Fundação Universidade Federal do ABC
Pró-reitoria de Pesquisa
Av. dos Estados, 5001, Santa Terezinha, Santo André/SP, CEP 09210-580
Bloco L, 3º Andar, Fone (11) 3356-7617
iniciacao@ufabc.edu.br

Relatório Final de Iniciação Científica
Referente ao Edital: PDPD 07/2023

Nome do aluno: Gabriel Alexandre Guenta Tsurushima
Assinatura do aluno:

Nome do orientador: Aritanan Borges Garcia Gruber
Assinatura do orientador:

Título do projeto: Técnicas de Programação Dinâmica em Grafos
Palavras-chave do projeto: programação dinâmica, algoritmos, otimização, grafos
Área do conhecimento do projeto: Ciência da Computação, Teoria dos Grafos
Bolsista: Sim, pelo PDPD - Programa Pesquisando Desde o Primeiro Dia (financiado pela UFABC)

Santo André, 2024

Contents

1 Resumo

Este projeto é uma introdução aos algoritmos em grafos, com foco em programação dinâmica. A programação dinâmica é uma técnica de resolução de problemas com inúmeras aplicações em otimização, frequentemente oferecendo uma abordagem eficiente para resolver problemas em diversas áreas, como Pesquisa Operacional e Teoria dos Grafos. Um dos pilares dessa técnica é a identificação de uma subestrutura ótima, que está associada a uma relação de recorrência a partir da qual é possível derivar um algoritmo.

Embora esses elementos sejam frequentemente desafiadores, eles constituem ferramentas poderosas que devem fazer parte do repertório de qualquer estudante de Ciência da Computação e áreas afins. Este trabalho traça o caminho para a aquisição dessas habilidades.

2 Introdução

O foco deste projeto é a aplicação da programação dinâmica em problemas da Teoria dos Grafos. O entendimento dessa técnica requer uma série de pré-requisitos, começando pelo

domínio de uma linguagem de programação — a escolha recaiu sobre C/C++, com o objetivo de explorar alguns dos aspectos mais modernos do C++. Além disso, é necessário o conhecimento de algoritmos e uma coleção fundamental de estruturas de dados básicas, como vetores, listas, árvores binárias, árvores binárias de busca, árvores AVL, filas, pilhas, heaps, além de algoritmos de busca, ordenação e, naturalmente, recursividade.

A partir daí, foi possível iniciar o estudo dos temas principais da pesquisa: grafos e programação dinâmica. Para construir uma base sólida e adquirir maturidade no assunto, exploramos alguns algoritmos introdutórios sobre grafos, como a busca em profundidade (DFS), a busca em largura (BFS) e a ordenação topológica de grafos acíclicos dirigidos (DAGs). Estudamos e implementamos diversos algoritmos clássicos de programação dinâmica, tanto unidimensionais quanto multidimensionais [4, 6]. Além disso, pela proximidade conceitual, também estudamos vários algoritmos gulosos.

Por fim, realizamos um breve estudo sobre a aplicação da programação dinâmica em modelos estocásticos, como o Modelo de Decisão de Markov (MDP). Nesse contexto, a técnica é utilizada para encontrar políticas ótimas em ambientes estocásticos, permitindo calcular a melhor sequência de decisões, visando minimizar custos ou maximizar recompensas ao longo do tempo.

Esse variado conjunto de exemplos evidencia a versatilidade da programação dinâmica como uma técnica fundamental para a resolução dos mais diversos problemas, permitindo lidar tanto com cenários determinísticos, como nos problemas de grafos mencionados anteriormente, quanto com cenários estocásticos.

3 Fundamentação teórica

3.1 Conteúdo base

Seguimos rigorosamente um planejamento de estudos. A primeira parte consistiu na familiarização com a linguagem C/C++ e alguns de seus recursos mais básicos, como entrada e saída, ponteiros, estruturas e gerenciamento de memória [1].

A seguir, discutiremos brevemente alguns dos tópicos estudados: vetores, algoritmos recursivos, listas encadeadas, filas, pilhas, busca em vetor ordenado, ordenação, árvores binárias e árvores binárias de busca.

Um *vetor* (também conhecido como *array*) é uma estrutura de dados que aloca e dispõe de forma sequencial um certo tipo de dado na memória RAM. Um vetor permite acessar um elemento através de um *índice* (ou uma *posição*). Para um vetor com n elementos o seu conjunto de índices é $\{0, 1, ..., n-1\}$. Assim, se v é um vetor de n elementos e i é um de seus índices, então $v[i]$ é o elemento que mora na posição i. Um vetor permite o acesso "aleatório" a qualquer um de seus elementos. Isto significa que o custo para acessar qualquer elemento é uma constante fixa (a alocação em posições consecutivas da memória facilita isso). A biblioteca padrão do C++(Standard Template Library) contém uma estrutura de dados, *vector*, chamada *vetor dinâmico*, que permite manipular estes objetos de uma forma robusta através de uma biblioteca que implementa diversas operações fundamentais, tais como estender um vetor através da adição de um ou mais elementos. A ideia de tal método consiste em manter inicialmente um vetor alocado com um certo tamanho. A cada adição de um novo elemento pode ocorrer uma de duas alternativas: ou (1) é possível acolher o novo elemento no espaço físico já reservado e, portanto, não há nada a fazer, ou (2) não há espaço físico suficiente e, em tal caso, um novo bloco de memória cujo tamanho é o dobro do atual é alocado e os elementos originais são despejados nesse novo bloco juntamente

com o novo habitante. Esta estratégia garante uma performance amortizada cujo consumo de tempo é constante, mantendo assim a principal propriedade de um vetor estático.

