Relatório Trabalho 1

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* ***Elenco Representativo*** *por meio do algoritmo* ***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 *Elenco Representativo* por meio de *Branch Bound*, na linguagem C.

O problema e sua descrição podem ser encontrados na especificações deste trabalho, e se encontra nas referências deste documento. Para os fins deste relatório, estaremos abordando os detalhes de modelagem e implementação da solução.

2. Modelagem

A modelagem desta solução foi baseada no algoritmo fornecido em COMBINATORIAL ALGORITHMS - GENERATION, ENUMERATION AND SEARCH, SEÇÃO 4.7, KREHER E STINSON. A ideia consiste em implementar backtracing para examinar cada uma das escolhas possíveis em ordem crescente.

A fim de reduzir a quantidade de travessias em determinada sub-árvore X', podemos inferir do cálculo do Cl se o custo encontrado até então é viável ou não, estaremos checando os seguintes casos para cada nodo:

1. Checa se a quantidade de atores selecionados até o momento é igual a quantidade total esperada de personagens, e se for igual então marcamos o nodo corrente como inviável para a solução X pois significa que não atingimos a quantidade de grupos necessárias.
2. Checa se a quantidade de grupos selecionados até o momento é menor que a quantidade total esperada dos mesmos, e se for então marcamos o nodo corrente como possível candidato para solução. E iteramos então o nodo em sua sequência.

A seguir calculamos pela função limitante B(X'), em que X' é alguma sub-árvore da solução X sendo testada como solução corrente. Por se tratar de um problema de maximização, podemos inferir que se o resultado de B(X') é menor que o menor custo ótimo encontrado até agora, então podemos cortar os galhos de restantes da árvore de estados, sem necessitar visitá-los (isso é chamado corte de otimalidade).

3. Análise das funções limitantes

Segue a implementação feita da função limitante conforme fornecida nas especificações:

```
static unsigned
alt_bounding_fn(struct bb_actor E[],
                size_t Em,
                struct bb_actor F[],
                size_t Fm,
                size_t n)
{
    unsigned sum_cost = 0, min_cost = F[0].c;
    for (size_t i = 0; i < Em; ++i)
        sum_cost += E[i].c;
    for (size_t i = 1; i < Fm; ++i)
        if (F[i].c < min_cost) min_cost = F[i].c;
    return sum_cost + (n - Em) * min_cost;
}
```

Em tese, ela realiza a soma do custo total dos atores já selecionados, com o produto do menor custo encontrado entre os atores ainda não selecionados e a quantidade de personagens ainda faltando.

Para otimizar a previsibilidade desta função, foi empregado a técnica de memoização, guardando os menores valores encontrados entre os atores não selecionados em um array ordenado de forma crescente. E então é possível obter os N menores valores realizando um backtracking a partir do último elemento adicionado no array.

```
static unsigned
default_bounding_fn(struct bb_actor E[],
                    size_t Em,
                    struct bb_actor F[],
                    size_t Fm,
                    size_t n)
{
    const unsigned leftover = n - Em;
    unsigned sum_cost = 0;
    unsigned memo[512]; // keep track of smaller costs
    size_t idx = 0;
    for (size_t i = 0; i < Em; ++i)
        sum_cost += E[i].c;
    // backtrack smaller costs
    memo[idx++] = F[0].c;
    for (size_t i = 1; i < Fm; ++i) {
        if (F[i].c <= memo[idx - 1]) {
            memo[idx++] = F[i].c;
        }
    }
    for (ssize_t i = idx - 1; i >= 0 && idx < leftover; --i)
        memo[idx++] = memo[i];
    for (size_t i = 0; i < leftover; ++i)
        sum_cost += memo[(idx - i) - 1];
```

```
    return sum_cost;
}
```

No geral, a função apresenta melhorias tanto em tempo, quanto menor quantidade nós visitados. E por obter a soma dos últimos N menores custos (ao invés de generalizar a partir do menor custo encontrado) ela deve ser mais precisa em sua previsão. É possível visualizar a diferença entre as duas funções no seguinte teste:

Alt function ./elenco -a < test/test.in

```
1 3
15
Visited nodes: 6
Elapsed time: 0.0060000000000000001 ms
Optimality cuts: 3
Feasibility cuts: 5
```

Default function ./elenco < test/test1.in

```
1 3
15
Visited nodes: 5
Elapsed time: 0.0050000000000000001 ms
Optimality cuts: 3
Feasibility cuts: 4
```

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 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 Toda a lógica necessária para a solução do problema por meio do *Branch Bound*
include/bb.h Onde se encontra funções e estruturas de acesso público
test/ Onde se encontra arquivos testes
elenco.c Executável exemplo que demonstra o uso da biblioteca conforme as especificações do trabalho
Makefile Makefile utilizado para a realização da *build* do executável **elenco**

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 usar globais de fato.

```
void
bb_solve(struct bb_input *in, bb_fn fn_bounding)
{
    struct _bb_ctx ctx = { .B = fn_bounding,
                           .opt_P = UINT_MAX,
                           .opt_X = calloc(in->m, sizeof *ctx.opt_X),
                           .E = calloc(in->m, sizeof *ctx.E),
                           .F = calloc(in->m, sizeof *ctx.F),
                           .X = calloc(in->m, sizeof *ctx.X),
                           .lens_S = calloc(in->l, sizeof *ctx.lens_S)

    _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 acabamos por aderir a ”más-praticas” de programação de nomeação de variáveis de um único caractere, ou não 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 **elenco** pelo comando do Makefile:

```
$ make
```

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

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

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

```
$ ./despacho -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

[Derenievicz 2022] [Kreher and Stinson 1999]

References

Derenievicz, G. (2022). 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.