

Felipe Roza Bonetti (12011BCC032), Pedro Henrique Marra Araújo (12011BCC008)

Implementação de Grafos em C para Recursos do Moodle

Uberlândia, Minas Gerais

2022

Sumário

Objetivos Gerais	4
Objetivos Específicos	4
Repositório	4
TADs	4
Vértice	5
Pilha de Vértices	6
Lista de Vértices	8
Lista de Listas de Vértices	9
Grafo	10
Comentário sobre os Algoritmos	16
Algoritmo de Kosaraju	17
Testando as Funções	20

Objetivos Gerais

Esse trabalho teve como objetivo geral a criação de um tipo abstrato de dados de implementação de grafos direcionados e ponderados de estrutura fixa, a qual será explicitada adiante, a fim de modelar um esquemático de recursos disponibilizados na plataforma Moodle.

Objetivos Específicos

Esse trabalho teve como objetivo específico a criação de um programa em linguagem C que estrutura o formato de grafos supracitados e que implementa funções básicas de qualquer grafo e funções específicas. Para tanto, foram criados os tipos abstratos vértice, pilha de vértices, lista de vértices e lista de listas de vértices. Além disso, foi criado um menu para a manipulação de qualquer grafo direcionado e ponderado com tal estrutura.

Repositório

Todos os códigos foram disponibilizados em um repositório no GitHub, que pode ser acessado clicando aqui. Nele, encontra-se, também, um executável com um menu de aplicação do código.

TADs

A seguir, uma breve documentação de todos os TADs criados para o trabalho, incluindo o próprio TAD do grafo em si.

Vértice

Dados: define uma estrutura (`struct vertice`) como vértice para o TAD grafo consistindo de um nome (`char[]`), de um tipo de pasta (`int`) e de uma ação (`int`) e também define um ponteiro para essa estrutura (`Vertice`). Escolheu-se usar esse apontamento, pois, dependendo da estrutura do vértice e seu tamanho, fazer, sempre, as funções por cópia poderiam ficar caras em relação ao espaço (por exemplo, se o tamanho do nome for colocado muito grande ou for colocado mais membros na estrutura do vértice). Para os TADs a seguir, vértice refere-se, em geral, ao ponteiro para essa estrutura do vértice.

Consistência dos Dados: não há nenhuma consistência para a entrada de dados, ficando a caráter do usuário verificá-las (até mesmo porque o esquemático do Moodle apresentado não tem todas as opções possíveis de tipo e ação, podendo ter mais outros tipos ou ações não listadas naquele grafo). Como os tipos de pastas e os tipos de ação são valores categóricos, colocou-se como inteiros e sugerimos usar (em relação ao informativo do Moodle, novamente, que talvez não tenha todos tipos ou ações possíveis):

Tipo: (1) Arquivo - (2) Página - (3) Pasta - (4) Tarefa - (5) URL.

Ação: (1) Download - (2) Upload - (3) Visualizar.

- `iguaisVertices`

Entrada: dois vértices.

Processo: verifica igualdade entre vértices (igualdade entre as estruturas de vértice).

Saída: 1, se iguais. 0, caso contrário.

Pilha de Vértices

Dados: define uma estrutura (`struct No`) consistindo de uma estrutura (`struct vertice`) — definida no TAD Vértice —, um ponteiro para o próximo nó (`struct No*`) e uma Pilha como (`struct No*`).

- **IniciaPilha**

Entrada: nenhuma.

Processo: cria uma pilha vazia.

Saída: retorna uma pilha vazia.

- **VaziaPilha**

Entrada: uma pilha válida.

Processo: valida a pilha e, caso válida, retorna se a pilha está vazia.

Saída: 1, pilha vazia. 0, caso contrário.

- **Topo**

Entrada: uma pilha válida e um vértice válido.

Processo: lê, se possível, o topo da pilha e copia seu valor para o vértice.

