Descrição das implementações:

Heap. C (Vanessa)

Função: MinHeap *criarHeap(int capacidade);

Objetivo:Criar e inicializar uma estrutura de dados do tipo heap mínimo (MinHeap).

Lógica de implementação:

- Alocação da estrutura principal:
  É alocada dinamicamente memória para a estrutura MinHeap que irá armazenar todos os dados da heap. Caso a alocação falhe, a função imprime uma mensagem de erro e retorna NULL.

- Inicialização dos atributos:
  tamanho: inicializado com 0, pois a heap começa vazia.
  capacidade: recebe o valor passado como argumento, que define o tamanho máximo do heap.

- Alocação dos vetores auxiliares:
  dados: vetor que armazenará os elementos do heap (pares de vértice e custo).
  posição: vetor que mantém a posição atual de cada vértice dentro do heap. Isso é útil para permitir acesso direto e rápido aos elementos.

- Verificação de alocação:
  Se a alocação de qualquer um dos vetores falhar, a função libera todas as memórias previamente alocadas e retorna NULL para evitar vazamento de memória.

- Inicialização do vetor de posições:
  Todos os valores do vetor posição são iniciados com -1, indicando que nenhum vértice foi inserido ainda.

- Retorno:
  A função retorna um ponteiro para o heap criada, pronta para ser utilizada em operações como inserção, remoção e atualização de custos.

Função: void inicializaHeap(MinHeap* h, int quant);

Objetivo: Reinicializa um heap mínimo (MinHeap) já criado, preparando-o para uma nova operação ou reutilização, sem a necessidade de desalocar e realocar memória.

Lógica de implementação:

- Verificação de validade da heap:
  A função verifica se o ponteiro h (referente ao heap) não é NULL. Isso evita erros ao tentar acessar memória inválida.

- Reinicialização do tamanho do heap:
  O campo tamanho da estrutura é redefinido para 0, esvaziando logicamente o heap. Isso indica que ele está novamente vazio.

- Reinicialização do vetor de posições:
  Um laço percorre todo o vetor posição, definindo todas as posições com -1. Isso significa que nenhum vértice está presente no heap no momento, preparando-a para futuras inserções.

Função: void trocar(ElementoHeap* a, ElementoHeap* b);

Objetivo: Realizar a troca de posição entre dois elementos da estrutura do heap, geralmente utilizada durante operações de ordenação como subir ou descer elementos na árvore binária do heap.

Lógica de implementação:

- Criação de uma variável temporária:
  Uma variável temp do tipo ElementoHeap é criada para armazenar temporariamente o conteúdo de a.

- Troca dos conteúdos:
  O conteúdo de a é sobrescrito com o conteúdo de b.
  Em seguida, b recebe o conteúdo previamente armazenado em temp.
  Esse processo efetua a troca de valores entre os dois ponteiros.

Função: void subir(MinHeap* h, int i);

Objetivo: Restaurar a propriedade do heap mínimo ao mover um elemento para cima na estrutura do heap, caso ele tenha um custo menor que o de seu pai.

Lógica de implementação:

- Verificação da posição:
  A função entra em um laço while que continua enquanto:
  i > 0 (o elemento não está na raiz), e o custo do elemento atual é menor que o custo do seu pai ((i - 1) / 2).

- Atualização das posições:

  Antes de trocar os elementos:
  A posição atual do vértice em h->posicao é atualizada para refletir a nova posição que ele assumirá após a troca.
  O mesmo é feito para o pai, garantindo que o vetor de posições continue correto após a movimentação.

  Troca dos elementos:
  Os elementos nas posições i (filho) e (i - 1) / 2 (pai) são trocados utilizando a função trocar.

- Atualização do índice:
  Após a troca, i é atualizado para a posição do pai, e o processo se repete caso o novo custo ainda seja menor do que o do novo pai.

Função: void descer(MinHeap* h, int i);

Objetivo: Manter a propriedade de heap mínima ao mover um elemento para baixo na estrutura do heap, caso ele tenha um custo maior que o de um de seus filhos.

Lógica de implementação:

- Inicialização de variáveis auxiliares:
  menor: inicialmente recebe o índice atual (i), ou seja, assume que o elemento está na posição correta.
  e e d: são os índices dos filhos esquerdo e direito, calculados como 2 * i + 1 e 2 * i + 2, respectivamente.

- Comparações com os filhos:
  O algoritmo verifica se os filhos estão dentro do limite do heap (h->tamanho) e compara seus custos com o do elemento atual:
  Se o filho esquerdo tem custo menor, menor é atualizado com o índice e.
  Se o filho direito tem custo menor, menor é atualizado com d.

- Troca de posições se necessário:
  Se menor for diferente de i, significa que o elemento atual não é o menor entre ele e seus filhos. Nesse caso:
  As posições dos vértices afetados são atualizadas no vetor posição.
  Os elementos são trocados usando a função trocar.

- Chamada recursiva:
  Após a troca, a função é chamada novamente para a nova posição (menor) do elemento, garantindo que ele desça até a posição correta.

Função : int insereHeap(MinHeap *H, int valor, int dist);

Objetivo: Inserir um novo elemento (vértice e custo) no heap mínimo, mantendo a propriedade da estrutura após a inserção.

Lógica de implementação:

- Validação da estrutura do heap:
  A função verifica se o ponteiro H é nulo. Caso seja, significa que o heap não foi criado corretamente, então uma mensagem de erro é exibida e a função retorna 0 (falha).

- Verificação de capacidade:
  Se o heap já atingiu sua capacidade máxima (ou seja, está cheio), a inserção não é possível. Nesse caso, a função exibe uma mensagem de erro e também retorna 0.

- Inserção do novo elemento:
  O novo elemento (representado por um valor e um custo) é colocado na primeira posição livre, ou seja, no índice atual de H->tamanho.

- Atualização do vetor de posições:
  O vetor posição é atualizado para indicar que o vértice valor agora está armazenado no índice correspondente do vetor dados.

- Reposicionamento com subir:
  Após a inserção, o elemento pode não estar na posição correta em relação à propriedade de heap mínima. A função subir é chamada para garantir que, se o custo for menor que o do pai, o elemento "suba" até a posição correta.

- Incremento do tamanho:
  O contador tamanho do heap é incrementado para refletir a adição do novo elemento.

- Retorno de sucesso:
  A função retorna 1 para indicar que a inserção foi concluída com sucesso.

Função: ElementoHeap remover(MinHeap* h);

Objetivo: Remover e retornar o elemento com o menor custo do heap mínimo, que está sempre na raiz (posição 0), mantendo a propriedade do heap após a remoção.