A expressividade de uma linguagem é de fundamental importância na composição de soluções para problemas. A linguagem C/C++ permite a construção de algoritmos recursivos o que constitui uma alternativa (e, muitas vezes, mais elegante) à forma mais popular, laços, de implementar processos algorítmicos repetitivos. Além disso, o uso de algoritmos recursivos, implementados na forma de funções, torna imperativa a necessidade de estabelecer claramente o que a sua função deve fazer.

Do ponto de vista sintático, uma função é recursiva se ela, diretamente ou indiretamente, chama a si mesma. É claro que uma tal construção pode ou não parar. O nosso interesse aqui reside nas funções recursivas que param.

É impossível catalogar todas as formas na qual um algoritmo recursivo pode se apresentar. No entanto, é possível apontar uma estratégia "geral" para a solução de problemas de forma recursiva. O primeiro passo consiste em estabelecer precisamente o problema que necessita ser resolvido. O segundo passo consiste em "mensurar" o tamanho do problema uma vez que a composição da solução dependerá diretamente da noção de tamanho. O processo de solução começa uma vez estabelecido estes dois passos iniciais. Primeiro, é necessário mostrar como resolver diretamente problemas de tamanho "pequeno". A seguir, é necessário considerar um problema de tamanho "grande". Este problema deve ser decomposto em um ou mais problemas de tamanho menor, mas que tenham a mesma natureza do problema original (ou seja, constituem uma versão "idêntica" do problema original envolvendo um problema menor). O próximo passo consiste em mostrar como as soluções para os problemas menores podem ser compostas de tal forma a obter uma solução para o problema original. As soluções para os problemas menores operacionalmente são obtidas através de uma chamada recursiva para a função que está sendo definida.

Uma lista ligada, também chamada de lista encadeada, é uma estrutura de armazenamento de certos tipos de dados em uma determinada ordem, assim como o vetor. Entretanto, os elementos da lista, chamados de nós ou células, não são necessariamente armazenados de forma sequencial na memória. Cada célula da lista contém como um de seus atributos o endereço da célula seguinte. Além disso, o endereço da célula seguinte é sempre diferente do endereço das células anteriores na lista. Por fim, o endereço de uma lista ligada é o endereço de sua primeira célula. Note que com o endereço da primeira célula é possível acessar todos os elementos da lista. Diferentemente do que ocorre no caso dos vetores, acessar o n-ésimo elemento a partir do início tem custo proporcional a n. Também diferentemente dos vetores, dada uma posição na lista é possível inserir ou remover um após tal posição em tempo constante.

Uma fila é uma estrutura de dados que manipula uma coleção de objetos e cuja interface possui duas operações básicas, uma de inserção, e a outra de remoção. Estas operações são compostas de tal forma a respeitar a seguinte propriedade invariante: o primeiro elemento inserido é o primeiro a ser removido (FIFO de First-In-First-Out), de forma similar a dinâmica de uma fila de pessoas.

Uma pilha é uma estrutura de dados que manipula uma coleção de objetos e cuja interface possui duas operações básicas: uma de inserção, e a outra de remoção. Estas operações são compostas de tal forma a respeitar a seguinte propriedade invariante: o último elemento a ser inserido é o primeiro a ser removido (LIFO de Last-In-First-Out).

O problema da busca em um vetor ordenado, isto é, dado um vetor de objetos sobre os quais está definida uma relação de ordem total, e uma chave de busca, decidir se a chave é um dos objetos do vetor, é mais um dos problemas clássicos da computação. Uma forma efi-

ciente de resolvê-lo - uma cujo consumo de tempo é proporcional ao logaritmo do número de elementos do vetor - é fornecida pelo algoritmo de busca binária que pode ser encarado como um algoritmo de divisão e conquista. A ideia do algoritmo consiste em testar se a chave de busca está presente no elemento que ocupa a posição "central" do vetor. Se este for o caso, o algoritmo devolve "sim" e para. Do contrário há duas alternativas: (1) a chave é menor que o elemento "central" ou (2) a chave é maior que o elemento central. No caso (1), o problema é resolvido recursivamente para o subvetor que consiste dos elementos que precedem o elemento "central". No caso (2), o problema é resolvido recursivamente para o subvetor que consiste dos elementos que sucedem o elemento "central". É evidente como resolver um problema "pequeno", isto é, um que envolve um vetor com zero elementos: basta devolver "não". Estudamos diversas variantes do algoritmo de busca binária que resolvem o problema original de forma mais robusta.

Há uma enorme variedade de algoritmos de ordenação, dois exemplos muito populares são o mergesort e o quicksort que também são instâncias da técnica de divisão e conquista. Embora o consumo de tempo do quicksort seja quadrático no número de elementos do vetor no pior caso, a sua performance na prática é destacável, sendo algumas de suas variações ainda muito presentes em bibliotecas padrão de algumas linguagens de programação.