Saída: 1, se sucesso em ler topo. 0, caso contrário. Retorna implicitamente o valor do topo no vértice de entrada.

- **Empilha**

Entrada: um endereço válido de pilha e um vértice.

Processo: empilha o vértice na pilha de vértices.

Saída: 1, se sucesso em empilhar. 0, caso contrário.

- **Desempilha**

Entrada: um endereço válido de pilha e um vértice.

Processo: desempilha o topo da pilha e copia sua informação para o vértice.

Saída: 1, se sucesso em desempilhar. 0, caso contrário. Retorna implicitamente o valor do topo no vértice de entrada.

- **EsvaziaPilha**

Entrada: um endereço válido de pilha.

Processo: esvazia a pilha.

Saída: 1, se sucesso em esvaziar. 0, caso contrário.

Lista de Vértices

Dados: define uma estrutura (`struct No`) consistindo de uma estrutura (`struct vertice`) — definida no TAD Vértice —, um ponteiro para o próximo nó (`struct No*`) e uma `Lista` como (`struct No*`). É uma lista de vértices necessária para o TAD de lista de listas de vértices (utilizado para a função Kosaraju).

- **IniciaLista**

Entrada: nenhuma.

Processo: cria uma lista vazia.

Saída: retorna uma lista vazia.

- **VaziaLista**

Entrada: uma lista válida.

Processo: valida a lista e, caso válida, retorna se a lista está vazia.

Saída: 1, lista vazia. 0, caso contrário.

- **Inserir**

Entrada: um endereço de lista válida e um vértice válido.

Processo: insere, se possível, o vértice na lista de vértices.

Saída: 1, se sucesso em inserir vértice. 0, caso contrário.

- **Remove**

Entrada: um endereço de lista válida e um vértice válido.

Processo: remove, se possível, o vértice na lista de vértices.

Saída: 1, se sucesso em remover vértice. 0, caso contrário.

- **EsvaziaLista**

Entrada: um endereço válido de lista.

Processo: esvazia a lista.

Saída: 1, se sucesso em esvaziar. 0, caso contrário.

Lista de Listas de Vértices

Dados: define uma estrutura (`struct NoLista`) consistindo de uma (`Lista`) — definida no TAD Lista —, um ponteiro para o próximo nó (`struct NoLista*`) e uma lista de listas de vértices (`ListaLista`). É utilizada, essencialmente, para a função Kosaraju.

- `IniciaListaLista`

Entrada: nenhuma.

Processo: cria uma lista de listas vazia.

Saída: retorna uma lista de listas vazia.

- `VaziaListaLista`

Entrada: uma lista de listas válida.

Processo: valida a lista de listas e, caso válida, retorna se a lista de listas está vazia.

Saída: 1, se lista de listas vazia. 0, caso contrário.

- `InsererLista`

Entrada: um endereço de lista de listas válida e uma lista de válida.

Processo: insere, se possível, a lista na lista de listas (não faz cópia da lista, coloca a própria lista de entrada na lista de listas).

Saída: 1, se sucesso em inserir a lista. 0, caso contrário.

- `EsvaziaListaLista`

Entrada: um endereço válido de lista de listas.

Processo: esvazia a lista de listas (esvazia todas as listas).

Saída: 1, se sucesso em esvaziar. 0, caso contrário.

Grafo

Dados: foram definidos os seguintes dados.

