MIS-APA

Alana Bispo de Souza
Breno Henrique Rosas do Nascimento
Levi Weber Costa

November 2020

1 Descrição formal do problema:

O Maximum Independente Set(MIS) ou Maior Conjunto Independente tem a seguinte descrição formal: dado um grafo $G = (V, E)$ um Independent Set é um subset $U \subseteq V$, tal que não existem nós adjacentes em U. Um Independent Set é maximal, se nenhum nó pode ser adicionado sem violar a independencia. E um Independent Set de máximo tamanho é chamado de máximo.

2 Justificativa da NP-Completude

O problema de escolha do Independent Set é NP-Completo uma vez que não existem algoritmos que o resolvam em tempo polinomial e o problema do máximo Clique pode ser reduzido polinomialmente ao Maximum Independente set e é NP-Completo.

3 Revisão de literatura

http://i.stanford.edu/pub/cstr/reports/cs/tr/76/550/CS-TR-76-550.pdf
Finding A Maximum Independent Set by Robert Endre Tarjan and Antony E. trojanowski.

Tarjan e Antony trabalham basicamente em cima do algoritmo de achar todos os subsets de um conjunto em $O(2^n)$, sendo n o tamanho do conjunto e otimiza em um algoritmo com complexidade de tempo $2^(n/3)$
Durante o artigo é explicado o que foi pensado em cada passo da otimização, provas do por que funciona e de complexidade, além de todo o psudeo código apresentado pelos mesmos.

Uma das formas utilizadas para resolver o problema, é a transformação do grafo em grafo complementar, e a utilização do algoritmo de máximo clique, que é equivalente ao Maximum Independent Set.

Outra possibilidade para resolução do problema é uma solução computacional, testar para todos os conjuntos de vértices se são um conjunto independente

http://people.idsia.ch/ grandoni/Pubblicazioni/FGK06soda.pdf

4 Metodologia a ser utilizada

A metodologia a ser utilizada será nossa a abordagem Algorítmica

Pensamos em 2 algoritmos que podem nos dar a resposta do problema, o primeiro é um muito citado nos artigos que lemos, de testar para todos os subsets em 2^n. Porém pensamos que podemos otimizar isso fazendo uma busca binária no tamanho do subset, já que queremos o maior subset e é um problema monotônico, onde se existe um subset de tamanho k, existe um subset de tamanho k-1. Associado a busca binária teríamos uma função recursiva que testaria todas as possibilidades de set com a quantidade especificada na busca binária de forma que só seja testado subsets daquele tamanho.

Então teríamos 3 algoritmos para implementar, o 2^n, o que foi pensado pelo grupo citado no parágrafo anterior, e o algoritmo da literatura Finding A Maximum Independent Set by Robert Endre Tarjan and Antony E. trojanowski em $2^{n/3}$ e comparar os tempos de execução.

5 Código de força bruta

A ideia por trás desse código, é literalmente testar todas os subconjuntos possíveis de vértices e pegar o maior subconjunto que retorna é um independent set. Para fazer isso usamos a ideia básica de trabalhar com a forma binária dos números, por exemplo gerar todos os subconjuntos de um conjunto de 2 elementos.

00
01
10
11

1 - Representa que o i-ésimo elemento do conjunto foi escolhido para ser testado.

0 - Não foi escolhido para ser testado.

Para cada subconjunto testamos todos os elementos escolhidos 2 a 2 perguntando se existe alguma aresta ligando os mesmos. E para cada subconjunto que retornar true para ser um independent set atualizamos nossa resposta caso esse subconjunto tenha tamanho maior que a nossa resposta até o momento.

Complexidade: $\theta(2^n * n^2)$

6 Implementação de busca binária

Essa parte foi a observação do grupo de que o problema do independent set é monotônico, ou seja, se existe um independent set de tamanho n, então com certeza existe de tamanhos $n-1, n-2, ...1, 0$. Então só precisaríamos testar para um tamanho maior que N, e se não existe um independent set de tamanho n, então com certeza não existe de tamanho $n+1, n+2...$

Com isso em mente fizemos uma busca binária dos números 0 até n, sendo n a quantidade de vértices, justamente a isso temos uma função que testa todas as combinações de subconjuntos com uma quantidade X de elementos, exemplo:

Quantidade de vértices = 5, testando para X = 3;
00111
01011
01101
01110
10011
10101
10110
11001
11010
11100

E fizemos isso basicamente gerando um vetor de n posições zeradas, e que as X últimas posições iguais a 1. A partir daí podemos usar a função "next permutation" do C++ que gera todas as permutações do vetor pra esse X, o que coincide com todas os subconjuntos possíveis. Para cada permutação fazemos igual no brute, testamos os elementos ativos 2 a 2 verificando se existe uma ligação entre eles, caso alguma das permutações retorne true para o independent set já finalizamos o check da busca binária e procuramos para um X maior, caso não encontre nenhuma permutação válida retornamos false no check e testamos para um número menor.

Complexidade: Para essa implementação o tempo varia para cada caso de teste, dependendo de qual é o tamanho do maximo independent set, o somatório da complexidade vai ser $\sum_{i=0}^{qt1} \binom{n}{med[i]} * n^2 + \sum_{i=0}^{qt2} var[i] * n^2$ onde qt1 é a quantidade de vezes que a busca binaria escolheu um tamanho de independent set maior que o máximo, e qt2 a quantidade de vezes que a busca binaria escolheu um independent set menor ou igual ao máximo sendo med o vetor dos tamanhos escolhidos errados e var o vetor da quantidade de iterações até encontrar uma permutação onde se tinha um independent set possível. Sendo assim podemos então analisar o pior caso que seria quando para cada uma dessas iterações, se

o independent set for encontrado ele é a ultima permutação e para as linhas de mais custo do triangulo de pascal seria aproximadamente $O(logn * \binom{n}{\frac{n}{2}} * n^2)$ mas na prática o resultado é bem melhor que isso.

