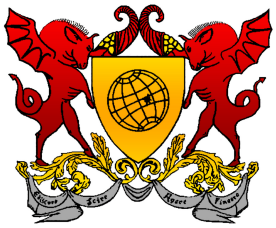
Universidade Federal de Viçosa
Campus Florestal
Ciência da Computação



Documentação 2º Trabalho Prático

Algoritmos e Estrutura de Dados

Grupo Victor (02658) e Adriano (02640)
Professor Thais Regina de Moura Braga Silva

Belo Horizonte, 8 de novembro de 2016

Sumário

Resumo

Este trabalho busca reproduzir dois dos mais conhecidos "problemas"intratáveis da computação de forma conjunta e simplificada. São eles o problema do caixeiro-viajante e o problema da mochila. O intuito é de reproduzir e estudar o comportamento de um "problema"computacional intratável. Para isso será utilizada a linguagem C e algumas ferramentas para auxiliar na elaboração e no desenvolvimento tanto do código fonte, como da documentação deste.

1 Metodologia

1.1 Organização e Planejamento do Projeto

O trabalho foi dividido em 3 fases:

1. a fase de planejamento;
2. a fase de produção;
3. a fase de teste e conclusão.

1.1.1 Fase de Planejamento

Na fase de planejamento elaboramos diagramas que representariam a estrutura do programa e a lógica a ser implementada, além de estudarmos as ferramentas que melhor auxiliariam na solução do problema proposto.

1.1.2 Fase de produção

Esta fase é composta por 3 etapas:

1. Criação de testes unitários para garantir estabilidade e correspondência ao planejamento feito na fase anterior;
2. desenvolvimento do código garantindo o sucesso dos testes criados;
3. aperfeiçoamento do código buscando maior legibilidade e menor complexidade de processamento.

1.1.3 Fase de teste e conclusão

Nesta fase iremos rodar o programa com vários valores de entrada com o intuito de estudar e documentar seu comportamento para diferentes tamanhos de entrada.

1.2 Ferramentas utilizadas

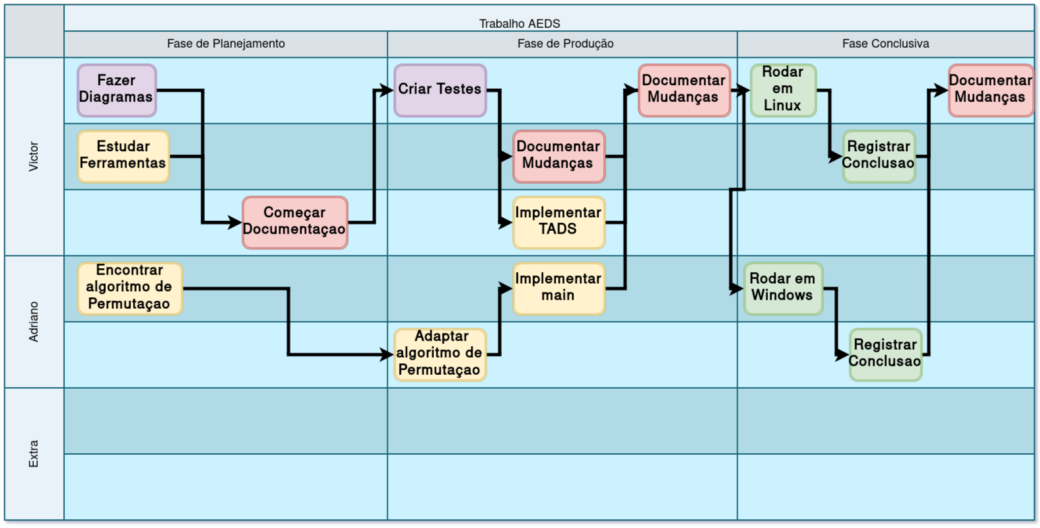
Durante a execução do projeto foi necessária a utilização de diversas ferramentas (listadas na tabela 1).

