UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
CENTRO DE CIÊNCIAS TÉCNOLÓGICAS – CCT
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO – BCC

JOÃO VITOR FRÖHLICH

FORMALIZAÇÃO E PROVA DE ALGORITMOS DE BUSCA DO MENOR CAMINHO EM GRAFOS UTILIZANDO COQ

JOINVILLE
2023

JOÃO VITOR FRÖHLICH

FORMALIZAÇÃO E PROVA DE ALGORITMOS DE BUSCA DO MENOR CAMINHO EM GRAFOS UTILIZANDO COQ

Trabalho de conclusão de curso submetido à Universidade do Estado de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação

Orientadora: Karina Giradi Roggia

JOINVILLE
2023

JOÃO VITOR FRÖHLICH

FORMALIZAÇÃO E PROVA DE ALGORITMOS DE BUSCA DO MENOR CAMINHO EM GRAFOS UTILIZANDO COQ

Trabalho de conclusão de curso submetido à Universidade do Estado de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação

Orientadora: Karina Giradi Roggia

BANCA EXAMINADORA:

Orientadora:

Dra. Karina Girardi Roggia
UDESC

Membros:

Dr. Cristiano Damiani Vasconcellos
UDESC

Dr. Gilmario Barbosa dos Santos
UDESC

Joinville, Junho de 2023

"*I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).* " (Linus Benedict Torvalds – What would you like to see most in minix?, [1991])

RESUMO

Um assistente de provas é um *software* que auxilia na formalização de provas matemáticas. Dentre as capacidades de um assistente de provas, como Coq, é possível modelar estruturas matemáticas complexas e *softwares*, que podem ser representados por meio de provas matemáticas. Entre essas estruturas matemáticas, podemos destacar os grafos, que possuem grande importância para a computação e algoritmos que visam modelar e explorar propriedades dessa estrutura. Porém, não foram encontradas muitas pesquisas sobre a formalização da implementação dos diferentes algoritmos que resolvem problemas da teoria de grafos. O objetivo deste trabalho é implementar alguns algoritmos de busca do menor caminho entre dois pontos em um grafo em Coq e provar a corretude dessas implementações.

Palavras-chave: Coq. Métodos Formais. Teoria de Grafos.

ABSTRACT

A proof assistant is a software that helps in the formalization of mathematical proofs. Among its capacities, a proof assistant like Coq is able to model complex mathematical structures and softwares, that can be represented as mathematical proofs. Graphs are a key mathematical structure in computer science with many algorithms dedicated to modeling and analyzing their properties. But little research has been done about formalizing the implementation from the different algorithms that solve the graph theory problems. The goal of this work is to implement some algorithms in Coq to find the shortest path between two points in a graph and verify their correctness.

Keywords: Coq. Formal Methods. Graph Theory.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Na computação, a matemática discreta desempenha um importante papel na modelagem e solução de problemas, especialmente pela utilização das diversas estruturas matemáticas complexas existentes, como categorias, grupos e monoides. Entre essas estruturas, podemos destacar os grafos, que são definidos como um conjunto não ordenado de vértices V, um conjunto de arestas E, disjunto de V, e uma função de incidência ψ, que associa a cada aresta do grafo um par não ordenado (e não necessariamente distinto) de vértices do grafo; a função de incidência também pode associar cada aresta do grafo a um par ordenado (e não necessariamente) distinto de vértices, onde o grafo é chamado de grafo direcionado (BONDY; MURTY et al., 1976). A principal característica de um grafo é que ele pode ser usado para representar elementos e as conexões entre eles, como por exemplo um grupo de pessoas e suas relações de amizade ou um conjunto de aeroportos e os voos entre eles. Considerando que as arestas podem ter pesos, se adquire a capacidade de representar outros problemas interessantes, como o menor tempo de deslocamento entre dois pontos em uma cidade ou calcular a capacidade necessária de distribuição de energia elétrica para suprir o consumo elétrico em uma cidade.

No contexto do uso de estruturas matemáticas complexas para a modelagem e solução de problemas na computação, é importante mencionar a área de verificação formal, que busca garantir que um programa ou algoritmo está correto em relação a uma especificação formal utilizando métodos matemáticos rigorosos para provar essa corretude, garantindo assim mais segurança às aplicações. Para auxiliar no processo de provar esses algoritmos, são utilizados softwares conhecidos como assistentes de provas, que permitem que os desenvolvedores especifiquem formalmente as propriedades a serem verificadas, enquanto o software executa as etapas da prova definidas pelo desenvolvedor de forma mecanizada, eliminando assim erros de lógica ou do raciocínio humano que podem ocorrer durante a execução manual de uma etapa de prova (GEUVERS, 2009). Alguns exemplos desses programas são aplicações como Coq e Lean, que trabalham com cálculo de construções indutivas, e Isabelle, que trabalha com lógica de alta ordem. Além de executar as etapas das provas e evitar erros lógicos, esses assistentes auxiliam na estruturação e organização das provas, permitindo o desenvolvimento de provas mais complexas e robustas. Um exemplo famoso do uso de assistentes é o teorema das quatro cores, conjecturado inicialmente em 1852 por Francis Guthrie, que buscava provar que as diferentes regiões de um mapa podiam ser coloridas usando apenas 4 cores, de forma que não existam duas regiões adjacentes com a mesma cor (GONTHIER et al., 2008). O teorema é considerado um marco para a verificação formal pois, após mais de um século de tentativas de se provar o teorema, haviam muitos casos específicos a serem provados, o que dificultava a organização da prova. Em 1976, Appel e Haken provaram o teorema com o auxílio dos computadores para organização dos diversos casos (APPEL; HAKEN, 1976), mas a limitada tecnologia da época tornou a prova controversa. Em 2005, Gonthier conseguiu desenvolver a prova do teorema no assistente de provas Coq, que centralizou a verificação formal em uma única aplicação e dispensou a necessidade de

acreditar em diversos programas diferentes para verificar casos específicos (GONTHIER, 2005). A partir do código da prova de Gonthier, foi desenvolvida a biblioteca Mathematical Components para o Coq, que se tornou uma ferramenta importante para a verificação formal de teoremas e provas matemáticas (MAHBOUBI; TASSI, 2022). Essa biblioteca oferece uma vasta coleção de definições, teoremas e algoritmos matemáticos formalmente verificados, permitindo aos usuários uma abordagem mais abstrata e elegante para a solução de problemas matemáticos complexos.

Apesar da prova do teorema das quatro cores ter sido um marco na história da utilização de assistentes de provas na prova de teoremas complexos, pouca pesquisa na área de grafos foi realizada desde então, mesmo considerando a relevância da verificação formal na construção de softwares mais seguros. Na área teórica, algumas teorias sobre isomorfismos de grafos foram provadas em (DOCZKAL; POUS, 2020a) e (DOCZKAL; POUS, 2020b), sendo que este último disponibilizou uma biblioteca pública [1] sobre teoria de grafos em Coq, implementada sobre a biblioteca Mathematical Components. Além destes, foi provada uma variante do Teorema de Wagner, que formalizou que um grafo que não contenha como subgrafo um grafo K_5 (grafo de 5 vértices onde cada nó possui uma aresta para todos os outros nós) ou um grafo $K_{3,3}$ (grafo bipartido com 3 nós em cada região, com cada nó possuindo uma aresta a todos os nós da região oposta) pode ter uma coloração de 4 cores (DOCZKAL, 2021). Isto se dá seguindo o teorema das quatro cores e o Teorema de Wagner, que prova que um grafo com as características acima é planar (WAGNER, 1937). Quanto à formalização de algoritmos de grafos, existem alguns trabalhos realizados em Isabelle/HOL disponíveis publicamente no Archive of Formal Proofs [2], incluindo a formalização de algoritmos famosos, como o algoritmo de Dijkstra (NORDHOFF; LAMMICH, 2012). Contudo, não foram encontrados artigos onde foram feitas formalizações de algoritmos de grafos em Coq. Portanto, o objetivo deste trabalho é explorar a área de formalização de algoritmos de grafos em Coq, especificamente de algoritmos da busca do menor caminho em grafos, provando também alguma propriedade sobre esses algoritmos, mostrando assim o potencial do desenvolvimento de pesquisas nessa área.

1.1 OBJETIVO GERAL

O objetivo geral deste trabalho é formalizar e provar em Coq algoritmos de busca do menor caminho entre dois pontos em grafos.

