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

JOÃO VITOR FRÖHLICH

FORMALIZAÇÃO E VERIFICAÇÃO EM COQ DO ALGORITMO DE DIJKSTRA

JOINVILLE
2023

JOÃO VITOR FRÖHLICH

FORMALIZAÇÃO E VERIFICAÇÃO EM COQ DO ALGORITMO DE DIJKSTRA

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
Coorientador: Paulo Henrique Torrens

JOINVILLE
2023

JOÃO VITOR FRÖHLICH

FORMALIZAÇÃO E VERIFICAÇÃO EM COQ DO ALGORITMO DE DIJKSTRA

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
Coorientador: Paulo Henrique Torrens

BANCA EXAMINADORA:

Orientadora:

Dra. Karina Girardi Roggia
UDESC

Coorientador:

Me. Paulo Henrique Torrens
University of Kent

Membros:

Dr. Cristiano Damiani Vasconcellos
UDESC

Dr. Gilmario Barbosa dos Santos
UDESC

Joinville, Novembro de 2023

AGRADECIMENTOS

Primeiramente, agradeço aos amigos que fiz durante toda a graduação, que não foram poucos, e que muito me ajudaram a seguir em frente, especialmente nos trabalhos mais difíceis, como este TCC em si.

Agradeço nominalmente aos meus amigos mais próximos durante cada período, Rodrigo, que foi meu companheiro de IC e procrastinação durante os primeiros anos; Machado, que foi um grande amigo de Discord durante a pandemia; Gio, Miguel e Karla, que me ajudaram a segurar os trancos após o retorno presencial; e por fim o Korhal, que foi um grande amigo durante o último ano dessa longa graduação e com quem eu compartilhei muitas fofocas.

Agradeço também a todos com quem fiz time na maratona, mais especificamente ao Machado e ao Granza, com quem fiz time durante a maior parte da graduação e com quem eu consegui até mesmo conquistar a primeira medalha da UDESC na nacional.

Agradeço aos amigos que fiz no laboratório Função e todas as partidas de truco que joguei lá. Agradeço especialmente ao Miguel com quem eu compartilhei as maiores dores do TCC.

Também agradeço aos professores que se dedicaram a lecionar cada disciplina que eu cursei nesta graduação, ainda que uns tivessem uma didática que não me agradasse, sei que se esforçaram para entregar os conteúdos de forma que todos os alunos entendessem.

Agradeço à professora Karina, por todo o tempo dedicado a esta orientação de TCC, e ao professor Torrens, que conseguiu arranjar um tempo para melhorar em muito este trabalho durante a última semana de escrita do texto.

Agradeço também aos meus pais que muito me apoiaram durante toda a graduação.

Por fim, agradeço a Udesc pela oportunidade de estudar e vivenciar tudo que estudei e vivenciei, e ao Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq pela bolsa de modalidade ITC-A de número 409707/2022-8 que apoiou este trabalho.

"*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 o Coq, é possível modelar estruturas complexas e *softwares*, que podem ser representados por meio destas provas. Entre essas estruturas, 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 em Coq o algoritmo de Dijkstra, que busca o menor caminho entre dois pontos em um grafo, e provar a corretude dessa implementação.

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 structures and softwares, that can be represented as 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 the Dijkstra's algorithm in Coq, that finds the shortest path between two points in a graph and verify its 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 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 ordem superior. Além de executar as etapas das provas e evitar erros lógicos, esses assistentes auxiliam na estruturação e organização das demonstrações, permitindo o desenvolvimento de provas mais complexas e robustas.

Um exemplo 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 propriedades 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 trabalhos onde foram feitas implementações e formalizações de algoritmos de grafos em Coq. Portanto, para que seja possível provar teoremas mais complexos dentro da teoria de grafos, é preciso focar na prova de teoremas já existentes para montar uma base sólida para auxiliar nestes teoremas, então o objetivo deste trabalho é explorar a área de implementação e formalização de algoritmos de grafos em Coq, especificamente do algoritmo de Dijkstra, provando também algumas propriedades sobre esse algoritmo, mostrando assim o potencial do desenvolvimento de pesquisas nessa área.

1.1 OBJETIVO GERAL

O objetivo geral deste trabalho é implementar e provar a corretude em Coq do algoritmo de Dijkstra, que busca o menor caminho entre dois pontos em um grafo.

