Relatório MC548

Trabalho Prático - 2011 S1

Rafael Coelho G. de Oliveira - 063787

Rafael Suguro R. Leite - 063910

William Marques Dias - 065106

Organização e descrição dos arquivos

O nosso programa se organiza de maneira bem simples. São apenas 2 arquivos, um com o quicksort implementado e outro com o resto da lógica/implementação da heurística.

- quicksort.c: arquivo com a implementação do QuickSort utilizado para ordenar a entrada de acordo com um critério escolhido para a escolha gulosa.
- saveWorld.c: arquivo que contém toda a implementação da heurísitca utilizada, GRASP.

Abaixo está o detalhamento das estruturas de dados e funções utilizadas.

Dados

Estação

```
typedef struct Estacao
{
    char nomeEstacao[10];
    float custoEstacao;
    int *vPontosEstacao;
    int nPontosEstacao;
} Estacao;
Estacao *vetorEstacoes;
```

A struct *`Estacao`* guarda os dados relevantes de uma estação. Esses dados são:

- `nomeEstacao`: Nome.
- `custoEstacao`: Custo da estação.
- `vPontosEstacao`: Vetor de pontos que a estação cobre.
- `nPontosEstacao`: Quantidade de pontos que a estação cobre.

Temos também um vetor com os dados das estações lidas do arquivo de entrada, o `*vetorEstacoes`. Ele é uma variável global e estará disponível a todo momento para quaisquer uma das funções.

Resultado

```
typedef struct Resultado
{
    int *vetorPontosCobertos;
    Estacao *vetorEstacoesResultado;
    int nEstacoesResultado;
    float custo;
}Resultado;
Resultado vetorResultado;
```

A struct `Resultado` guarda os dados que serão apresentados no fim da execução como o melhor resultado possível calculado pela heurística. Os dados são:

- `vetorPontosCobertos`: Este vetor contém informações sobre cada ponto do nosso universo a ser coberto, ou seja, o valor de cada posição indica quantas vezes aquele ponto está sendo coberto pelo nosso conjunto escolhido de estações. Cada posição desse vetor corresponde ao ponto. Logo, vetorPontosCobertos[3] dispõe da informação da multiplicidade do ponto 3. Caso o ponto 3 esteja coberto por 4 estações, seu valor será 4, caso não seja coberto por nenhuma estação, seu valor será 0.
- `vetorEstacoesResultado`: Vetor que contém as estações selecionadas pela heurística para serem instaladas.
- `nEstacoesResultado`: Quantidade de estações a serem instaladas.
- `custo`: Custo final, é a soma dos custos de cada estação.

O `vetorResultado` é uma variável que armazena o valor de uma das iterações do algoritmo guloso aleatório e da busca local. Ela é modificada a cada iteração. Já o `vetorResultadoFinal` é a variável que armazena o valor final da heurística, modificada se o valor encontrado em `vetorResultado` é melhor que o valor atual.

Funções

`void leDados()`

Função que popula o `vetorEstacoes` com os dados do arquivo de entrada.

`void buscaGulosa()`

Função que constrói a primeira solução gulosa aleatória e entrega em `vetorResultado`. A partir desta solução que a busca local será efetuada.

`void buscaLocal()`

Função que implementa a busca local para melhorar a solução dada pela busca gulosa aleatória.

`int calcula_RCL()`

Função que calcula a lista restrita de candidatos que será utilizada na construção da solução inicial. A função retorna um inteiro que limita o tamanho do vetor.

`void imprimeResultado()`

Função que apenas irá imprimir a solução escolhida como a melhor pela nossa heurística.

`void removeEstacao(int i)`

Função que remove uma estação redundante da solução.

`void comparaResultado()`

Função que compara o melhor resultado encontrado com o resultado atual após a realização da busca local. Se o resultado atual é melhor que o melhor resultado encontrado, ele substitui este último.

`int main(int argc, char *argv[])`

Na main estamos apenas lendo o arquivo de entrada e tratando ele como stdin e depois chamando cada uma das funções acima durante um período de 1 minuto antes de imprimir o resultado.