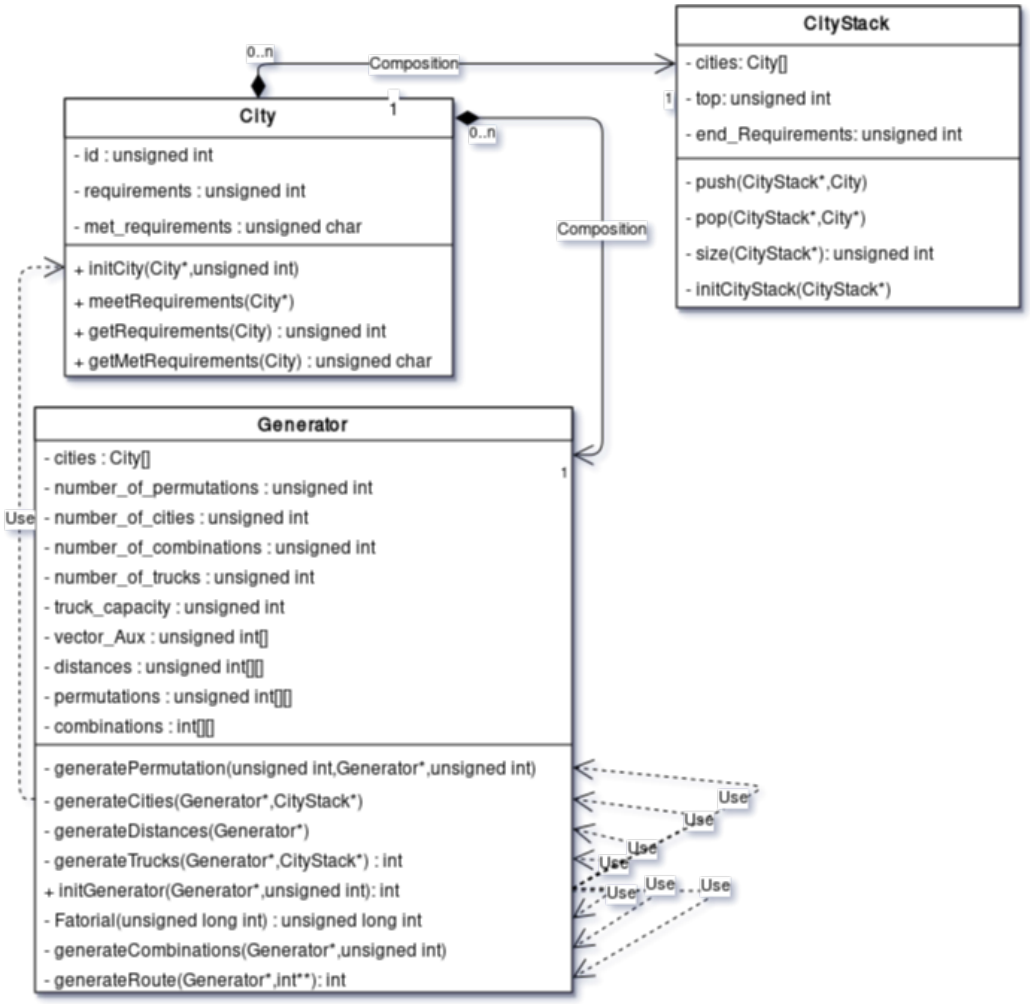
Nome	Function	Função
Tex-Studio	Writing Documentation	Escrever Documentação
Check	Unit-testing	Teste unitário
cmake	Cross-platform compiling	Compilação multiplataforma
draw.io	UML Designer	Desenhar UML
Atom	Text-editor	Editor de Texto
tablesgenerator.com	Generate LaTeX tables	Gerador de tabelas LaTeX
Github	Version-control	Controle de Versão
slack.com	Chatting and code sharing	Contato e compartilhamento de código

