Inteligência Artificial

Problema da Mínima Latência

PUC
RIO

Bianca Faria Dutra Fragoso

Pedro Felipe Magalhães

Introdução

O MLP é uma variante do problema do caixeiro viajante, sendo definido a seguir. Seja G=(V, A) um grafo direcionado completo, onde V = {v0, …, vn} é o conjunto de vértices, v0 representa o depósito e os outros vértices, os clientes, e A={(i,j) : i, j ∈ V, i ≠ j} é o conjunto de arcos, sendo que cada um é associado com o tempo de viagem entre os vértices i e j. O objetivo do MLP é encontrar um circuito hamiltoniano em G que minimiza o tempo total de espera (latência) dos clientes. O tempo de latência do i-éssimo cliente, ou l(i), é a soma de todos os tempos de viagem do depósito até o i-éssimo cliente presente no circuito Hamiltoniano.

Esse trabalho mostra como foi a metodologia da implementação de alguns algoritmos que chegam perto da melhor solução para o problema da mínima latência. Os resultados obtidos pelos nossos algoritmos, comparados com os melhores obtidos até hoje também são mostrados, tanto quanto a comparação entre os tempos para que eles achassem tais soluções.

1. Busca Gulosa com profundidade limitada e poda

1.1 O algoritmo

1.1.1 Descrição:

A Busca Gulosa com profundidade limitada e poda, utiliza uma busca em profundidade (DFS) com limite de profundidade e quantidade de nós adjacentes visitados a DFS retorna a menor latência encontrada a partir de um nó recebido. O ideal para achar um resultado exato, seria deixar a DFS rodar com profundidade = número de nós, no entanto, por se tratar de um problema exponencial temos que usar artifícios para limitar essa busca, e continuar tendo uma precisão aceitável.

A partir de um nó “S” nosso algoritmo tem a heurística de que o nó que retorna a melhor latência na DFS dentre todos os nós adjacentes a “S” é o melhor nó para adicionarmos ao caminho ideal. Assim, partindo do depósito, rodamos a DFS nos “k” melhores(menores peso da aresta) nós adjacentes ao depósito, adicionamos o nó que retornar a menor latência e aplicamos o mesmo procedimento ao nó adicionado. Fazemos isso até que o caminho esteja completo.

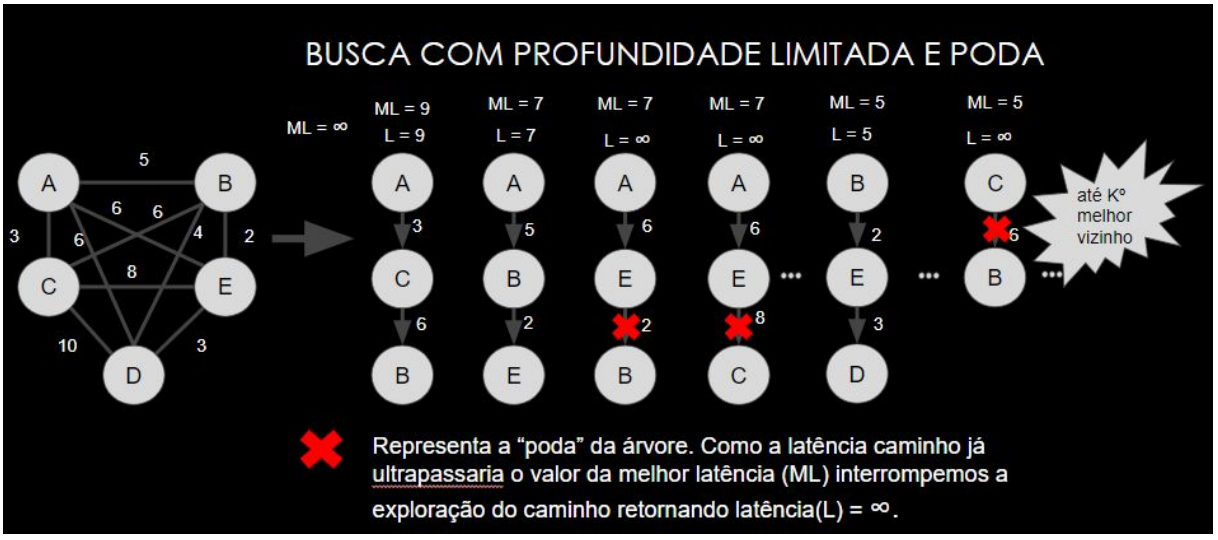
Primeiramente para que a DFS ficasse viável, adaptamos ela para receber uma profundidade limite, um “k” que define o número de nós adjacentes que serão explorados. Além disso, mantemos uma variável com a melhor latência encontrada o que permite a “poda” de ramos que já estouraram esse limite. Com essas modificações conseguimos reduzir drasticamente o número de nós explorados pela DFS.

