Universidade de Brasília
Trabalho 2 - Estrutura de Dados

Gustavo Carvalho, Tomás Rosemberg

Junho 2017

1 Introdução

Propõe-se o desenvolvimento de um algoritmo capaz de encontrar a saída de um labirinto, representado na forma de um grafo por listas encadeadas, utilizando o menor caminho possível. Em cada vértice, tem-se a possibilidade de encontrar uma porta ou uma chave. Não é possível passar por vértices que contenham portas sem que antes tenha-se pego a chave desta. Um exemplo de um labirinto é visto na Figura 1. Neste, as portas são representadas por letras maiúsculas, as chaves por letras minúsculas, o início pelo rótulo **II** e o fim por **FF**.

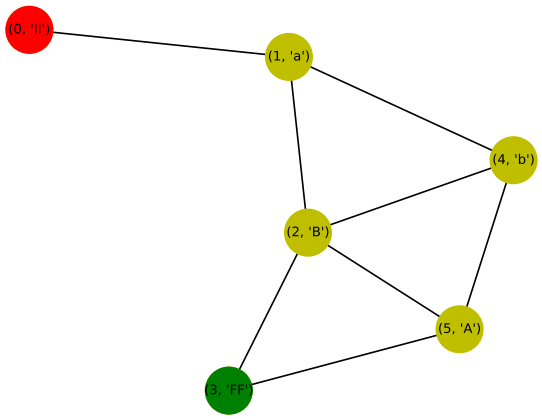

Figura 1: Exemplo de um labirinto representado em grafo

2 Módulos

O programa desenvolvido é dividido em três partes. Um módulo para leitura dos dados de entrada, um contendo o Tipo Abstrato de Dados desenvolvido para representar o labirinto e as estruturas necessárias para obter a solução do problema e, finalmente, um programa principal que engloba todas as implementações. A seguir, descreve-se em detalhes cada uma das partes do projeto.

2.1 Tipo Abstrato de Dados

Para representar o labirinto por meio dos grafos, utilizou-se as estruturas descritas nas Tabelas 1 e 2. Utilizou-se estas na adaptação desenvolvida do algoritmo Breadth-first search (BFS).

Tabela 1: Discriminação do conteúdo dos Grafos

V	Número de vértices
E	Número de aresta
K	Número de chaves
D	Número de portas
Keylog	Chaveiro com as chaves coletadas
Adj	Vetor de listas de adjacência que representa o grafo

Tabela 2: Discriminação do conteúdo de Adj

Primeiro	Primeira célula da lista de adjacência do vértice
Cor	Branco (não visitado), Cinza (encontrado), Preto (pesquisado)
Dist	Distância ao vértice inicial
Chave	Caso exista, contém a chave presente no vértice
Porta	Caso exista, contém a porta presente no vértice

Inicialmente, todos os vértices do grafo são inicializados a uma distância infinita do vértice inicial. A medida que o algoritmo BFS avança, a menor distância encontrada é atualizada em cada vértice do grafo. Ao final da execução, é possível descobrir a menor distância possível do vértice **FF** ao vértice **II**.

Descreve-se nas próximas secções, com mais detalhes, as funções desenvolvidas para trabalhar com os grafos e BFSs, bem como o algoritmo proposto para a solução do problema. Suprime-se os códigos na explicação para facilitar a visualização do leitor. Estes podem ser encontrados em arquivo anexo.

2.1.1 Funções da fila

Funções que permitem manipular as listas utilizados no algoritmo BFS. A descrição da funcionalidade de cada uma é vista na Tabela 3.

Trivialmente, tem-se complexidade $\mathcal{O}(1)$ para todas as funções aqui descritas.

Tabela 3: Descrição das funções de fila

FazQ	Cria uma fila
AdicionaQ	Adiciona elemento à fila
InicializaQ	Cria fila, adiciona elemento e inicializa distância inicial à 0
VaziaQ	Verifica se a lista encontra-se vazia
RetiraQ	Retira elemento da fila

2.1.2 Funções de Grafos

Funções que permitem manipular os grafos utilizadas no algoritmo BFS. A descrição da funcionalidade de cada uma é vista na Tabela 4.

Tabela 4: Descrição das funções de grafos

FazGrafo	Cria um grafo
AdicionaVertice	Adiciona vértice ao grafo
AdicionaChave	Adiciona chave ao vértice do grafo
AdicionaPorta	Adiciona porta ao vértice do grafo
GrafoCopia	Cria uma cópia do grafo

Trivialmente, tem-se complexidade $\mathcal{O}(1)$ para as funções AdicionaVertice. AdicionaChave e AdicionaPorta. Já para FazGrafo e GrafoCopia, tem-se complexidade $\mathcal{O}(V)$.

2.1.3 Funções de tratamento de chaves e portas

Funções que permitem tratar os vértices especiais que contém chaves ou portas. A descrição da funcionalidade de cada uma é vista na Tabela 5.