1.2 OBJETIVOS ESPECÍFICOS

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

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

- Aprofundar o estudo e o conhecimento sobre o assistente de provas Coq
- Implementar o algoritmo de Dijkstra no assistente de provas Coq
- Provar a corretude da implementação do algoritmo de Dijkstra

A implementação e a prova realizada estão disponíveis em:

<https://github.com/joao-frohlich/dijkstra-spec>

1.3 TRABALHOS RELACIONADOS

A partir de um levantamento da literatura, foram identificados alguns trabalhos relacionados ao presente trabalho. (MOHAN; WANG; HOBOR, 2020) traz uma implementação do algoritmo de Dijkstra utilizando o compilador CompCert C, que é um compilador implementado e verificado utilizando o assistente de provas Coq, com a prova sendo desenvolvida em Coq. Além desse trabalho, a implementação e prova do algoritmo de Dijkstra foram desenvolvidas para os assistentes de provas Isabelle (NORDHOFF; LAMMICH, 2012), ACL2 (MOORE; ZHANG, 2005) e Jahob (MANGE; KUHN, 2007). Por fim, existem coleções de provas sobre grafos. A biblioteca Certigraph[3] traz provas sobre diversos algoritmos de grafos implementados em CompCert C e desenvolvidas no assistente de provas Coq. Já a biblioteca Archive of Formal Proofs[4] traz provas sobre diversos algoritmos, incluindo de grafos, implementados e desenvolvidas no assistente de provas Isabelle. O diferencial deste trabalho consiste em fazer tanto a implementação quanto a prova do algoritmo de Dijkstra no assistente de provas Coq.

1.4 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 discutidos pseudocódigos envolvendo busca em grafos e o algoritmo de Dijkstra, no Capítulo 4 é apresentada uma descrição sobre o assistente de provas Coq, no Capítulo 5 é descrita a formalização e prova da corretude do algoritmo e, por fim, no Capítulo 6 são apresentadas as conclusões sobre este trabalho.

[3] <https://github.com/CertiGraph/CertiGraph>
[4] <https://www.isa-afp.org/>

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, formalmente $\delta_0 : E \to V$, indicando qual o vértice de origem da aresta, e por fim δ_1 é uma função de mapeamento de uma aresta para um vértice, formalmente $\delta_1 : E \to V$, 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, sendo a origem da seta o vértice de origem da aresta; e a ponta da seta, 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.

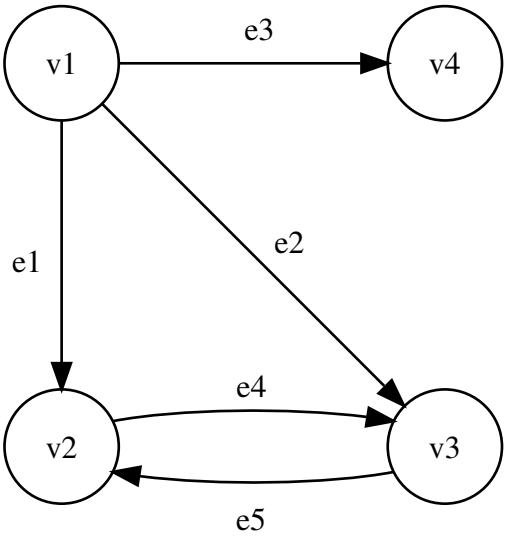

Figura 1 – Grafo direcionado
Fonte: O autor

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 inteiros positivos não nulos, ou seja, a definição de φ se torna $\varphi : E \to \mathbb{Z}^+$. Dessa forma, será mais fácil, de um ponto de vista computacional, trabalhar com pesos nas arestas. 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, as etiquetas das arestas são substituídas por seu peso, 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	3
e2	1
e3	5
e4	5
e5	1

Tabela 2 – Definição de φ no exemplo 2

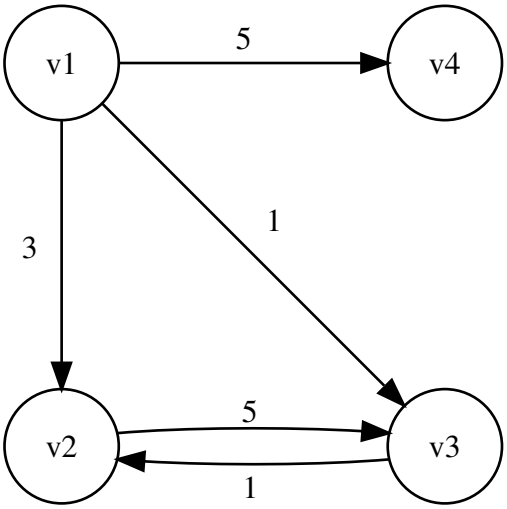

Figura 2 – Grafo direcionado ponderado
Fonte: O autor

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
e2	v1	v3	3
e3	v3	v2	1
e4	v2	v1	2