1.2 OBJETIVOS ESPECÍFICOS

Com base no objetivo geral, os seguintes objetivos específicos são definidos:

- Estudar os principais algoritmos determinísticos de busca do menor caminho de grafos
- Estudar os principais algoritmos heurísticos de busca do menor caminho em grafos

[1] <https://github.com/coq-community/graph-theory>
[2] <https://www.isa-afp.org/topics/computer-science/algorithms/graph/>

- Implementar alguns algoritmos de busca do menor caminho em assistente de provas, que serão escolhidos de acordo com critérios a serem estabelecidos
- Provar a corretude da implementação dos algoritmos definidos

1.3 ESTRUTURA DO TRABALHO

O trabalho está organizado da seguinte forma: no Capítulo 2 são apresentados conceitos e definições da teoria de grafos, bem como será discutido o problema da busca do menor caminho, no Capítulo 3 são apresentados algoritmos que implementam a busca do menor caminho, no Capítulo 4 é apresentada uma descrição sobre o assistente de provas Coq e a biblioteca Mathematical Components, no Capítulo 5 é descrita a formalização e prova da corretude dos algoritmos e, por fim, no Capítulo 6 são apresentadas conclusões parciais sobre o trabalho.

2 TEORIA DE GRAFOS

Para melhor compreender o conteúdo deste trabalho, é importante definir alguns conceitos de teoria de grafos.

2.1 GRAFOS DIRECIONADOS

Um grafo direcionado é uma tupla $G = \langle V, E, \delta_0, \delta_1 \rangle$, onde V é um conjunto de vértices, E é um conjunto de arestas, disjunto de V, δ_0 é uma função de mapeamento de uma aresta para um vértice $\delta_0 : E \to V$, indicando qual o vértice de origem da aresta, e por fim δ_1 é uma função de mapeamento de uma aresta para um vértice $\delta_1 : E \to V$, indicando qual o vértice destino da aresta. Nessa definição, são permitidas arestas paralelas e laços. A fim de restringir arestas paralelas e laços, que não serão tratados neste trabalho, serão impostas duas restrições: $\forall e \in E, \delta_0(e) \neq \delta_1(e)$ (restringe laços) e $\forall e_1, e_2 \in E, \delta_0(e_1) = \delta_0(e_2) \wedge \delta_1(e_1) = \delta_1(e_2) \implies e_1 = e_2$ (restringe arestas paralelas).

Um grafo direcionado pode ser representado graficamente em um plano, onde os vértices são representados como círculos e as arestas como setas, onde a origem da seta é o vértice de origem da aresta, e a ponta da seta é o vértice destino da aresta. Um exemplo de grafo é dado a seguir para ilustrar esses conceitos, onde o grafo $G = \langle V, E, \delta_0, \delta_1 \rangle$ tem $V = \{v_1, v_2, v_3, v_4\}$, $E = \{e_1, e_2, e_3, e_4, e_5\}$ e δ_0 e δ_1 mapeadas como descrito na Tabela 1, com uma ilustração na Figura 1:

	δ_0	δ_1
e1	v1	v2
e2	v1	v3
e3	v1	v4
e4	v2	v3
e5	v3	v2

Tabela 1 – Definição de δ_0 e δ_1 no exemplo 1

Dois vértices distintos u e v são ditos vizinhos se $\exists e \in E, (\delta_0(e) = u \wedge \delta_1(e) = v) \vee (\delta_0(e) = v \wedge \delta_1(e) = u)$. Se $\delta_0(e) = u \wedge \delta_1(e) = v$, então v é diretamente alcançável a partir de u.

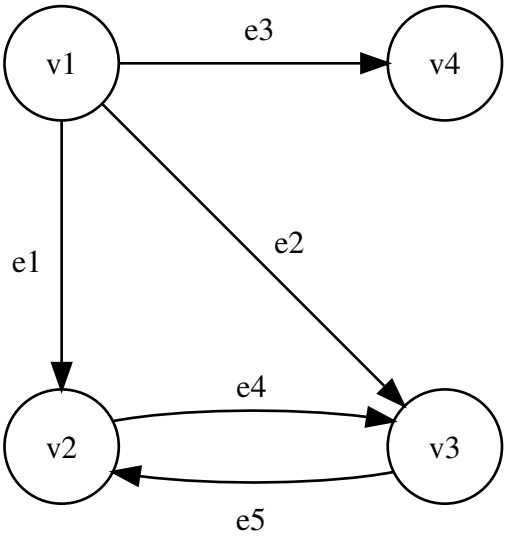

Figura 1 – Grafo direcionado
Fonte: O autor

2.2 GRAFOS DIRECIONADOS PONDERADOS

O conceito de peso pode ser adicionado à definição de grafos direcionados através da adição de mais uma função φ, definida como $\varphi : E \to \mathbb{R}$. No escopo deste trabalho, o peso será restrito a valores reais positivos não nulos, ou seja, a definição de φ se torna $\varphi : E \to \mathbb{R}^+$. Assim, a tupla representando um grafo direcionado ponderado se torna $G = \langle V, E, \delta_0, \delta_1, \varphi \rangle$.

Na ilustração de um grafo direcionado ponderado, a etiqueta das arestas são substituídas pelo peso da aresta, mapeado em φ. O exemplo anterior será modificado para representar essas mudanças da seguinte forma: $G = \langle V, E, \delta_0, \delta_1, \varphi \rangle$, com os valores de φ mapeados na Tabela 2. A representação gráfica deste novo grafo pode ser observada na Figura 2:

	φ
e1	2.5
e2	1.0
e3	5.0
e4	5.0
e5	0.5

Tabela 2 – Definição de φ no exemplo 2

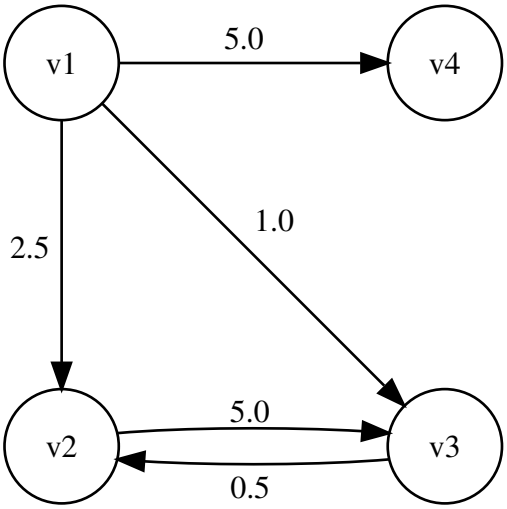

Figura 2 – Grafo direcionado ponderado
Fonte: O autor

2.3 CAMINHOS FINITOS

O conceito de caminho finito no grafo pode ser definido como uma lista não vazia de arestas $C = [e_1, e_2, ..., e_n]$, onde $\forall i \in \{1, n-1\}$, $\delta_1(e_i) = \delta_0(e_{i+1})$, ou seja, o destino de uma aresta no caminho deve ser a origem da aresta seguinte no caminho, com exceção da última aresta do caminho.

As funções δ_0, δ_1 e φ (em grafos ponderados) podem ser definidas para cada caminho finito C no grafo, onde

$$\delta_0(C) = \delta_0(e_1)$$

$$\delta_1(C) = \delta_1(e_n)$$

$$\varphi(C) = \sum_{i=1}^{|C|} \varphi(e_i)$$

Dados dois vértices distintos u e v, se $\exists C, \delta_0(C) = u \wedge \delta_1(C) = v$, então v é alcançável a partir de u.

2.3.1 Ciclos

Um ciclo é definido como qualquer caminho C onde $\delta_0(C) = \delta_1(C)$. Perceba que um caminho pode conter um ciclo como subcaminho.

2.3.2 Menor Caminho

Seja um grafo G, um conjunto de vértices V de G e dois vértices $u, v \in V$. O menor caminho de u para v será o caminho C' tal que $\forall C\,(\delta_0(C) = u \wedge \delta_1(C) = v), \varphi(C') = min(\varphi(C))$. No caso onde $u = v$, o menor caminho será sempre 0. Note que pode existir mais do que um menor caminho, como pode ser observado no exemplo a seguir: $G = \langle V, E, \delta_0, \delta_1, \varphi \rangle$, com

$V = \{v_1, v_2, v_3\}$, $E = \{e_1, e_2, e_3, e_4\}$ e δ_0, δ_1 e φ definidos na Tabela 3. Há também a ilustração deste grafo na Figura 3.