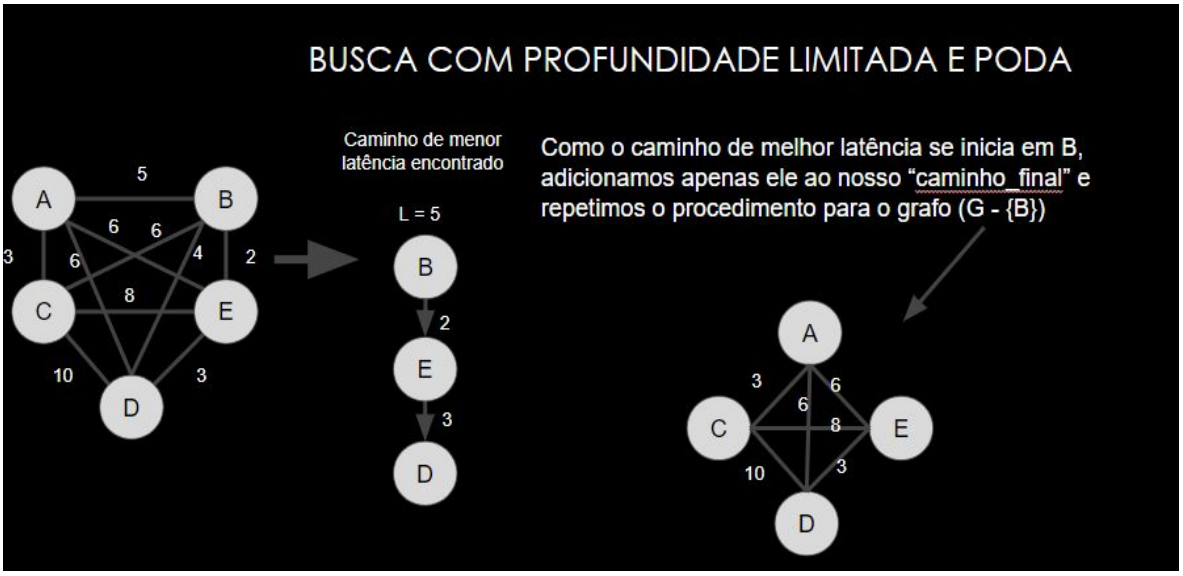
1.1.2 Pseudo-código:

```
busca_gulosa_dfs(inicial,p,k)
path <- append(inicial)
vizinhos <- pegaMelhoresVizinhos(inicial)
for i = 1 .. numero de cidades:
for v = 1 .. k
        se vizinhos[v] nao esta no path:
                lat =  dfs_latencia(vizinhos[v],p,k)
path <- append(vizinhos[v] com menor lat retornada)
path <- append(inicial)
retorna path
```

dfs_latencia (inicial , p, k)
se p == 0:
    se latencia < melhor latência calculada:
        atualiza melhor latência
    retorna latência
se latencia > melhor latência calculada:
    retorna MAX_INT
for i = 1 .. k:
    lat = **dfs_latencia**(vizinhoOrd[i],p,k)
retorna melhor lat

1.1.3 Exemplo:





1.2 Os resultados

	Melhores resultados já encontrados	Nossos melhores resultados
dantzig42	12528	13047(+4%,profundidade = 8, k = 4)
gr48	102378	106799 (+4%, profundidade = 11, k = 4)
brazil58	512361	538013 (+5%, profundidade = 6, k = 4)
gr120	363454	381391 (+5%, profundidade = 10, k = 4)
pa561	658870	726279 (+10%, profundidade = 9, k = 4)

1.3 O tempo

	Tempo para achar melhor solução encontrada	Tempo para achar nossa melhor solução
dantzig42	0.17	0.174s (profundidade = 8, k = 4)
gr48	0.31	14.284s (profundidade = 11, k = 4)
brazil58	0.55	0.024s(profundidade = 6, k = 4)
gr120	9.54	34.369s (profundidade = 10, k = 4)
pa561	1155.32	120.095s (profundidade = 7, k = 4)

2. Simulated Annealing

2.1 O algoritmo

Simulated Annealing, é uma meta-heurística para otimização que consiste numa técnica de busca local probabilística, e se fundamenta numa analogia com a termodinâmica.

A metaheurística usada é uma metáfora de um processo térmico, dito *annealing* ou *recozimento*, utilizado em metalurgia para obtenção de estados de baixa energia num sólido. O processo consiste de duas etapas: na primeira, a temperatura do sólido é aumentada para um valor próximo de 1100°C, na segunda, o resfriamento deve ser realizado lentamente.