1. **struct grafo:** uma estrutura do grafo. Consiste no número de vértices (**int**), no número de arcos (**int**) e uma lista de vértices (**ListaVertices**).
 2. **struct noVert:** nó da lista de vértices do grafo. Possui a informação do vértice (**struct vertice**), a lista de adjacência desse vértice (**ListaAdjacencia**) e o ponteiro para o próximo nó (**struct noVert***).
 3. **ListaVertices:** ponteiro para um nó de vértice.
 4. **struct noAdj:** nó da lista de adjacência de um determinado vértice. Possui um vértice, um peso da aresta e um nó para o próximo nó (**struct noAdj***).
 5. **ListaAdjacencia:** ponteiro para um nó da lista de adjacência.
 6. **struct visitaTempo:** estrutura para guardar os tempos das ordenações topológicas. Possui um vértice, seu tempo de descoberta na ordenação (**int**) e seu tempo de finalização (**int**).
 7. **Tempo:** um ponteiro para a (**struct visitaTempo**) e é, em essência, um vetor de resultados de visitação da ordenação topológica.
 8. **struct distancia:** estrutura para guardar as distâncias de um vértice para o algoritmo de Dijkstra. Possui um vértice e sua distância (**int**).
 9. **Distancia:** um ponteiro para a (**struct distancia**) e é, em essência, um vetor de resultados de distâncias do algoritmo de Dijkstra.
- **criarGrafoNulo**

Entrada: nenhuma.

Processo: cria um grafo nulo (nenhum vértice e nenhuma aresta).

Saída: um grafo nulo.
 - **EhNulo**

Entrada: um grafo válido.

Processo: verifica se o grafo é nulo.

Saída: 1, se grafo válido é nulo. 0, caso contrário.

- **criarGrafoVazio**

Entrada: um vetor de vértices e seu tamanho.

Processo: cria um grafo vazio (com os vértices do vetor de vértices).

Saída: um grafo vazio.

- **EsvaziaGrafo**

Entrada: um grafo válido.

Processo: esvazia o grafo (deixa na condição de vazio).

Saída: 1, se esvaziado com sucesso. 0, caso contrário.

- **AnulaGrafo**

Entrada: um grafo válido.

Processo: anula o grafo (deixa na condição de nulo).

Saída: 1, se anulado com sucesso. 0, caso contrário.

- **ApagaGrafo**

Entrada: um endereço válido de grafo.

Processo: apaga o grafo.

Saída: 1, se apagado com sucesso. 0, caso contrário.

- **inserirArco**

Entrada: um grafo válido, um vértice **v1**, outro vértice **v2** e o peso do arco entre eles.

Processo: insere, se possível, o arco $v1 \rightarrow v2$ no grafo, implementada de forma a não inserir arcos múltiplos.

Saída: 1, se inserido com sucesso. 0, caso contrário.

- **inserirVertice**

Entrada: um grafo válido e um vértice válido.

Processo: insere, se possível, o vértice no grafo.

Saída: 1, se inserido com sucesso. 0, caso contrário.

- **removerArco**

Entrada: um grafo válido, um vértice **v1** e outro vértice **v2**.

Processo: remove, se possível, o arco $v1 \rightarrow v2$.

Saída: 1, se sucesso. 0, caso contrário.

- **removerVertice**

Entrada: um grafo válido e um vértice.

Processo: remove, se possível, o vértice do grafo e os arcos incidentes a ele.

Saída: 1, se removido com sucesso. 0, caso contrário.

- **CopiaGrafo**

Entrada: um grafo válido e um endereço válido de grafo.

Processo: copia, se possível, grafo entrado para o endereço de grafo entrado.

Saída: 1, se sucesso em fazer cópia do grafo. 0, caso contrário.

- **grauVertice**

Entrada: um grafo válido, um vértice válido e um endereço de inteiro.

Processo: encontra, se possível, o grau do vértice no grafo e copia o valor para o endereço entrado.

Saída: 1, se possível encontrar e copiar grau. 0, caso contrário. Retorna, implicitamente, o grau no endereço de entrada.

- **maiorGrau**

Entrada: um grafo válido, um vértice e um endereço de inteiro.

Processo: busca, se possível, o vértice de maior grau do grafo, faz uma cópia dele para o vértice entrado.

Saída: 1, se possível encontrar o vértice de maior grau. 0, caso contrário. Retorna, implicitamente, o grau desse vértice de maior grau no endereço de inteiro entrado.

- **BuscaEmProfundidade**

Entrada: um grafo válido, um vértice válido, um endereço de vértice e um endereço de inteiro.

