Steiner Cycle Problem

Guilherme Martinez Sampaio
RA:073177

Glossário

Heurística

Para resolver o problema utilizei a meta heurística Grasp utilizando processamento paralelo para abrir varias instancias ao mesmo tempo, então a base do algoritmo foi:

Enquanto temos tempo:

1. Criamos uma solução randomica de forma construtiva.

2. Realizamos uma busca local sobre esta solução tentando melhora-la

3. Comparamos com a melhor solução que já encontramos

Solução randomica construtiva

Criamos uma solução randomica de três maneiras que descrevo abaixo. A primeira usa apenas os nós terminais e é chamada de *randomSolution()* enquanto a segunda utiliza de um fator aleatório para adicionar vértices não terminais que se chama *randomSolutionWithNoise()*. Já a terceira solução consiste em montar uma solução de forma gulosa chamada de *greedRandomSolution()*.

randomSolution(): Criamos uma solução aleatória pegando todos os vértices terminais do problema e arranjando eles de forma aleatória. Como sempre temos um grafo completo, sempre teremos uma solução valida para este caso.

randomSolutionWithNoise(): Esta implementação é bem parecida com a implementação do *randomSolution()* porém ele só muda com o fator de adicionar vértices não terminais na solução com intuito de aumentar a diversidade da solução. Sendo assim utilizo um fator randomico para verificar se um numero é menor que um limitante (no caso 10%) para adicionar o vértice não terminal a solução.

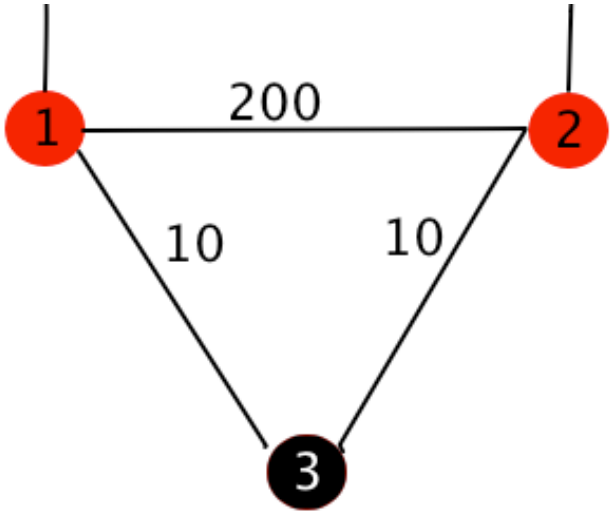
greedRandomSolution(): Por ultimo esta implementação pega um nó terminal de forma aleatória e analisa todos as arestas vizinhas pegando a de menor custo. Para balancear a

escolha pegamos dividimos o valor do peso das arestas que levam a nós terminais por 2 para aumentar a chances delas serem escolhidas convergindo mais rápido para a solução.

Busca Local

Na fase de busca local tentamos melhorar o resultado da solução tentando convergir para mínimos locais. Para isso utilizamos o seguinte principio gulosos.

Para reduzir o valor máximo da solução partimos do principio de que podemos remover uma aresta de peso alto da solução e achar um caminho mínimo entre estes dois vértices que tenha valor menor que a resta que liga a eles. A imagem abaixo ilustra bem o principio guloso adotado neste algoritmo.



A figura acima mostra um exemplo onde podemos melhorar o valor da solução removendo a aresta de maior peso e procurando um caminho de peso menor para substitui-lo. Assim podemos remover a aresta 1->2 e substituir pelo caminho 1->3->2 reduzindo o valor de 200 para 20.

Executamos esta interação ate que tenhamos passado por um grande numero de aresta ou limitado por um inteiro para não perder muito tempo.

No caso do grafo ser do tipo TSP, onde todos os vértices são terminais a busca local foi substituída pela heurística 2-OPT para garantir que haverá valores melhores com o passar do tempo e não apenas uma variação aleatória.

Execução

Para executar o programa subimos o numero de instancias igual a quantidade de core's disponíveis. No caso se temos um computador com um processador de 2 núcleos utilizamos duas instancias do Grasp rodando simultaneamente. Uma utilizando *randomSolution()* e a outra usando *randomSolutionWithNoise()*. Para tirar proveito de todo o recurso da maquina. Utilizei a biblioteca *OpenMP* para realizar o processamento paralelo utilizando o esquema Fork and Join executando cada instancia em cada core depois juntando o resultados de todas quando o tempo acaba.