	δ_0	δ_1	φ
e1	v1	v2	4.0
e2	v1	v3	3.0
e3	v3	v2	1.0
e4	v2	v1	2.0

Tabela 3 – Definição de δ_0, δ_1 e φ no exemplo 3

Neste grafo podemos observar que existem diversos caminhos entre os vértices v_1 e v_2, como por exemplo $C_1 = [e_1]$, $C_2 = [e_2, e_3]$ e $C_3 = [e_1, e_4, e_1]$. Perceba que o caminho C_3 possui um ciclo como subcaminho e, por causa disso, existem infinitos caminhos entre v_1 e v_2, mas isso não faz com que existiam infinitos menores caminhos, como será discutido a seguir. Entre v_1 e v_2 existem dois menores caminhos, que são C_1 e C_2, onde $\varphi(C_1) = \varphi(C_2) = 4.0$. Note que mostrar que $\nexists C\,(\delta_0(C) = v_1 \wedge \delta_1(C) = v_2 \wedge \varphi(C) \leq 4.0)$ é trivial.

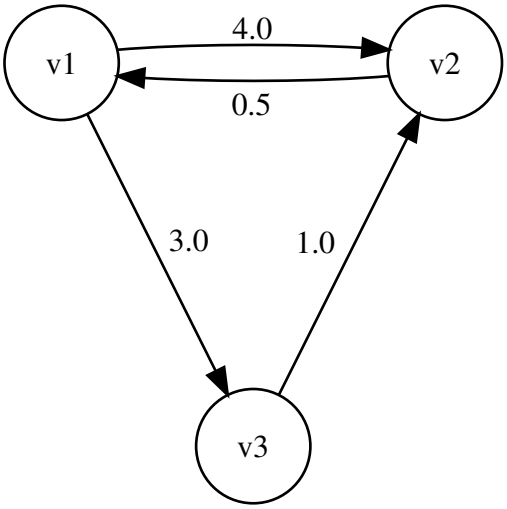

Figura 3 – Grafo para exemplificar o menor caminho
Fonte: O autor

2.3.3 Infinitos Menores Caminhos

Aqui será feita uma breve discussão sobre a possibilidade da existência de infinitos menores caminhos. Como visto no exemplo anterior, existiam infinitos caminhos entre v_1 e v_2. Como é possível garantir que não existem infinitos menores caminhos?

Na definição de grafos direcionados ponderados, o contradomínio da função φ foi restrito a valores positivos não nulos para evitar não apenas este problema, como também o possível problema do valor do menor caminho ser infinitamente negativo.

Para explicar melhor, suponha $\varphi : E \to \mathbb{R}_0^+$, ou seja, o contradomínio de φ é o conjunto dos números reais não negativos. Seja um grafo G com um caminho $C = [e_1, e_2]$, onde $\delta_0(C) =$

$\delta_1(C) = v$ (ou seja, C é um ciclo), $\varphi(e_1) = \varphi(e_2) = 0.0$ e $u = \delta_1(e_1) = \delta_0(e_2)$. Entre v e u que, por definição, é distinto de v, existem infinitos caminhos de custo 0.0, que é o menor caminho possível baseado na restrição de φ.

Suponha agora $\varphi : E \to \mathbb{R}$ e caminhos podendo ser infinitos. Considerando o exemplo anterior, alterando a função de peso para $\varphi(e_1) = -1.0$, considere um caminho $C_1 = C++[e_1] = [e_1, e_2, e_1]$. Note que $\varphi(C_1) = -2.0$. Seja outro caminho $C_2 = C++C_1$. Assim, $\varphi(C_2) = -3.0$. Note que quanto mais o ciclo C é utilizado na composição do caminho, menor se torna o peso do caminho. Se o ciclo C for utilizado infinitamente, $\varphi(C_\infty) = -\infty$.

Computacionalmente, a existência de infinitos menores caminhos, de um menor caminho infinito ou até mesmo de infinitos caminhos pode ser problemática. Por conta disso é que foram impostas as restrições no contradomínio da função φ e na definição de caminhos para este trabalho

2.4 REPRESENTAÇÃO COMPUTACIONAL

Um grafo pode ser representado computacionalmente de diversas maneiras. Aqui serão apresentadas duas das maneiras mais comumente utilizadas.

2.4.1 Matriz de Adjacências

Seja um grafo direcionado ponderado G com n vértices, a matriz de adjacências que representa esse grafo é uma matriz $n \times n$. Para toda posição $M_{i,j}$, onde i representa o vértice de origem e j o vértice de destino de uma possível aresta, tem-se:

$$M_{i,j} = \begin{cases} \varphi(e) & \forall e \mid \delta_0(e) = i \wedge \delta_1(e) = j \\ 0 & \text{se } i = j \\ -1 & \text{caso contrário} \end{cases} \tag{1}$$

Para o grafo da Figura 2, a matriz equivalente seria representada como na Tabela 4:

	v_1	v_2	v_3	v_4
v_1	0	2.5	1.0	5.0
v_2	-1	0	5.0	-1
v_3	-1	0.5	0	-1
v_4	-1	-1	-1	0

Tabela 4 – Matriz de adjacência

No caso de um grafo sem pesos, $\varphi(e)$ é substituído por 1.

A representação por matriz de adjacências normalmente é utilizada para representar grafos com muitas arestas ou em algoritmos que façam uso do acesso aleatório para verificar as arestas [1].

[1] Nesse caso, o acesso aleatório consiste em verificar a aresta sem precisar explorar todas as arestas do vértice de origem, apenas verificando a posição na matriz

2.4.2 Listas de Adjacências

Uma lista de adjacências é uma lista de pares definida para cada vértice i do conjunto V de um grafo direcionado ponderado G, onde

$$L_i = [(\delta_1(e), \varphi(e))], \forall e \mid \delta_0(e) = i$$

No grafo da Figura 2, as listas de adjacências seriam

L_1 = [(v_2, 2.5); (v_3, 1.0); (v_4, 5.0)]
L_2 = [(v_3, 5.0)]
L_3 = [(v_2, 0.5)]
L_4 = []

No caso de um grafo sem pesos, L_i é a lista de vértices diretamente alcançáveis a partir de i, ou seja, $L_i = [\delta_1(e)], \forall e\,(\delta_0(e) = i)$.

A representação por listas de adjacências normalmente é utilizada para representar grafos que não contenham muitas arestas. Sua utilização também é recomendada em algoritmos que exploram todas as arestas de um vértice.

3 ALGORITMOS DE BUSCA DO MENOR CAMINHO

Uma possível solução para o problema do menor caminho, discutido em 2.3.2 consiste em testar todos os caminhos possíveis entre dois pontos. Mas considere que alguém queira se deslocar da cidade de São Paulo para a cidade do Rio de Janeiro. Uma possível rota seria fazer um desvio por Belo Horizonte e só então se deslocar para o Rio de Janeiro, como pode ser observado na Figura 4, porém essa rota é muito mais longa do que ir pela rodovia BR-116, como mostrado na Figura 5.

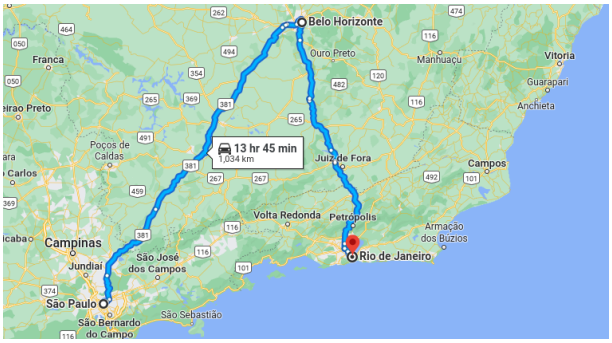

Figura 4 – Trajeto de São Paulo a Rio de Janeiro, passando por Belo Horizonte
Fonte: O autor (Google Maps)

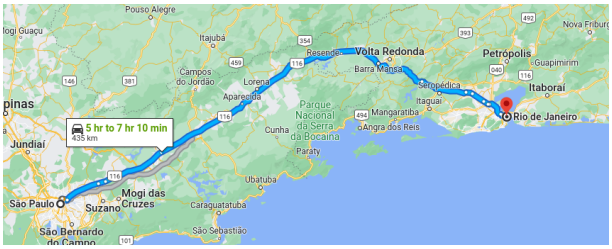

Figura 5 – Trajeto de São Paulo a Rio de Janeiro, passando pela rodovia BR-116
Fonte: O autor (Google Maps)

Partindo do princípio de que não adianta desviar muito do destino no meio do caminho, diversos algoritmos foram desenvolvidos para otimizar a busca pelo menor caminho entre dois pontos, dos quais alguns serão tratados neste trabalho. Afim de que se possa trabalhar na formalização destes algoritmos, é importante compreender o funcionamento dos mesmos. Por isso, esta seção discutirá tanto o funcionamento dos algoritmos a serem tratados como conterá pseudocódigos da implementação deles. Esta seção foi baseada majoritariamente em (CORMEN et al., 2022).