De forma análoga, o algoritmo de *Simulated Annealing* substitui a solução atual por uma solução próxima (i.e., na sua vizinhança no espaço de soluções), escolhida de acordo com uma função objetivo e com uma variável T (dita *Temperatura*, por analogia). Quanto maior for T , maior a probabilidade de aceitar soluções piores dentro da vizinhança calculada. À medida que o algoritmo progride, o valor de T é decrementado, começando o algoritmo a convergir para uma solução ótima.

2.1.1 A solução inicial

Como solução inicial, usamos a solução calculada pelo nosso algoritmo já citado antes de Busca Gulosa em Profundidade Limitada com Poda, que já nos fornece uma boa solução e por isso melhora o resultado da *Simulated Annealing*. Testamos antes com soluções iniciais randômicas, mas os resultados foram piores.

2.1.2 A temperatura

Depois de rodarmos o algoritmo para nossas instâncias, notamos que a melhor temperatura inicial seria 1000, e realmente em algumas referências, vimos que o valor normalmente utilizado como inicial é algo perto disso.

Vamos diminuindo a temperatura com um fator de resfriamento de 0.002, ou seja, vamos diminuindo a temperatura em 0.2% a cada iteração.

2.1.3 A probabilidade

O que torna esse algoritmo especial é exatamente o cálculo da probabilidade baseado nas experiências químicas de resfriamento de amostras.

Calculamos um , como sendo a diferença entre o valor da função objetiva do vizinho novo e o valor da função objetiva da solução corrente.

A probabilidade p, então, é calculada como sendo:

Assim, como a temperatura vai diminuindo a cada iteração, a probabilidade de aceitar vizinhos piores também diminui.

2.1.4 As vizinhanças

Usamos 2 vizinhanças para chegarmos a melhores resultados.

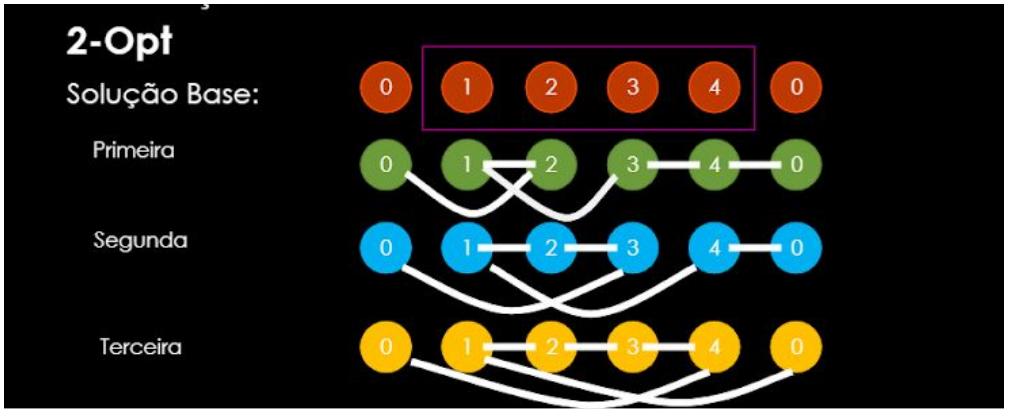
O que pudemos observar, é que as duas vizinhanças usadas tinham poucas diferenças para instâncias pequenas, mas conforme o número de nós fosse aumentando, as diferenças começavam a crescer.

2.1.4.1 2 – Opt

Usamos o 2-Opt como nossa vizinhança principal, pois é a que fornecia melhores resultados.

Portanto, a vizinhança que usávamos a cada iteração era ela, e como segunda opção a Swap.

Um exemplo de aplicação do nosso 2-Opt numa solução é mostrado na figura abaixo:

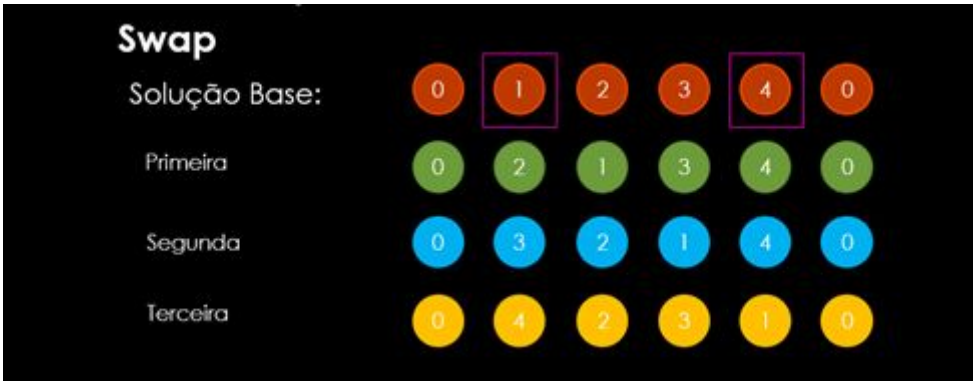


O 2-Opt fixa um elemento, nesse caso o 1 e vai invertendo os trechos do vetor a partir do 1. Na primeira ele apenas inverte o 1 e 2, na segunda ele inverte 1,2,3 para 3,2,1 e na terceira inverte 1,2,3,4 para 4,3,2,1. É como se ele tivesse cruzando arestas.