Um outro algoritmo, o *heapsort*, encara um vetor como uma árvore binária semi-completa e explora a estrutura de tal árvore para produzir um algoritmo cujo consumo de tempo no pior caso é proporcional a $n \lg n$. Um vetor $v[1:n]$ de objetos é um *heap* se o elemento que mora na posição i de v, $v[i]$, é menor ou igual ao elemento que mora na posição $i/2$, $v[i/2]$, - divisão inteira de i por 2 -, chamado de *pai* de i, para cada $1 < i \leqslant n$. Note que a relação "é pai de" no conjunto dos índices induz uma árvore binária cuja altura é igual ao piso de $\lg n$. O algoritmo explora esta estrutura para ordenar um vetor de forma eficiente. É importante destacar que todos estes algoritmos são baseados em comparações. Além disso, tanto o *mergesort* quanto o *heapsort* realizam uma quantidade "ótima" de comparações. Uma combinação perspicaz do *quicksort* com o *heapsort* está presente na biblioteca padrão do C++, implementada na função *sort*.

A *tabela de dispersão* (*hash table*) é uma estrutura de dados amplamente utilizada para acelerar funções como a de inserção, remoção e busca - idealmente em tempo constante O(1) - mapeando chaves para posições em um vetor, permitindo o acesso a um ou mais objetos através de uma mesma chave. Há alguns cuidados que devem ser tomados ao implementar uma tabela de dispersão como, por exemplo, uma boa escolha de uma função de espalhamento h. Dizemos que dois objetos p e q *colidem* se $h(p) = h(q)$. Uma tal função deve conseguir espalhar bem o conjunto das chaves de tal forma que para cada valor v no contradomínio de h, o número esperado de chaves que são mapeadas por h para a posição v é uma constante independente da posição.

A definição de árvore binária é um exemplo de uma definição recursiva. Dizemos que uma coleção finita X de objetos, chamados de *nós* (ou vértices), é uma *árvore binária* se (1) X é vazio, ou (2) X não é vazio e existe um elemento na coleção, digamos r, chamado de *raiz*, tal que a coleção $X - \{r\}$ pode ser particionada em duas subcoleções Y, chamada de *subárvore esquerda* de r, e Z, chamada de *subárvore direita* de r, tais que Y e Z são árvores binárias. A implementação usual em C++ consiste de uma estrutura que armazena uma certa informação e de dois ponteiros, o primeiro aponta para a subárvore esquerda e, o segundo, para a direita. A árvore vazia é representada por `nullptr`, embora haja alternativas melhores.

Uma *árvore binária de busca* é uma árvore binária cujos nós possuem um atributo chamado de *chave* sobre o qual está definida uma relação de ordem total, e dotada da seguinte pro-

priedade: para cada nó r da árvore, se x é um nó que está na subárvore esquerda de r, então a chave de x é menor que a chave de r, e se x é um nó que está na subárvore direita de r, então a chave de x é maior que a chave de r. Os algoritmos envolvendo árvores binárias constituem ótimos exemplos da expressividade de algoritmos recursivos. É claro que em virtude da equivalência de recursão e iteração, todo algoritmo recursivo pode ser expresso como um algoritmo iterativo, como consta em alguns dos exemplos do repositório[2]. No entanto, a naturalidade de alguns algoritmos recursivos é notória e destaca a importância da técnica.

3.2 Grafos e Programação Dinâmica

Um *grafo* consiste de dois conjuntos finitos, o conjunto dos *vértices*, este não-vazio, e o conjunto dos *arcos*. Cada arco é um par ordenado de vértices distintos. Um grafo é uma estrutura extremamente versátil, pois permite modelar uma gama muito rica de problemas dos mais variados tipos. Há duas formas populares de se representar um grafo computacionalmente: (1) através de uma matriz, digamos M, cujas linhas e colunas são indexadas pelo conjunto dos vértices, chamada de *matriz de adjacência*, tal que para cada par u, v de vértices, a entrada $M[u][v]$ é igual a 1 quando o grafo contém o arco (u, v), e é igual a 0 quando (u, v) não é um dos arcos do grafo; e (2) através de um vetor de listas, chamado de *listas de adjacência*, tal que para cada vértice u está associada uma lista cujos elementos são os vértices v tais que o par (u, v) é um arco do grafo; a ordem na qual tais elementos estão elencados é irrelevante.

Algoritmos em grafos desempenham um papel central no estudo de algoritmos, oferecendo soluções eficientes para problemas como busca, detecção de ciclos e determinação de caminhos mínimos. Entre os algoritmos de busca, destacam-se a busca em profundidade e a busca em largura em virtude de suas inúmeras aplicações, ambos descritos detalhadamente em [4].

A programação dinâmica é uma técnica poderosa utilizada para resolver, em particular, problemas de otimização fornecendo como subproduto, com frequência, algoritmos eficientes, i.e., cujo consumo de tempo é polinomial no tamanho da entrada. A ideia central é observar que uma solução ótima para um certo problema de otimização contém soluções ótimas para versões mais simples deste mesmo problema de otimização; esta observação costuma ser rotulada como *propriedade da subestrutura ótima*. Uma vez identificado tais subproblemas uma tabela é utilizada para armazenar as soluções para os subproblemas de tal forma a evitar a repetição de computações desnecessárias.