3.1 ALGORITMOS DE BUSCA EM GRAFOS

Antes de se aprofundar nos algoritmos propriamente ditos, é importante entender algumas técnicas de teoria de grafos, que serão utilizadas na implementação dos algoritmos a serem utilizados neste trabalho.

3.1.1 Busca em profundidade

A técnica da busca em profundidade (DFS) consiste em tentar visitar todos os vértices seguindo uma ordem de profundidade no grafo. Um vértice é dito descoberto uma vez que tenha sido visitado pela busca. Sendo assim, a busca se dá da seguinte forma: seja v o vértice descoberto mais recentemente, uma busca é realizada pelos vértices diretamente alcançáveis a partir de v que ainda não foram descobertos. Uma vez que se descubra um novo vértice, a busca é imediatamente realizada a partir desse novo vértice descoberto. Quando um vértice descoberto u não alcança diretamente nenhum vértice não descoberto, encerramos a busca nesse vértice e a busca continua a partir do seu antecessor, isto é, do vértice que descobriu u.

Afim de representar o antecessor de um vértice em uma busca, os vértices do conjunto V passarão a ser representados como tuplas: $v = \langle L_v, \pi_v, C_v \rangle$, onde L_v denota a identificação do vértice, π_v indica o antecessor do vértice v e C_v indica a coloração do vértice, que será utilizada para registar se o vértice já foi descoberto pela busca ou não.

Para a implementação do algoritmo, será utilizada uma função recursiva, que se mostra muito eficiente para esse tipo de busca, como mostrado no Algoritmo 1. Neste algoritmo, $C_v = Branco$ é a coloração de um vértice não descoberto e $C_v = Cinza$ é a coloração de um vértice descoberto. Além disso, $\pi_v = NIL$ indica que o vértice v não possui um predecessor definido.

Algoritmo 1 Algoritmo de DFS

```
1: Função DFS(G, u):
2:     Para todo v ∈ V Faça
3:         C_v ← Branco
4:         π_v ← NIL
5:     DESCOBRE-DFS(G, u)
1: Função DESCOBRE-DFS(G, u):
2:     C_u = Cinza
3:     Para todo e ∈ E | δ_0(e) == u Faça
4:         Se C_{δ_1(e)} == Branco Então
5:             π_{δ_1(e)} = u
6:             DESCOBRE-DFS(G, δ_1(e))
```

3.1.2 Busca em largura

Diferente da busca em profundidade, a busca em largura consiste em armazenar os vértices recém descobertos em uma fila, e explorar os mesmos seguindo a ordem em que foram adicionados. Em outras palavras, seja v o vértice na frente da fila. Para cada vértice u alcançável diretamente a partir de v e que ainda não tenha sido descoberto, u é inserido ao final da fila. Após se explorar cada aresta que tem v como origem, v é removido da fila, e o algoritmo segue até que a fila esteja vazia.

A mesma representação de V usada em 3.1.1 é utilizada neste algoritmo, e a implementação pode ser conferida no Algoritmo 2.

Algoritmo 2 Algoritmo de BFS

```
1:  Função BFS(G, u):
2:      Para todo v ∈ V Faça
3:          C_v ← Branco
4:          π_v ← NIL
5:      C_u ← Cinza
6:      Fila ← {u}
7:      Enquanto Fila não está vazia Faça
8:          u ← FRENTE(Fila)
9:          Para todo e ∈ E | δ_0(e) == u Faça
10:             Se C_{δ_1(e)} == Branco Então
11:                 C_{δ_1(e)} = Cinza
12:                 π_{δ_1(e)} = u
13:                 INSERE(Fila, δ_1(e))
14:         REMOVE-FRENTE(Fila)
```

3.1.3 Relaxamento

A técnica de relaxamento é um procedimento realizado na execução dos algoritmos de busca do menor caminho que serão tratados neste trabalho, que visa atualizar o limite superior da menor distância de um vértice de origem o para qualquer vértice do grafo G. Para representar este limite superior, será adicionada uma nova função no grafo $\varphi_V : V \to \mathbb{R}^+$, que mapeia cada vértice $v' \in V$ a um valor real não negativo que representa o limite superior da menor distância com que se alcança v' a partir de um vértice de origem o.

O processo de relaxamento de uma aresta $e \in E$ com origem em u e destino em v consiste em testar se alcançar v através de u passando pela aresta e, considerando $\varphi_V(u)$ e $\varphi(e)$, gera um custo menor que o limite superior de menor caminho de v e, em caso positivo, atualizar os valores de $\varphi_V(v)$ e π_v. Note que manter o antecessor de um vértice na busca pelo menor caminho será útil no processo de recuperar os vértices que compõem o menor caminho. O processo de relaxamento de um vértice u' consiste em executar o procedimento de relaxamento em todas as arestas partindo de u'. A implementação do relaxamento pode ser observada no Algoritmo 3.

Algoritmo 3 Relaxamento

1: **Função** RELAXA-ARESTA(G, e)
2: **Se** $\varphi_V(\delta_1(e)) > \varphi_V(\delta_0(e)) + \varphi(e)$ **Então**
3: $\varphi_V(\delta_1(e)) = \varphi_V(\delta_0(e)) + \varphi(e)$
4: $\pi_{\delta_1(e)} = \delta_0(e)$

1: **Função** RELAXA-VERTICE(G, u)
2: **Para todo** $e \in E \mid \delta_0(e) == u$ **Faça**
3: RELAXA-ARESTA(G, e)

3.2 ALGORITMO DE DIJKSTRA

O Algoritmo de Dijkstra é um algoritmo que resolve o problema do caminho mínimo de um vértice de um grafo direcionado ponderado para qualquer outro vértice. O algoritmo consiste de uma generalização da busca em largura para grafos ponderados. Seja G um grafo, V o conjunto de vértices de G, $o \in V$ um vértice de origem e S um conjunto de vértices que já possuem o menor caminho determinado de o até si. O algoritmo consiste em repetidamente selecionar um vértice $v \in V - S$ tal que o limite superior $\varphi_V(v)$ seja o menor entre os vértices do conjunto $V - S$, adicionar o vértice v a S e relaxar o vértice v, fazendo isso até que $V - S$ esteja vazio. Note que uma vez que um vértice $v \in V - S$ é selecionado, então o menor caminho até v está determinado. Para o problema do menor caminho entre dois pontos, a condição de parada do algoritmo se dá quando o vértice destino pertence a S.

Durante a inicialização do algoritmo, o antecessor de cada vértice é definido como nulo e o limite superior de custo de cada vértice é definido como infinito, o que não poderia ocorrer pelas definições deste trabalho. Contudo, a definição deste limite superior de custo infinito garante que quando um vértice for alcançado pela busca pela primeira vez, seu limite superior sempre será atualizado. A única exceção na definição de um limite superior ocorre para o vértice de origem da busca, cujo valor inicial é 0.

Uma propriedade que pode ser provada, e é importante para provar a complexidade de tempo do algoritmo, é a de que cada aresta do grafo é relaxada no máximo uma vez. Esta propriedade será explorada durante a formalização do algoritmo.