Tabela 1: Lista de Software e Ferramentas

2 Diagramas, Licença e Códigos fontes utilizados

2.1 Diagramas




CityStack
- cities: City[]
- top: unsigned int
- end_Requirements: unsigned int
- push(CityStack*,City)
- pop(CityStack*,City*)
- size(CityStack*): unsigned int
- initCityStack(CityStack*)
0..n
Composition
1
City
- id : unsigned int
- requirements : unsigned int
- met_requirements : unsigned char
+ initCity(City*,unsigned int)
+ meetRequirements(City*)
+ getRequirements(City) : unsigned int
+ getMetRequirements(City) : unsigned char
Composition
Generator
- cities : City[]
- number_of_permutations : unsigned int
- number_of_cities : unsigned int
- number_of_combinations : unsigned int
- number_of_trucks : unsigned int
- truck_capacity : unsigned int
- vector_Aux : unsigned int[]
- distances : unsigned int[][]
- permutations : unsigned int[][]
- combinations : int[][]
- generatePermutation(unsigned int,Generator*,unsigned int)
- generateCities(Generator*,CityStack*)
- generateDistances(Generator*)
- generateTrucks(Generator*,CityStack*) : int
+ initGenerator(Generator*,unsigned int): int
- Fatorial(unsigned long int) : unsigned long int
- generateCombinations(Generator*,unsigned int)
- generateRoute(Generator*,int**): int
Use
Use
Use
Use
Use
Use
Use

2.2 Licença

2.3 Códigos Fontes incorporados ao programa

- Arquivo FindCheck.cmake utilizado para conferir se libCheck está instalado
- Programa que adaptamos para gerar todas as possíveis rotas

3 Expectativas e Impressões

3.1 Pré-projeto

Como sabíamos que o programa a ser desenvolvido no Trabalho Prático não era algo simples e que nós teríamos pouco tempo para nos encontrar, devido a disponibilidade de cada um, decidimos planejar todo o desenvolvimento antes de começarmos a implementá-lo. Assim poderíamos desenvolver partes diferentes do código independente de outras, uma vez que sabíamos qual seria, até então, os parâmetros e o retorno de cada parte do código, além de termos uma noção do todo antes mesmo de implementar uma função.

3.2 Dificuldades encontradas durante desenvolvimento

Contudo nem tudo foi como planejado...

1. Muitos TADs que elaboramos no pré-projeto foram abandonados, ou porque podia ser representado de uma forma mais simples, ou porque sua implementação exigiria maior alteração no código de arranjo, ou por apresentarem ambos os problemas, como o caso do TAD path, o que acarretaria num desgaste maior do que a não utilização deste.

2. Nosso planejamento inicial partia do pressuposto de que a carga de caminhões deveria ser gerada aleatoriamente, igualmente a demanda de cada cidade, isso acarretou num aumento considerável da complexidade do código. Por fim o código originado desse planejamento nunca alcançou uma versão estável, exigindo um replanejamento da lógica de nosso algoritmo.

3. O código de arranjo que utilizamos além de simplesmente desenhar na tela o resultado, utilizava muitas variáveis globais e só fazia a permutação de n elementos tomados n a n (P(n,n)) o que exigiu uma grande adaptação para que atendesse a nossas necessidades.

3.3 Soluções encontradas para os problemas encontrados

1. Para substituir o TAD Path utilizamos uma abordagem um pouco diferente, a qual chamamos de *Combinations*. Sua lógica era basicamente a seguinte: Cada permutação é, posteriormente a sua criação, dividida em diferentes *Combinations* que são vetores contendo 0's a mais entre seus itens, representando o fim do percurso de um caminhão e o início da trajetória de outro. Assim solucionaríamos o problema de dividir as rotas entre os caminhões e ainda poderíamos utilizar o algoritmo de arranjo que tínhamos, cuja única limitação, que apresentava um desafio para nós, era a de que ele não podia gerar as permutações de n elementos tomados por valores diferentes de n;

2. O replanejamento se baseou na mudança da variável que determinava a capacidade dos caminhões para uma constante previamente definida e na alteração da forma como a demanda de cada era escolhida, antes esta era gerada aleatoriamente e depois ela passou a ser definida pelo usuário.