7 Otimização da busca binária

Essa otimização foi feita pensando no caso inicial da busca binária, sabemos que se estamos testando os números de 0 até n, o primeiro número testado é o chão de $\frac{n}{2}$, e que todas as permutações do vetor de $\frac{n}{2}$ 0's e $\frac{n}{2}$ 1's é o maior número da n-ésima linha do triângulo de pascal. Com isso criamos um treshold baseado na quantidade de arestas do grafo, pois um grafo que tem muitas arestas tende a ter um independent set menor, e com poucas arestas maior. Para fazer isso alteramos a busca binaria para não escolher simplesmente o número do meio inicialmente, se o número de arestas for maior que o treshold começamos dividindo o intervalo por 10 inicialmente e depois por 9, e assim por diante até ficar no padrão da divisão por 2.

O mesmo somatório da busca binária normal, a heurística melhorara casos específicos, onde o somatório para está busca binaria é menor

8 BackTracking

Na solução de backtracking o algoritmo é da seguinte forma: Com uma função recursiva vamos guardar o estado do independent set atual e andar de 1 a n escolhendo colocar ou não o nó atual no independent set. Para isso, testamos se o nó possui uma aresta para um nó já usado, se já possui retornamos a recursão para o estado do próximo vértice sem adicionar o nó, se não possui retornamos o máximo entre a recursão colocando o nó + 1 e a recursão sem colocar o nó no independent set. E para o caso base, quando a posição é maior doque n retornamos 0, e com isso no final o resultado da função vai ser o maior independent set no grafo.

complexidade: $O(iqt * (n^2 + n))$ sendo iqt a quantidade de independent sets que o grafo de entrada possui, uma vez que a recursão vai gerar todos os independent sets possíveis e para cada um deles percorrer $n^2 + n$

9 Resultados computacionais finais

Separamos 4 tipos de casos de teste.

9.1 Quantidade de vértices entre 1 e 20 e quantidade de arestas entre 1 e n

Nesses casos de teste os resultados da busca binária otimizada e da busca binária normal ficaram bem parecidos, enquanto o backTracking teve o menor

tempo na maioria dos casos, e como esperado, o código de força bruta demorou consideravelmente mais, numa crescente exponencial com a quantidade de vértices, sendo em torno de 10 vezes mais lento que os resultados da busca binária e os resultados do backTracking costuma ser bem mais rápido, menos quando o resultado do maior independent set é algo próximo de n que ai a busca binária sempre retorna resultados rápido, pois o checker sempre para quando chega em um resultado válido e vai para o próximo número a ser testado.

9.2 Quantidade de vértices entre 1 e 20 e quantidade de arestas entre n e $\frac{n*(n-1)}{2}$

Nesse caso o código de brute force mostrou resultados semelhantes, já que nele o que mais importa é a quantidade de vértices independente das arestas, já as 2 buscas binárias deram uma piorada no tempo, já que agora o checker vai ter uma maior dificuldade em retornar true para um independent set,com uma diferença considerável entre as duas, pois a otimização faz com que não seja testado casos muito grandes. Já o backTracking começa a ter uma vantagem clara sobre ambos pois terão poucas possibilidades de formar um independentset para ser testado.

9.3 Quantidade de vértices entre 20 e 30 e quantidade de arestas entre n e $\frac{n*(n-1)}{2}$

Aqui começamos a ver como o código de força bruta é ineficiente, passando a demorar minutos para computar qual o maior independent set. Já entre as buscas binárias e o backtracking mostrou uma superioridade ainda maior, sendo em alguns casos centenas de vezes mais rápido que a busca binária, e milhares de vezes mais rápido que o de força bruta. um exemplo expressivo disso foi o seguinte:

Resultado com busca binaria: 4
1608.448000 s
Resultado com backtracking apenas: 4
0.041000 s
Resultado testando todas as possibilidades: 4
8827.445000 s
Resultado com busca binaria otimizada: 4
52.704000 s

9.4 Quantidade de vértices entre 20 e 30 e quantidade de arestas entre 1 e n

O brute force apresentou os mesmos resultados do 9.3, as buscas binárias em casos onde o independent set era muito grande, se mostraram extremamente rápidas, mas em casos médios ainda tiveram complexidade pior que a do backtracking. Este continuou tendo um resultado solido. Um exemplo desses casos é o seguinte:

Resultado com busca binaria: 22
0.073000 s
Resultado com backtracking apenas: 22
464.014000 s
Resultado testando todas as possibilidades: 22
4644.445000 s
Resultado com busca binaria otimizada: 22
0.086000 s

10 Conclusões e possíveis extensões

Concluímos que a implementação do backtracking foi a que se mostrou com melhor em tempo de execução médio nos casos testados. Porém em casos onde o máximo independente set era próximo de n, a busca binária é muito mais rápida, isso se deve a complexidade do backtracking ser ligada a quantidade de independent sets possíveis, uma vez que esta gera todos eles em sua recursão. Em contrapartida a busca binaria ao mesmo tempo que pode levar a tempos muito maiores calculando possibilidades que não formam um independent set, economiza muito tempo calculando alguns conjuntos específicos.

Uma extensão possível seria criar um algoritmo que integra a solução de backtracking com a solução de busca binaria, para casos com arestas proximas do máximo se utilizaria a busca binaria e casos com quantidade média ou pequena de arestas seria usado o algoritmo de backtracking.