A implementação deste algoritmo pode ser observada no Algoritmo 4

Algoritmo 4 Algoritmo de Dijkstra

```
1: Função PEGA-MENOR-VALOR(L)
2:     ret ← NIL
3:     aux ← ∞
4:     Para todo v ∈ L Faça
5:         Se φ_V(v) < aux Então
6:             aux = φ_V(v)
7:             ret = v
8:     Retorna ret
1: Função DIJKSTRA(G, s)
2:     Para todo v ∈ V Faça
3:         π_v = NIL
4:         Se v == s Então φ_V(v) = 0
5:         Senãoφ_V(v) = ∞
6:     L ← [s]
7:     Enquanto L não está vazia Faça
8:         cur ← PEGA-MENOR-VALOR(L)
9:         REMOVE(L, cur)
10:        Para todo e ∈ E | δ_0(e) == cur Faça
11:            Se φ_V(δ_1(e)) > φ_V(δ_0(e)) + φ(e) Então
12:                INSERE(L, δ_1(e))
13:                RELAXA-ARESTA(G, e)
```

3.3 ALGORITMO DE BUSCA A*

O Algoritmo de Busca A* é um algoritmo que visa otimizar o Algoritmo de Dijkstra para a busca da menor distância entre dois pontos. A diferença entre eles está na forma de considerar o próximo vértice a ser relaxado (no caso da Busca A*, expandido). Enquanto o Algoritmo de Dijkstra considera apenas a distância percorrida até cada vértice, a Busca A* considera tanto a distância percorrida até o vértice, como também se utiliza de uma função heurística $h : V \to \mathbb{R}^+$, que mapeia cada vértice a um valor positivo não nulo, que representa uma aproximação da distância de cada vértice até o vértice destino.

Dessa forma, sendo G um grafo, V o conjunto de vértices de G, $o \in V$ um vértice de origem e S um conjunto de vértices que já possuem o menor caminho determinado pelo algoritmo de o até si, o algoritmo consiste em repetidamente analisar o vértice $v \in V - S$ tal que $\varphi_V(v) + h(v)$ seja o menor entre todos os vértices do conjunto $V - S$. Assim como na versão da busca pelo menor caminho entre dois pontos, a condição de parada é o vértice de destino ser analisado.

4 COQ

4.1 ASSISTENTES DE PROVAS

Os assistentes de provas são programas que auxiliam o desenvolvimento de provas formais, mas diferente de provadores automáticos, não provam teoremas com o apertar de um botão e necessitam que uma pessoa guie a prova (SILVA, 2019). Esses programas são muito úteis no processo de realizar provas matemáticas, como as que serão realizadas nesse trabalho, pois conseguem mecanizar os passos e organizar os teoremas utilizados nelas.

Os assistentes de provas começaram a ser desenvolvidos por volta da década de 1960 e 1970 quando, apesar dos provadores automáticos de teoremas ainda estarem em desenvolvimento, foi notado que a capacidade destes programas estava estagnando (HARRISON; URBAN; WIEDIJK, 2014). Nessa mesma época, alguns dos assistentes criados foram Automath, LCF, Mizar e PVS (GEUVERS, 2009).

Contudo, quando os assistentes de provas estavam começando a ser desenvolvidos, os matemáticos estavam céticos quanto a esses programas. Isso se deve ao fato de que erros na teoria central do assistente ou na sua implementação podem levar o assistente a provar sentenças falsas (SILVA, 2019). Mas este ceticismo foi diminuindo com o passar do tempo porque os assistentes que foram criados se mostraram mais confiáveis no processo de prova do que as provas tradicionais, uma vez que várias provas puderam ser desenvolvidas por causa dos assistentes e foram aceitas, como o teorema das quatro cores e o teorema dos números primos (GEUVERS, 2009).

Além de provas formais, os assistentes de provas podem ser utilizados na verificação formal de softwares e hardwares. O processo de verificação consiste em verificar a corretude de um sistema, através da descrição deste em termos matemáticos, permitindo assim que a corretude seja expressa como teoremas matemáticos (AVIGAD; MOURA; KONG, 2021).

Entre as principais vantagens da utilização de assistentes de provas, podemos destacar:

- A verificação mecânica é rápida e confiável;
- As provas são processadas interativamente, com informações sobre os estados da prova;
- Existem comandos que permitem buscar teoremas e lemas já provados;
- As provas podem ser automatizadas com métodos não-deterministas;
- Programas verificados podem ser extraídos para outras linguagens.

Quanto ao problema da falta de confiabilidade em assistentes de provas, o matemático Nicolaas Govert de Bruijn propôs como solução que o núcleo do assistente, isto é, o programa verificador, fosse um programa muito pequeno, cuja verificação possa ser realizada à mão, dando assim maior confiabilidade ao sistema (BARENDREGT; GEUVERS, 2001). Esta solução é conhecida como conceito de de Bruijn.

4.2 O ASSISTENTE DE PROVAS COQ

O Coq é um assistente de provas que usa como núcleo o formalismo do Cálculo de Construções Indutivas (BERTOT; CASTÉRAN, 2013). Semelhante ao Capítulo 3, a maior parte do seguinte texto é baseado no trabalho de (SILVA, 2019). Segundo o autor, o Coq é desenvolvido pelo instituto de pesquisa francês INRIA, desde 1984. Começou sendo chamado de CoC, uma alusão ao Cálculo de Construções, que era a base de implementação do assistente, mas passou a se chamar Coq quando foi estendido para suportar os tipos indutivos do Cálculo de Construções Indutivas. Além do desenvolvimento contínuo feito pelo INRIA, a comunidade de usuários do assistente desenvolve ferramentas e bibliotecas para a linguagem, como por exemplo a biblioteca *Mathematical Components* (MAHBOUBI; TASSI, 2022), que traz uma formalização para diversos campos da matemática.

O Coq é utilizado na Ciência da Computação principalmente como uma ferramenta de métodos formais para verificação de programas. Um exemplo da utilização do Coq nessa área pode ser encontrado em (LEROY et al., 2016), que implementou um compilador da linguagem C totalmente verificado em Coq. Além da verificação formal de programas, o assistente pode ser utilizado para provas de teoria de grafos, como pode ser observado na prova do teorema das quatro cores formalizada por (GONTHIER, 2005), bem como nos teoremas formalizados em (DOCZKAL; POUS, 2020b) utilizando a biblioteca *Mathematical Components*.

O assistente de provas Coq é dividido em quatro componentes:

- A linguagem de programação e especificação *Gallina*, que implementa o Cálculo de Construções Indutivas. Esta linguagem garante que qualquer programa ou prova escrita nela sempre termine.
- A linguagem de comandos *vernacular*, que faz a interação com o assistente.
- Um conjunto de táticas para realização das provas, que são traduzidas para termos em *Gallina*.
- A linguagem $\mathcal{L}$tac para implementar novas táticas e automatizar provas.

Para a utilização do Coq de forma interativa, as ferramentas CoqIDE, VSCoq e ProofGeneral podem ser utilizadas. Para este trabalho, a ferramenta VSCoq será utilizada. Como não existem mudanças sintáticas entre as ferramentas, os códigos mostrados neste trabalho podem ser replicados em qualquer ferramenta.

Para declarar provas no Coq, são utilizados os comandos `Theorem`, `Lemma`, `Example` ou `Corollary` seguidos dos termos que serão provados. Estes são equivalentes, cuja diferença real é refletida apenas na leitura da prova. Para iniciar uma prova, é utilizado o comando `Proof` e, para finalizar uma prova, são utilizados os comandos `Qed` ou `Defined`, indicando que a prova foi aceita, ou o comando `Admitted`, indicando que a prova foi aceita mas está incompleta. Também

pode ser utilizado o comando `Abort` para finalizar a prova, porém indicando que a prova não foi aceita.

Novos tipos podem ser definidos em Coq através dos comandos `Record`, para tipos não indutivos como, por exemplo, estruturas matemáticas como grafos, grupos e anéis, e `Inductive` e `Coinductive`, para tipos indutivos como, por exemplo, os números naturais. Funções também podem ser definidas, com o uso dos comandos `Definition`, para funções não recursivas, e `Fixpoint` para funções recursivas. No caso de funções recursivas, um dos argumentos precisa ter a redução explícita, por conta da propriedade da linguagem *Gallina* de sempre terminar os programas.

Os teoremas já provados e as definições podem ser expandidas com o comando `Check` e buscadas com os comandos `Search`, que realiza uma busca pelo nome dos teoremas e definições; e `Locate`, que realiza uma busca por símbolos usados nos mesmos.