Há duas formas de se implementar uma solução utilizando essa técnica, a *top-down*, implementada através de uma função recursiva, e a *bottom-up*, implementada através de uma função iterativa. O preço computacional pago pela versão recursiva pode em alguns casos ser vantajoso uma vez que, nesse caso, tão somente os subproblemas "relacionados" ao problema original no processo de decomposição são solucionados.

Um problema notório é o do caminho mínimo em grafos ponderados. Um grafo ponderado é um par (G, w) em que G é um grafo e w é uma função que associa a cada arco de G um número real, dito o *custo* do arco. Um *caminho* em um grafo é uma sequência finita de vértices na qual vértices adjacentes na sequência formam um arco; dizemos que um caminhe *une* o seu vértice inicial ao seu vértice final. A noção de custo é facilmente levantada para caminhos: o *custo* de um caminho é a soma dos custos dos arcos presentes no caminho. Um caminho π que une os vértices s e t é dito *mínimo* se o custo de π é menor ou igual ao custo de qualquer outro caminho que une s e t. A forma básica do problema do caminho mínimo apresenta duas variantes, a primeira consiste em fixar uma *origem* e determinar, se existir,

um caminho mínimo que leva da origem até cada um dos demais vértices, e a segunda, é a versão do problema na qual não há uma origem fixa e o objetivo consiste em determinar um caminho mínimo, se existir, entre cada par de vértices. Há uma série de algoritmos populares para encontrar caminhos mínimos: o algoritmo de Dijkstra resolve a versão (1) do problema quando o custo dos arcos é não-negativo; o algoritmo de Bellman-Ford também resolve a versão (1) do problema quando todo *ciclo* — i.e., um caminho cujos únicos vértices que se repetem são o inicial e o final — do grafo tem custo não-negativo; e o algoritmo de Floyd-Warshall que resolve a versão (2) do problema nas mesmas condições do algoritmo de Bellman-Ford. Vale destacar que os três algoritmos constituem instâncias da técnica de programação dinâmica [4].

Além dos problemas de caminho mínimo em grafos, outro exemplo clássico de aplicação da programação dinâmica envolve Problemas de Decisão de Markov (MDP). Vamos definir a versão de um MDP estudado neste trabalho. Os ingredientes de um MDP são:

- um conjunto finito de *estados*;
- um conjunto finito de *decisões*;
- para cada estado s, é dado um conjunto $D(s)$, cujos elementos são as decisões disponíveis no estado s;
- para cada estado s e cada decisão a em $D(s)$, é dada uma distribuição de probabilidade $P(s,a)$ sobre o conjunto dos estados, que fornece a probabilidade de que o sistema se mova para um certo estado t quando no estado s é selecionada uma decisão a, disponível em s;
- para cada estado s e cada decisão a em $D(s)$, é dada uma recompensa $r(s,a)$ que é um número real; e
- um número natural N, chamado de *horizonte*.

Em virtude da natureza estocástica do problema, o objetivo aqui consiste em determinar uma política ótima. Uma *política* é uma função que associa uma decisão a cada estado e a cada inteiro n no conjunto $\{1,\ldots,N\}$. O *valor* de uma política é a esperança da recompensa recebida num horizonte de N tomadas de decisão. Um política é *ótima* se o seu valor é máximo.

4 Metodologia

A metodologia deste projeto é dividida em três tipos de atividades: estudo individual das técnicas e algoritmos envolvidos, reuniões periódicas com o orientador e atividades de programação através da resolução de problemas.

4.1 Estudo Individual

O roteiro de estudos foi guiado pelas referências [1, 4, 6, 5]. A cada semana um planejamento foi estabelecido em conjunto com o orientador, definindo os tópicos a serem estudados e os exercícios práticos a serem realizados. O aluno seguiu o planejamento, dedicando-se ao estudo teórico e aplicando os conceitos por meio de implementações e exercícios retirados da bibliografia recomendada. As dificuldades encontradas foram discutidas e resolvidas durante as reuniões semanais e por meio de mensagens quando necessário.

4.2 Reuniões com o Orientador

As reuniões semanais com o orientador desempenharam um papel fundamental no desenvolvimento do projeto. Esses encontros proporcionaram a oportunidade para discutir o progresso do estudo, validar os conceitos aprendidos e superar as dificuldades enfrentadas. Sempre que houvesse dificuldade na compreensão de um tópico pelo aluno, eram realizadas reuniões adicionais, onde o orientador apresentava explicações detalhadas e respondia às dúvidas do aluno. Essa interação direta foi essencial para garantir uma compreensão mais profunda dos conteúdos abordados.

4.3 Atividades de Programação

A implementação prática dos conceitos foi realizada em C++, com foco na resolução de problemas. Os exercícios foram selecionados pelo orientador, provenientes de repositórios de problemas, como `AtCoder`[9] e `Codeforces`[8], que oferecem exercícios práticos alinhados com o conteúdo estudado. Essa prática permitiu consolidar o aprendizado teórico, reforçando a compreensão das técnicas e algoritmos estudados e sua aplicação em cenários reais.