Heurística

A heurística escolhida para a resolução do problema foi a GRASP. Resolvemos utilizar esta heurística porque ela tenta combinar as vantagens do uso da aleatoriedade e da busca gulosa para a construção de soluções. Além disso, as soluções geradas pelo uso desta heurística são de boa qualidade. A busca local é rapidamente executada e, geralmente, ela encontra melhores soluções que aquelas que foram geradas inicialmente.

A idéia da GRASP é construir uma solução inicial baseada em uma busca gulosa com elementos aleatórios, de maneira que essa busca não fique determinística. Depois da solução inicial entregue uma busca local é feita para aprimorar o resultado encontrado. Esse resultado da busca local é armazenado e começa o processo de encontrar uma nova solução. Caso essa segunda solução seja melhor que a primeira, definiremos a solução ideal agora como a segunda solução, ao invés da primeira.

Esse processo será feito por no máximo 1 minuto e não há problema em repetí-lo porque cada busca gulosa aleatória gerará uma solução possivelmente diferente da anterior, o que nos dará uma gama de diversas soluções.

Descrição da implementação

A implementação começa com a ordenação das estações através do algoritmo quicksort. As estações são colocadas em ordem crescente de acordo com a razão custo da estação/ números e pontos cobertos.

Em seguida, calculamos a lista restrita de candidatos (RCL) que será usada para gerarmos uma solução inicial com o algoritmo semi-guloso. Primeiramente, comparamos a razão da estação posicionada em primeiro com a razão daquela posicionada em último. Se elas forem iguais, usamos os custos das estações para criarmos a lista, caso contrário, usamos as próprias razões.

As estações incluídas na lista possuem custo ou razão inferiores a max, onde max = cmax + (alpha * (cmin - cmax)) e alpha = 0.5. A RCL limita o número de estações para a construção de uma solução inicial.

Durante o processo de criação da solução inicial, usamos a aleatoriedade para escolher a primeira estação que será incluída na solução. Utilizamos uma função randômica em conjunto com o horário do relógio da máquina. As próximas inserções seguem uma escolha gulosa sobre o vetor de estações.

Os critérios de parada para a construção da solução inicial respeitam duas condições: ou todos os elementos do vetor de pontos cobertos possuem valor igual ou superior a 1, ou todas as estações da RCL foram adicionadas. Nesse caso, se a instância criada não é válida, ou seja, existem pontos não cobertos, incrementamos em 1 o tamanho da lista e inserimos a nova estação na solução.

Uma vez criada a solução inicial, fazemos uma busca local para melhorar a solução encontrada. Nessa etapa, procuramos por estações redundantes e as removemos da solução. Uma estação é considerada redundante se todos os pontos que ela cobre são também cobertos por outras estações. Nesse caso, ela pode ser removida da solução com segurança.

Ao final da busca local, comparamos a solução atual com a melhor solução encontrada até o momento. Se a solução atual é melhor que a melhor solução encontrada até o momento, nos a armazenamos. Enquanto o tempo de execução não passar de 1 minuto, o programa começa uma nova iteração e faz todo o ciclo a procura de uma nova solução.

Resultados

Os arquivos foram rodados em MacOs com o comando t*ime* antes para medirmos o tempo. Lembrando que nosso algoritmo configura um *signal handler* pra receber um *alarm* em 58 segundos e dá 2 segundos para a função de impressão o resultado (o que é um tempo gigantesco, porém preferimos deixar uma margem grande).

Todos os testes foram rodados 10 vezes. O resultado apresentado no tempo foi a média, e para cada teste, todos os 10 resultados apresentados foram os mesmos, o que nos leva a crer que de fato a solução apresentada é a melhor.

Na saída dos testes (até o teste 3 apenas), estamos colocando os pontos que cada estação

cobre, para facilitar a visualização de que a solução cobre todo o universo do problema. O programa, no entanto, não imprime os pontos de cada estação no seu final.

