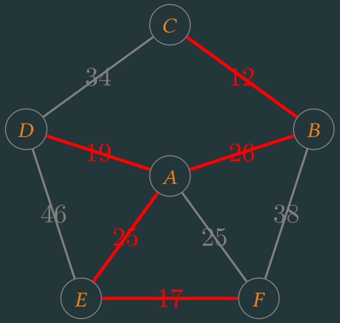


图 19: Prim 算法示例过程

未完待续 . . .

小结

小结

•

问与答