Relatório Trabalho 2

Lucas Müller - GRR20197160

[1]Centro Politécnico – Universidade Federal do Paraná (UFPR)
815300-000 – Curitiba – PR – Brasil

Resumo. *Este documento específica a descrição, modelagem e implementação de uma solução para o problema* ***Envio perigoso*** *por meio da técnica* ***Branch & Bound****, escrito em* ***linguagem C****.*

1. Introdução

Esse documento descreve o passo-a-passo para a modelagem e implementação de uma solução para o problema de *Envio perigoso* por meio de *Branch & Bound*, na linguagem C.

O problema consiste de encontrar a encontrar uma atribuição de itens em viagens, tal que cada viagem tenha a soma de pesos dentro do limite de carga (C) especificado. Também é preciso que cada par fornecido como restrições seja respeitado (pares de itens que não devem ser carregados na mesma viagem), e que a quantidade de viagens (k) seja minimizado.

Este relatório estará abordando os detalhes de modelagem e implementação para chegar em um solução que resolva este problema.

2. Modelagem

A modelagem deste problema foi baseada no algoritmo fornecido em COMBINATORIAL ALGORITHMS - GENERATION, ENUMERATION AND SEARCH, SEÇÃO 4.7, KREHER E STINSON. A ideia consiste em implementar, por meio da técnica de Branch & Bound, uma solução capaz de obter a solução ótima que resolva o problema, a partir da análise de todas as soluções viáveis possíveis.

A fim de reduzir a quantidade de travessias na árvore de soluções, foram empregadas técnicas de podagem para as sub-árvore de soluções, denominadas **corte de viabilidade** e **corte de otimalidade**. Onde:

Corte de Viabilidade Seleciona todas as possíveis viagens em Cl (vetor de escolhas) para x_l (item da iteração corrente) sob a condição de que x_l não vá sobrecarregar o peso limite C das viagens. Desta forma, é possível poupar de percorrer sub-árvores em que não é possível chegar a uma solução por conta da restrição de peso limite.

Corte de Otimalidade É verificado se o sub-conjunto de soluções corrente pode ser um candidato à ser uma solução ótima, isso é feito comparando o resultado de $B()$ para cada elemento de Cl com a solução ótima opt_K (quantidade mínima de viagens) corrente. Caso o resultado de $B()$ seja superior a opt_K para algum elemento Cl, então já sabemos não é possível obter uma melhor solução com este sub-conjunto, e portanto não é preciso continuar percorrendo.

O problema foi modelado da seguinte forma:

$$
\begin{aligned}
&\text{min: } max\{v(i) | i \in I\} \\
&\quad \text{S.T} \\
&\qquad w_{v(1)} < C \\
&\qquad w_{v(2)} < C \\
&\qquad ... \\
&\qquad w_{v(n-1)} < C \\
&\\
&\qquad a_1 \neq b_2 \\
&\qquad a_2 \neq b_2 \\
&\qquad ... \\
&\qquad a_{p-1} \neq b_{p-1}
\end{aligned}
$$

Onde:

I :conjunto contendo todos os itens a serem carregados
v :função que devolve o índice da viagem a qual o item pertence
n :quantidade itens em I
p :quantidade pares de itens que não podem ser carregados em uma mesma viagem
$w_{v(i)}$:peso acumulado para viagem de item i
$a_j \neq b_j$:pares de itens que não podem ser carregados em uma mesma viagem, tal que $j < p$

3. Análise das funções limitantes

Segue a função limitante conforme fornecida na especificação:

$$B_{\text{dada}}(E) = \max\{k, \sum_{i=1}^{n} w_i/C\}$$

E a sua implementação equivalente:

```
static unsigned
alt_bounding_fn(const struct bb_actor E[],
                const size_t En,
                const unsigned C,
                const unsigned k)
{
    float estimated_k = E[0].w;
    for (size_t i = 1; i < En; ++i)
        estimated_k += E[i].w;
    estimated_k = ceil(estimated_k / C);
    return (k > estimated_k) ? k : estimated_k;
}
```

Em outras palavras, esta função retorna o máximo entre as viagens já escolhida (E), e a soma de todos os pesos dividida pela capacidade do caminhão. O problema dessa função é que ela apenas considera as soluções já encontradas, desconsiderando os itens ainda não selecionados.