Processo: busca, se possível, todos os vértices atingíveis no grafo a partir do vértice entrado em uma busca em profundidade.

Saída: 1, se sucesso em colocar a ordem dos vértices encontrados na busca no vetor de entrada. 0, caso contrário. Retorna, também, implicitamente o tamanho do vetor no endereço de inteiro de entrada.

- **BuscaEmProfundidadeTempo**

Entrada: um grafo válido, um vértice válido, um endereço de tempo e um endereço de inteiro.

Processo: busca, se possível, todos os vértices atingíveis no grafo a partir do vértice entrado em uma busca em profundidade e faz uma ordenação topológica deles.

Saída: 1, se sucesso em colocar a ordenação topológica encontrada no vetor de entrada. 0, caso contrário. Retorna, também, implicitamente o tamanho do vetor no endereço de inteiro de entrada.

- **existeCaminho**

Entrada: um grafo válido, um vértice **v1** e outro vértice **v2**.

Processo: verifica, se possível, se há um caminho **v1** \rightarrow **v2** (caminho entre **v1** e **v2**). Note que não verifica se existe, necessariamente, caminho entre **v2** e **v1**, uma vez que o grafo é direcionado.

Saída: 1, se sucesso em encontrar o caminho. 0, caso contrário.

- Transposto

Entrada: um grafo válido e um endereço válido de grafo.

Processo: faz uma cópia, se possível, do transposto do grafo (um grafo com todas os arcos invertidos) entrado para o endereço de grafo entrado.

Saída: 1, se sucesso em fazer a cópia do grafo transposto. 0, caso contrário.

- BuscaTodosTempos

Entrada: um grafo válido e um endereço de tempo.

Processo: busca, se possível, todos os vértices do grafo a partir do primeiro vértice da lista de vértices em uma busca em profundidade e faz uma ordenação topológica deles. Nesse caso, busca até mesmo em grafos desconexos ou fracamente conexos (nesse caso, começa da componente fortemente conexa do primeiro vértice e vai buscando todas as outras componentes desse tipo até não ter mais vértice não ordenado topologicamente).

1, se sucesso em colocar a ordenação topológica encontrada no vetor de entrada. 0, caso contrário. Não retorna, implicitamente, o tamanho desse vetor, pois esse é do tamanho da quantidade de vértices do grafo no momento da busca, já que ele, necessariamente, ordena todos os vértices, mesmo aqueles não conectados entre si.

- Kosaraju

Entrada: um grafo válido e um endereço de uma lista de listas de vértices.

Processo: encontra, se possível, todas as componentes fortemente conectadas do grafo de acordo com o algoritmo de Kosaraju.

Saída: 1, se sucesso em encontrar as componentes. 0, caso contrário. Retorna, implicitamente, cada componente fortemente conectada como uma lista de vértices da lista de listas de vértices endereçada na entrada.

- Dijkstra

Entrada: um grafo válido, um vértice e um endereço de distância.

Processo: busca, se possível, as distâncias entre o vértice entrado e todos os outros vértices do grafo de acordo com o algoritmo de Dijkstra.

Saída: 1, se sucesso em colocar as distâncias como um vetor nesse endereço. 0, caso contrário. Não retorna, implicitamente, o tamanho desse vetor, pois esse tamanho é a quantidade de vértices do grafo menos um — uma vez que o algoritmo, necessariamente, retorna as distâncias de todos os vértices diferentes do entrado. Caso o vértice entrado não tenha um caminho para outro determinado vértice, o valor referente a essa distância, por padrão, é `INT_MAX`.