5 Resultados e discussão dos resultados

Uma boa forma de mensurar o domínio e aprendizado é através da aplicação dos conceitos estudados, por meio de implementação e resolução de problemas. Além dos algoritmos apresentados neste documento houveram outros que foram implementados e se encontram em um repositório público[2].

5.1 Algoritmos Introdutórios

No intuito de criar uma base sólida para prosseguir com o estudo dos temas deste projeto, foram resolvidos problemas de tópicos introdutórios e fundamentais para a compreensão de algoritmos mais avançados. Evidenciando a aplicabilidade do conceito de vetores e demais estruturas de dados básicas vistas até o momento, segue, para ilustrar, a resolução de um exercício do `Neps Academy`[3] que foi resolvido utilizando vetores dinâmicos e dicionários, duas estruturas da biblioteca padrão, STL, do C++.

```
void solve(){
  int n; cin >> n;
  map<int, int> cnt;
  vector<int> v(n);

  for(auto &x : v) cin >> x;
  for(auto &y : v) cnt[y]++;
  int maxocrr = 0;
  for(auto &par: cnt) if(par.second > maxocrr)
        maxocrr = par.second;
  for(const auto x: cnt) if(x.second == maxocrr)
        cout << x.first << "␣";
```

```
  cout << endl;
}
```

O seguinte exemplo, que é uma implementação do algoritmo de ordenação por inserção, ilustra o uso de estruturas e listas ligadas.

```
struct node {
        int info;
        node *next;
};

node *make_node(int info, node *next){
        node *p = new node();
        p->info = info; p->next = next;
        return p;
}

node *build(int *b, int *e){
        if(b==e) return nullptr;
        node *h = build(b+1, e);
        return make_node(*b,h);
}

node *isorting(node *h){
        if(!h || !h->next) return h;
        node dummy;
        dummy.next = h;
        node *prev = h, *cur = h->next;

        while(cur){
        if(cur->info >= prev->info){
                prev = cur;
                cur = cur->next;
        }
        node *tmp = &dummy;
        while(cur->info > tmp->next->info){
                tmp = tmp->next;
        }
        prev->next = cur->next;
        cur->next = tmp->next;
        tmp->next = cur;
        cur = prev->next;
        return dummy.next;
}

void pr_list(node *h) {
        if (h == nullptr) return;
        cout << h->info << "␣";
```

```
        pr_list(h->next);
}
```

Um algoritmo bem relevante em árvores binárias é a varredura, que é um algoritmo relativamente simples se implementado de forma recursiva (isso devido ao fato de árvores binárias serem estruturas naturalmente recursivas, assim como listas ligadas) porém menos natural se implementado de forma iterativa. Eis, então, uma versão iterativa do algoritmo de varredura em árvores binárias, usando `union` e pilhas:

```
template <typename T>
struct node {
  T info;
  node *left, *right;
  node() {};
  node(T i, node *l, node *r) {
    info = i; left = l; right = r;
  }
};

template <typename T>
inline node<T> *null() {
  static node<T> p = node<T>();
  return &p;
}

template <typename T>
inline bool is_null(node<T>* p) {
  return p == null<T>();
}

node<int> *build(int b, int e) {
  if (b == e) return null<int>();
  int m = b + (e-b)/2;
  node<int> *left = build(b, m);
  node<int> *right = build(m+1, e);
  return new node<int>(m, left, right);
}

template <typename T>
union obj {
  T value;
  node<T> *tree;
  obj(){}
};

template <typename T>
struct vert {
  int tag;
```

```
  obj<T> uni;
};

template<typename T>
void empilha(stack<vert<T>>&pilha, T valor) {
  vert<T> vert;
  vert.tag = 1;
  vert.uni.value = valor;
  pilha.push(vert);
}

template<typename T>
void empilha(stack<vert<T>>&pilha, node<T>* noh){
  vert<T> vert;
  vert.tag = 0;
  vert.uni.tree = noh;
  pilha.push(vert);
}

template <typename T>
void erd(node <T> *r) {
  if(r == nullptr) return;
  stack<vert<T>> pilha;
  empilha(pilha, r);
  vert<T> x;
  while(!pilha.empty()){
    x = pilha.top(); pilha.pop();
    if(x.tag == 0 && x.uni.tree != nullptr){
      empilha(pilha, x.uni.tree->right);
      empilha(pilha, x.uni.tree->info);
      empilha(pilha, x.uni.tree->left);
    } else if(x.tag == 1 && x.uni.value != 0){
      cout << x.uni.value << endl;
    }
  }
}
```

A STL do C++ permite utilizar o tipo `variant` que simplifica a implementação do algoritmo anterior, como mostrado no código abaixo:

```
template <typename T>
void erd(node<T>* r) {
  stack<variant<T, node<T>*>> s;
  s.push(r);
  while (!s.empty()) {
    auto x = s.top(); s.pop();
    if (holds_alternative<node<T>*>(x)) {
      auto r = get<node<T>*>(x);
```

```
      if (is_null(r)) continue;
      s.push(r->right);
      s.push(r->info);
      s.push(r->left);
    } else {
      T info = get<T>(x);
      cout << info << '␣';
    }
  }
}
```

