Trabalho Prático 3 - Algoritmos I

Eduardo Santos Birchal

Matrícula: 2024023970

1 Introdução

O trabalho a seguir envolve um algoritmo para determinar a maior equipe de duendes possível onde nenhum dos duendes brigam entre si, com o número de duendes nunca excedendo 40. Para tal, foi necessário aplicar conceitos de programação dinâmica para que uma solução suficientemente eficiente fosse alcançada.

2 Modelagem

O problema apresentado se trata de um problema de Conjunto Independente Máximo, onde os duendes são representados como vértices em um grafo não direcionado, e suas inimizades são representadas como arestas. O objetivo é encontrar o maior conjunto de vértices tal que nenhum par de vértices compartilha uma aresta.

Para um grafo com N vértices, uma abordagem de força bruta testaria todos os 2^N subconjuntos possíveis; em vez disso, o problema é modelado usando a técnica Meet-in-the-Middle. Ela divide o conjunto de vértices V em duas partições disjuntas, V_{esq} e V_{dir}, cada uma com tamanho aproximado de $N/2$. A complexidade resultante se torna algo em torno de $O(2^{N/2})$, o que é tratável ($2^{20} \approx 10^6$). A modelagem da solução é estruturada em três etapas:

1. O grafo é particionado. Um conjunto independente global S será a união de um conjunto independente $S_{esq} \subseteq V_{esq}$ e um conjunto independente $S_{dir} \subseteq V_{dir}$, desde que não existam arestas conectando S_{esq} a S_{dir}.

2. Para processar a metade direita, utiliza-se a técnica de Sum-Over-Subsets (SOS) DP, adaptada para maximização. O objetivo é construir uma tabela que, dada uma bitmask representando o conjunto de vértices permitidos em V_{dir}, retorne em tempo constante o tamanho do maior conjunto independente contido inteiramente nesse conjunto. Isso elimina a necessidade de recalcular a solução ótima da direita para cada tentativa da esquerda.

3. O algoritmo itera sobre todos os conjuntos independentes válidos da esquerda. Para cada conjunto S_{esq}, identificam-se os vizinhos proibidos na direita. O complemento desses vizinhos define os vértices permitidos. A solução ótima é então obtida somando-se $|S_{esq}|$ com o valor pré-calculado pela SOS DP para os vértices permitidos.

3 Solução

A solução foi implementada em C++, utilizando bitmasks para representar conjuntos de vértices e operações de conjuntos. O uso de bitmasks permite que operações como união, interseção e verificação de pertinência sejam realizadas em tempo $O(1)$ utilizando instruções nativas do processador.

3.1 Representação do Grafo

O grafo é representado por um vetor de adjacência onde cada posição i armazena um inteiro de 64 bits (`long long`). O k-ésimo bit desse inteiro está ligado se, e somente se, existe uma aresta entre o vértice i e o vértice k.

3.2 Divisão e Pré-processamento

O conjunto de vértices é dividido em duas metades: V_{esq} (índices 0 a $N/2-1$) e V_{dir} (índices $N/2$ a $N-1$). Para otimizar as consultas futuras, calculam-se vetores auxiliares que mapeiam as adjacências da metade direita para um intervalo normalizado de $[0, |V_{dir}|-1]$. Isso evita operações de deslocamento de bits repetitivas durante a execução dos algoritmos principais.

3.3 Programação Dinâmica SOS

A tabela de Programação Dinâmica, denotada por `sos_dp`, é um vetor de tamanho $2^{|V_{dir}|}$. Cada posição `mask` armazena um par $(Tamanho, MelhorMascara)$, representando o maior subconjunto independente contido no subconjunto de vértices definido por `mask`.

O preenchimento da tabela ocorre em dois estágios:

1. **I**tera-se por todas as máscaras possíveis de V_{dir}. Se uma máscara representa um conjunto independente, ela é inicializada com seu próprio tamanho. Caso contrário, recebe valor zero.

2. **A**plica-se o princípio da inclusão-exclusão dinâmica. Para cada bit i e cada máscara M, se o bit i está ligado em M, o valor de $DP[M]$ é atualizado comparando-se o valor atual com o valor de $DP[M \setminus \{i\}]$. Dessa forma, todo conjunto guarda o valor do seu maior subconjunto independente.

O critério de maximização considera primeiramente o tamanho do conjunto. Em caso de empate, utiliza-se a ordem lexicográfica (preferência por vértices de índice menor) para fazer o desempate.

```
Para cada bit i de 0 até n_dir-1:
    Para cada máscara M de 0 até 2^n_dir-1:
        Se bit i está ligado em M:
            Opcao1 = DP[M] // Mantém bit i
            Opcao2 = DP[M ^ (1<<i)] // Remove bit i
            DP[M] = Max(Opcao1, Opcao2)
```

Figura 1: Pseudocódigo da Propagação SOS

3.4 Meet-in-the-Middle

A exploração da metade esquerda é realizada através de uma função recursiva de backtracking. A função constrói conjuntos independentes incrementalmente, decidindo para cada vértice $u \in V_{esq}$ se ele deve ser incluído ou não.