Programa

O código consiste basicamente na implementação da classe *Grasp* contida no arquivo *GRASP.cpp*. Neste arquivo se encontram todos os métodos para gerar as soluções aleatórias e fazer a busca local.

Eu criei uma outra classe chamada *GreedyHeuristic* (contida no arquivo *GreedHeuristic.cpp*) que se baseia em pegar um vértice terminal de forma aleatória e procurar o caminho mínimo para todos os outros vértices terminais (ocultando os vértices da solução e os outros vértices terminais), adiciona o menor caminho a solução fazendo isso até fechar o ciclo. Porém sua execução foi retirada do trabalho por levar tempo proporcional a O(V^4) levando tempo não praticável para as instancias de 1000 vértices.

O código fonte do programa pode ser encontrado na integra no repositório do Github (https://github.com/gsampaio/Steiner-Cycle). Ao baixai-lo você encontrara um arquivo arquivo run.sh que se encarrega de compilar e executar para todas as instancias dada problema.

Resultados

Os resultados abaixo foram rodados por até no máximo 20 minutos numa maquina Core 2 Duo de 2.2GHZ e 8gb de RAM com duas instancias da classe Grasp rodando simultaneamente.

ARQUIVO	TEMPO DE EXECUÇÃO	SOLUÇÃO
AC_1000	1193.41	1500
AC_200	1192.11	1493
EC_1000	1193.65	4498
EC_200	1192.11	300
IC_1000	1193.03	2499
IC_200	1192.07	100
cc10-2p	1196.67	43171840
design432	1192.01	54
es10fst01	1192.06	752530985
es10fst09	1192.10	740216055
es10fst09_no_leaf	1192.08	669096325
es10fst12	1192.04	483387888
es10fst12_no_leafs	1192.03	794458970
hc7p	1192.29	901120
i320-001	1192.15	679023
odd_wheel	1192.04	28
random_1000v_100t	1198.76	49109
random_100v_10t	1192.09	4042
random_100v_25t	1192.16	13579
random_200v_20t	1192.08	9539
random_200v_50t	1192.34	25095
random_500v_50t	1192.66	22250
se03	1192.04	107
tsp_berlin52	1192.39	26021

ARQUIVO	TEMPO DE EXECUÇÃO	SOLUÇÃO
tsp_bier127	1204.34	625419
tsp_burma14	1192.03	5917
tsp_d1291	1284.08	1725134
tsp_d493	1212.13	443326
tsp_kroA200	1200.85	339160
wrp3-20_no_leaf	1192.24	490000000
wrp3-30_no_leaf	1192.21	1401000000

O tempo limite para execução do algoritmo foi limitado em 20 minutos. Mesmo assim percebemos que algumas instancias do TSP não conseguiram retornar antes de 1200 segundos devido ao fato da implementação da busca local ser diferente para estes casos do problema.

Conclusão

Embora não tenha acesso aos valores exatos para obter uma melhor comparação podemos perceber claramente que o algoritmo funciona muito mal quando executa instancias do tsp mesmo com a implementação do 2opt, com isso levo a acreditar que o opt esta implementado de maneira ineficiente, tanto por levar mais tempo por executar tanto por apresentar uma implementação possivelmente

Em contra partida o algoritmo tende a funcionar melhor em grafos onde existe um bom balanceamento entre vértices terminais e não terminais, uma vez que ele tenta sempre melhorar as piores arestas/caminhos.

O procedimento de busca local de procurar por um caminho de custo menor é custoso uma vez que precisamos montar um subgrafo sem os itens da solução, isso faz com que esta analise seja mais cara do que outras desenvolvidas por outras heuristicas.

Referencia Bibliográfica

- Christian, Nilsson (2003). *Heuristics for the Traveling Salesman Problem* *http://www.ida.liu.se/~TDDB19/reports_2003/htsp.pdf*
- L. Pitsoulis and M. G. C. Resende (2002) *Greedy randomized adaptive search procedures* http://www2.research.att.com/~mgcr/doc/grasp-hao.pdf
- Open MP http://openmp.org/wp/