Comentário sobre os Algoritmos

As funcionalidades solicitadas foram implementadas pelas seguintes funções:

- (a) *“Criação do grafo, com inserção/remoção de vértices e arestas”*: as de inserção já estavam implementadas, só mudando um pouco elas para apontar erro caso algo de errado ocorra. As funções de remoção foram implementadas de forma análoga, de forma a manter correta a estrutura do grafo (por exemplo, ao remover um vértice, todos os arcos de entrada e saída deste são, também, excluídos).
- (b) *“Busca do vértice de maior grau, que, para a trilha, representa um recurso com peso importante no fluxo”*: `maiorGrau`.
- (c) *“Dados dois recursos (vértices), verificar se existe caminho entre os mesmos”*: `existeCaminho` retorna somente se existe, não volta o caminho (não solicitado).
- (d) *“A partir de um vértice, encontrar o menor caminho para os outros vértices a ele conectados”*: `Dijkstra` (como já foi discutido em sala de aula, não explicaremos ele aqui, uma vez que a implementação foi bem parecida com a mostrada em sala de aula).
- (e) *“Usando busca em profundidade, encontrar recursos fortemente conectados”*: `Kosaraju`, que utiliza buscas em profundidade para achar esses recursos fortemente conectados (esse, sim, será explicado a diante).
- (f) *“Impressão do grafo”*: `imprimirListaAdj`, que imprime a lista de adjacência do grafo, que nada mais é que o grafo por si só.

Algoritmo de Kosaraju

Uma componente fortemente conectada de um grafo direcionado é um subconjunto de vértices que, entre quaisquer dois vértices v_1 e v_2 , existe um caminho entre v_1 e v_2 . Tome como exemplo o seguinte grafo direcionado ponderado.

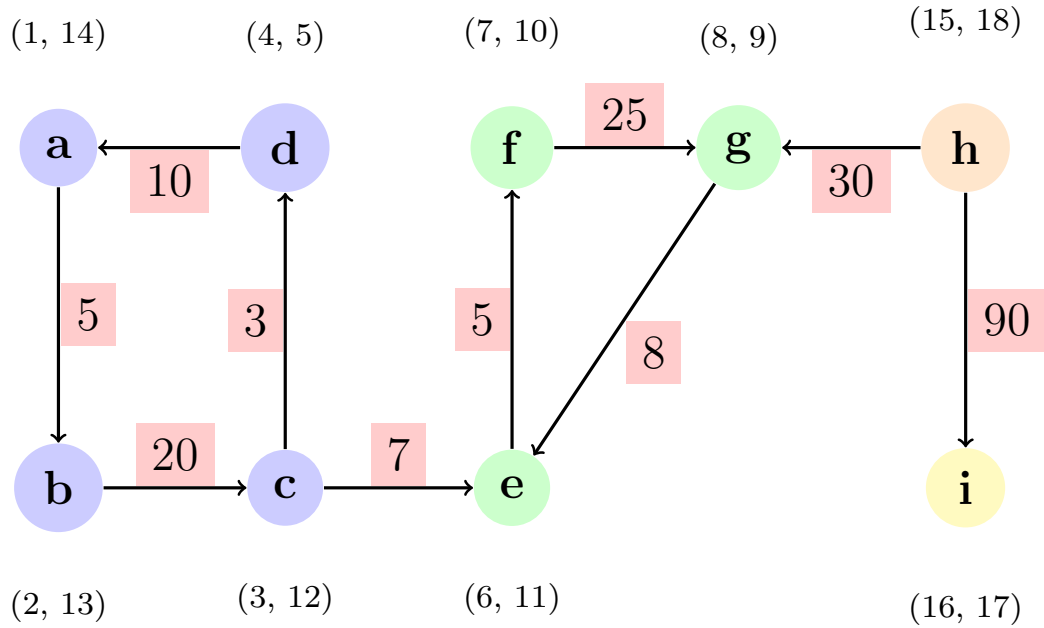


Figura 1: grafo direcionado ponderado, disponibilizado no arquivo `3.in`, com as componentes fortemente conexas marcadas por cor e com os tempos da ordenação topológica.