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 $\varphi(C_1) = \varphi(C_2) = 4$. Note que mostrar que $\nexists C\,(\delta_0(C) = v_1 \wedge \delta_1(C) = v_2 \wedge \varphi(C) \leq 4)$ é trivial.

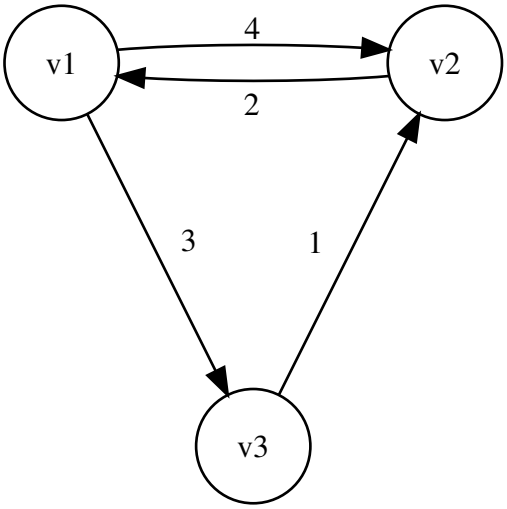

Figura 3 – Grafo para exemplificar o menor caminho
Fonte: O autor

2.3.3 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 $\varphi : E \to \mathbb{Z}_0^+$, ou seja, o contradomínio de φ é o conjunto dos números inteiros 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), $\varphi(e_1) = \varphi(e_2) = 0$ 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, que é o menor caminho possível baseado na restrição de φ.

Suponha agora $\varphi : E \to \mathbb{Z}$ e caminhos podendo ser infinitos. Considerando o exemplo anterior, alterando a função de peso para $\varphi(e_1) = -1$, considere um caminho $C_1 = C + +[e_1] = [e_1, e_2, e_1]$, onde $++$ representa a concatenação de listas. Note que $\varphi(C_1) = -2$. Seja outro caminho $C_2 = C + +C_1$. Assim, $\varphi(C_2) = -3$. 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	3	1	5
v_2	-1	0	5	-1
v_3	-1	1	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, 3); (v_3, 1); (v_4, 5)]
L_2 = [(v_3, 5)]
L_3 = [(v_2, 1)]
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 ALGORITMO DE DIJKSTRA

Uma possível solução para o problema do menor caminho, discutido na Seção 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.