Um problema de otimização em um conjunto finito pode ser resolvido de uma maneira conceitualmente simples explorando-se todo o espaço de busca, o que chamamos de *busca exaustiva*. Essa abordagem pode ser extremamente ineficiente em termos de tempo computacional, caso, por exemplo, o espaço de busca tenha tamanho exponencial em relação ao tamanho do problema, mas é útil para problemas pequenos ou quando não conhecemos uma alternativa melhor. A seguir, apresentamos um exemplo que ilustra a busca exaustiva: encontrar um emparelhamento máximo em um grafo bipartido.

```
size_t max_bip_matching(graph& G, it b, it e) {
    set<int> matched;
    size_t max_size = 0;
    auto backt = [&](auto&& backt, it b) -> void {
        if (b == e)
            max_size = max(max_size, matched.size());
        else {
            backt(backt, b+1);
            for (int k: G[*b]) {
                if (matched.find(k) != matched.end())
                    continue;
                matched.insert(k);
                backt(backt, b+1);
                matched.erase(k);
            }
        }
    };
    backt(backt, b);
    return max_size;
}

map<int, int> max_bip_matching_wt(graph& G, it b, it e) {
    map<int, int> matching;
    set<int> matched;
    map<int, int> max_size_matching;
    auto backt = [&](auto&& backt, it b) -> void {
        if (b == e)
            if (max_size_matching.size() < matching.size())
                max_size_matching = matching;
```

```
        else {
            backt(backt, b+1);
            for (int k: G[*b]) {
                if (matched.find(k) != matched.end())
                    continue;
                matching[*b] = k;
                matched.insert(k);
                backt(backt, b+1);
                matching.erase(*b);
                matched.erase(k);
            }
        }
    };
    backt(backt, b);
    return max_size_matching;
}
```

5.2 Algoritmos em grafos de nível introdutório

5.2.1 Busca em Profundidade

O problema "Counting Rooms" do CSES[7] consiste em, dado um grid retangular representando um mapa, contar o número de salas (regiões conectadas de espaços vazios) no mapa. O mapa é composto por células que podem ser espaços vazios ('.') ou paredes ('#'). O problema pode ser resolvido usando-se uma busca em profundidade que percorre o grid marcando as células vazias visitadas e contando o número de salas.

```
using graph = vector<vector<char>>;
const vector<int> nx {0, 1, 0, -1}, ny {1, 0, -1, 0};
void dfs_count(graph& g, int x, int y, vector<vector<bool>>&
visited) {
    int n = g.size();
    int m = g[0].size();
    visited[x][y] = true;
    for(int i = 0; i < 4; i++) {
        int dx = x + nx[i], dy = y + ny[i];
        if (dx >= 0 && dx < n && dy >= 0 && dy < m &&
            g[dx][dy] == '.' && !visited[dx][dy]){
            dfs_count(g, dx, dy, visited);
        }
    }
}

void solve() {
    int n, m; cin >> n >> m;
    graph g(n, vector<char>(m));
    for (int i = 0; i < n; i++) {
        string s; cin >> s;
```

```
        for (int j = 0; j < m; j++) {
            g[i][j] = s[j];
        }
    }
    int ans = 0;

    vector<vector<bool>> visited(n, vector<bool>(m, false));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if(g[i][j] == '.' && !visited[i][j]){
                dfs_count(g, i, j, visited);
                ans++;
            }
        }
    }
    cout << ans << '\n';
}
```

5.2.2 Busca em Largura

O problema "Móbile" do NEPS[3] consiste em, dado um grafo não direcionado, onde cada vértice representa uma peça de um móbile e cada aresta representa uma conexão entre duas peças, determinar se é possível construir um móbile equilibrado, ou seja, se todas as peças ficam na mesma posição horizontal. A resolução é feita utilizando o algoritmo de busca em largura para determinar a distância de cada vértice do ponto inicial, após isso, basta ordenar e percorrer o vetor de distância verificando se todas as peças que estão na mesma distância têm o mesmo número de conexões, se isso acontecer, o móbile é equilibrado.

```
void bfs(int s, vector<vi>& graph, vector<optional<int>>&
dist) {
  dist[s] = 0;
  queue<int> q; q.push(s);
  while (!q.empty()) {
    int u = q.front(); q.pop();
    for (auto& v: graph[u]) {
      if (dist[v].has_value()) continue;
      dist[v] = dist[u].value() + 1;
      q.push(v);
    }
  }
}

void solve(){
  int n; cin >> n;
  vector<vector<int>>graph(n+1);
  for (int i = 0; i < n; i++) {
    int u, v; cin >> u >> v;
    graph[v].push_back(u);
```

```
  }
  vector<optional<int>> dist(n+1);
  bfs(0, graph, dist);
  bool ok = true;

  vector<pair<int, int>> vp(n+1);
  for(int i = 0; i <= n; i++)
    vp[i] = {dist[i].value(), i};


  sort(vp.begin(), vp.end());
  int j = 0;
  for (int i = 1; i <= n; i++, j++) {
    if (vp[i].first != vp[j].first) continue;
    if(graph[vp[j].second].size()!=graph[vp[i].second].size())
      ok = false; break;
  }
  cout << (ok? "bem" : "mal") << '\n';
}
```