Tivemos que adaptar ao nosso problema, pois ele não poderia alterar nem a primeira e nem a última cidade, que são o depósito.

2.1.4.2 Swap

Como a Swap fornecia resultados piores que a 2-opt, usávamos ela como vizinhança secundária, ou seja, só usávamos ela quando o melhor valor da função objetiva se mantivesse por duas iterações seguidas. Isso nos permitia fugir de mínimos locais.



Na Swap, um número é fixado, nesse caso o 1, e ele troca de lugar com os nós seguintes. Na primeira, o 1 troca de lugar com o 2, na segunda o 3 troca de lugar com o 1 e na terceira o 4 troca de lugar com o 1.

2.2 Os resultados

	Melhores resultados já encontrados	Nossos melhores resultados
dantzig42	12528	12562(0.27%,profundidade = 3, k = 4)
gr48	102378	103976 (1.5%, profundidade = 8, k = 4)
brazil58	512361	513483 (0.21%, profundidade = 6, k = 4)
gr120	363454	381391 (4.9%, profundidade = 9, k = 4)
pa561	658870	726279 (+10%, profundidade = 9, k = 4)

2.3 O tempo

	Tempo para achar melhor solução encontrada	Tempo para achar nossa melhor solução
dantzig42	0.17	2.48
gr48	0.31	4.71
brazil58	0.55	7.31
gr120	9.54	54.87
pa561	1155.32	2555.94

3. Outros algoritmos

Alguns algoritmos foram implementados enquanto desenvolvíamos o trabalho, no entanto não deram resultados tão bons quanto os citados acima.

3.1 Busca gulosa usando Kruskal

Nos baseamos no algoritimo descrito em:
http://lcm.csa.iisc.ernet.in/dsa/node186.html

Primeiramente temos que guardar todas as arestas do grafo ordenadas de forma crescente. (utilizamos um heap) Em seguida vamos desempilhando o heap e só adicionamos a aresta desempilhada, se com a adição dela todos os vértices teriam grau <= 2 e não formasse um ciclo(só poderia formar um ciclo se fosse a última aresta.

Ao final teríamos um caminho hamiltoniano só com as arestas, bastando procurar pelo vértice que corresponde ao depósito para formarmos o caminho válido que começa e termina no depósito.

3.2 Busca Local

Implementamos inicialmente uma busca local. Porém, ela não teve resultados tão bons quanto a Simulated Annealing.

No entanto, para instâncias maiores o resultado dela é até bom e não demora muito.

4. A divisão de tarefas

A maioria do trabalho foi feito em conjunto, com envolvimento total dos dois participantes.

Tivemos algumas discussões para decidir qual seriam os melhores algoritmos que poderiam ser implementados.

Nos reunimos várias vezes para implementarmos o código juntos principalmente no início da construção do código.

Depois, continuamos aprimorando os algoritmos, cada um com uma ideia diferente para melhorar.

5. Como rodar programa

Se executado por um duplo clique no executável, ou pelo terminal só que sem parâmetros passados, o programa executará a busca gulosa e a Simulated Annealing para todas as instâncias(menos a de 561 cidades porque é mais demorado).(as instâncias devem estar em uma pasta chamada dataset na mesma pasta raiz do executável)

No entanto, quando executado pelo terminal, pode-se passar como parâmetro qual arquivo de instância você quer que o programa execute as buscas e com qual profundidade e “k” da busca gulosa.

ex:

./IA-latencia.exe "../dataset/dantzig42.tsp" 7 5

./IA-latencia.exe "../dataset/dantzig42.tsp" 7 5

./{nomeDoExe} [path_para_o_tsp] [p [k]]

6. Conclusão

Ao longo do trabalho, conseguimos implementar vários algoritmos e percebemos que uns são bem melhores que outros.

A Simulated Annealing se assemelha a uma busca local, no entanto ela tem resultado muito melhores, o que demonstra a diferença que uma meta – heurística faz num algoritmo.

Além disso, percebemos que se fizéssemos algo completamente randômico, como escolher a solução inicial randomicamente, teríamos

resultados bem piores do que se já começássemos com uma solução já relativamente boa, conseguida através de uma busca gulosa.

Pudemos também chegar a conclusão de como é importante usarmos técnicas que não sejam buscas cegas, pois mesmo usando uma busca gulosa e uma meta-heurística, conforme íamos aumentando o tamanho das nossas instâncias, o tempo aumentava consideravelmente.