Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada

Giovanni Franco de Paula Rosário

Rodrigo Lafayette da Silva

Caixeiro Alugador: Algoritmos Exatos

Natal - RN

Outubro/2018

Giovanni Franco de Paula Rosário

Rodrigo Lafayette da Silva

Caixeiro Alugador: Algoritmos Exatos

Relatório escrito pelos alunos Giovanni Franco de Paula Rosário e Rodrigo Lafayette para a disciplina DIM0605 - Projeto e Análise de Algoritmos, ofertada pelo Departamento de Informática e Matemática Aplicada(DIMAp) da Universidade Federal do Rio Grande do Norte(UFRN) e ministrada pela professora Dra. Elizabeth Ferreira Gouvea.

Natal - RN

Outubro/2018

Sumário

Resumo

Este relatório apresenta abordagens exatas para resolver instâncias do problema do Caixeiro Alugador. Primeiro, apresentamos a sua formulação e sua história ao longo dos material encontrado na literatura, em seguida mostraremos os algoritmos produzidos (falando em aspectos como suas complexidades) e, por fim, iremos relatar os resultados das execuções.

1 Introdução

O Caixeiro Alugador (CA) é um problema baseado no Caixeiro Viajante (CV) onde há mais restrições, estas justificadas pelo seu contexto, que é o de aluguel de carros para locomoção de turistas. Basicamente é um CV onde o caixeiro viaja usando carros alugados que possuem custos de aluguel diferentes e se movem entre as cidades com custos diferentes (simbolizando combustível gasto e a velocidade em que o carro se movimenta, por exemplo).

Considere um grafo completo G(V,A,C), onde V é o conjunto de vértices que representam as cidades e são indexados pelo conjunto dos Naturais, A o conjunto de arestas que ligam cada par de vértices e representam as estradas e C um conjunto de carros.

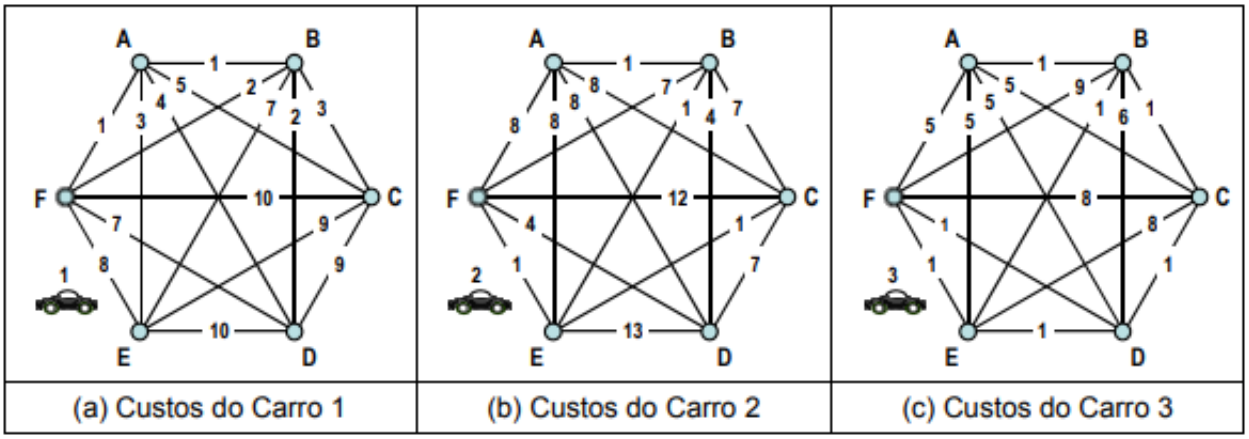

Figura 1: Custo dos carros.
fonte: "O Problema do Caixeiro Alugador - Um Estudo Algorítmico", Paulo A. (2011)

As restrições do problema são as do Caixeiro Viajante e adicionalmente as seguintes:

1. Há vários carros para aluguel, cada um com custo de viagem entre cidades diferente. Os custos são abstraídos em um único valor (com o fim de facilitar a modelagem) e são representados por uma entrada na matriz de adjacência do carro.

$$\forall c_i \in C\ (\{i\} \times \{< o, d >: o \in V \wedge d \in V\} \in Custos(c_i))$$

2. Um carro só pode ser devolvido se a operadora que o alugou estiver presente na cidade na qual ele se encontra.

3. Se um carro c alugado em uma cidade i for devolvido em uma cidade j, onde $j \neq i$ haverá uma cobrança associada ao custo de devolução, se $i = j$ então o custo será 0 (nesse caso o problema equivale ao CV).

4. Todo caminho do caixeiro (chamado de *tour* na literatura) deve começar e terminar na mesma cidade, que é o que acontece na maioria dos passeios turísticos.