3. A solução encontrada para eliminar as variáveis globais foi passar o endereço dessas como parâmetro dentro da função recursiva de permutação, assim elas poderiam ser alteradas por todas as instâncias de recursividade da função. Já a solução encontrada para adaptar a saída da função foi simplesmente a criação de um ponteiro para ponteiros, formando assim uma espécie de matriz em que cada linha poderia ter tamanhos variados, nessa "matriz"cada linha representava uma permutação e cada item da linha era um inteiro representando o índice de uma cidade no vetor de cidades.

4 Referência

O programa foi feito dividido em TADS (city,citystack e generator), sendo que cada um compreende uma parte crucial do programa:

4.1 city

```
1          typedef struct{
2          unsigned int id;
3          unsigned int requirements;
4          unsigned char met_requirements;
5          }City;
6
7          int initCity(City* c, unsigned int req);
8          void meetRequirements(City*);
9          unsigned int getRequirements(City);
10         unsigned char getMetRequirements(City);
```

4.1.1 Variáveis

- ```
  1          unsigned int id;
  ```

- ```
  1          unsigned int requirements;
  ```

- ```
  1          unsigned char met_requirements;
  ```

4.1.2 Funções

- ```
  1          int initCity(City* c, unsigned int req);
  ```

  - Função que inicializa a cidade
  - Parâmetros:
    - City* c: Ponteiro para endereço de cidade a ser inicializada
    - req: Demanda da cidade

- ```
  1          void meetRequirements(City*);
  ```
  ```

- Função altera valor de met_requirements para 1, "atendendo"à demanda da cidade
- Parâmetros:
  - City* c: Ponteiro para endereço de cidade a ser alterada

```
unsigned int getRequirements(City);
```

- Função que retorna demanda da cidade
- Parâmetros:
  - City c: Cópia da cidade a ter demanda lida

```
unsigned char getMetRequirements(City);
```

- Função que retorna se cidade teve demanda atendida
- Parâmetros:
  - City c: Cópia da cidade a ter variável met_requirement lida

4.2 citystack

```
#define MAXSTACKSIZE 200
#include "city.h"
typedef struct {
City cities[MAXSTACKSIZE];
unsigned int end_Requirements; // ARMAZENA O SOMATORIO↩
    DE DEMANDA
unsigned int top;
} CityStack;

void initCityStack(CityStack* cs);
void push(CityStack* cs,City c);
void pop(CityStack* cs,City* c);
unsigned int size(CityStack* cs);
```

4.2.1 Variáveis

- ```
  City cities[MAXSTACKSIZE];
  ```

- ```
  unsigned int end_Requirements;
  ```

- ```
  unsigned int top;
  ```

4.2.2 Funções

- ```
  void initCityStack(CityStack* cs);
  ```

  - Função que inicializa a pilha de cidade
  - Parâmetros:
    - CityStack* cs: Ponteiro para endereço de pilha de cidade a ser inicializada

- ```
  void push(CityStack* cs, City c);
  ```

  - Função que empilha cidade
  - Parâmetros:
    - CityStack* cs: Ponteiro para endereço de pilha de cidade a ter cidade empilhada
    - City c: Cópia de valores de cidade a ser empilhada

- ```
  void pop(CityStack* cs, City* c);
  ```

  - Função que desempilha cidade

- Parâmetros:
  - CityStack* cs: Ponteiro para endereço de pilha de cidade a ter cidade desempilhada
  - City *c: Ponteiro de cidade a ser desempilhada

```
unsigned int size(CityStack* cs);
```

- Função que retorna tamanho da pilha
- Parâmetros:
  - CityStack* cs: Ponteiro para endereço de pilha de cidade a ser lida

4.3 generator

