Thief Orienteering Problem (ThOP)

ES85248 - Gustavo Canal Uliana ES85282 - Fabio Henrique Martins Godoy

7 de dezembro de 2017

1 Introdução

Com o desenvolvimento constante da computação, o surgimento de novos problemas é constante. Entretanto, um grupo de problemas já há muito tempo conhecidos, ainda é bastante estudado. Os integrantes desse grupo são problemas famosos, seja pela sua relevância histórica, utilizade ou dificuldade.

Esse documento retrata uma séries de soluções para o problema ThOP, sendo esse uma combinação de dois problemas clássicos dos citados acima: O Problema do Caixeiro Viajante e o Problema da Mochila.

No Problema do Caixeiro Viajante, temos um grafo completo e valorado de n vértices. O objetivo é encontrar um ciclo hamiltoniano no grafo (um ciclo que passe por todos os vértices do grafo sem passar duas vezes pelo mesmo vértice) de custo mínimo.

No Problema da Mochila, consideramos que temos uma mochila de capacidade máxima de peso K, e um consjunto de n Itens, sendo que cada item possui um valor e um peso. O objetivo é selecionar um grupo de itens para colocar na mochila, de modo a obter o maior valor possível sem ultrapassar a capacidade K da mochila.

Ambos os problemas são categorizados como NP-Difícil, logo não se conhece um algoritmo para resolvê-los em tempo polinomial. Muitas vezes, métodos heurísticos e aproximações são usadas, para fornecer um resultado aceitável dentro de um tempo útil.

2 O Problema

Aqui explicaremos mais sobre o ThOP.

Assim como no PVC, existe um grafo valorado de n vértices, que representa os possíveis pontos a serem visitados. Diferentemente do problema original, entretanto, não é necessário que o ciclo passe por todos os pontos, somente que não ocorra repetições. O peso a aresta Aij representa a distancia entre os pontos i e j.

Cada ponto possui sua lista de itens, e assim como no problema da mochila, cada item possui seu custo e valor.

O objetivo é encontrar uma rota e plano de coleta de itens com ganho máximo para uma pessoa, aqui chamado de "ladrão". O ladrão possui os seguintes parâmetros:

- **Capacidade da Mochila**: Capacidade máxima de peso que pode carregar;
- **Velocidade Max**: A velocidade com que ele anda quando a mochila está vazia;
- **Velocidade Min**: A velocidade com que ele anda quando a mochila está na capacidade máxima;
- **Tempo Limite**: Tempo limite que o ladrão possui para fazer sua rota e retornar ao ponto inicial;

A Velocidade do ladrão reduz linearmente de acordo com o peso atual da mochila. Com sua velocidade atual e o valor de uma aresta Aij, podemos facilmente calcular o tempo que ele leva pra viajar de i para j.

Assim, a rota e plano de coleta encontrados devem respeitar as restrições de tempo e capacidade da mochila. Obviamente, um item só pode ser coletado se seua cidade está inclusa na rota.

Cada item está em uma única cidade.

3 Heurísticas

3.1 Método Guloso (Greedy)

O primeiro método heurístico analisado será uma solução gulosa. Para isso, definiremos o conceito de rank de Item e rank de Ponto.

Dado um Item qualquer, seu rank é um valor no intervalo [0, 1) que indica a qualidade daquele item em relação aos outros, sendo 0 o melhor valor e 1 o pior valor. Para calcular o rank de um item, primeiramente calculamos o custo benefício (valor/peso) de todos os itens. Os itens serão então ordenados em função desse custo benefício. Uma vez que os n itens são ordenados, cada item receberá o rank i/n, onde i é sua posição na lista ordenada. Assim, o item com melhor custo benefício receberá o rank 0, e o item com pior custo benefício recebera o rank (n-1)/n (para valores grandes de n, aproximadamente 1).

O rank de uma Cidade qualquer é dado pela soma do rank dos seus itens.

Usaremos o rank de uma cidade no processo de decisão guloso: em qualquer momento da solucao, escolhemos ir para a cidade com o menor rank disponível. Uma vez lá, pegaremos os itens com melhores custo benefícios que não prejudiquem muito o custo benefício atual da mochila.

O pseudo-código abaixo ilustra o algoritmo guloso:

```
Solucao Greedy(){
    Solucao S;
    Cidade atual = inicial;
    List<Cidades> cidades = cidades ordenadas por rank;
    for(Cidade C in cidades){
        if(verificaTempo(atual, C)){
            S.pegaItens(C);
            S.incluiCidade(C);
            atual = C;
        } else {
            continue;
        }
    }
    return S;
}

void pegaItens(C){
  List<Item> itens = itens da cidade C ordenadas por custo beneficio
  double ratio = mochila.valor / mochila.peso; //Custo beneficio
      atual da mochila
```

```
  foreach(Item I de itens){
    if(I.ratio > ratio * fator de piora){//Se custo beneficio do
        item for no max fator% do ratio da mochila
      inclui I na solucao;//Isso impede que incluir itens piore a
          solucao ate um valor lim. Algoritmo executado com ratio
          90%.
    }
  }
}

bool verificaTempo(Cidades atual, Cidade C);
//Verifica se e possivel ir da cidade atual para C, pegar o item de
    menor peso, e ir ate a cidade Final
//Sem estourar o limite de tempo ou mochila
```

3.2 Busca Local - Vizinhanças

Foram consideradas as seguintes vizinhanças durantes as buscas locais:

- **Incluir cidade**: Incluir cidade no final da solução, onde o efeito no tempo será minimizado;
- **Trocar Cidade**: Remover uma cidade e inserir outra no final, novamente buscando minimizar o impacto no tempo;
- **Trocar Item**: Troca os itens atuais de uma cidade por outros;
- **Troca Ordem Cidades**: Troca ordem de visita das cidades. Não tem efeito no valor da solução, mas pode melhorar o tempo;

As vizinhanças eram analisadas em sequência, e a melhor era escolhida.

3.3 Simulated Annealing

Método baseado no Simmulated Annealing padrão, sem nenhuma alteração significativa. A "temperatura inicial"definida foi de 750, com fator de redução de 0.1, máximo de iterações de 100. A perturbação feita na solução buscava trocar algumas cidades por outras aleatórias. Segue o psudo-código da perturbação:

```
Solucao perturbar(){
    Lista<Cidades> possiveis = cidades nao inclusas na solucao atual

    int qtdPerturb = valor aleatorio entre 0 e solucaoAtual.size()/2

    for(i in 0 to qtdPerturb){
        remove uma cidade aleatoria da solucao
    }

    while(for possivel inserir cidades na solucao){
    Cidade C = cidade aleatoria de possiveis
        solucaoAtual.incluiCidade(C);
    solucaoAtual.pegaItens(C);
    }


    for(i in 0 to qtdPerturb){
    Cidade c = cidade aleatoria da solucao
        solucaoAtual.trocaItens(C)//troca itens atuais por itens
            aleatorios
    }

    retorna solucao modificada;
}
```

3.4 GRASP

4 Resultados e Conclusões