5. Carros com características exatamente iguais podem ser alugados por valores diferentes na mesma operadora ou em operadora diferente, portanto é possível abstrair a operadora e trabalhar apenas com as informações do carro.

6. O custo de devolução do carro pode ser calculado de acordo com o caminho $i, \dot{...}, \dot{j}$ (onde i é a cidade em que a locação foi feita e j a cidade de devolução) ou mediante outros fatores (podendo até mesmo ser fixo).

2 Modelagem do problema

Os vários custos associados aos carros foram representados por tuplas (uma matriz tridimensional a nível de programação) $\{C \times V \times V\} \times CUSTOS$, ou seja, são relações de adjacência para cada carro indexando os seus respectivos custos.

Um caminho (válido) é modelado como $path \in \{x : x \in V^+ \wedge\ head(x) = last(x) \wedge (\forall y, z \in 0..size(tail(x))\ : (tail(x))[y] = (tail(x))[z] \rightarrow y = z)\}$. Um caminho parcial ou "subcaminho" é um caminho sem repetição de cidades que não possui todas as cidades (os vértices), seja $SUBPATH$ o conjunto de todos os subcaminhos.

Uma lista válida de carros usados é definida como $cars \in \{x : x \in C^+ \wedge (\forall y, z \in 0..size(x) : x[y] = x[z] \rightarrow y = z)\}$.

3 Algoritmo proposto: Backtraking

O algoritmo de Backtraking consiste em fazer uma busca da melhor combinação de cidades e carros usados entre todas as combinações possíveis.

O nosso implementação é dividida em 3 funções principais responsáveis por controlar a geração de combinações (`alg`), a comparação dos custos de combinações geradas (`calcular_custo_caminho`) e o calculo do custo de uma combinação (`calcular_custo_sub_caminho`

O tipo de dados usamos para representar a lista de carros usados e seu custo total (com devoluções) foi o `custoXcarro`, que é especificada como segue:

```
#define a_sz 20
int ORDEM;
int CARS;
int ORIGEM = 0;

#define RANDSEED 40
//#define CUSTO_TRECHO_MAX 50
//#define CUSTO_DEVOLUCAO_MAX 10

struct custoXcarro {
    int custo;
    vector <int> carro;
};
```

A função `alg`, que serve de "esqueleto"para o algoritmo e gera as combinações de cidades, tendo em sua condição do laço o predicado que possibilita parar apenas quando não houverem mais combinações a serem testadas. Como fixamos a cidade inicial (que é sempre uma padrão) o laço mais externo, que acaba definindo a complexidade da função, tem sua complexidade descrita por $\theta(n-1)!$.

```
// (n-1)! ?
int alg(int matriz[a_sz][a_sz][a_sz], int s, vector<int> &caminho, vector<int> &carros_usados, int matriz_custo_devolucao[a_sz][a_sz][a_sz]) {

    // A origem e o destino são sempre os mesmos, então fixei.
    vector<int> vertice;
    for (int i = 0; i < ORDEM; i++)
            vertice.push_back(i);


    // Não temos caminho ainda e usaremos min: N x N -> N.
    int menor_custo = INT_MAX;

    // É um do while pois podemos só ter uma permutação.
    do {
        /* PRINTAR PERMUTACAO ATUAL
        for (int i = 0; i < vertice.size(); i++){
            std::cout << vertice[i]+1;
            if(i+1 != vertice.size())
                std::cout << " -> ";
        }
        cout << endl;
        */

        /* PRINTAR MELHOR CUSTO PARA PERMUTACAO ATUAL*/
        custoXcarro melhor_custo = calcular_custo_caminho(matriz, vertice, matriz_custo_devolucao);

        int custo_atual = melhor_custo.custo;

        /*
        cout << "Melhor custo para a permutação : " << custo_atual << endl;
        */

        /* ATUALIZAR MELHOR CAMINHO */
        if (custo_atual < menor_custo) {
            menor_custo = custo_atual;
            caminho = vertice;
            carros_usados = melhor_custo.carro;
        }

    } while (next_permutation(vertice.begin()+1, vertice.end()));
    caminho.push_back(ORIGEM);
    return menor_custo;
}
```

A função `calcular_custo_caminho` irá, para cada permutação de caminho chamada na função `alg`, calcular o melhor custo possível para esse caminho. O calculo é feito começando o caminho com cada um dos carros disponíveis. Após usar o carro para o primeiro trecho, torna o carro indisponível e chama a função recursiva calcular_sub_caminho para calcular os trechos a frente. Após isso, é adicionado o custo das devoluções e verificado se o resultado para esse cenário é o melhor até o momento.