Lógica de implementação:

- Verificação do estado do heap:
  A função verifica se o ponteiro h é nulo ou se o heap está vazio (tamanho == 0).
  Se qualquer dessas condições for verdadeira, retorna um elemento inválido ({-1, -1})
  e exibe uma mensagem de erro indicando que o heap está vazio.

- Captura do elemento mínimo:
  O elemento mínimo está sempre na raiz do heap, ou seja, na posição 0 do vetor dados.
  Esse elemento é armazenado em min para ser retornado depois.

- Atualização do vetor de posições:
  A posição do vértice removido é marcada como -1 no vetor posicao, indicando que
  ele não está mais no heap.

- Reposicionamento do último elemento:
  O último elemento do heap (na posição tamanho - 1) é movido para a raiz (posição 0)
  para ocupar o espaço deixado pela remoção.

- Atualização do vetor de posições para o novo elemento da raiz:
  Se o heap ainda tiver elementos (tamanho > 0), a posição do novo elemento na raiz é
  atualizada para 0.

- Redução do tamanho:
  O tamanho do heap é decrementado, pois um elemento foi removido.

- Restauração da propriedade do heap:
  A função descer é chamada a partir da raiz para mover o elemento recém-colocado
  para baixo, caso seja necessário, garantindo que a propriedade de heap mínima seja
  mantida.

- Retorno do elemento removido:
  Por fim, a função retorna o elemento mínimo que foi removido da estrutura.

Função: void atualizaHeap(MinHeap* h, int v, int novoCusto);

Objetivo: Atualizar o custo de um elemento específico no heap e ajustar sua posição para manter a propriedade de heap mínimo

Lógica de implementação:

- Verificação inicial:
  A função verifica se o ponteiro h é nulo ou se o vértice v não está presente no heap
  (posição igual a -1).

Caso uma dessas condições seja verdadeira, exibe uma mensagem de erro informando que o vértice não foi encontrado e retorna sem fazer alterações.

- Localização do elemento:
  Obtém o índice i do elemento v no vetor dados, usando o vetor posição.

- Atualização do custo:
  O custo do elemento na posição i é atualizado para novoCusto.

- Reposicionamento do elemento:
  Chama a função subir para mover o elemento para cima, caso seu novo custo seja menor que o do pai, garantindo a manutenção da propriedade de heap mínimo.

Fila. C (Carina)

Função: fila* criarFila(int tamanho);

Objetivo: Criar e inicializar uma fila dinâmica capaz de armazenar até um número especificado de elementos.

Lógica de implementação:

- Alocação da estrutura fila:
  A função aloca memória para a estrutura fila, que vai representar a fila em si.

- Alocação do vetor de itens:
  Aloca um vetor de inteiros com tamanho igual ao parâmetro tamanho. Esse vetor será usado para armazenar os elementos da fila.

- Inicialização dos índices:

  início é inicializado com -1, indicando que a fila está vazia e não há um elemento inicial válido.
  fim também é inicializado com -1, indicando que não há elementos na fila.

- Retorno do ponteiro:
  A função retorna o ponteiro para a fila recém-criada e inicializada, pronta para operações de enfileiramento e desenfileiramento.

Função: int estaVazia(fila* f);

Objetivo: Verificar se a fila está vazia.

- Condição de fila vazia:
  A fila é considerada vazia quando o índice inicio está definido como -1. Isso significa que não há elementos armazenados.

- Retorno:
  A função retorna o resultado da expressão f->inicio == -1, que será:
  1 (verdadeiro), se a fila estiver vazia;
  0 (falso), caso contrário.

Função: void enfileirar(fila* f, int valor);

Objetivo: Adicionar um novo elemento ao final da fila.

Lógica de implementação:

- Verificação da fila vazia:
  A função verifica se a fila está vazia pelo índice fim ser igual a -1.
  Se estiver vazia, tanto inicio quanto fim são inicializados para 0, indicando que o primeiro elemento será inserido na posição zero do vetor.

- Fila não vazia:
  Se a fila já possui elementos, o índice fim é incrementado para apontar para a próxima posição livre onde o novo elemento será inserido.

- Inserção do elemento:
  O valor passado é armazenado na posição fim do vetor itens, representando a inserção do elemento no final da fila.

Função: int desenfileirar(fila* f);

Objetivo: Remover e retornar o elemento que está no início da fila.

Lógica de implementação:

- Captura do elemento:
  A função armazena o elemento que está na posição início do vetor itens em uma variável item. Esse é o elemento que será removido da fila.

- Verificação se a fila ficará vazia após remoção:
  Se o índice inicial for igual ao índice fim, significa que existe apenas um elemento na fila.
  Nesse caso, ambos os índices de início e fim são definidos como -1, indicando que a fila ficou vazia.

- Fila com mais de um elemento:
  Se ainda houver mais elementos, o índice inicial é incrementado para apontar para o próximo elemento da fila.

- Retorno do elemento removido:
  A função retorna o valor armazenado em item, que era o elemento removido do início da fila.

Grafo. C :

Função: grafo* cria_Grafo(int num); (Carina)

Objetivo: Criar e inicializar um grafo com um número especificado de vértices.

Lógica de implementação:

- Alocação da estrutura do grafo:
  A função começa alocando memória para a estrutura principal do grafo. Caso a alocação falhe, exibe uma mensagem de erro e retorna NULL.

- Inicialização do número de vértices:
  O campo nro_vertices da estrutura é configurado para o valor num, que representa a quantidade total de vértices do grafo.

- Alocação do vetor de vértices:
  É alocado um vetor para armazenar os vértices. Se a alocação falhar, o grafo é liberado da memória e a função retorna NULL.

- Alocação da lista de adjacências:
  A lista de adjacências é um vetor de ponteiros para nós (arestas). Essa estrutura será usada para representar as conexões entre os vértices. Caso a alocação falhe, libera a memória alocada anteriormente e retorna NULL.

- Alocação da matriz de pesos:
  A matriz pesos é um vetor de ponteiros para vetores de floats, formando uma matriz quadrada para armazenar os pesos das arestas entre vértices. A alocação é feita em duas etapas:
  Primeiro, aloca o vetor de ponteiros.
  Depois, para cada linha, aloca o vetor de pesos.
  Se alguma alocação falhar, todas as alocações anteriores são liberadas, exibindo mensagem de erro e retornando NULL.

- Inicialização dos pesos e listas:
  Para cada vértice:
  Inicializa a lista de adjacências como NULL, indicando que ainda não há conexões.
  Inicializa a matriz de pesos com FLT_MAX, que pode representar a ausência de conexão entre vértices

- Inicialização dos vértices:
  Cada vértice recebe:
  Um identificador único (id) baseado na posição no vetor.
  Variáveis booleanas (temAgua, temPosto, temFogo, fogoApagado, jaPegouFogo) iniciadas como falsas.
  Campos adicionais como área e ponteiros (posto) inicializados com valores padrão.