5.2.3 Dijkstra

O problema "Caminho das Pontes" do NEPS[3] consiste em, dado um grafo ponderado, com n vértices e m arestas, encontrar o caminho de custo mínimo entre o vértice 0 (origem) e o vértice $n-1$ (destino), vértices esses que são representados como pontes no problema.

```
using ii = pair<int,int>;
void solve(){
  int n, m; cin >> n >> m;
  vector<vector<ii>> AL(n);
  n += 2;
  for(int i = 0; i < m; i++) {
    int u, v, w; cin >> u >> v >> w;
    AL[u].emplace_back(v, w);
    AL[v].emplace_back(u, w);
  }
  vector<optional<int>> dist(n);
  dist[0] = 0;
  set<ii> pq;
  pq.emplace(0, 0);

  while (!pq.empty()) {
    auto [d, u] = *pq.begin();
    pq.erase(pq.begin());
    for (auto &[v,w]: AL[u]) {
      if (!dist[u]) continue;
      if (dist[v] && dist[u].value() + w >= dist[v].value())
            continue;
```

```
      if (dist[v])
            pq.erase(pq.find({dist[v].value(), v}));
      dist[v] = dist[u].value() + w;
      pq.emplace(dist[v].value(), v);
    }
  }
  cout << dist[n-1].value() << '\n';
}
```

5.3 Programação Dinâmica Unidimensional

Além do exemplo abaixo, também implementamos os problemas Weighted Interval Scheduling[6] e Cut Rod[4], cuja leitura é recomendada, presentes no repositório do `Github`[2]. Um excelente exemplo de programação dinâmica unidimensional é o problema "Frog 1" do `Atcoder`[9]: Dado uma sequência de pedras em um caminho, um sapo precisa pular de uma pedra para outra, começando da primeira pedra e chegando à última pedra. O sapo pode pular para a próxima pedra (distância 1) ou para a segunda próxima pedra (distância 2). O objetivo é encontrar o custo mínimo total para o sapo chegar à última pedra.

Uma questão bastante interessante foi a descoberta de um problema de performance ao utilizarmos o `std::optional` na resolução deste problema. Embora essa ferramenta tenha sido amplamente empregada em exemplos anteriores para melhorar a legibilidade e a "elegância" do código, até então não havia apresentado nenhum impacto significativo na performance dos algoritmos. No entanto, na implementação do código abaixo, observou-se uma diferença drástica no tempo de execução esperado do algoritmo:

```
void solve(){
    int n; cin >> n;
    vector<int>v(n);
    for(auto &x: v)
        cin >> x;
    vector<optional<int>> memo(n, nullopt);
    auto dp = [&](auto&& dp, int i) -> int{
        if(memo[i]) return *memo[i];
        if (i == 0) return 0;
        if (i == 1) return abs(v[1] - v[0]);
        return *memo[i] = min(dp(dp, i-1)+abs(v[i-1]-v[i]),
                              dp(dp, i-2)+abs(v[i-2]-v[i]));
    };
    cout << dp(dp, n-1) << '\n';
}
```

Tempo de execução: 2210 ms.

Mesmo com o uso de `pragmas` não conseguimos melhoras significativas, decidimos então testar o uso de um vector de inteiros(no lugar do optional de inteiros) com um valor simbólico de 1e9 para representar o que seria a ausência de valor do optional e a performance melhorou drasticamente atingindo o resultado esperado: Tempo de execução: 7ms.

Por questão de analise, decidimos testar a performance com o uso da estrutura `unordered_map` da STL no lugar do vector e o resultado foi condizente com o esperado:

```
void solve(){
    int n; cin >> n;
    vector<int>v(n);
    for(auto& x: v)
        cin >> x;
    unordered_map<int, int> memo;
        auto dp = [&](auto&& dp, int i) -> int{
        if(memo[i]) return *memo[i];
        if (i == 0) return 0;
        if (i == 1) return abs(v[1] - v[0]);
        return *memo[i] = min(dp(dp, i-1)+abs(v[i-1]-v[i]),
                              dp(dp, i-2)+abs(v[i-2]-v[i]));
    };
    cout << dp(dp, n-1) << '\n';
}
```

Tempo de execução: 12ms.

Uma consulta a documentação do `std::optional` evidentemente apontou que há um *overhead* inevitável em sua utilização, o que surpreende é que tal *overhead* tenha sido tão grande em um problema tão simples.