Nesse exemplo, temos as seguintes componentes fortemente conexas, marcadas com diferentes cores: $\{a, b, c, d\}$, $\{e, f, g\}$, $\{h\}$ e $\{i\}$. Por exemplo, de a , conseguimos ir para b , c e d (e a recíproca para todos vértices é válida). Também, por exemplo, de i , não conseguimos ir para nenhum outro vértice de forma que, desse outro vértice, consigamos voltar para i , fazendo com que o i sozinho seja uma componente fortemente conectada.

O algoritmo de Kosaraju aproveita-se do fato que, dado o grafo transposto (grafo com os mesmos vértices, porém com todos os arcos invertidos)

de determinado grafo, conseguimos encontrar essas componentes de forma sistemática com buscas em profundidade em vértices não-visitados ordenados por tempo mínimo de finalização da ordenação topológica no grafo inicial. Abaixo, o grafo transposto da Figura 1 e a pilha de vértices em relação ao tempo de finalização de busca em profundidade.

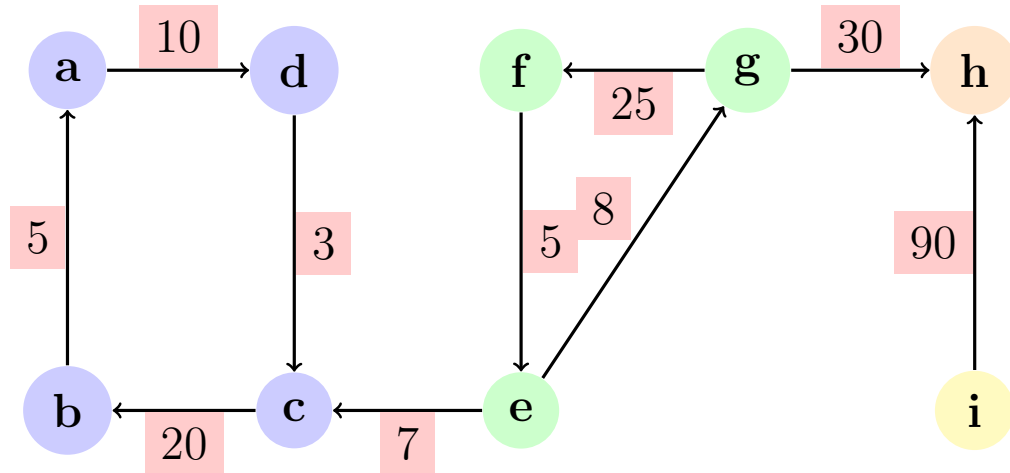


Figura 2: grafo transposto da Figura 1.

h
i
a
b
c
e
f
g
d

Figura 3: Pilha dos vértices a serem feitos buscas em profundidade para encontrar as componentes fortemente conectadas no grafo exemplo da Figura 1.

Para cada vértice da pilha da Figura 3, desempilhe e faça uma busca em

profundidade a partir desse vértice. Se aparecer algum vértice ainda não-visitado nessa busca, esse conjunto de vértices buscado é uma componente fortemente conexa ainda não encontrada.

Por exemplo, desempilhe h e faça a busca a partir de h , a qual encontra só o próprio vértice h . Como h foi visitado pela primeira vez agora, $\{h\}$ é uma componente fortemente conectada. Desempilhe i e faça a busca a partir de h , a qual encontra i e h . Como encontrou i , ainda não-visitado, $\{i, h\}$ é outra componente. Desempilhe a e faça a busca a partir de a , a qual encontra $\{a, b, c, d\}$. Como encontrou quatro vértices ainda não-visitados, $\{a, b, c, d\}$ é outra componente. Desempilhe b e faça a busca a partir de b , a qual encontra $\{a, b, c, d\}$. Como não encontrou nenhum vértice não-visitado (visitou todos ao desempilhar a), não adiciona esse conjunto à lista de componentes fortemente conectadas. Desempilha e e faça a busca a partir de e , a qual encontra $\{e, f, g\}$. Como todos vértices desse conjunto não foram visitados, adicione esse conjunto como uma componente na lista. Agora, todos os próximos vértices não encontrarão componentes ainda não descobertas. O algoritmo para quando a pilha está vazia.