Tabela 5: Descrição das funções de tratamento de chaves e portas

ConvIndicePorta	Mapeia porta para índice do vetor ($A \to 0$, $B \to 1$, ...)
ConvIndiceChave	Mapeia chave para índice do vetor ($a \to 0$, $b \to 1$, ...)
PassaPorta	Retorna 1 caso seja possível passar pela porta, 0 c.c.
NovaChave	Retorna 1 caso exista nova chave no vértice, 0 c.c.
AtualizaKeyLog	Atualiza chaveiro com a nova chave encontrada

Trivialmente, tem-se complexidade $\mathcal{O}(1)$ para todas as funções aqui descritas.

2.1.4 Funções de busca em largura

São as funções mais relevantes do projeto. Realizam a busca do menor caminho de saída do labirinto possível, considerando chaves e portas. Utilizam adaptações do algoritmo BFS clássico. A descrição da funcionalidade de cada uma é vista na Tabela 6.

Tabela 6: Descrição das funções de tratamento de chaves e portas

bfs	Responsável por chamar as funções bfs1 e bfs2
bfs1	Procura a menor distância possível à saída, com no máximo uma chave
bfs2	Procura a menor distância possível à saida, com todas as chaves disponíveis, sendo limitada por uma profundidade de busca
Minimo	Verifica o mínimo entre dois valores, tratando -1 como infinito

O funcionamento do algoritmo de busca funciona da seguinte maneira:

1. Função bfs inicializa a distância mínima para infinito
2. Função bfs chama bfs1 para procurar a menor distância possível à saída, utilizando no máximo uma chave pelo caminho.
3. Dada a distância mínima encontrada por bfs1, limita-se o espaço de busca de bfs2 por esta distância. Visto que se deseja uma solução ótima, bfs2 não necessita procurar além da melhor solução já encontrada. Assim, bfs2 procura a melhor solução possível, limitada pela solução não ótima de bfs1, utilizando todas as chaves disponíveis no caminho.
4. Dentre as soluções encontradas por bfs1 e bfs2, a função bfs analisa a melhor solução com a chamada de Mínimo.
5. Então, bfs retorna a melhor solução encontrada no passo anterior.

Para as funções bfs1 e bfs2, o pior caso possível é dado por $\mathcal{O}(K! \times (V + E))$. Entretanto, espera-se que o algoritmo bfs1 limite a ordem de bfs2, de acordo com a solução não ótima encontrada. Como a ordem da função bfs é dada pela soma das ordens de bfs1 e bfs2, temos que a ordem de bfs é $\mathcal{O}(K! \times (V + E))$. Finalmente, a ordem de Minimo é trivialmente $\mathcal{O}(1)$.

2.1.5 Funções de desalocar memória

Funções responsáveis por liberar toda a memoria dinâmica utilizada, evitando *leaks* de memoria. Estas são descritas na Tabela 7.

Tabela 7: Descrição das funções de tratamento de chaves e portas

freeQ	Libera memória utilizada na lista da BFS
freelista	Libera memória utilizada na lista de adjacência
freeG	Libera memória utilizada no grafo

Tem-se complexidade $\mathcal{O}(V)$ no pior caso destas funções.

2.2 Leitura

Nesta seção, desenvolveu-se as funções responsáveis pela interpretação dos dados de entrada que definem o ambiente do labirinto. Vê-se a seguir a descrição da funcionalidade e ordem de complexidade de cada uma delas.

2.2.1 Função ComparaDados

Função responsável por tratar as entradas de texto escritas pelo usuário. Seta uma flag para cada um dos comandos digitados, de modo que se possa imprimir ao final da execução do programa somente as informações solicitadas. Também é tratado um possível erro de digitação, em que um comando invalido é inserido.

Tem-se trivialmente complexidade $\mathcal{O}(1)$.

2.2.2 LeituraDeEntrada

Para ser possível a manipulação das entradas, criamos o arquivo leitura o qual é responsável pela recepção dos dados e manipulação dos mesmos. Conforme consta nas definições de entradas do projeto, receberemos strings as quais selecionarão as saídas desejadas. Tais strings serão "passos"para que a saída seja a quantidade mínima de passos necessários para sair do labirinto, "tempo"para imprimir o tempo de execução do programa e "tudo"para que seja impresso todas as opções anteriores. Para a manipulação das entradas em strings, utilizamos a função *strtok* para separar as palavras recebidas e chamamos a função "ComparaDados"para cada palavra.

Tem-se complexidade $\mathcal{O}(E + K + D)$.

