Relatório do Projeto 2 – Agrupamento

Disciplina: Projeto de Algoritmos - SCC5900

Aluno: Cézanne Alves – 10846548

Este relatório visa discutir detalhes da implementação dos algoritmos de Kruskal e Prim, e análise de um agrupamento por árvore geradora mínima (MST) do 2º projeto da disciplina.

Kruskal

A principal estrutura de dados no algoritmo de kruskal é o ***Union Find Disjoint Set*** (UFDS)

Altura vs Peso

Caso fossemos implementar um UFDS em árvore sem compressão de caminho, fica claro que unir árvores de forma que a maior fique em cima minimiza a altura da árvore resultante, diminuindo o custo de futuras operações de consulta. Entretanto, ao realizar a compressão de caminho, a altura de várias sub-árvores muda, o que torna custoso recalcular a altura da raiz, já que esta é o máximo das alturas dos filhos (recursivamente).

Quando há alguma compactação de caminho, geralmente é utilizada união por tamanho (quantidade de nós) ou ***rank***, que é altura que uma árvore teria se seus ramos não tivessem sido compactados. A estratégia escolhida na implementação foi a união por *rank*.

Path compression

A compactação de caminho pode ser feita apontando um nó para seu avô a cada passo e seguindo direto para o avô, pulando de dois em dois nós, aproveitando assim a subida até o topo da árvore para compactar o caminho do nó consultado até a raiz. Essa estratégia é conhecida como ***Halving***.

```
int Find(int x) {
   while(x != id[x]) {
      id[x] = id[id[x]]; // só uma linha!
      x = id[x];
   }
   return x;
}
```

```
int find(int x){
    if(id[x]==x)
        return x;
    return id[x] = find(id[x]);
}
```

Essa implementação é segura pois, com o nó raiz aponta para si mesmo, não há risco de ultrapassar a raiz ao pular de dois em dois nós. Entretanto, após a conclusão da consulta o caminho do nó consultado terá metade do tamanho anterior.

Uma outra opção chamada de compressão (***Compression***) é aproveitar que o custo da subida pela árvore é O(log(n)) e descer pelo mesmo caminho atualizando os nós para apontarem direto para a raiz. O trabalho adicional é completamente amortizado já que O(2log(n)) = O(log(n)), e o caminho percorrido é completamente compactado "de graça" em uma só consulta, deixando as próximas consultas para os nós daquele caminho em tempo constante até que a arvore cresça novamente. Esse procedimento é facilmente implementado de maneira recursiva.

Tarjan e Leeuwen [1] fizeram um estudo detalhado dos algoritmos de UFDS e mostraram que usando tanto compression quanto halving, e tanto união por rank quanto por tamanho, a complexidade de instanciar e executar m operações em um UFDS de n elementos é de

$$\Theta\big(m\,\alpha(m,n)\big), se\ m \geq n$$

$$\Theta\big(n + m\,\alpha(n,n)\big), se\ m < n$$

Onde $\alpha(m,n)$ é a função inversa de Akerman que cresce ao infinito, mas tão lentamente que é abaixo de 5 para qualquer valor prático.

Como no kruskal, $m \in O(n^2)$ é garantido que $\alpha(m,n) \ll \log(n)$ e as operações do UFDS são amortizadas pela ordenação das arestas, que tem custo $O(E\log(E)) = O(E\log(N))$

Prim

O algoritmo de Prim tem custo O(E log(N)), onde E se dá porque cada aresta é considerada duas vezes (uma para cada nó incidente), e log(N) se dá porque, ao considerar uma aresta, pode-se atualizar o custo de um nó numa fila de prioridade com os N nós.

Como o python não fornece suas estruturas padrões uma fila de prioridade que permita mudar de maneira eficiente a prioridade de um elemento, para simplificar a implementação foi utilizada uma fila de prioridade comum, e os nós foram reinseridos, permitindo nós repetidos, bastando checar se um nó removido já foi processado antes, e descartar caso sim.