- Retorno do grafo:
  Após todas as alocações e inicializações, a função retorna o ponteiro para o grafo criado, pronto para ser utilizado.

Função: struct no* criarNo(int v); (Vanessa)

Objetivo: Criar um novo nó para ser usado na lista de adjacência do grafo, representando uma conexão para um vértice específico.

Lógica de implementação:

- Alocação de memória para o nó:
  A função começa alocando memória para um novo nó da lista. Se a alocação falhar, exibe uma mensagem de erro e retorna NULL.

- Inicialização dos campos do nó:
  O campo idVertice recebe o valor v, que é o identificador do vértice que esse nó representa na lista de adjacência.
  O ponteiro prox é inicializado com NULL, indicando que ainda não há próximo nó ligado a este.

- Retorno do nó criado:
  Após a alocação e inicialização, o ponteiro para o novo nó é retornado, pronto para ser inserido na lista de adjacências.

Função: void infoVertice(struct grafo* grafo, int id, bool temAgua, float area); (Vanessa)

Objetivo: Atualizar as informações de um vértice específico do grafo, definindo se ele possui água e sua área.

Lógica de implementação:

- Acesso ao vértice:
  Utiliza o índice id para acessar diretamente o vértice desejado dentro do vetor vértice da estrutura do grafo.

- Atualização dos atributos:
  Atualiza o campo área do vértice com o valor passado no parâmetro área.
  Atualiza o campo booleano temAgua com o valor passado no parâmetro temAgua, indicando se o vértice possui água ou não.

Função: void adicionaAresta(grafo* grafo, int origem, int destino, float peso); (Carina)
Objetivo: Adicionar uma aresta entre dois vértices (origem e destino) em um grafo não direcionado, atribuindo um peso a essa conexão.

Lógica de implementação:

- Criação do nó para o destino:
  Cria um novo nó que representa o vértice destino usando a função criarNo.
  Esse novo nó é inserido no início da lista de adjacência do vértice origem, apontando para o nó que estava anteriormente na lista.
  Atualiza a lista de adjacência de origem para começar com esse novo nó.
  Atualiza a matriz de pesos na posição [destino][origem] com o valor peso.

- Criação do nó para a origem:
  Similarmente, cria um novo nó para o vértice origem.
  Insere esse nó no início da lista de adjacência do vértice destino.
  Atualiza a matriz de pesos na posição [origem][destino] com o valor peso.

- Representação de grafo não direcionado:
  Ao adicionar a aresta em ambas as listas de adjacência (origem → destino e destino → origem) e atualizar ambos os pesos simetricamente, a função garante que o grafo seja tratado como não direcionado.

Função: void imprimirGrafo(grafo* gr); (Vanessa)

Objetivo: Exibir no console todas as informações do grafo, incluindo detalhes de cada vértice, seus atributos e suas conexões com pesos.

Lógica de implementação:

- Verificação inicial:
  A função primeiro verifica se o ponteiro do grafo é nulo. Se for, retorna imediatamente para evitar erros.

- Iteração pelos vértices:
  Percorre todos os vértices do grafo, usando um laço que vai de 0 até nro_vertices - 1.

- Impressão dos atributos do vértice:
  Para cada vértice, imprime seu índice, área (area), se possui água (temAgua), e se possui posto (temPosto).

- Exibição dos detalhes do posto:
  Caso o vértice tenha um posto (temPosto == true) e o ponteiro para o posto não seja nulo, imprime informações específicas do posto:
  ID do posto
  Quantidade de brigadistas (qntBombeiros)
  Velocidade média e capacidade de água do caminhão associado

- Impressão das conexões:
  Em seguida, imprime as conexões do vértice atual:
  Se não houver conexões (lista de adjacência vazia), imprime "Nenhuma".
  Caso contrário, percorre a lista ligada de adjacência, imprimindo para cada nó o ID do vértice adjacente e o peso da aresta correspondente.

Função: void libera_Grafo(grafo* gr); (Carina)

Objetivo: Liberar toda a memória alocada para o grafo, incluindo suas listas de adjacência, pesos, vértices e demais estruturas internas, evitando vazamentos de memória.

Lógica de implementação:

- Verificação inicial:
  A função verifica se o ponteiro do grafo é diferente de NULL para garantir que há algo a ser liberado.

- Liberação das listas de adjacência:
  Para cada vértice do grafo, percorre a lista ligada de nós (arestas) e libera cada nó individualmente. Isso é feito com um laço que segue os ponteiros prox até o fim da lista, liberando cada nó no caminho.

- Liberação do vetor de listas de adjacência:

Após liberar todos os nós das listas, libera o vetor listaAdj que armazenava os ponteiros para o início de cada lista.

- Liberação da matriz de pesos:
  Para cada linha da matriz pesos, libera o vetor de pesos que foi alocado dinamicamente. Depois libera o vetor de ponteiros pesos.

- Liberação dos postos associados aos vértices:
  Para cada vértice, verifica se existe um posto associado (posto != NULL). Caso exista, libera a memória do posto e também a do caminhão associado a esse posto.

- Liberação do vetor de vértices:
  Após liberar os postos, libera o vetor que armazena os vértices.

- Liberação da estrutura principal do grafo:
  Finalmente, libera a memória do próprio grafo (gr).

Fogo. C (Carina)

Função: void propagarFogo(grafo *grafo, int inicio);
Objetivo: Simular a propagação de fogo em um grafo, iniciando por um vértice específico, e expandindo para os vértices adjacentes de forma probabilística.

Lógica de implementação:

- Inicialização da fila e do vetor de visitados:
  Cria uma fila para gerenciar os vértices que terão o fogo propagado.
  Cria um vetor booleano visitado para controlar quais vértices já foram processados e evitar repetição.

- Configuração da semente aleatória:
  Inicializa a semente do gerador de números aleatórios com time(NULL) para garantir variabilidade na simulação.

- Verificação do vértice inicial:
  Checa se o vértice inicial está dentro dos limites válidos, se já está em chamas (temFogo == true) e se o fogo ainda não foi apagado (fogoApagado == false).
  Se sim, adiciona o vértice à fila e marca como visitado.

- Propagação do fogo:
  Enquanto a fila não estiver vazia, retira o próximo vértice atual para processar.
  Se o fogo já tiver sido apagado neste vértice, ignora e continua para o próximo.
  Percorre toda a lista de adjacência deste vértice, verificando seus vizinhos.