Uma melhor estratégia seria realizar a soma da quantidade de viagens dos itens mais selecionados, com a quantidade de viagens para os itens que ainda estão para serem selecionados, assim pode-se chegar a uma definição mais precisa para $B()$:

$$B_{\text{criada}}(E, F) = \max\{k, \sum_{i=1}^{|E|} w_i/C\} + \sum_{j=1}^{|F|} w_j/C$$

E sua implementação equivalente:

```
static unsigned
bounding_fn(const struct bb_item E[],
            const size_t En,
            const struct bb_item F[],
            const size_t Fn,
            const unsigned C,
            const unsigned k)
{
    float estimated_Ek = E[0].w, estimated_Fk = F[0].w;

    for (size_t i = 1; i < En; ++i)
        estimated_Ek += E[i].w;
    estimated_Ek = ceil(estimated_Ek / C);
    if (k > estimated_Ek) estimated_Ek = k;

    for (size_t i = 1; i < Fn; ++i)
        estimated_Fk += F[i].w;
    estimated_Fk = ceil(estimated_Fk / C);

    return estimated_Ek + estimated_Fk;
}
```

A nova função apresenta melhorias tanto em tempo, quanto em travessia (menos nós acessados). É possível visualizar a diferença entre as duas funções no seguinte teste:

Alt function ./envio -a < test/test3.in

```
5 1 2 3 4 1 2 3 4 6
6
Visited nodes: 38
Elapsed time: 0.037999999999999999 ms
Optimality cuts: 18
Feasibility cuts: 36
```

Default function ./envio < test/test3.in

```
5 1 2 3 4 1 2 3 4 6
6
Visited nodes: 21
Elapsed time: 0.025000000000000001 ms
Optimality cuts: 6
Feasibility cuts: 19
```

4. Implementação

4.1. Linguagem escolhida

A linguagem C foi escolhida por permitir uma manipulação precisa da memória. Nesta implementação é essencial que o tempo de execução do algoritmo não seja contaminado por chamadas de sistema, e portanto busca-se reutilizar memória sempre que possível. Todas as alocações dinâmicas necessárias acontecem uma única vez, antes da execução do algoritmo.

4.2. Arquivos

O repositório do projeto consiste dos seguintes arquivos:

src/input.c Realiza o parsing dos dados de entrada em uma estrutura 'struct bb_input'
src/bb.c Contém a lógica necessária para a solução do problema por meio do *Branch & Bound*
include/bb.h Funções e estruturas de acesso público para uso no **envio.c**
test/ Arquivos testes utilizados para esta implementação
envio.c Executável exemplo que demonstra o uso da biblioteca conforme as especificações do trabalho
Makefile Makefile utilizado para a realização da *build* do executável **envio**

4.3. Contexto de execução

Para criar um programa mais robusto, que funcione em situações de paralelismo, foi criado uma estrutura 'struct _bb_ctx' que permite guardar e manipular recursos de escopo "global", sem precisar usar globais de fato.

```
void
bb_solve(const struct bb_input *in, const bb_fn fn_bounding)
{
    struct _bb_ctx ctx = {
        .B = fn_bounding,
        .opt_K = UINT_MAX,
        .opt_X = calloc(in->n, sizeof *ctx.opt_X),
        .X = calloc(in->n, sizeof *ctx.X),
        .Cl = calloc(in->n, sizeof *ctx.Cl),
        .acc_weights = calloc(in->n, sizeof *ctx.acc_weights),
    };
```

```
    _bb_solve(in, &ctx, 0);
... // limpeza dos recursos alocados
}
```

4.4. Nomeação de símbolos

A fim de simplificar a leitura do código-fonte, foi decidido nomear símbolos de funções e variáveis com os mesmos caracteres utilizados na literatura e especificação deste trabalho. Por conta disso optou-se aderir a "más-praticas" de programação de nomeação de variáveis contendo um único caractere, ou que não sejam auto-explicativas. Mas o seu entendimento é facilitado para quem está familiarizado com as especificações.

4.5. Testagem

Para testar o funcionamento do programa basta seguir as seguintes etapas:

1. Gerar o executável **envio** pelo comando do Makefile:

```
$ make
```

2. Testar o executável com entradas de *test/*:

```
$ ./envio < text/test1.in
```

3. Testar tempo de execução com argumentos de comando *[-a][-o][-f]*

```
$ ./envio -a -o -f < test/test5.in
```

   Onde:
   -a Usa a função limitante alternativa, fornecida nas especificações do trabalho
   -o Desabilita cortes por otimalidade
   -f Desabilita cortes por viabilidade

5. References

[Guedes 2023] [Kreher and Stinson 1999]

References

Guedes, A. (2023). Segundo trabalho. In *Otimização - Trabalho 2*.

Kreher, D. L. and Stinson, D. R. (1999). section 4.7. In *Combinatorial Algorithms - Generation, Enumeration and Search*. CRC Press.