Quando se está utilizando as ferramentas interativas, o estado da prova pode ser acompanhado durante todo o seu desenvolvimento. Nas ferramentas citadas, 3 janelas são exibidas, sendo uma para edição de texto, uma para mostrar o estado da prova e outra para exibir mensagens do sistema. Um exemplo da ferramenta VSCoq pode ser observado na Figura 6. O estado da prova é composto por dois conjuntos, onde um é o conjunto de hipóteses, ou contexto da prova, e o outro é o conjunto de objetivos da prova. Na janela do estado da prova, os conjuntos são divididos por uma linha horizontal, com o conjunto de hipóteses da prova estando acima desta linha e o conjunto de objetivos abaixo da linha.

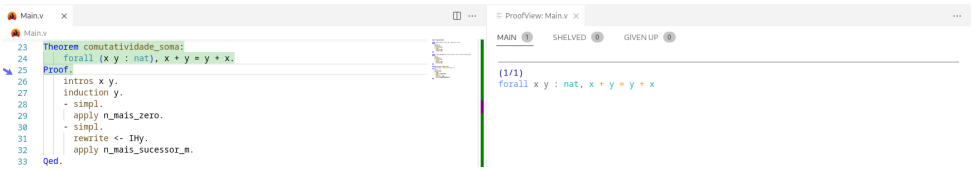

Figura 6 – Exemplo de um teorema provado no VSCoq
Fonte: O autor

Para manipular o estado da prova, existe um conjunto de táticas que podem ser utilizadas. Entre elas, pode-se citar as táticas (BARRAS et al., 1999):

- `intros`: move variáveis quantificadas universalmente e premissas de implicações do objetivo para o contexto;
- `simpl`: aplica reduções ao objetivo para algo que ainda seja legível, sem normalizar por completo;
- `reflexivity`: dado um objetivo com uma igualdade $t = u$, tenta verificar que t e u são iguais por definição;
- `rewrite` {teorema}: Sendo {teorema} uma igualdade, busca pelo lado esquerdo da igualdade no objetivo e o substitui pelo lado direito. Pode ser utilizado no sentido inverso com `rewrite` $\leftarrow$ {teorema}.

- `apply` {teorema}: utiliza unificação para equiparar os tipos presentes em {teorema} com o objetivo ou com uma hipótese, quando especificada.
- `destruct`: realiza uma análise de caso gerando um subobjetivo para cada construtor do tipo indutivo ou coindutivo selecionado;
- `induction`: Semelhante ao comando `destruct`, porém aplicando o princípio da indução.

Exemplo 1 (Tipos indutivos, definições, definições indutivas e provas)**.** Para exemplificar a utilização do assistente de provas Coq, serão mostrados nesta seção alguns exemplos de códigos. Inicialmente, será mostrada a definição de números naturais. Um número natural pode ser `O` (representando o 0) ou o sucessor de um número, representado pela função `S`. Desta forma, dizemos que 1 é representado por `S O`, 2 por `S (S O)` e assim por diante. Em Coq, a definição é como segue:

```
Inductive nat : Set :=
  | O
  | S: nat → nat.
```

A seguir, serão realizadas duas definições, sendo uma não recursiva e outra recursiva. A função `pred` retorna o predecessor de um número natural ou `O`, quando a entrada for `O`. Já a função `soma` realiza a operação de soma de dois números naturais de maneira recursiva.

```
Definition pred (n : nat) : nat :=
  match n with
  | O ⇒ O
  | S n' ⇒ n'
  end.

Fixpoint soma (n : nat) (m : nat) : nat :=
  match n with
  | O ⇒ m
  | S n' ⇒ S (soma n' m)
  end.
```

Uma definição indutiva pode ser utilizada para descrever relações, como por exemplo a relação `par`, que representa a paridade de números naturais. A relação `par` associa números naturais (`nat`) a provas (`Prop`).

```
Inductive par : nat → Prop :=
| par_O : par O
| par_SS (n : nat) (H : par n) : par (S (S n)).
```

Seguem agora alguns exemplos de teoremas provados utilizando as definições acima. O lema `auxiliar_1_ex_1` prova que o zero é um elemento neutro da operação de soma, enquanto que o lema `auxiliar_2_ex_1` prova que, dado dois números naturais `n` e `m` quaisquer, o sucessor

da soma de `n` e `m` é igual à soma de `n` com o sucessor de `m`. Por fim, utilizando estes dois lemas auxiliares, o teorema `exemplo_1` prova a comutatividade da soma. Além disso, o código `exemplo_2` mostra um exemplo do teste de paridade do número 4.

```
Lemma auxiliar_1_ex_1 : forall (n : nat),
soma n 0 = n.
Proof.
    intros.
    induction n.
    - reflexivity.
    - simpl.
    rewrite IHn.
    reflexivity.
Qed.

Lemma auxiliar_2_ex_1 : forall (n m : nat),
    S (soma n m) = soma n (S m).
Proof.
    intros.
    induction n.
    - reflexivity.
    - simpl.
    rewrite IHn.
    reflexivity.
Qed.

Theorem exemplo_1 : forall (n m : nat),
    soma n m = soma m n.
Proof.
    intros.
    induction n.
    - simpl.
    rewrite auxiliar_1_ex_1.
    reflexivity.
    - simpl.
    rewrite IHn.
    apply auxiliar_2_ex_1.
Qed.

Example exemplo_2 : par (S (S (S (S 0)))).
Proof.
    apply par_SS. apply par_SS. apply par_0.
```

```
Qed.
```

■

Por fim, o assistente de provas Coq conta com uma grande variedade de materiais para aprender a utilizá-lo, tais como o livro Coq'Art (BERTOT; CASTÉRAN, 2013), o projeto Software Foundations [1] e o livro Certified Programming with Dependent Types (CHLIPALA, 2022).

4.3 MATHEMATICAL COMPONENTS

A biblioteca *Mathematical Components* começou a ser desenvolvida a partir da prova do teorema das quatro cores no Coq, e busca formalizar a matemática para o assistente de provas, cobrindo desde teorias sobre estruturas básicas, como listas, números e conjuntos finitos até tópicos mais avançados sobre diferentes tipos de álgebra (MAHBOUBI; TASSI, 2022).

Esta biblioteca implementa diversos açúcares sintáticos além das estruturas, bem como implementa algumas mudanças sintáticas. Para este trabalho, aproveita-se da formalização de grafos como estruturas algébricas e da simplificação dos processos de provas para diminuir a carga necessária para a implementação dos algoritmos selecionados e para a prova da especificação dos mesmos.

Alguns trabalhos que utilizaram essa biblioteca para provas em grafos, além da prova do teorema das quatro cores, são (DOCZKAL; POUS, 2020b), que formaliza teoremas básicos e sobre isomorfismos em grafos; e (DOCZKAL; POUS, 2020a), que prova a integridade da axiomatização de isomorfismo de grafos através da reescrita de grafos.

As adições e mudanças na sintaxe que serão descritas aqui são baseadas no livro (MAHBOUBI; TASSI, 2022). Uma das principais adições sintáticas da biblioteca são as notações pós-fixas, sendo um bom exemplo a utilização da notação .+1 para denotar o sucessor de um número, que na sintaxe do Coq utiliza-se apenas a definição do tipo indutivo, `S`.

As principais mudanças sintáticas são percebidas nas táticas utilizadas nas provas, que o livro chama de *script* de prova. Destacam-se as seguintes mudanças com as táticas apresentadas anteriormente:

- `intros` é trocado por `move` ⇒ , fazendo essencialmente a mesma coisa;
- `simpl` é trocado por `rewrite` /=, porém com um poder maior de simplificação;
- `reflexivity` é trocado por `by` [] , onde [] é um conjunto de táticas que pode ser combinada com o comando `by` para ajudar a concluir a prova. Além disso, o comando `by` [] consegue concluir um conjunto maior de provas que o comando `reflexivity`;
- `rewrite` {`teorema`} é mantido, mas {teorema} ser precedido por ! e –, fazendo com que a reescrita seja aplicada repetidamente ou no sentido inverso, respectivamente;

[1] https://softwarefoundations.cis.upenn.edu/

- `apply {teorema}` é trocado por `apply: {teorema}`, fazendo essencialmente a mesma coisa;
- `destruct` é trocado por `case:`, fazendo essencialmente a mesma coisa;
- `induction` é trocado por `elim:`, fazendo essencialmente a mesma coisa.