- Verificação dos vizinhos para propagação:
  Para cada vizinho (adj), verifica se:
  O vértice é válido (dentro do intervalo).
  O vizinho não está em chamas (temFogo == false).
  O fogo ainda não foi apagado no vizinho (fogoApagado == false).
  O vizinho ainda não foi visitado na propagação atual.
  O vizinho não possui um posto (temPosto == false), ou seja, o fogo não se propaga para locais protegidos.

- Probabilidade de propagação:
  Utiliza um número aleatório para decidir se o fogo se propaga para o vizinho, comparando com a constante CHANCE_PROPAGACAO.

- Se o fogo se propagar:
  Atualiza o estado do vizinho para estar em chamas (temFogo = true).
  Marca que o vizinho já pegou fogo (jaPegouFogo = true).
  Exibe mensagem de sucesso.
  Enfileira o vizinho para que sua adjacência também seja processada.
  Marca o vizinho como visitado.
  Caso contrário, exibe mensagem de falha na propagação.

- Liberação da memória:
  Ao final, libera a memória alocada para a fila e para o vetor de visitados.

Brigadistas. C:

Função: postoBrigadistas* criarPosto(int idVertice, int qntBombeiros, float velocidade); (Carina)

Objetivo: Criar e inicializar uma estrutura do tipo postoBrigadistas, que representa um posto de brigadistas em um vértice do grafo, com seus atributos configurados.

Lógica de implementação:

- Alocação de memória para o posto:
  Reserva espaço na memória para uma nova estrutura postoBrigadistas usando malloc.
  Verifica se a alocação foi bem-sucedida; em caso negativo, imprime mensagem de erro e retorna NULL.

- Inicialização dos atributos do posto:
  Define o id e idVertice com o valor passado no parâmetro idVertice.

Configura o número de bombeiros (qntBombeiros) conforme o parâmetro recebido.
Inicializa a flag utilizadoNestaIteracao como false, indicando que o posto ainda não foi usado na iteração atual.

- Alocação e configuração do caminhão:
  Aloca memória para o caminhao associado ao posto.
  Caso a alocação falhe, libera a memória do posto e retorna NULL para evitar vazamento de memória.
  Se alocação bem-sucedida, define a velocidade média do caminhão (velocidadeMedia) com o valor do parâmetro velocidade.
  Inicializa a capacidade de água do caminhão com uma constante CAP.

- Retorno:
  Após todas as inicializações, retorna o ponteiro para o novo posto criado.

Função: bool existe(int guardaVerticeNum[], int tam, int valor); (Carina)

Objetivo: Verificar se um determinado valor (valor) está presente em um vetor de inteiros (guardaVerticeNum), que tem tamanho tam.

Lógica de implementação:

- Percorrer o vetor:
  A função utiliza um laço for para iterar sobre cada elemento do vetor, do índice 0 até tam - 1.

- Comparação dos valores:
  Para cada elemento, verifica se o valor armazenado no índice atual é igual ao valor buscado (valor).

- Retorno caso o valor seja encontrado:
  Se algum elemento for igual ao valor, a função retorna true imediatamente, indicando que o valor existe no vetor.
- Retorno caso o valor não seja encontrado:

  Se o laço terminar sem encontrar o valor, a função retorna false, indicando que o valor não está presente no vetor.

Função: void distribuirPostos(grafo* grafo, int qnt, float velocidade); (Carina)

Objetivo:Distribuir três postos de brigadistas em vértices distintos do grafo, configurando cada posto com uma quantidade de bombeiros (qnt) e a velocidade média do caminhão (velocidade).

Lógica de implementação:

- Inicialização do gerador de números aleatórios:
  A função chama srand(time(NULL)) para garantir que a geração de números aleatórios seja diferente a cada execução.

- Selecionar 3 vértices distintos aleatoriamente:
  Um array verticesEscolhidos[3] guarda os índices dos vértices escolhidos.
  Um contador num_posto acompanha quantos vértices já foram selecionados.

- Enquanto não forem escolhidos 3 vértices:
  Gera-se um valor aleatório valorAleatorio entre 0 e grafo->nro_vertices - 1.
  Verifica-se, com a função existe, se o vértice já foi escolhido.
  Se não foi, adiciona o vértice em verticesEscolhidos e incrementa num_posto.

- Criar e atribuir os postos nos vértices escolhidos:
  Para cada um dos três vértices selecionados:
  Marca-se o vértice com temPosto = true.
  Cria-se um posto de brigadistas, usando a função criarPosto, que recebe o id do vértice, a quantidade de bombeiros e a velocidade do caminhão.
  O posto criado é atribuído ao campo posto do vértice.

- Exibir os vértices onde os postos foram distribuídos:
  Imprime na tela os índices dos vértices onde os postos foram alocados.

Função: CaminhoMinimoTempo* encontrarCaminhoComTempo(grafo* grafo, int verticeOrigem, float velocidadeBrigadista); (Vanessa)

Objetivo: Calcular o caminho mínimo em tempo a partir de um vértice origem até todos os demais vértices do grafo, considerando a velocidade do brigadista para determinar o tempo gasto em cada aresta.

Lógica de implementação:

- Inicialização das estruturas auxiliares:
  dist: vetor de distâncias (tempos) mínimas calculadas até cada vértice, inicializado com FLT_MAX (infinito).
  somaTempos: vetor que armazena a soma acumulada dos tempos para chegar a cada vértice, também inicializado com infinito.

anterior: vetor que armazena o vértice anterior no caminho mínimo para reconstrução do caminho.
visitado: vetor booleano para marcar se um vértice já foi processado.
Criação do MinHeap (heap mínimo):
Usado para selecionar o próximo vértice com menor tempo estimado (dist) a ser processado.
Se a criação do heap falhar, libera as estruturas e retorna NULL.

- Configuração inicial:
  Define a distância e soma de tempos do vértice origem como zero.
  Insere o vértice origem na heap com custo 0.
  Processamento da heap (algoritmo similar ao Dijkstra):
  Enquanto a heap não estiver vazia:
  Remove o elemento com menor custo (dist) da heap.
  Marca o vértice atual como visitado.
  Percorre todos os vértices adjacentes ao vértice atual.
  Para cada vértice adjacente, calcula o tempo para chegar até ele através do vértice atual, baseado no peso da aresta dividido pela velocidade do brigadista.
  Se o novo tempo calculado for menor que o tempo registrado para aquele vértice:
  Atualiza dist, somaTempos e o vértice anterior.
  Insere ou atualiza o vértice na heap para refletir o novo menor custo.

- Liberação e retorno:
  Libera a heap e o vetor de visitados.
  Cria e retorna uma estrutura CaminhoMinimoTempo contendo os vetores de distâncias, tempos acumulados e anteriores para uso posterior.