O algoritmo, nesse exemplo, resulta na lista $[\{i\}, \{h\}, \{a, b, c, d\}, \{e, f, g\}]$ de componentes fortemente conectadas.

Testando as Funções

Além do grafo da Figura 1, utilizaremos outros dois grafos como exemplo. Seguem eles.

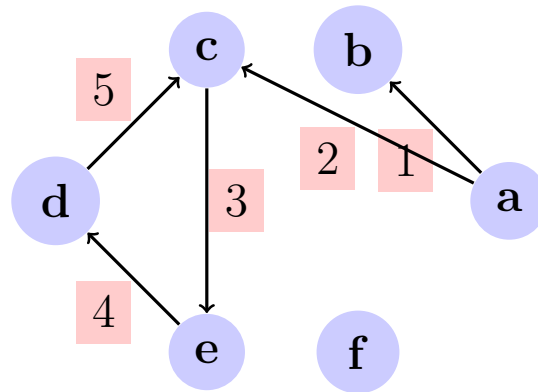


Figura 4: segundo grafo para testes, disponibilizado em 1.in.

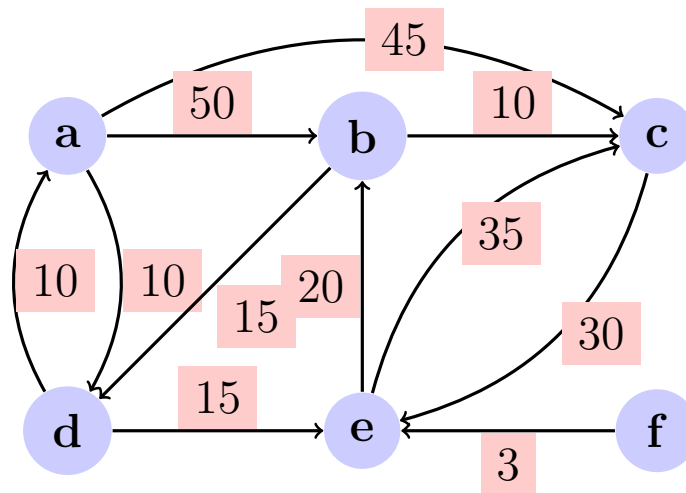


Figura 5: terceiro grafo para testes, disponibilizado em 2.in.

Para testar, criamos três grafos (disponibilizados em `1.in`, `2.in` e `3.in`), relevando o valor do tipo e ação (só o nome para nós aqui é importante, para facilitar os exemplos). Todas as funções foram disponibilizadas a partir do menu disponibilizado em `main.exe`. Abaixo uma figura desse menu.

- [1] Criar grafo nulo.
- [2] Criar grafo por arquivo.
- [3] Inserir v|rtice (v).
- [4] Inserir arco (v -> w).
- [5] Remover v|rtice (v).
- [6] Remover arco (v -> w).
- [7] Imprimir grafo.
- [8] Ver v|rtice de maior grau.
- [9] Verificar se existe caminho entre v e w.
- [10] Dado v, verificar todos os menores caminhos a outros v|rtices.
- [11] Busca em profundidade a partir de v.
- [12] Verificar recursos fortemente conectados.
- [13] Esvaziar grafo (deixar na condi|ção de grafo vazio).
- [14] Anular grafo (deixar na condi|ção de grafo nulo).
- [15] Destruir grafo.
- [16] Sair do sistema.