Uma das grandes vantagens da biblioteca é a possibilidade de eliminar os casos triviais gerados pelos comandos `case:` e `induction:` usando a sequência de caracteres // logo antes do fim da declaração da tática que, assim como na sintaxe padrão, é denotada por . ou ; .

Exemplo 2 (Provas na *Mathematical Components*)**.** Para exemplificar a utilização da biblioteca *Mathematical Components*, o mesmo exemplo mostrado no Exemplo 1 será feito aqui também, porém como as definições de tipos e funções são idênticas na biblioteca padrão do Coq e na *Mathematical Components*, apenas as provas serão realizadas aqui. Contudo, algo importante a se ressaltar são os comandos utilizados para importar a biblioteca, que estão presentes no começo do código.

```
From mathcomp Require Import all_ssreflect.
Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.

Lemma auxiliar_1_ex_3 : forall (n : nat),
    soma n 0 = n.
Proof.
elim => [//| n IH].
by rewrite /= IH.
Qed.

Lemma auxiliar_2_ex_3 : forall (n m : nat),
    S (soma n m) = soma n (S m).
Proof.
move => n m.
elim: n => [//| n IH].
by rewrite /= IH.
Qed.

Theorem exemplo_3 : forall (n m : nat),
    soma n m = soma m n.
Proof.
move => n m.
elim: n => [|n IH].
```

```
    by rewrite auxiliar_1_ex_3.
by rewrite /= IH auxiliar_2_ex_3.
Qed.

Example exemplo_4 : par (S (S (S (S O)))).
Proof.
apply: par_SS.
by apply: par_SS par_0.
Qed.
```

■

5 FORMALIZAÇÃO E PROVA

Inicialmente será realizada a formalização e prova de uma implementação de caminhos ponderados, que possui tanto definições de caminhos como teoremas sobre estas definições que serão futuramente utilizadas. Para gerenciamento de versões do assistente de provas Coq e da biblioteca *Mathematical Components*, foi utilizado o gerenciador de pacotes da linguagem *OCaml*, *opam*. Neste trabalho, foram utilizadas as versões 8.17.0 do Coq e 1.17.0 da *Mathematical Components*. Estas escolhas foram feitas pelo fato da versão 1.17.0 da *Mathematical Components* ser a versão mais recente compatível com os códigos do livro (MAHBOUBI; TASSI, 2022) e a versão 8.17.0 do Coq ser a versão estável mais recente no momento do início da implementação deste trabalho.

Para a implementação, notou-se a necessidade de adaptar as definições descritas no Capítulo 2, as quais serão descritas no decorrer deste capítulo. O código completo da implementação e os *scritps* de prova podem ser encontrados em: <https://github.com/joao-frohlich/dijkstra-coq>.

Para um aprofundamento a respeito dos tipos e comandos utilizados nesta seção, sugere-se a leitura da documentação da *Mathematical Components*[1].

5.1 CAMINHOS FINITOS

Caminhos finitos serão representados como uma sequência de elementos de um tipo finito qualquer. Na *Mathematical Components*, tipos finitos são representados como instâncias `finType` da interface `Finite`, que por sua vez é uma subclasse da interface `Countable` e esta, por fim, é uma subclasse da interface `Choice`, cuja instância representa um tipo com um operador de escolha, e da interface `Equality`, cuja instância representa um tipo com igualdade decidível.

Para o peso dos caminhos, será declarada uma função que recebe dois elementos do tipo finito de um caminho, que simula uma aresta, e atribui um peso para estes parâmetros. Apesar de não definir a função para todos os elementos e mantê-la tão genérica quanto possível, será criada uma função auxiliar para descrever que o peso de uma aresta cujo início e fim é o mesmo vértice é zero.

Apesar da existência da estrutura de dados de lista na biblioteca padrão do Coq, será utilizada a estrutura de dados `seq` da *Mathematical Components*, pelo fato desta possuir mais suporte ao `finType`. Antes de mostrar qualquer trecho de código, vale ressaltar que os mesmos serão inseridos dentro de uma `Section`, para permitir a declaração de `Variable` e `Hypothesis` que, uma vez declaradas dentro de um `Section`, passarão para o contexto dos novos teoremas e poderão ser utilizadas nas funções definidas.

A primeira parte da implementação consiste na declaração da `Section`, junto com um `finType` e uma declaração de função de peso para uma aresta.

```
Section CaminhoPonderado.
```

[1] O índice da documentação pode ser encontrada em: <https://math-comp.github.io/htmldoc_1_17_0/index.html>

```
Variable T : finType.
Variable φ : T → T → nat.

Definition φ' (x y : T) :=
    if x == y then 0
    else φ x y.
```

Com estas definições, um caminho pode ser representado pelo tipo `seq T`. A partir disso, é possível definir como calcular o peso de um caminho qualquer através da seguinte função recursiva:

$$\varphi_C(c) := \begin{cases} 0 & \text{se c == [] ou c == \_ :: []} \\ (\varphi\ x\ y) + \varphi_C(\text{y :: c'}) & \text{se c == x :: y :: c'} \end{cases} \tag{2}$$

Nesta função recursiva é utilizada a decomposição de lista, onde no primeiro caso são consideradas listas vazias ou listas de um elemento, ou seja, caminhos sem arestas e, por definição, com peso nulo, e no segundo caso sendo consideradas listas com dois ou mais elementos, onde se soma o peso da aresta composta pelos dois primeiros vértices do caminho, x e y, ao cálculo recursivo da função φ_C sobre a lista de vértices sem o primeiro elemento.

Contudo, mesmo que seja perceptível que a lista decresce a cada chamada recursiva da função, o algoritmo do Coq não consegue concluir isso, por causa da operação `y :: c'`. Sendo assim, é preciso modificar um pouco a função φ_C, para que a redução na chamada recursiva fique explícita:

```
Fixpoint φ_C (c : seq T) : nat :=
    match c with
    | [::]    ⇒ 0
    | x :: c' ⇒ φ' x (head x c') + φ_C c'
    end.
```

Perceba a mudança sintática na definição de listas vazias. Isto se dá por causa das notações utilizadas pela *Mathematical Components*, que faz com que uma sequência vazia seja declarada como `[::]`. A função `head` recebe como parâmetros um elemento `x : T` e uma sequência `s : seq T`, onde `T` é um tipo qualquer, retornando `x`, se `s = [::]`, e `y`, caso `s = y :: s'`.

É importante definir também uma função para concatenação de caminhos que obedeça à restrição de que para dois caminhos serem concatenados, o último elemento do primeiro caminho deve ser igual ao primeiro elemento do segundo caminho, bem como à restrição dos dois caminhos não serem vazios. A função pode ser definida em Coq da seguinte forma:

```
Definition concat_caminho (x1 x2 : T) (s1 s2 C1 C2 : seq T) : seq T :=
    if (C1 == x1 :: s1) && (C2 == x2 :: s2)
        && (last x1 s1 == x2) then C1 ++ s2
    else [::].
```

A função `last` recebe como parâmetros um elemento `x : T` e uma sequência `s : seq T`, onde `T` é um tipo qualquer. O retorno dessa função é o último elemento da sequência `x :: s`.

Com a função para calcular o peso de um caminho qualquer definida e uma função de concatenação de caminhos, o próximo passo é provar que esta implementação está correta. Para isso, deve ser realizada a especificação formal das propriedades sobre caminhos ponderados. Para este escopo, foram definidas as seguintes propriedades, com seus respectivos teoremas declarados em Coq:

- Todo peso de caminho é positivo;

```
Lemma peso_caminho_positivo (c : seq T) : 0 <= peso_caminho c.
```

- A adição de um vértice ao início de um caminho qualquer C faz com que o peso deste novo caminho seja igual ao peso de C somado ao peso definido para a aresta que liga este novo vértice ao primeiro vértice de C;

```
Lemma peso_cons_caminho (x : T) (c : seq T) :
    peso_caminho (x :: c) = φ' x (head x c) + peso_caminho (c).
```

- O peso da concatenação de dois caminhos C_1 e C_2, desde que obedecendo às restrições impostas pela função de concatenação de caminhos, será igual à soma do peso de C_1 com o peso de C_2.

```
Lemma peso_concat_caminho (x1 x2 : T) (c1 c2 C1 C2 : seq T) :
    C1 = x1 :: c1 → C2 = x2 :: c2 → last x1 c1 == x2 →
    peso_caminho (concat_caminho x1 x2 c1 c2 C1 C2) =
    peso_caminho C1 + peso_caminho C2.
```

As duas primeiras provas são triviais e podem ser resolvidas pela tática `by []`. Para a prova da terceira especificação, sugere-se a prova de dois lemas auxiliares para facilitar o processo da prova do teorema principal, sendo eles:

```
Lemma peso_mesmo_vertice (x : T) : φ' x x = 0.

Lemma peso_concat_seq (x1 x2 : T) (s1 s2 : seq T) :
    φ_C ((x1 :: s1) ++ (x2 :: s2)) =
    φ_C (x1 :: s1) + φ_C (x2 :: s2) + φ' (last x1 s1) x2.
```

Note que o segundo lema auxiliar trata da concatenação de sequências, sem restrições, enquanto que a função `concat_caminho` e o teorema principal implementam restrições semelhantes às descritas no Capítulo 2 para caminhos.