A cada chamada recursiva, mantêm-se duas máscaras de estado:

- `current_mask`: Vértices escolhidos até o momento na metade esquerda.
- `forbidden_right`: Máscara de vértices em V_{dir} que são vizinhos de algum vértice em `current_mask`.

Ao atingir o caso base (todos os vértices da esquerda processados), o algoritmo calcula o conjunto de vértices permitidos na direita ($Allowed = All_{dir} \setminus forbidden\_right$) e consulta a tabela SOS em $O(1)$ para obter o melhor complemento correspondente a $Allowed$. O tamanho total é $|current\_mask| + DP[Allowed].tamanho$.

4 Análise de Complexidade

A eficiência do Meet-in-the-Middle se dá por causa do tradeoff entre tempo e espaço. Para analisar o custo computacional, pode-se dividi-lo nas duas fases principais do algoritmo, assumindo N vértices no total e uma divisão balanceada onde $|V_{esq}| \approx |V_{dir}| \approx N/2$.

4.1 Complexidade de Tempo

1. **Metade Direita (Construção da Tabela SOS):**
   - Para a inicialização, o algoritmo itera sobre todas as $2^{N/2}$ máscaras possíveis da metade direita. Para cada máscara, verifica-se a independência dos vértices percorrendo seus bits, o que leva $O(N)$. O custo final é $O(N \cdot 2^{N/2})$.
   - Para a propagação, o algoritmo itera sobre cada um dos $N/2$ bits. Para cada bit, percorre-se toda a tabela de tamanho $2^{N/2}$. O custo final é $O(\frac{N}{2} \cdot 2^{N/2})$.
2. **Metade Esquerda (Busca Recursiva):** A função recursiva explora o espaço de subconjuntos da metade esquerda. No pior caso, ela visita todas as $2^{N/2}$ folhas da árvore de recursão. Graças ao pré-cálculo da SOS DP, o processamento em cada folha (a verificação de compatibilidade e a consulta à tabela) é realizado em tempo $O(1)$ através de operações bit a bit. O custo final é $O(2^{N/2})$.

Somando as etapas, a complexidade temporal total é dominada pelo termo da programação dinâmica:

$$T(N) = O(N \cdot 2^{N/2})$$

Para $N = 40$, temos $N/2 = 20$. O número de operações é proporcional a $40 \cdot 2^{20} \approx 4 \cdot 10^7$, o que é trivial para a maioria dos processadores modernos, levando menos de 1 segundo. Enquanto isso, a força bruta $O(2^N)$ exigiria 10^{12} operações, o que poderia levar horas ou dias, possivelmente impedindo que a equipe de duendes seja formada a tempo do Natal.

4.2 Complexidade de Espaço

O consumo de memória é dominado pela tabela de Programação Dinâmica `sos_dp`.

- A tabela armazena uma entrada para cada subconjunto possível da metade direita.
- O tamanho da tabela é, portanto, $2^{N/2}$ entradas.
- Cada entrada armazena um par de inteiros (8 bytes).

$$S(N) = O(2^{N/2})$$

Para $N = 40$, a tabela ocupa $2^{20} \times 8$ bytes ≈ 8 MB de memória RAM.

5 Considerações Finais

Esse Trabalho Prático necessitou a descoberta e adaptação de mais de um algoritmo conhecido para que a eficiência da solução seja viável; uma habilidade muito importante para qualquer programador, já que é através da combinação de conhecimentos existentes que novas inovações podem surgir. Além disso, ele demonstrou não só a importância da Programação Dinâmica para a otimização de algoritmos NP-difíceis, como também o impacto que a representação da informação tem sobre a performance de um programa: a representação dos conjuntos como bitmasks permitiu que várias operações complexas sejam reduzidas a $O(1)$.

Uma das grandes dificuldades para a resolução do trabalho foi a pesquisa a respeito do algoritmo, que demorou a levar a resultados verdadeiramente frutíferos, e também a intuição a respeito dos bitmasks. Com essas habilidades dominadas, porém, novas ferramentas surgem no arsenal do programador, que certamente serão de grande utilidade futura.

Referências

[1] Jon Kleinberg, Éva Tardos. *Algorithm Design*. Pearson Education, 1st edition, 2005.

[2] Ryan E. Dougherty. *Exact "Fast" Algorithm for the Maximum Independent Set Problem*. `https://www.youtube.com/watch?v=6jdWxjTASms`

[3] Wikipedia. *Maximal independent set*. https://en.wikipedia.org/wiki/Maximal_independent_set

[4] GeeksForGeeks. *Sum over Subsets | Dynamic Programming*. https://www.geeksforgeeks.org/dsa/sum-subsets-dynamic-programming/

[5] GeeksForGeeks. *Meet in the middle*. https://www.geeksforgeeks.org/dsa/meet-in-the-middle/