Teste 1 (Arquivo /testes/teste1)

Entrada:

```
10 pontos
94 estações
```

Saída:

```
Custo: 18.85
Total: 3
S48 - 1 3 5 6 8
S59 - 2 5 7 8
S74 - 1 4 7 8 9 10
```

Tempo de execução: `real 0m58.017s`

Teste 2 (Arquivo testes/teste2)

Entrada:

```
50 Pontos
475 Estações
```

Saída:

```
Custo: 22.10
Total: 4
S110 - 4 5 8 9 10 12 14 15 19 20 21 22 26 27 29 31 32 33 34 35 38 39 41 43 44 45 47
S160 - 1 2 5 6 18 19 21 24 26 27 29 31 34 36 39 40 45 46 47 48 49
S423 - 1 2 4 7 8 9 11 13 16 17 19 22 23 24 25 26 27 28 30 31 32 34 38 39 43 47 50
S429 - 3 4 5 7 9 10 12 13 14 15 16 18 19 23 31 32 33 34 37 38 41 42 45 48
```

Tempo de execução: `real 0m58.017s`

Teste 3 (Arquivo testes/teste3)

Entrada:

```
100 Pontos
532 Estações
```

Saída:

```
Custo: 101.04
Total: 15
```

```
S20 - 3 4 5 8 12 14 27 40 42 43 56 72 74 78 81 85 94
S75 - 8 9 24 27 35 49 53 55 57 87 91 94
S128 - 17 20 39 52 64 69 72 74 79 86 100
S129 - 2 5 6 17 23 34 36 45 49 55 61 75 81 85 95
S157 - 2 7 11 26 29 41 47 56 57 77 82 86 90 91 93
S199 - 1 10 13 19 38 42 52 54 57 58 73 86 97 98
S203 - 2 10 27 45 48 49 60 61 66 71
S228 - 9 17 42 43 52 57 62 68 76 77 89 96
S287 - 4 9 21 28 30 45 52 53 83 84 90 95 97
S258 - 1 3 14 28 29 31 32 42 50 51 67 80 83 86 91
S408 - 9 13 15 26 28 36 61 63 67 69 85 98
S422 - 22 29 41 44 49 51 57 58 69 70 76 83 85
S428 - 4 28 29 33 38 42 46 58 63 65 68 69 72 85 88 95 96
S520 - 7 16 30 37 38 49 79 86 87 88 98 99 100
S523 - 2 10 16 18 25 27 36 59 88 92
```

Tempo de execução: `real 0m58.019s`

Teste 4 (Arquivo testes/teste4)

Entrada:

```
500 Pontos
108 Estações
```

Saída:

```
Custo: 162.22
Total: 17
S4
S9
S11
S17
S18
S24
S28
S36
S42
S52
S58
S56
S63
S75
S82
S88
S102
```

Tempo de execução: `real 0m58.015s`

Teste 5 (Arquivo testes/teste5)

Entrada:

```
50 Pontos
475 Estações
```

Saída:

```
Custo: 34.83
Total: 6
S561
S1167
S1355
S1395
S1444
S1828
```

Tempo de execução: `real 0m58.019s`

Teste 6 (Arquivo testes/teste6)

Entrada:

```
50 Pontos
475 Estações
```

Saída:

```
Custo: 79.85
Total: 13
S122
S127
S206
S213
S253
S408
S670
S682
S697
S839
S845
S858
S1030
```

Tempo de execução: `real 0m58.020s`

Conclusão

Após os testes, verificamos que uma meta-heurística bem implementada retorna bons valores

para instâncias grandes em pouco tempo. No algoritmo implementado, no entanto, a solução poderia ser melhorada com uma modificação da busca local.

Porém, segundos testes realizados, o tempo de processamento necessário para melhorar uma solução seria muito custoso em comparação com o ganho efetivo da solução. Isso é um *trade-off* recorrente em problemas de otimização.

Vale a pena, também, salientar a complexidade de implementação do GRASP. Comparado com as outras meta-heurísticas vistas em curso, o GRASP é de fácil implementação e conseguiu garantir bons resultados.