3 Main

Módulo principal do programa. Responsável por unir todas as funções desenvolvidas para implementar a funcionalidade final desejada.

```
int main(int argc, char const *argv[]) {
  // Inicializa variaveis do programa
  TipoFlags Flag;
  struct timeval beginTime, endTime, beginT, endT;
  unsigned long time1, time2;
  int ret, s, g;

  gettimeofday(&beginTime, NULL);
  Flag.Passos = 0;
  Flag.Tempo = 0;

  Grafo G;
  G = LeituraDeEntrada(&Flag, &s, &g);
  gettimeofday(&endTime, NULL);

  // Processa dados obtidos no procedimento anterior
  // Obtem tempo de computacao
  gettimeofday(&beginT, NULL);
  ret = bfs(G, s, g);
  gettimeofday(&endT, NULL);

  // Imprime resultados solicitados pelo usuario de acordo com as flags obtidas previamente
  if(Flag.Passos){
      printf("%d\n", ret);
  }
  if(Flag.Tempo){
    time1 = ((endT.tv_sec * 1000000 + endT.tv_usec) - (beginT.tv_sec * 1000000 + beginT.tv_usec));
    time2 = ((endTime.tv_sec * 1000000 + endTime.tv_usec) - (beginTime.tv_sec * 1000000 + beginTime.tv_usec));
    printf("%ld\n",time1);   // Tempo de computacao
```

```
    printf("%ld\n",time2);   // Tempo de entrada e alocacao de estruturas
  }


  freeG(G);
  // Libera memoria utilizada no programa
  printf("\tTEMPO DE COMPUTAÇÃO:%ld\n",time1);
  return 0;                 // Fim de programa
}
```

Tem-se complexidade $\mathcal{O}(K! \times (V + E))$, já que é a função bfs que define a ordem do programa a ser impressa.

4 Resultados

Com o programa completo, simulou-o em diversas situações. Para averiguar seu funcionamento, utilizou-se de exemplos de grafos de diferentes tamanhos e complexidades, comparando a solução encontrada com a resposta correta. Viu-se que para todos os casos o programa foi capaz de encontrar a solução correta com relativa facilidade.

Os resultados obtidos são resumidos na Tabela 9, em que se observa os diferentes tamanhos dos grafos de entrada e os respectivos tempos de computação. É evidente que, devido à otimização realizada, buscando limitar o espaço de busca da BFS, obteve-se resultados muito superiores à ordem do pior caso. Para efeitos de comparação, removeu-se a otimização de limitação do espaço de busca da bfs2 pela bfs1. Estes resultados são apresentados na Tabela 10. O tamanho de cada entrada é descrito na Tabela 8.

Tabela 8: Descrição das entradas

Entrada	1	2	3	4	5	6	7	8	9	10	11	12
Vértices	6	7	12	7	14	26	22	21	318	5000	5	4500
Arestas	8	6	21	11	25	37	27	30	743	21241	4	20274
Chaves	2	2	2	2	6	8	2	2	20	8	1	5
Portas	2	2	3	2	6	8	3	7	158	300	2	10

É evidente que os resultados apresentados na Tabela 9 foram muito superiores aos da Tabela 10. De fato, percebe-se que no programa com otimização a Entrada 9 foi concluída em apenas 226 mil µs, enquanto sem a otimização o programa levou mais de um dia para rodar e teve sua execução interrompida. Outro indicador importante é a memória utilizada em cada caso. À Entrada 10, por exemplo, apresentou-se uma melhoria de 680% no uso de memória.

De fato, observa-se pela análise de ordem o motivo da demora excessiva para a Entrada 9. Neste caso, como $K = 20$ e $V = 743$, tem-se complexidade na ordem de 10^{21}. De fato, a execução deste programa demoraria um tempo fora de qualquer limite aceitável. Por outro lado, para a entrada 10, como $K = 8$ e $V = 21241$, tem-se complexidade na ordem de 10^9. Observe que a ordem dos dois exemplos é extremamente diferente, justificando o motivo da Entrada 10 ter funcionado para ambos os programas. Após a otimização, apesar da ordem de complexidade dos dois casos ser mantida, o caso médio é extremamente

Tabela 9: Resultados Com Otimização

Entrada	Mallocs	Bytes	Tempo Total
1	71	7.004	33
2	69	7.116	28,6
3	165	9.646	41,4
4	105	7.846	28,2
5	281	14.474	65,8
6	3371	132.944	956
7	245	12.606	64,4
8	191	11.084	42,4
9	1.255.754	79.772.492	226.741,2
10	225.319	7.449.936	64.370,0
11	22	5.650	5,6
12	63.074	1.662.206	11.593

Tabela 10: Resultados Sem Otimização

Entrada	Mallocs	Bytes	Tempo Total
1	45	6.328	120
2	44	6.408	130
3	125	8.732	160
4	80	7.292	140
5	649	27.208	320
6	6127	290.384	1.950
7	177	11.004	180
8	121	8.984	160
9	*	*	*
10	118.856.752	5.068.421.992	49.579.813
11	19	5.602	120
12	553.895	21.286.143	307.326

diferente, favorecendo uma execução possivelmente rápida e efetiva. Destaca-se que para uma análise de ordem qualquer, é importante observar que dada uma mesma ordem entre dois algoritmos, estes podem se comportar de maneiras extremamente diferentes. Pode-se explicar tais comportamentos diferenciados tanto pelo caso médio como pela constante multiplicativa desconsiderada na análise de ordem.