UNIVERSIDADE DE SÃO PAULO
CAMPUS DE SÃO CARLOS
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO (ICMC)

EDUARDO MACIEL DE MATOS 12563821
JOÃO PEDRO RIBEIRO DA SILVA 12563727
JOÃO OTÁVIO DA SILVA 12563748
LEONARDO MINORU IWASHIMA 12534738
NICOLAS DE GÓES 12563780

INTRODUÇÃO A TEORIA DA COMPUTAÇÃO (SCC-0505)

TRABALHO 02: DOCUMENTAÇÃO

São Carlos
2022

1. Estruturação

O código foi todo modularizado em funções, de maneira a simplificar o entendimento e possíveis manutenções. No código em si, cada função foi nomeada e comentada a fim de esclarecer seu funcionamento.

2. Fluxo

A execução do programa segue o seguinte fluxo:

- Inclusão das bibliotecas utilizadas;
- Declaração das structs utilizadas;
- Declaração do cabeçalho das funções utilizadas;
- Criação da máquina de Turing;
- Inicialização e leitura das informações da máquina. Isso inclui:
    - Alocação de memória e inicialização dos estados;
    - Alocação de memória e leitura dos símbolos terminais;
    - Alocação de memória e leitura do alfabeto estendido de fita;
    - Leitura do estado final; e
    - Alocação de memória e leitura das transições.
- Leitura das cadeias;
- Validação das cadeias através de recursão;
- Destruição da máquina, ou seja, liberação das alocações de memória.

3. Desempenho e alocações de memória

Na construção do projeto, o grupo procurou um meio termo entre desempenho, uso de memória e simplicidade do código. Os estados, os símbolos terminais, o alfabeto estendido de fita e as cadeias foram salvos em arrays dinâmicos, que alocam a memória correta em tempo de execução dependendo do seu número de elementos. Por outro lado, os arrays de transições, o vetor que representa a fita e as strings das cadeias possuem alocação estática, ou seja, com número fixo de elementos independente da execução do programa.

Nos casos onde foram utilizadas alocações dinâmicas, a implementação era simples de ser feita, além de não afetar o desempenho do programa. Já nos casos onde a alocação foi feita de maneira estática, realizar alocações dinâmicas teria uma implementação bem mais complexa, envolvendo diversas realocações de memória, que podem afetar o desempenho do programa se feitas em excesso. Como o enunciado do trabalho já informa o número máximo de transições, o comprimento máximo da fita e o comprimento máximo das cadeias, sendo todos esses números pequenos, a utilização de alocação estática foi preferível pelo grupo.

Todas as alocações de memória possuem uma verificação em caso de erro na alocação, interrompendo a execução do programa. Além disso, todas as alocações são liberadas ao final do programa, não havendo vazamentos de memória.

Em termos de tempo, a complexidade assintótica para cada estado q é $O(n_q * m_q)$, sendo m_q o número de transições desse estado q e n_q o número de vezes que esse estado q é alcançado durante o processamento. Portanto, o número total de passos da máquina seria $\sum_{i=0}^{N} (n_{q_i} * m_{q_i})$, para cada estado q em $q_0, q_1, \ldots, q_N$ da máquina.

4. Structs

4.1. Transição:

```
typedef struct transicao_st
{
    char simboloLido;
    char simboloEscrito;
    char movimentoCabeca;
    int indiceDestino;
} TRANSICAO;
```

char simboloLido: símbolo de fita lido na posição apontada pela cabeça.
char simboloEscrito: novo símbolo de fita a ser escrito na posição apontada pela cabeça.
char movimentoCabeca: movimento que a cabeça fará após a transição (R - direita, L - esquerda, S - mantém).
int indiceDestino: inteiro que representa o índice do estado para o qual a transição leva.

4.2. Estado:

```
typedef struct estado_st
{
    int ehFinal;
    int qtdTransicoes;
    TRANSICAO *transicoes[MAX_TRANSICOES_ESTADO];
} ESTADO;
```

int ehFinal: inteiro que sinaliza se é estado de aceitação;
int qtdTransicoes: inteiro que indica a quantidade de transições que o estado faz.
*TRANSICAO *transicoes[MAX_TRANSICOES_ESTADO]:* vetor de transições com máximo de 50 elementos, que indica todas as transições que este estado atual faz.

4.3. Máquina:

```
typedef struct maquina_st
{
    int qtdEstados;
    int indiceEstadoAtual;
    char *simbolosTerminais;
    int qtdSimbolosTerminais;
    char *simbolosDeFita;
    int qtdSimbolosDeFita;
    ESTADO **estados;
    char fita[23];
    int indiceCabeca;
} MAQUINA;
```

int qtdEstados: quantidade de estados que a máquina vai ler.
int indiceEstadoAtual: índice que indica o estado em que a máquina se encontra no momento.
*char *simbolosTerminais:* caracteres representando os símbolos terminais indicados.
int qtdSimbolosTerminais: quantidade de símbolos terminais indicados pelo usuário.
*char *simbolosDeFita:* caracteres representando os símbolos da fita indicados.
int qtdSimbolosDeFita: quantidade de símbolos de fita indicados pelo usuário.
*ESTADO **estados:* vetor de ponteiros para os estados da máquina.
char fita[23]: vetor de caracteres representando a fita.
int indiceCabeca: posição que a cabeça aponta no vetor fita.

5. Validação de Cadeia

A função de validação de cadeia é dividida em duas partes: na primeira, é recebida a máquina de turing e a cadeia a ser processada, é feita a preparação da máquina, restaurando a cabeça e o estado atual para suas posições iniciais e colocando a cadeia na fita. Após isso, é chamada a segunda parte, que é a função recursiva que de fato faz o processamento da cadeia.

O caso base é quando o estado atual é um estado de aceitação, ou seja, toda a cadeia já foi processada e validada, desta forma, a função retorna 1, indicando que a cadeia é válida.

Se a chamada da função não entrou no caso base, é feita a verificação se há alguma transição saindo do estado atual e com o mesmo símbolo que o caractere apontado pela cabeça, se sim, é feita uma chamada recursiva com o símbolo da fita sobrescrito, o estado atual e posição da cabeça atualizados (de acordo com a transição). Se não há nenhuma transição com o caractere atual, a função retorna 0.

Ao fim das recursões: Ou a máquina ficou presa em um estado, sem ter transições compatíveis saindo dele. Ou a máquina alcançou um estado de aceitação, nesse caso, a cadeia é aceita.

6. Discussões

O grupo ficou satisfeito com a solução desenvolvida. A organização e a modularização do código ajudaram no entendimento e na implementação de novas funcionalidades. Apesar de algumas alocações estáticas, a utilização de espaço do trabalho ainda é bastante otimizada, tendo em vista que as poucas informações que foram alocadas estaticamente tratam-se de arrays e strings pequenas. Em relação ao tempo, o desempenho do programa também é bastante otimizado, tendo em vista que ele foi codificado em C. Além disso, como visto no tópico 3, a complexidade do programa depende fortemente da complexidade da solução fornecida na entrada, sendo difícil chegar a uma complexidade assintótica precisa para o programa.