Função: void simularApagarTodosOsFocos(grafo* grafo, float velocidadeBrigadista, float intervaloPropagacao, postoBrigadistas* postos[]); (Vanessa)

Objetivo: Gerenciar a simulação do combate aos incêndios no grafo, onde brigadistas são enviados a vértices com fogo para apagá-lo, controlando o uso de água dos postos, o tempo de deslocamento, e monitorando o status dos incêndios em cada vértice.

Lógica de implementação:

- Apagar fogo e atualizar recursos:
  Quando um brigadista chega a um vértice com fogo e consegue apagá-lo:
  Reduz a capacidade de água do caminhão do posto correspondente pelo volume de água necessária.
  Imprime uma mensagem informando que o fogo foi apagado, o posto que fez o combate, o vértice do posto e a água restante.

Marca o vértice como sem fogo (temFogo = false) e como fogo apagado (fogoApagado = true).
Atualiza o tempo total de deslocamento acumulado, somando o tempo para chegar ao fogo.
Marca o posto como utilizado na iteração atual para evitar reutilização múltipla no mesmo ciclo.
Incrementa o contador de brigadistas usados na iteração atual.

- Relatar fogo ainda ativo:
  Se o vértice ainda tem fogo ativo (não apagado), o código verifica:
  Se existe um caminho válido (melhorCaminho) e se o tempo para chegar é conhecido (anterior[i] != -1), imprime o tempo estimado para chegada ao fogo.
  Caso contrário, indica que o fogo está ativo, mas não há posto disponível ou está dentro do tempo (não podendo agir ainda).

- Liberar memória do caminho calculado:
  Após o uso do caminho mínimo para cada vértice, a memória alocada para o mesmo é liberada, evitando vazamentos.

- Atualizar tempo da simulação:
  Incrementa o tempo simulado pela quantidade definida de intervalo de propagação.

- Verificar se ainda há incêndios ativos:
  Percorre todos os vértices verificando se algum ainda tem fogo ativo e não apagado.
  Se encontrar algum, marca a variável incendioAtivo como true e interrompe a verificação.

- Encerrar simulação quando todos os incêndios forem apagados:
  Se nenhum incêndio estiver ativo, imprime mensagem indicando que todos os focos foram apagados.
  Exibe o tempo total de deslocamento acumulado dos brigadistas durante toda a simulação.

Função: void reabastecerAguaPosto(grafo* grafo, postoBrigadistas* posto); (Vanessa)

Objetivo: Simular o reabastecimento de água do caminhão de um posto de brigadistas, encontrando o caminho mais rápido até um vértice que possua água disponível, e atualizar a capacidade do caminhão após o reabastecimento.

Lógica de implementação:

- Identificação do vértice do posto:
  Obtém o ID do vértice onde o posto está localizado (idPosto).

- Calcular caminho mínimo até todos os vértices:
  Usa a função encontrarCaminhoComTempo para calcular os tempos mínimos do posto até todos os outros vértices no grafo, considerando a velocidade média do caminhão do posto.

- Encontrar o vértice com água mais próximo:
  Inicializa menorTempo com o maior valor possível (FLT_MAX) e verticeComAgua como -1 (indicando que ainda não encontrou nenhum).

- Percorre todos os vértices do grafo:
  Se o vértice possui água (temAgua == true) e o tempo calculado para chegar até ele (caminhoReabastecimento->dist[j]) for menor que menorTempo, atualiza menorTempo e verticeComAgua com esse vértice.

- Simular reabastecimento se houver local acessível:
  Se encontrou algum vértice com água acessível (verticeComAgua != -1):
  Imprime mensagem informando que o posto vai se deslocar até esse vértice para reabastecer.

  Calcula o tempo total de reabastecimento como o tempo de ida e volta (2 vezes o menor tempo encontrado).
  Restaura a capacidade do caminhão para o valor máximo definido (CAP).
  Caso não encontre local com água acessível:
  Imprime mensagem indicando que o posto não conseguiu reabastecer por não existir local acessível com água.

- Liberar memória alocada:
  Libera os arrays e a estrutura alocada para o caminho mínimo calculado para evitar vazamentos de memória.

Função: void reabastecerAgua(grafo* grafo, postoBrigadistas* postos[]); (Carina)

Objetivo: Verificar quais postos de brigadistas foram utilizados na iteração atual da simulação para apagar incêndios, e simular o reabastecimento de água desses postos, se necessário.

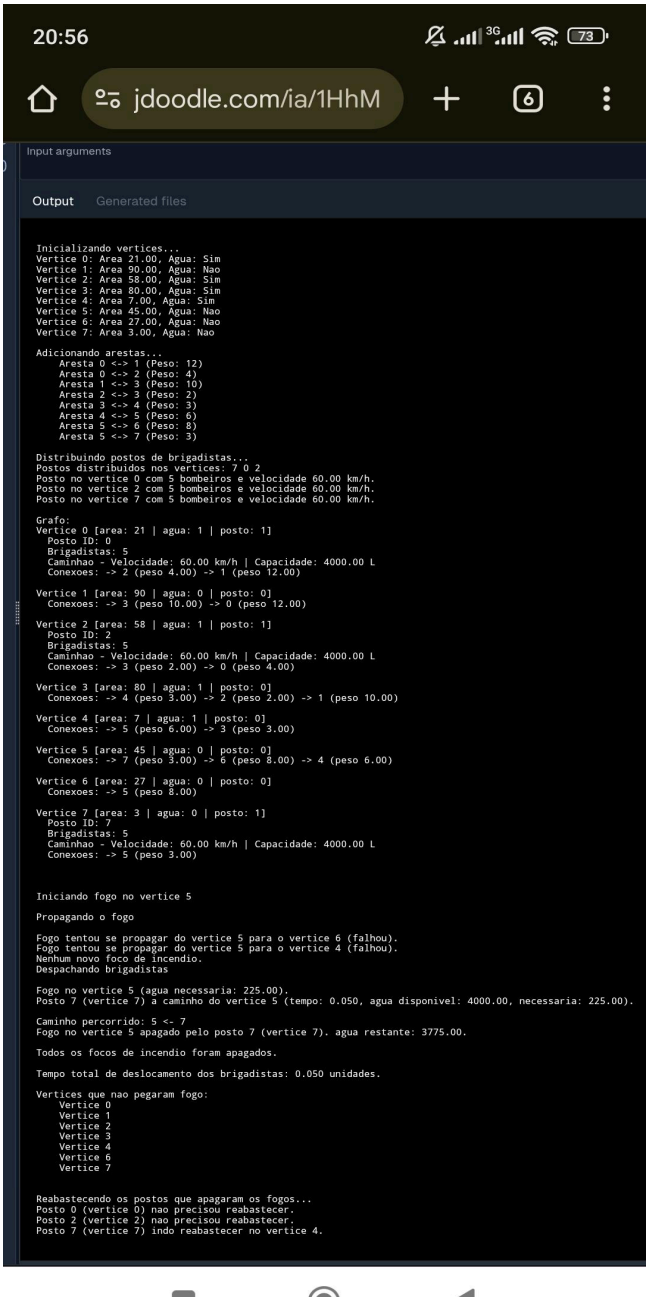
Lógica de implementação:

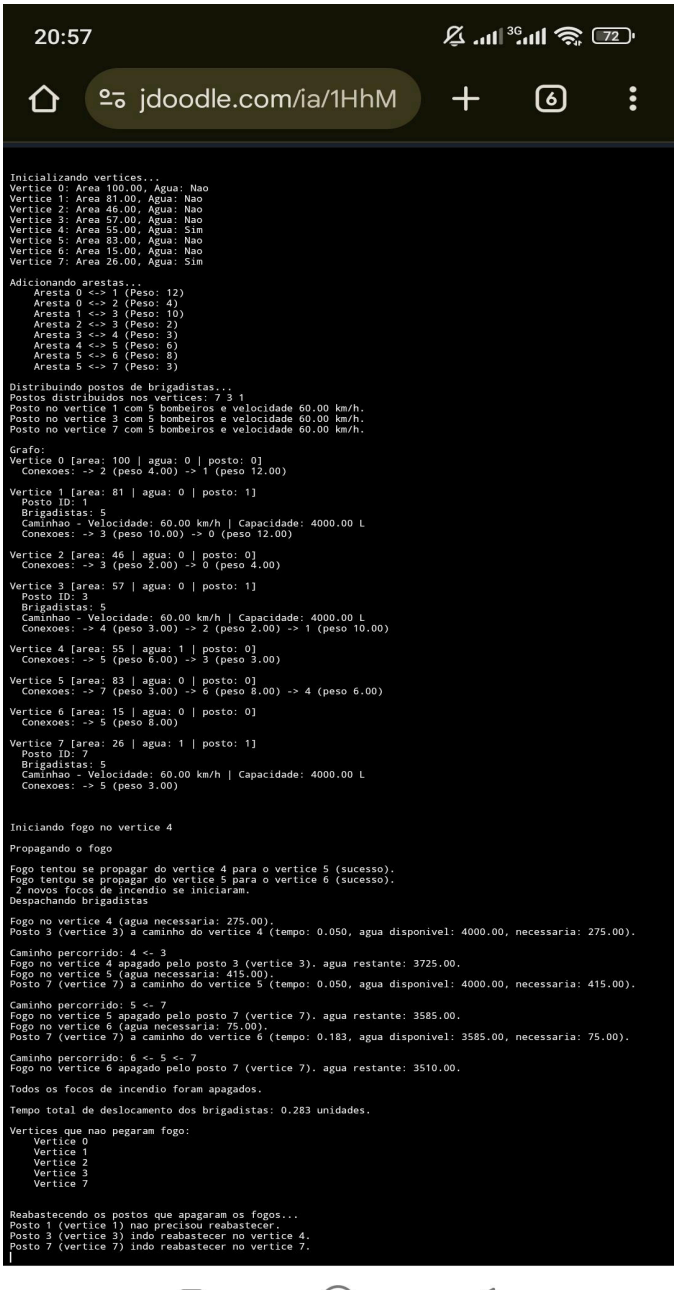
- Início da simulação de reabastecimento:
  Imprime uma mensagem informando que o sistema está verificando os postos que precisam reabastecer.

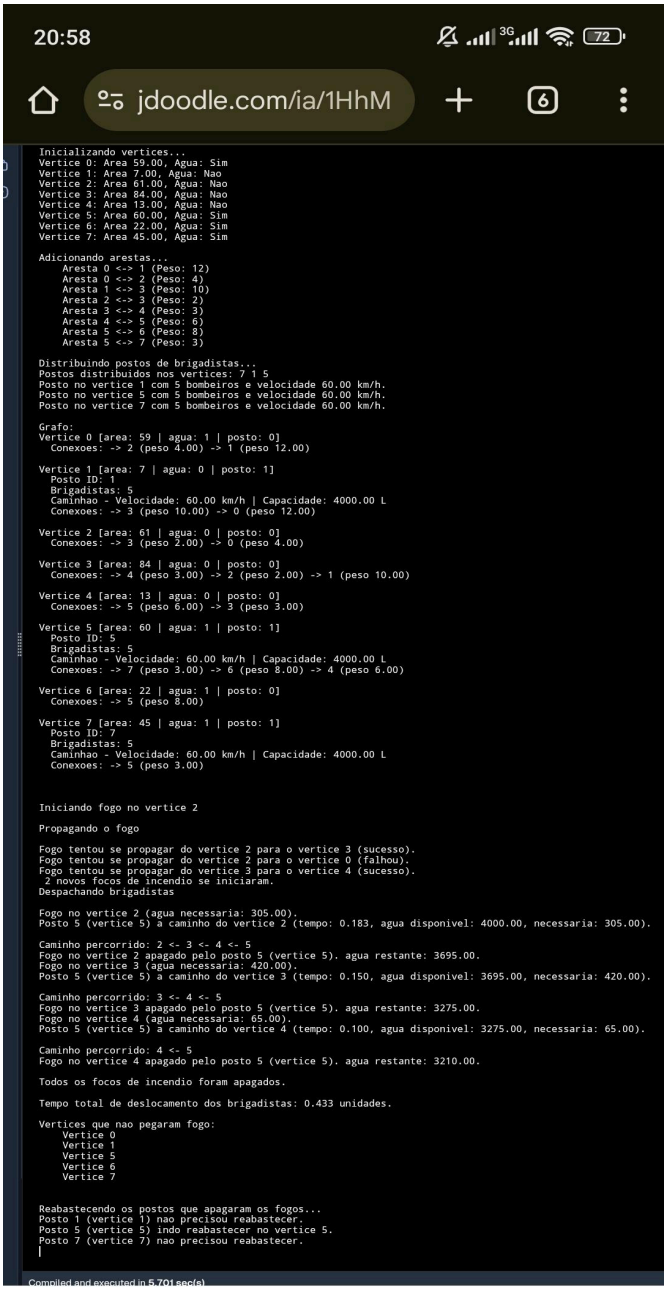
- Percorrer os três postos de brigadistas:

Utiliza um for de 0 a 2 (total de 3 postos).
Para cada posto:
Verifica se ele foi utilizado na iteração atual, por meio do campo utilizadoNestaIteracao.
Se sim, chama a função reabastecerAguaPosto para simular o reabastecimento desse posto específico.
Se não foi utilizado, imprime uma mensagem indicando que esse posto não precisou reabastecer, informando seu ID e vértice.