5.4 Programação Dinâmica Multidimensional

Além do exemplo abaixo, também implementamos os problemas: Matrix Chain Multiplication[4], Longest Common Subsequence[4] e Optimal Binary Search Trees[4]. As implementações estão disponíveis no diretório "dp and greedy" do repositório[2]

O problema escolhido é o `Kongey Donk` disponível na plataforma do `Codeforces`[8]. O problema consiste em encontrar o caminho de maior valor em um grid bidimensional, onde cada célula representa um galho com um valor representando a quantidade de bananas naquele galho. O objetivo é começar em qualquer uma das células da primeira coluna à esquerda e encontrar o caminho de maior valor até a última coluna, podendo se mover apenas para as células adjacentes na coluna à direita.

```
using ll = long long;
using vi = vector<long long>;
void solve() {
    ll n, h; cin >> n >> h;
    vector<vi> tree(n, vi(h)), dp(n, vi(h, 0));
    for (ll i = 0; i < n; i++)
        for (ll j = 0; j < h; j++)
            cin >> tree[i][j];

    auto step = [&](auto&& step, ll i, ll j) -> ll {
        if (j == h || i == n || i < 0) return 0;
```

```
        if (dp[i][j] > 0) return dp[i][j];
        ll q = 0;
        if (i == 0 || i == n-1)
            q = max(step(step, i, j+1),
                    i == 0 ? step(step, i+1, j+1) : step(step, i-1, j+1));
        else q = max(step(step, i, j+1), max(step(step, i+1, j+1),
                                             step(step, i-1, j+1)));
        return dp[i][j] = q + tree[i][j];
    };
    ll ans = step(step, 0, 0);
    for (ll i = 1; i < n; i++) {
        ans = max(ans, step(step, i, 0));
    }
    cout << ans << '\n';
}
```

5.5 Programação Dinâmica em Grafos

O problema escolhido é praticamente uma aplicação direta do algoritmo de Floyd Warshall, sendo disponibilizado na plataforma CSES[7] pelo nome de "Shortest Routes II". Dado um grafo ponderado com n cidades e m estradas, o objetivo é encontrar a distância mínima e se existe um caminho entre cada par de cidades a e b com distância c dadas por q consultas. Importante notar que um algoritmo que calcula o caminho mínimo a partir de um só vértice, como o Bellman-Ford(implementação no repositório do projeto[2]) e o Dijkstra poderia ser chamado várias vezes para responder todas as consultas. O que pode ser pouco eficiente comparado com o Floyd Warshall que calcula de uma vez o caminho mínimo entre cada vértice do grafo.

```
using ll = long long;
using graph = vector<vector<ll>>;
void solve(){
    const ll INF = 1e18;
    ll n, m, q; cin >> n >> m >> q;
    graph g(n, vector<ll>(n, INF));

    for (int i = 0; i < n; i++) g[i][i] = 0;
    for (int i = 0; i < m; i++) {
        ll u, v, w; cin >> u >> v >> w;
        u--; v--;
        g[u][v] = min(w, g[u][v]);
        g[v][u] = min(w, g[v][u]);
    }

    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
```

```
            }
        }
    }
    for (int i = 0; i < q; i++) {
        ll a, b; cin >> a >> b;
        a--; b--;
        cout << (g[a][b] == INF? -1 : g[a][b]) << '\n';
    }
}
```

6 Conclusões e Trabalhos Futuros

O estudo desses conceitos, principalmente programação dinâmica, costumam apresentar um desafio para grande parte dos alunos, porém um bom planejamento permite que esse trajeto seja realizado de forma fluída e com uma boa estruturação da base para o aprendizado de algoritmos e técnicas mais complexas.

O orientando passou por desafios nada triviais na compreensão e implementação dos algoritmos e técnicas estudadas, esses que forneceram uma fundação sólida nos conceitos base para o aluno explorar a técnica de otimização com programação dinâmica em diferentes tipos de problemas, fornecendo uma boa capacitação para futuras pesquisas(objetivo deste projeto), principalmente nas áreas de Otimização Combinatória, Teoria dos Grafos e Pesquisa Operacional.

No futuro, há muitas oportunidades de expandir este trabalho, uma abordagem promissora e que recebeu interesse do orientando é a aplicação dos princípios da programação dinâmica em modelos estocásticos, focando na área de Pesquisa Operacional. O envolvimento de incertezas nas transições de estados promovem desafios interessantes e com uma diversa gama de aplicações. Seguir por esse eixo abrirá novas oportunidades de pesquisa na otimização de sistemas dinâmicos e incertos, como os encontrados em operações logísticas e sistemas financeiros.

Referências

[1] Feofilof, Paulo, *Projeto de Algoritmos(em C)*. Disponível em: `https://www.ime.usp.br/~pf/algoritmos/index.html`. Acesso em 26 mar. 2024.

[2] Tsurushima, Gabriel A. G., *Dynamic Programming Techniques in Graphs*. Disponível em: `https://github.com/tsurubr/DP_Graphs`. Acesso em: 24 set. 2024.

[3] NEPS ACADEMY, *Exercícios de programação*. Disponível em: `https://neps.academy/br/exercises`. Acesso em 22 mar. 2024.

[4] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., *Introduction to algorithms(4th ed.)*, MIT Press, 2022.

[5] Eric V. Denardo. *Dynamic Programming: Models and Applications*. Dover Publications, 2003.

[6] Tardos, E., Kleinberg, J., *Algorithm Design*, Pearson, 2013.

[7] CSES, *CSES Problemset*. Disponível em: https://cses.fi/problemset/. Acesso em 22 sep. 2024.

[8] Codeforces, *Codeforces Problemset*. Disponível em: https://codeforces.com/problemset. Acesso em 24 sep. 2024.

[9] AtCoder, *AtCoder Contests*. Disponível em: https://atcoder.jp/contests/. Acesso em 24 sep. 2024.