```
#include "city.h"
#include "citystack.h"
#define MAX 100
#define MAXX 20000
typedef struct{
City *cities;
unsigned int number_of_permutations; // contador de quantas linhas tem a permutacao
unsigned int number_of_cities;
unsigned int number_of_combinations; // contador de quantas linhas tem a matriz de combinacoes
unsigned int number_of_trucks;
unsigned int truck_capacity;
unsigned int vector_Aux[MAX]; // vetor universal para usar em recursoes etc.
unsigned int distances[MAX][MAX]; // matriz para armazenar as distancias entre as cidades.
int **permutations; // ponteiro de ponteiros para armazenar as permutacoes geradas, posteriormente as combinacoes.
int **combinations; // matriz para guardar as combinacoes de ZEROS
}Generator;

```

```
18          int generateTrucks(Generator* g, CityStack* cs);
19          void generateDistances(Generator* g);
20          void generateCities(Generator* g, CityStack* cs);
21          int initGenerator(Generator* g, unsigned int n); // ↩
                inicializa o gerador
22          unsigned long int Fatorial(unsigned long int n); // ↩
                calcula fatorial recursivo
23          void generatePermutation(unsigned int nivel, Generator↩
                * g, unsigned int n); // gera permutacoes ↩
                recursivas
24          void generateCombinations(Generator* g, unsigned int ↩
                last); // gera combinacoes com zeros
25          int generateRoute(Generator* g, int **bestRoute); // ↩
                gera rotas e calcula a demanda e menor rota
```

4.3.1 Variáveis

- ```
  1          City *cities;
  ```

- ```
  1          unsigned int number_of_permutations;
  ```

- ```
  1          unsigned int number_of_cities;
  ```

- ```
  1          unsigned int number_of_combinations;
  ```

- ```
  1          unsigned int number_of_trucks;
  ```

- ```
  1          unsigned int truck_capacity;
  ```

- ```
  1          unsigned int vector_Aux[MAX];
  ```
  ```

- ```
  unsigned int distances[MAX][MAX];
  ```

- ```
  int **permutations;
  ```

- ```
  int **combinations;
  ```

4.3.2 Funções

- ```
  int generateTrucks(Generator* g, CityStack* cs);
  ```

  - Função que gera a quantidade de caminhões
  - Parâmetros:
    - CityStack* cs: Ponteiro para endereço de pilha de cidades a ser lida
    - Generator* g: Ponteiro para endereço de gerador para alterar valor de quantidade de caminhões

- ```
  void generateDistances(Generator* g);
  ```

  - Função que gera a matriz de distâncias
  - Parâmetros:
    - Generator* g: Ponteiro para endereço de gerador onde está contida a matriz de distancias

- ```
  void generateCities(Generator* g, CityStack* cs);
  ```

  - Função que lê a demanda das cidades
  - Parâmetros:

    - Generator* g: Ponteiro para endereço de gerador onde está contida a variável que armaze a quantidade de cidades
    - CityStack* cs: Ponteiro para pilha das cidades cujas demandas serão definidas

- ```
  int initGenerator(Generator* g, unsigned int n);
  ```

  - Função que inicializa o gerador chamando todas as Funções geradoras nele contidas
  - Parâmetros:
    - Generator* g: Ponteiro para endereço de gerador a ser inicializado
    - unsigned int n: numero de cidades requisitado pelo usuário

- ```
  unsigned long int Fatorial(unsigned long int n);
  ```

  - Função de Fatorial
  - Parâmetros:
    - unsigned long int n: valor a ser calculado o fatorial

- ```
  void generatePermutation(unsigned int nivel, ↩
      Generator* g, unsigned int n);
  ```

  - Função que gera permutação de n elementos tomados n a n
  - Parâmetros:
    - unsigned int nivel: valor que representa nível de recursão atual
    - Generator* g: Ponteiro para endereço de gerador cuja "matriz"de permutações será alterada
    - unsigned int n: numero de elementos
  ```