Complexidade = $CARS * calcularsubcaminho(n-1)$ onde n = Tamanho do caminho

```
/*
* FUNÇÃO INICIAL PARA CALCULAR UMA O MENOR CUSTO PARA UMA PERMUTACAO DO CAMINHO.
* Simula os cenários começando com cada um dos carros e chamando a função recursiva
* calcular_custo_sub_caminho para calcular os proximos passos.
*/
custoXcarro calcular_custo_caminho (int matriz[a_sz][a_sz][a_sz], vector <int> caminho, int matriz_custo_devolucao[a_sz][a_sz][a_sz]) {
        // Outra permutação, outra possível solução, é necessário resetar o custo.
        int custo_atual = 0;
        //começa no primeiro carro
        int carro_atual = 0;
        //carros disponiveis
        bool carros_disponiveis[CARS];

        custoXcarro melhor_custo;
        melhor_custo.custo = INT_MAX;

        //seta todos os carros como disponiveis
        for (int i = 0; i < CARS; i++)
            carros_disponiveis[i] = true;
        //Ponto de partida
        int s = ORIGEM;
        /* PARA TODO CARRO DISPONIVEL, COMEÇAR CENARIO COM ELE */
        for (int carro_i = 0; carro_i < CARS; carro_i++) {
            /* COMEÇA O CAMINHO COM CARRO_I */
            custoXcarro aux;
            carros_disponiveis[carro_i] = false; //Aluga o carro;
            custoXcarro aux_recursive = calcular_custo_sub_caminho(matriz, s+1, caminho, carro_i, carros_disponiveis); //Calcula o resto da rota com ele
            aux.carro.push_back(carro_i);
            aux.carro.insert(aux.carro.end(), aux_recursive.carro.begin(),aux_recursive.carro.end());
            aux.custo = matriz[carro_i][caminho[s]][caminho[s+1]] + aux_recursive.custo;
            /* VOLTA DISPONIBILIDADE DO CARRO PARA O BackTrack */
            carros_disponiveis[carro_i] = true;

            /* ACRESCENTAR OS CUSTOS DAS DEVOLUCOES NO CUSTO FINAL */
            int custo_final_devolucoes = 0;
            int pivot = 0;
            for (int j = 0; j < aux.carro.size(); j++)
                if (aux.carro[j] != aux.carro[pivot]) {
                    custo_final_devolucoes += matriz_custo_devolucao[aux.carro[pivot]][caminho[pivot]][caminho[j]];
                    pivot = j;
                }

            /* ultima devolucao de carro */
            custo_final_devolucoes += matriz_custo_devolucao[aux.carro[pivot]][caminho[pivot]][ORIGEM];

            aux.custo += custo_final_devolucoes;

            /*SE CUSTO FINAL FOR MELHOR ATÉ AGORA, SALVA */
            if (aux.custo < melhor_custo.custo)
                melhor_custo = aux;
        }
        return melhor_custo;
}
```

A função calcular_custo_sub_caminho verifica o caso de parada de ter chego no ultimo vértice. Se chegou, retorna o custoXcarro utilizado para o ultimo trecho. Se não chegou, entra em recursão para o restante do caminho com o carro atual e também com cada um dos carros restantes. Verifica o resultado de cada recursão e retorna o melhor resultado.

Complexidade = $CARS * calcularsubcaminho(n-1)$, onde $n = size(subcaminho)$ e calcularsubcaminho(0) = $O(1)$. Esse calculo desconsidera operações que não são previstas no modelo, no caso as que são específicas da implementação (como inserções em lista, cópias de valores e outras operações do gênero).

