MINIMUM-COST SPANNING TREES

Algoritmo de Prim

Constrói uma Árvore Geradora Mínima (MST) através de uma sequência de subárvores em expansão.

1. Início: Começa com um único vértice arbitrário, formando uma subárvore inicial.
2. Expansão Gananciosa: A cada iteração, escolhe a aresta de menor peso que conecta um vértice dentro da árvore a um vértice fora dela. Esse vértice fora da árvore é adicionado à árvore.
3. Repetição: O processo é repetido até que todos os vértices do grafo estejam incluídos na árvore geradora mínima.

Eficiência:

- Matriz de Adjacência Sem Heap: $\Theta(|V|^2)$. Usando uma matriz de adjacência, o algoritmo verifica todas as arestas em cada iteração, virando em uma complexidade quadrática.
- Lista de Adjacência com Heap: $\Theta((|V| + |E|) \log |V|)$. Usando uma lista de adjacência e uma fila de prioridade (heap), o algoritmo melhora a eficiência ao reduzir o tempo de busca da aresta de menor peso.

- Vértice Mais Próximo: à cada iteração, a aresta escolhida deve ser a de menor peso entre aquelas que conectam a árvore a um vértice externo. Isso é garantido ao manter uma fila de prioridade (ou heap) que ajuda a encontrar rapidamente a aresta de menor peso.
- Semelhança com Dijkstra: Ambos os algoritmos começam com um único vértice e expandem de forma gananciosa. No entanto, Dijkstra é usado para encontrar o caminho mais curto de um ponto a outro, enquanto Prim é usado para encontrar a MST.
- Grafos com Pesos Negativos: Prim pode lidar com pesos negativos nas arestas, pois se concentra na expansão da árvore de maneira gananciosa, independentemente dos pesos. No entanto, os ciclos negativos não afetam a construção da MST.

Algoritmo de Kruskal

O algoritmo de Kruskal é um método gananciosо para encontrar a MST de um grafo, eficaz em grafos esparsos (grafos com relativamente poucas arestas em comparação com o número de vértices).

- Ele começa com cada vértice como uma árvore separada e une árvores usando as arestas de menor peso sem formar ciclos.(A seleção da aresta de menor peso é feita ordenando todas as arestas do grafo inicialmente.)
- Utiliza a estrutura de dados Union-Find (ou Disjoint Set), isso permite verificar rapidamente se dois vértices estão na mesma árvore (subconjunto) e uni-los se não estiverem. A mesma gerencia os subconjuntos e garantir a eficiência das operações de união e busca.

Disjoint Set

1. find(DS ds, int x): Encontra o representante (ou líder) do conjunto ao qual o elemento x pertence.
2. union(DS ds, int x, int y): Une os conjuntos contendo x e y.
3. Implementação: quick-find and quick-union

Quick-Find
Array: cada índice representa um elemento e o valor armazenado é o representante do conjunto e cada conjunto é implementado como uma linked list.

Operações:

- create disjointSubset(n): cria n linked lists, cada uma com um elemento representativo.
- find: $\Theta(1)$ – Acesso direto ao representante.
- union: $\Theta(n)$ – Concatenar listas e atualizar representantes.

Otimização(union by size): anexar lista mais curta para a maior.

Quick-Union
Usa uma árvore com ponteiros para os pais, onde a raiz é o representante do conjunto.

Operações:

- create disjointSubset(n): Inicializa nnn árvores unitárias, cada elemento é seu próprio representante.
- find: $\Theta(n)$ – Percorre do elemento até a raiz da árvore.
- union: $\Theta(1)$ – A raiz de xaponta para a raiz de y.

Otimização: Union by Size/Rank – Liga a árvore menor à maior para manter a profundidade baixa.

- Size: Número de nós.
- Rank: Altura da subárvore.

Quick-Find vs Quick-Union

- Quick-Find: Rápido para encontrar (find), mas lento para unir (union). Melhoria com união por tamanho.
- Quick-Union: Rápido para unir (union), mas pode ser lento para encontrar (find) sem otimizações. Melhoria com união por tamanho ou altura (rank).