O primeiro lema pode ser resolvido através da aplicação do teorema `eq_refl`, enquanto que o segundo pode ser resolvido por indução em `s1` com generalização de `x1`.

Após a prova dos lemas auxiliares, a prova do teorema principal pode ser realizada simplificando o lado esquerdo da igualdade para `x1 :: s1 ++ s2`, a partir da utilização das hipóteses do teorema e da abertura da definição de `concat_caminho`. Desse estado, pode ser realizada a análise de casos sobre `s2`, gerando como subobjetivos provar o teorema para `s2 = [::]` e provar o teorema para `s2 = x2' :: s2`.

O primeiro subobjetivo é trivial. O segundo subobjetivo transforma o lado esquerdo da igualdade em `x1 :: s1 ++ x2' :: s2`, o que permite a aplicação do lema auxiliar `peso_concat_seq`. Após uma aplicação do teorema `peso_cons_caminho` e manipulação na soma, o subobjetivo é provado por reflexividade e o teorema está provado.

6 CONSIDERAÇÕES PARCIAIS

A Teoria de Grafos é uma área muito importante no estudo da matemática e da ciência da computação, com diversos resultados e aplicações na resolução de problemas. Dentre as várias provas realizadas nessa área, algumas só foram possíveis com o uso de assistentes de provas, que são ferramentas que auxiliam no processo e organização das demonstrações. Tais aplicações não são restritas somente à teoria de grafos, podendo também ser utilizadas em outras áreas da matemática ou na especificação formal de sistemas computacionais complexos.

Entre as provas da teoria de grafos que foram realizadas em assistentes de provas, destaca-se a prova do Teorema das Quatro Cores que, além da importância no processo de aceitação dos assistentes de provas pela comunidade matemática, também serviu de base para a implementação de uma biblioteca focada no desenvolvimento de definições e teoremas sobre diversos campos da matemática discreta para o assistente de provas Coq, a *Mathematical Components*. Apesar disso, não existem muitas provas sobre teoria de grafos implementadas no Coq.

Com a escassez de formalização sobre algoritmos e teoremas já provados, percebe-se a necessidade de criar uma base sólida para a prova de novos teoremas para a teoria de grafos com o uso de assistentes de prova. Sendo assim, uma das primeiras contribuições deste trabalho pode ser percebida na implementação e especificação formal sobre caminhos ponderados, onde não foram encontrados estudos ou trabalhos sobre isso. Contudo, ainda está sendo estudada a modelagem de grafos no Coq de acordo com as definições deste trabalho.

Sendo assim, os resultados obtidos na formalização de caminhos ponderados indicam que é possível estender esse conceito para as modelagens de grafos existentes, modelando assim grafos ponderados e permitindo a formalização de algoritmos e teoremas que trabalhem com este conceito. Por isso, os seguintes itens são propostos para serem entregues no TCC2, bem como um cronograma para a execução de cada item:

1. Modelar grafos ponderados no Coq;
2. Implementar e formalizar o Algoritmo de Dijkstra;
3. Estudar heurísticas que geram o menor caminho na Busca A*;
4. Implementar e formalizar o Algoritmo de Busca A*.

Etapas	**2023/1**	**2023/2**				
	Jul	**Ago**	**Set**	**Out**	**Nov**	**Dez**
1						
2						
3						
4						

Tabela 5 – Cronograma Proposto para o TCC2

REFERÊNCIAS

APPEL, K; HAKEN, W. Every planar map is four colorable. **American Mathematical Society**, v. 82, n. 5, 1976. Citado na página 9.

AVIGAD, Jeremy; MOURA, Leonardo De; KONG, Soonho. Theorem proving in lean. **Online: https://leanprover. github. io/theorem_ proving_ in_ lean/theorem_ proving_ in_ lean. pdf**, 2021. Citado na página 23.

BARENDREGT, Henk; GEUVERS, Herman. Proof-assistants using dependent type systems. In: **Handbook of automated reasoning**. [S.l.: s.n.], 2001. p. 1149–1238. Citado na página 23.

BARRAS, Bruno et al. The coq proof assistant reference manual. **INRIA, version**, v. 6, n. 11, 1999. Citado na página 25.

BERTOT, Yves; CASTÉRAN, Pierre. **Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions**. [S.l.]: Springer Science & Business Media, 2013. Citado 2 vezes nas páginas 24 e 28.

BONDY, John Adrian; MURTY, Uppaluri Siva Ramachandra et al. **Graph theory with applications**. [S.l.]: Macmillan London, 1976. v. 290. Citado na página 9.

CHLIPALA, Adam. **Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant**. [S.l.]: MIT Press, 2022. Citado na página 28.

CORMEN, Thomas H et al. **Introduction to algorithms**. [S.l.]: MIT press, 2022. Citado na página 18.

DOCZKAL, Christian. A variant of wagner's theorem based on combinatorial hypermaps. 2021. Citado na página 10.

DOCZKAL, Christian; POUS, Damien. Completeness of an axiomatization of graph isomorphism via graph rewriting in coq. In: **Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs**. [S.l.: s.n.], 2020. p. 325–337. Citado 2 vezes nas páginas 10 e 28.

DOCZKAL, Christian; POUS, Damien. Graph theory in coq: Minors, treewidth, and isomorphisms. **Journal of Automated Reasoning**, Springer, v. 64, p. 795–825, 2020. Citado 3 vezes nas páginas 10, 24 e 28.

GEUVERS, Herman. Proof assistants: History, ideas and future. **Sadhana**, Springer, v. 34, p. 3–25, 2009. Citado 2 vezes nas páginas 9 e 23.

GONTHIER, Georges. **A computer-checked proof of the four colour theorem**. 2005. Citado 2 vezes nas páginas 10 e 24.

GONTHIER, Georges et al. Formal proof–the four-color theorem. **Notices of the AMS**, v. 55, n. 11, p. 1382–1393, 2008. Citado na página 9.

HARRISON, John; URBAN, Josef; WIEDIJK, Freek. History of interactive theorem proving. In: **Computational Logic**. [S.l.: s.n.], 2014. v. 9, p. 135–214. Citado na página 23.

LEROY, Xavier et al. Compcert-a formally verified optimizing compiler. In: **ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress**. [S.l.: s.n.], 2016. Citado na página 24.

MAHBOUBI, Assia; TASSI, Enrico. **Mathematical Components**. Zenodo, 2022. Disponível em: <https://doi.org/10.5281/zenodo.7118596>. Citado 4 vezes nas páginas 10, 24, 28 e 31.

NORDHOFF, Benedikt; LAMMICH, Peter. Dijkstra's shortest path algorithm. **Archive of Formal Proofs**, January 2012. ISSN 2150-914x. <https://isa-afp.org/entries/Dijkstra_Shortest_Path.html>, Formal proof development. Citado na página 10.

SILVA, Rafael Castro Goncalves. **Uma certificação em COQ do algoritmo W monádico. 2019. 78 p**. Tese (Doutorado) — Dissertação (Mestrado)-Universidade do Estado de Santa Catarina, Programa de . . . , 2019. Citado 2 vezes nas páginas 23 e 24.

WAGNER, Klaus. Über eine eigenschaft der ebenen komplexe. **Mathematische Annalen**, Springer, v. 114, n. 1, p. 570–590, 1937. Citado na página 10.