No total, a complexidade do algoritmo é $O(n! \times card(carros)^n)$, onde n é o número de cidades e card(carros) é o tamanho do conjunto de carros.

```
custoXcarro calcular_custo_sub_caminho(int matriz[a_sz][a_sz][a_sz], int s, vector <int> sub_caminho,
                                       int carro_atual, bool carros_disponiveis[]) {
    //SE CHEGOU NO ULTIMO VERTICE
    if (s == sub_caminho.size()-1) {
        custoXcarro aux;
        aux.custo = matriz[carro_atual][sub_caminho[s]][ORIGEM];
        aux.carro.push_back(carro_atual);
        return aux;
    }

    custoXcarro melhor_custo;

    /* TESTAR CENARIO DE CONTINUAR COM CARRO ATUAL */
    custoXcarro aux;
    custoXcarro aux_recursive = calcular_custo_sub_caminho(matriz, s+1, sub_caminho, carro_atual, carros_disponiveis);
    aux.custo = matriz[carro_atual][sub_caminho[s]][sub_caminho[s+1]] + aux_recursive.custo;

    aux.carro.push_back(carro_atual);
    aux.carro.insert(aux.carro.end(), aux_recursive.carro.begin(),aux_recursive.carro.end());


    melhor_custo = aux;

    /* TESTAR CENARIO DE TROCAR PARA CADA CARRO DISPONIVEL */
    for (int carro_i = 0; carro_i < CARS; carro_i++) {
        if (carros_disponiveis[carro_i]) {
            custoXcarro aux2;
            /* TROCA PARA CARRO_I, FAZ O PASSO ATUAL COM O CARRO_I E VAI NA RECURSAO COM CARRO_I */

            carros_disponiveis[carro_i] = false;
            custoXcarro aux2_recursive = calcular_custo_sub_caminho(matriz, s+1, sub_caminho, carro_i, carros_disponiveis);

            /* CUSTO = Passo Atual + Restante do caminho */
            aux2.custo = matriz[carro_i][sub_caminho[s]][sub_caminho[s+1]] + aux2_recursive.custo;
            aux2.carro.push_back(carro_i);
            aux2.carro.insert(aux2.carro.end(), aux2_recursive.carro.begin(),aux2_recursive.carro.end());
            carros_disponiveis[carro_i] = true;

            if (aux2.custo <= melhor_custo.custo)
                melhor_custo = aux2;
        }
    }

    return melhor_custo;
}
```

4 Resultados

A máquina que usamos para executar nossos experimentos tem as seguintes especificações:

- 8gb de Memória RAM.
- Processador Intel Core i5-6300HQ (2.30GHz 4)
- Sistema operacional Ubuntu 16.04 64 bits.

Os resultados obtidos nas instâncias usadas na literatura (adaptadas para o nosso padrão de entradas) foram os seguintes.

Instância	Dimensão	Número de carros	Tempo de execução (s)
Egito9n.car	9	4	163.734
Mauritania10n.car	10	2	38.1411
Bolivia10n.car	10	3	319.527
Etiopia10n.car	10	4	2385.07
Colombia11n.car	11	2	458.4
AfricaSul11n.car	11	3	4271.02

Mali11n.car	11	4	Não terminou (8h de execução)
Angola12n.car	12	2	6024.16

A diferença do tempo de execução das instâncias obedece o esperado, aumentar um pouco qualquer um dos fatores (carros e cidades) pode elevar muito o custo computacional, dado o número de combinações possíveis.

O autor do artigo "O Problema do Caixeiro Alugador - Um Estudo Algorítmico", Paulo Asconavieta não respondeu a nossa tentativa de contato e por isso não obtivemos os resultados da execução dos algoritmos dele (os resultados da própria tese não seriam suficientes dado o ambiente de execução diferente).

5 Dificuldades Enfrentadas e Conclusões

Nós perdemos bastante tempo tentando concluir o algoritmo usando programação dinâmica, que acabou não ficando pronto a tempo desta entrega. Felizmente tivemos sucesso em adaptar as instâncias medir os tempos de execução de forma adequada.

O fato do Paulo Asconavieta não ter mandado os códigos a tempo deste relatório não ter seu prazo de submetimento encerrado pode parecer ter sido um obstáculo no nosso caminho para alcançar o objetivo final do experimento (a comparação), por outro lado nós temos certeza que o nosso algoritmo foi projetado por nós e a nossa inspiração, se muito, foi apenas conceitual. O sucesso com a primeira técnica se dá provavelmente pelo fato de já termos experiências prévias com *backtraking*, então é de se imaginar que um vislumbre do algoritmo com programação dinâmica iria nos ajudar muito (embora a tese pareça indicar que a técnica pode não ter sido usada de forma pura).

Já temos algumas ideias do que fazer em termos de heurísticas, mas não definimos ainda.

6 Referências

[1] Paulo Henrique Asconavieta da Silva; Marco Cesar Goldbarg; Elizabeth Ferreira Gouvêa. The Car Renter Salesman Problem: An Algorithmic Study, In: 42o Simpósio Brasileiro de Pesquisa Operacional, Bento Gonçalves, 2010.

[2] RIOS, BRENNER HUMBERTO OJEDA; Elizabeth Ferreira Gouvêa; QUESQUEN, GREIS YVET OROPEZA. A hybrid metaheuristic using a corrected formulation for the Traveling Car Renter Salesman Problem, Em: 2017 IEEE Congress on Evolutionary Computation (CEC), Donostia (2017).

[3] FELIPE, DENIS; Elizabeth Ferreira Gouvêa; Marco Cesar Goldbarg. Scientific algorithms for the Car Renter Salesman Problem, In: 2014 IEEE Congress on Evolutionary Computation (CEC), Beijing (2014).

[4] André Villela; Satoru Ochi. An efficient hybrid algorithm for the Traveling Car Renter Problem, Expert Systems With Applications 64 páginas 132–140 (2016).