Breve análise dos resultados obtidos nas simulações:






20:58
jdoodle.com/ia/1HhM
Inicializando vertices...
Vertice 0: Area 59.00, Agua: Sim
Vertice 1: Area 7.00, Agua: Nao
Vertice 2: Area 61.00, Agua: Nao
Vertice 3: Area 84.00, Agua: Nao
Vertice 4: Area 13.00, Agua: Nao
Vertice 5: Area 60.00, Agua: Sim
Vertice 6: Area 22.00, Agua: Sim
Vertice 7: Area 45.00, Agua: Sim
Adicionando arestas...
Aresta 0 <-> 1 (Peso: 12)
Aresta 0 <-> 2 (Peso: 4)
Aresta 1 <-> 3 (Peso: 10)
Aresta 2 <-> 3 (Peso: 2)
Aresta 3 <-> 4 (Peso: 3)
Aresta 4 <-> 5 (Peso: 6)
Aresta 5 <-> 6 (Peso: 8)
Aresta 5 <-> 7 (Peso: 3)
Distribuindo postos de brigadistas...
Postos distribuidos nos vertices: 7 1 5
Posto no vertice 1 com 5 bombeiros e velocidade 60.00 km/h.
Posto no vertice 5 com 5 bombeiros e velocidade 60.00 km/h.
Posto no vertice 7 com 5 bombeiros e velocidade 60.00 km/h.
Grafo:
Vertice 0 [area: 59 | agua: 1 | posto: 0]
Conexoes: -> 2 (peso 4.00) -> 1 (peso 12.00)
Vertice 1 [area: 7 | agua: 0 | posto: 1]
Posto ID: 1
Brigadistas: 5
Caminhao - Velocidade: 60.00 km/h | Capacidade: 4000.00 L
Conexoes: -> 3 (peso 10.00) -> 0 (peso 12.00)
Vertice 2 [area: 61 | agua: 0 | posto: 0]
Conexoes: -> 3 (peso 2.00) -> 0 (peso 4.00)
Vertice 3 [area: 84 | agua: 0 | posto: 0]
Conexoes: -> 4 (peso 3.00) -> 2 (peso 2.00) -> 1 (peso 10.00)
Vertice 4 [area: 13 | agua: 0 | posto: 0]
Conexoes: -> 5 (peso 6.00) -> 3 (peso 3.00)
Vertice 5 [area: 60 | agua: 1 | posto: 1]
Posto ID: 5
Brigadistas: 5
Caminhao - Velocidade: 60.00 km/h | Capacidade: 4000.00 L
Conexoes: -> 7 (peso 3.00) -> 6 (peso 8.00) -> 4 (peso 6.00)
Vertice 6 [area: 22 | agua: 1 | posto: 0]
Conexoes: -> 5 (peso 8.00)
Vertice 7 [area: 45 | agua: 1 | posto: 1]
Posto ID: 7
Brigadistas: 5
Caminhao - Velocidade: 60.00 km/h | Capacidade: 4000.00 L
Conexoes: -> 5 (peso 3.00)
Iniciando fogo no vertice 2
Propagando o fogo
Fogo tentou se propagar do vertice 2 para o vertice 3 (sucesso).
Fogo tentou se propagar do vertice 2 para o vertice 0 (falhou).
Fogo tentou se propagar do vertice 3 para o vertice 4 (sucesso).
2 novos focos de incendio se iniciaram.
Despachando brigadistas
Fogo no vertice 2 (agua necessaria: 305.00).
Posto 5 (vertice 5) a caminho do vertice 2 (tempo: 0.183, agua disponivel: 4000.00, necessaria: 305.00).
Caminho percorrido: 2 <- 3 <- 4 <- 5
Fogo no vertice 2 apagado pelo posto 5 (vertice 5). agua restante: 3695.00.
Fogo no vertice 3 (agua necessaria: 420.00).
Posto 5 (vertice 5) a caminho do vertice 3 (tempo: 0.150, agua disponivel: 3695.00, necessaria: 420.00).
Caminho percorrido: 3 <- 4 <- 5
Fogo no vertice 3 apagado pelo posto 5 (vertice 5). agua restante: 3275.00.
Fogo no vertice 4 (agua necessaria: 65.00).
Posto 5 (vertice 5) a caminho do vertice 4 (tempo: 0.100, agua disponivel: 3275.00, necessaria: 65.00).
Caminho percorrido: 4 <- 5
Fogo no vertice 4 apagado pelo posto 5 (vertice 5). agua restante: 3210.00.
Todos os focos de incendio foram apagados.
Tempo total de deslocamento dos brigadistas: 0.433 unidades.
Vertices que nao pegaram fogo:
Vertice 0
Vertice 1
Vertice 5
Vertice 6
Vertice 7
Reabastecendo os postos que apagaram os fogos...
Posto 1 (vertice 1) nao precisou reabastecer.
Posto 5 (vertice 5) indo reabastecer no vertice 5.
Posto 7 (vertice 7) nao precisou reabastecer.
Compiled and executed in 5.701 sec(s)