- ```
  1 void generateCombinations(Generator* g, unsigned int last);
  ```

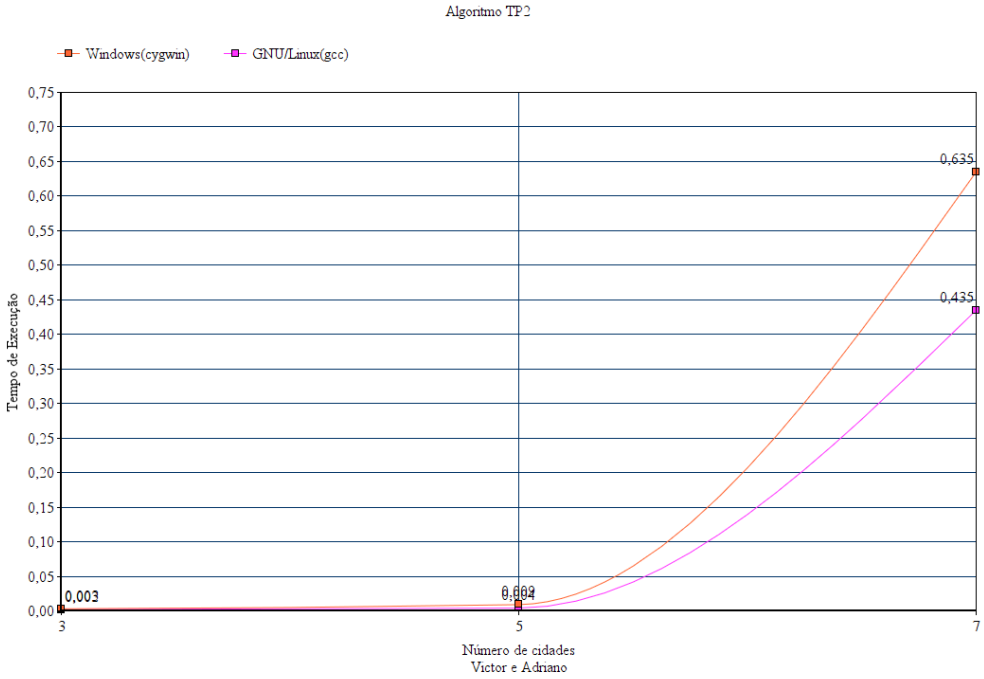
  - Função que gera combinações possíveis de serem dividas as permutações entre os caminhões
  - Parâmetros:
    - Generator* g: Ponteiro para endereço de gerador cuja "matriz"de combinações será alterada
    - unsigned int last: Limitador de número máximo de caminhões

- ```
  1 int generateRoute(Generator* g, int **bestRoute);
  ```

  - Função que encontra melhor rota de combinações baseado na matriz distâncias
  - Parâmetros:
    - Generator* g: Ponteiro para endereço de gerador cujas "matrizes"de combinações e de distâncias serão estudadas
    - int **bestRoute: Ponteiro para vetor que representa melhor rota válida encontrada

5 Conclusão

Após muitos testes em duas máquinas diferentes (i7-3537U 8gb ram e DualCore 2.0GHz 2gb ram), rodando em sistemas operacionais diferentes (Windows,ArchLinux,ElementaryOS) conseguimos fazer uma média aritmética entre os valores (tempo por quantidade de cidade) obtidos em ambos os sistemas(Windows e GNU/Linux) de cada computador e representá-la em um gráfico construído e renderizado no site http://www.onlinecharttool.com/



6 Agradecimentos

A Augusto pela ajuda e disposição em sugerir e discutir formas de resolver problemas encontrados no decorrer do desenvolvimento do programa;

A Athena por compartilhar um algoritmo de arranjo encontrado por ela, que, apesar de não utilizá-lo diretamente, foi de grande ajuda na compreensão do algoritmo que nós havíamos encontrado;

A Joaquim pela contribuição na elaboração da documentação;

A Universidade Federal de Viçosa Campus Florestal por nos proporcionar um ambiente de criatividade e inúmeras oportunidades;

A professora Thais Regina de Moura Braga Silva pela imensa preocupação e dedicação ao nosso aprendizado.