Isso aumenta a limite do total de desempilhamentos de O(N) para O(E) mas isso se amortiza com o processamento das E arestas. Além disso, a fila de prioridade cresce de tamanho O(N) para O(E), elevando o custo das operações de enfileirar e desenfileirar de O(log(n)) para O(log(E)), mas como $E < V^2$,

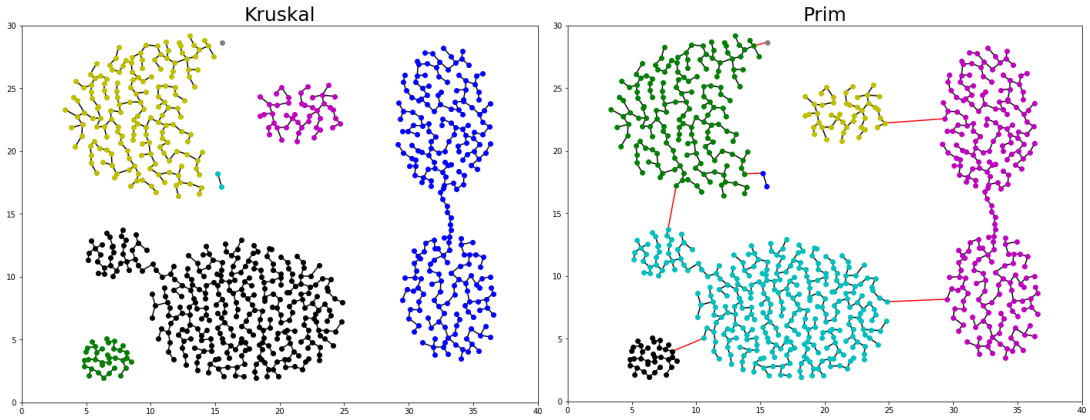
$$\log(E) \in O(\log(N^2)) = O(2\log(V)) = O(\log(N))$$

Portanto a implementação feita mantém a complexidade de $O(E \log(N))$

Análise do agrupamento

```
unq, counts = np.unique(smallers[:,0], return_counts=True)
print("number of distances up to max_edge(MST) with at least 2 edges: ", len(counts[counts>1]))
```

```
number of distances up to max_edge(MST) with at least 2 edges:  6696
```



Apesar do grafo possuir arestas de tamanho repetido, inclusive entre arestas menores que a maior aresta da MST, os dois algoritmos induziram o mesmo agrupamento. Os mapeamentos de nós para rótulos não são iguais, mas são isomórficos.

É possível confirmar a equivalência dos agrupamentos com o ***Rand Index***, aqui foi utilizada a versão ajustada, cujo valor varia no intervalo [-1, 1] e tende a dar valores próximos a 0 para agrupamentos aleatórios ou independentes, e exatamente 1 para agrupamentos equivalentes.

```
print(kk.part.relabeled() == prim_part) # mappings ar different
metrics.adjusted_rand_score(prim_part, list(kk.part)) # but isomorphic
```

```
False

1.0
```

Percebe-se visualmente que os agrupamentos não são ideais, mas podemos confirmar isso comparando com o agrupamento de referência:

```
reference = [int(line) for line in open("classes.txt")]
metrics.adjusted_rand_score(prim_part, reference)
```

```
0.8042069683967059
```

Isso se dá por uma limitação da estratégia de agrupamento. Ao agrupar por espaçamento máximo, um nó vai para o cluster que tem um ponto mais próximo a ele, independente da sua distância para o centro do cluster. Neste conjunto de dados, os *clusters* em Azul (figura do Kruskal) possuem uma trilha de pontos entre si, assim com os *clusters* em preto.

Apesar do exemplo ser sintético, e os clusters em azul serem obviamente pensados para evidenciar essa deficiência da estratégia, a situação dos clusters em preto não é muito incomum, e agrupamentos por espaçamento máximo podem desempenhar mal em cenários onde há sobreposição dos *clusters.*

[1] TARJAN, Robert E.; VAN LEEUWEN, Jan. Worst-case analysis of set union algorithms. **Journal of the ACM (JACM)**, v. 31, n. 2, p. 245-281, 1984.