20:58
jdoodle.com/ia/1HhM
Output
Generated files
Inicializando vertices...
Vertice 0: Area 79.00, Agua: Nao
Vertice 1: Area 97.00, Agua: Sim
Vertice 2: Area 7.00, Agua: Nao
Vertice 3: Area 81.00, Agua: Nao
Vertice 4: Area 8.00, Agua: Nao
Vertice 5: Area 81.00, Agua: Sim
Vertice 6: Area 68.00, Agua: Sim
Vertice 7: Area 82.00, Agua: Nao
Adicionando arestas...
Aresta 0 <-> 1 (Peso: 12)
Aresta 0 <-> 2 (Peso: 4)
Aresta 1 <-> 3 (Peso: 10)
Aresta 2 <-> 3 (Peso: 2)
Aresta 3 <-> 4 (Peso: 3)
Aresta 4 <-> 5 (Peso: 6)
Aresta 5 <-> 6 (Peso: 8)
Aresta 5 <-> 7 (Peso: 3)
Distribuindo postos de brigadistas...
Postos distribuidos nos vertices: 7 4 2
Posto no vertice 2 com 5 bombeiros e velocidade 60.00 km/h.
Posto no vertice 4 com 5 bombeiros e velocidade 60.00 km/h.
Posto no vertice 7 com 5 bombeiros e velocidade 60.00 km/h.
Grafo:
Vertice 0 [area: 79 | agua: 0 | posto: 0]
Conexoes: -> 2 (peso 4.00) -> 1 (peso 12.00)
Vertice 1 [area: 97 | agua: 1 | posto: 0]
Conexoes: -> 3 (peso 10.00) -> 0 (peso 12.00)
Vertice 2 [area: 7 | agua: 0 | posto: 1]
Posto ID: 2
Brigadistas: 5
Caminhao - Velocidade: 60.00 km/h | Capacidade: 4000.00 L
Conexoes: -> 3 (peso 2.00) -> 0 (peso 4.00)
Vertice 3 [area: 81 | agua: 0 | posto: 0]
Conexoes: -> 4 (peso 3.00) -> 2 (peso 2.00) -> 1 (peso 10.00)
Vertice 4 [area: 8 | agua: 0 | posto: 1]
Posto ID: 4
Brigadistas: 5
Caminhao - Velocidade: 60.00 km/h | Capacidade: 4000.00 L
Conexoes: -> 5 (peso 6.00) -> 3 (peso 3.00)
Vertice 5 [area: 81 | agua: 1 | posto: 0]
Conexoes: -> 7 (peso 3.00) -> 6 (peso 8.00) -> 4 (peso 6.00)
Vertice 6 [area: 68 | agua: 1 | posto: 0]
Conexoes: -> 5 (peso 8.00)
Vertice 7 [area: 82 | agua: 0 | posto: 1]
Posto ID: 7
Brigadistas: 5
Caminhao - Velocidade: 60.00 km/h | Capacidade: 4000.00 L
Conexoes: -> 5 (peso 3.00)
Iniciando fogo no vertice 6
Propagando o fogo
Fogo tentou se propagar do vertice 6 para o vertice 5 (falhou).
Nenhum novo foco de incendio.
Despachando brigadistas
Fogo no vertice 6 (agua necessaria: 340.00).
Posto 7 (vertice 7) a caminho do vertice 6 (tempo: 0.183, agua disponivel: 4000.00, necessaria: 340.00).
Caminho percorrido: 6 <- 5 <- 7
Fogo no vertice 6 apagado pelo posto 7 (vertice 7). agua restante: 3660.00.
Todos os focos de incendio foram apagados.
Tempo total de deslocamento dos brigadistas: 0.183 unidades.
Vertices que nao pegaram fogo:
Vertice 0
Vertice 1
Vertice 2
Vertice 3
Vertice 4
Vertice 5
Vertice 7
Reabastecendo os postos que apagaram os fogos...
Posto 2 (vertice 2) nao precisou reabastecer.
Posto 4 (vertice 4) nao precisou reabastecer.
Posto 7 (vertice 7) indo reabastecer no vertice 5.
Compiled and executed in 5.651 sec(s)

Análise Geral:

Estas simulações demonstram a dinâmica de um incêndio se espalhando por uma rede e a resposta dos brigadistas. Observamos variações nos seguintes aspectos:

- Ponto de início: Cada execução inicia o fogo em um vértice diferente (4, 6, 2 e 5).
- Padrão de propagação: A topologia da rede e as condições de propagação levam a diferentes caminhos e sucessos na expansão do fogo.
- Deslocamento de recursos: Os brigadistas são enviados para os locais afetados, com diferentes tempos de deslocamento dependendo da distância e estratégia.
- Sucesso no combate: Em todos os casos mostrados, o fogo é eventualmente apagado nos vértices atingidos.
- Reabastecimento: A menção ao reabastecimento de postos sugere uma gestão contínua dos recursos para combate a incêndio.
- As diferentes saídas ilustram como a mudança no ponto inicial do incêndio altera significativamente o curso dos eventos na simulação.

De maneira geral, avaliando a eficiência do algoritmo com base nos resultados apresentados, podemos afirmar as seguintes questões sobre a sua eficiência):

- Supressão do Incêndio: Em todas as simulações mostradas, o algoritmo conseguiu, eventualmente, conter e apagar o fogo em todos os vértices que foram atingidos. Isso sugere que a estratégia de alocação e deslocamento de brigadistas é eficaz em impedir a propagação descontrolada do incêndio na rede simulada.
- Tempo de Deslocamento: Os tempos totais de deslocamento dos brigadistas reportados nas diferentes simulações parecem razoáveis, variando de 0.033 a 0.433 unidades. Isso indica que o algoritmo consegue mobilizar os recursos em um tempo relativamente curto em relação à dinâmica da propagação do fogo.
- Gerenciamento de Recursos: A menção ao "reabastecimento dos postos" sugere que o algoritmo considera a disponibilidade e o reabastecimento de recursos (como água), o que é crucial para a sustentabilidade da operação de combate a incêndio e, portanto, para a eficiência a longo prazo.
- Resposta Dinâmica: O algoritmo demonstra uma capacidade de responder dinamicamente ao foco inicial do incêndio e à sua propagação, deslocando brigadistas para os locais necessários.

Possíveis melhorias e discussão sobre os desafios encontrados:

As possíveis melhorias são:

- Modelos de Propagação Mais Complexos: Incorporar modelos de propagação que considerem mais fatores ambientais, como direção e intensidade do vento, tipo de vegetação, umidade, e topografia. Isso tornaria a simulação mais realista e exigiria estratégias de combate mais adaptáveis.

- Intensidade do Fogo: Simular diferentes níveis de intensidade do fogo, que poderiam influenciar a velocidade de propagação e a quantidade de recursos necessários para combatê-lo.

- Visualização Gráfica: Uma representação visual da rede, da propagação do fogo e do movimento dos brigadistas facilitaria a compreensão do comportamento do algoritmo e a identificação de áreas para melhoria.

Desafios encontrados:

Os principais desafios encontrados durante a implementação do código foram duas funcionalidades específicas: a propagação do fogo e o cálculo do caminho mínimo para os brigadistas.

A função de propagação do fogo apresentou desafios por exigir um controle sobre a dinâmica do incêndio. Não era possível permitir que todos os vértices pegassem fogo simultaneamente, pois era necessário simular a propagação progressiva, de forma controlada, respeitando a lógica de adjacência e probabilidades, o que demandou o uso de estruturas como filas e controle de vértices visitados.

Já a função de encontrar o caminho mínimo envolveu uma dificuldade adicional por lidar com variados postos de brigadistas. Como havia três postos disponíveis, foi necessário calcular, para cada foco de incêndio, qual posto conseguiria alcançá-lo no menor tempo possível, levando em conta as distâncias no grafo e as velocidades dos caminhões. Essa lógica exigiu não apenas a implementação de um algoritmo eficiente de caminhos mínimos (Dijkstra), mas também a inclusão desse algoritmo à lógica de simulação.