Digite uma das opcoes:

O usuário pode tanto criar um grafo nulo (opção 1) e ir adicionando vértices e arestas a forma que desejar, ou criar um grafo a partir de um arquivo (opção 2). Se escolher criar a partir de arquivo, precisa entrar o diretório do arquivo, que deve ter a quantidade de vértices, quantidade de arestas, os vértices e as arestas (com peso), como abaixo.

```

6
5
f 10 11
a 1 2
b 3 4
c 4 5
d 6 7
e 8 9
a 1 2 1 b 3 4
a 1 2 2 c 4 5
c 4 5 3 e 8 9
e 8 9 4 d 6 7
d 6 7 5 c 4 5

```

A seguir, o exemplo do funcionamento de algumas funções. Testamos só a de maior grau, o algoritmo de Dijkstra e o algoritmo de Kosaraju, uma vez que essas três funções precisam de todas as outras funções como auxiliares, porém todas as funções disponibilizados no menu estão funcionando perfeitamente.

Entrando o grafo da Figura 4 (1.in), abaixo seguem os resultados da chamada de algumas funções.

`grau(("c", 4, 5)) = 3.`

Figura 6: Vértice de maior grau é o c.

`dist(("a", 1, 2), ("c", 4, 5)) = 2.`
`dist(("a", 1, 2), ("b", 3, 4)) = 1.`
`dist(("a", 1, 2), ("e", 8, 9)) = 5.`
`dist(("a", 1, 2), ("d", 6, 7)) = 9.`
`dist(("a", 1, 2), ("f", 10, 11)) = Infinito.`

Figura 7: Distâncias de todos os vértices em relação ao vértice a usando Dijkstra.

`Componente[1] = { ("f", 10, 11) }`
`Componente[2] = { ("e", 8, 9) ("d", 6, 7) ("c", 4, 5) }`
`Componente[3] = { ("b", 3, 4) }`
`Componente[4] = { ("a", 1, 2) }`

Figura 8: Achando as componentes fortemente conexas usando Kosaraju.

Entrando o grafo da Figura 5 (2.in), abaixo seguem os resultados da chamada de algumas funções.

```
grau(("e", 9, 10)) = 5.
```

Figura 9: Vértice de maior grau é o e.

```
dist(("a", 1, 2), ("d", 7, 8)) = 10.  
dist(("a", 1, 2), ("b", 3, 4)) = 45.  
dist(("a", 1, 2), ("c", 5, 6)) = 45.  
dist(("a", 1, 2), ("f", 11, 12)) = Infinito.  
dist(("a", 1, 2), ("e", 9, 10)) = 25.
```

Figura 10: Distâncias de todos os vértices em relação ao vértice a usando Dijkstra.

```
Componente[1] = { ("c", 5, 6) ("e", 9, 10) ("b", 3, 4) ("d", 7, 8) ("a", 1, 2) }  
Componente[2] = { ("f", 11, 12) }
```

Figura 11: Achando as componentes fortemente conexas usando Kosaraju.

Entrando o grafo da Figura 1 (3.in), abaixo seguem os resultados da chamada de algumas funções.

`grau(("g", 0, 0)) = 3.`

Figura 12: Vértice de maior grau é o g.

```
dist(("a", 0, 0), ("b", 0, 0)) = 5.  
dist(("a", 0, 0), ("i", 0, 0)) = Infinito.  
dist(("a", 0, 0), ("h", 0, 0)) = Infinito.  
dist(("a", 0, 0), ("g", 0, 0)) = 62.  
dist(("a", 0, 0), ("f", 0, 0)) = 37.  
dist(("a", 0, 0), ("e", 0, 0)) = 32.  
dist(("a", 0, 0), ("d", 0, 0)) = 28.  
dist(("a", 0, 0), ("c", 0, 0)) = 25.
```

Figura 13: Distâncias de todos os vértices em relação ao vértice a usando Dijkstra.

```
Componente[1] = { ("f", 0, 0) ("g", 0, 0) ("e", 0, 0) }  
Componente[2] = { ("b", 0, 0) ("c", 0, 0) ("d", 0, 0) ("a", 0, 0) }  
Componente[3] = { ("i", 0, 0) }  
Componente[4] = { ("h", 0, 0) }
```

Figura 14: Achando as componentes fortemente conexas usando Kosaraju.