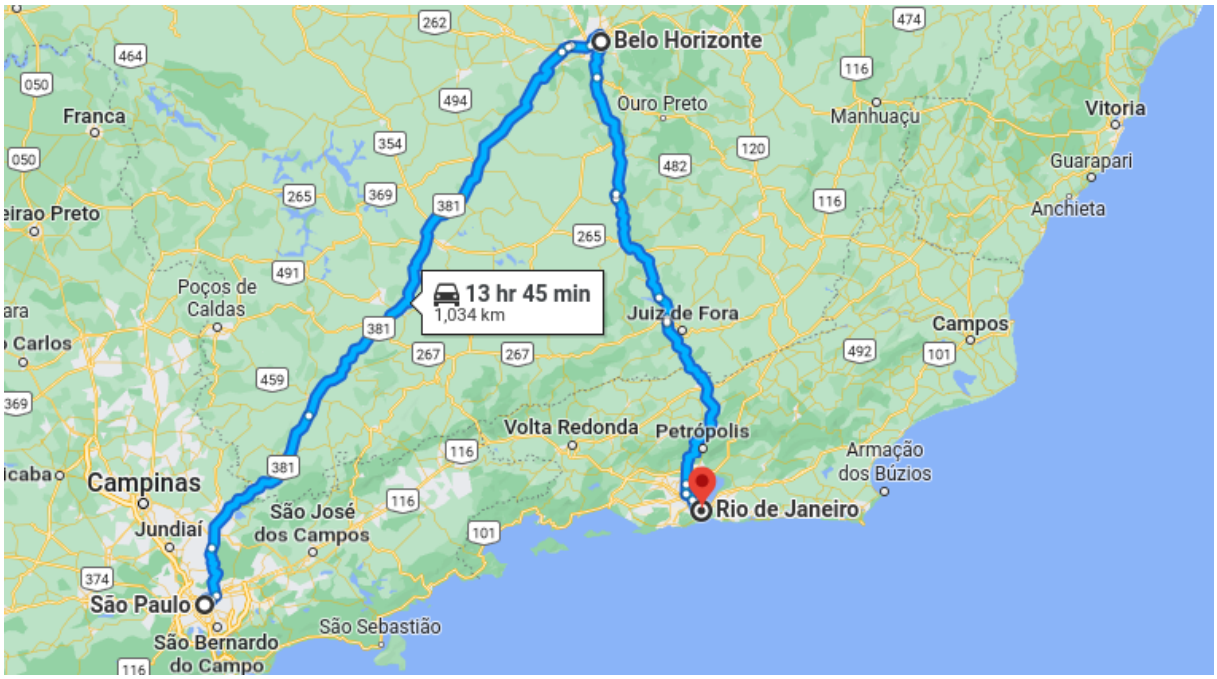

Figura 4 – Trajeto de São Paulo a Rio de Janeiro, passando por Belo Horizonte
Fonte: O autor (Google Maps)

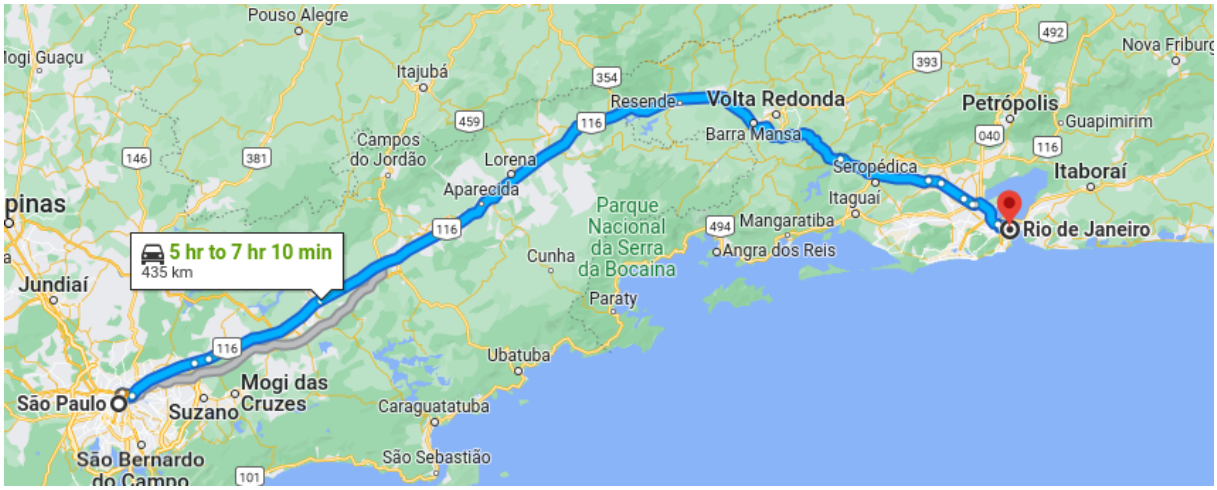

Figura 5 – Trajeto de São Paulo a Rio de Janeiro, passando pela rodovia BR-116
Fonte: O autor (Google Maps)

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 um em específico será tratado neste trabalho, o algoritmo de Dijsktra. Afim de que se possa trabalhar na formalização deste algoritmo, é importante compreender o funcionamento dele. Por isso, este capítulo discutirá o funcionamento deste algoritmo e conterá pseudocódigo da sua implementação. Este capítulo foi baseado majoritariamente em (CORMEN et al., 2022).

3.1 ALGORITMOS DE BUSCA EM GRAFOS

Antes de se aprofundar no algoritmo propriamente dito, é importante entender algumas técnicas de teoria de grafos, que são a base da implementação do algoritmo de Dijkstra.

3.1.1 Busca em largura

A busca em largura é uma técnica de busca em grafos, que consiste em tentar visitar todos os vértices seguindo uma ordem de largura no grafo. Um vértice é dito descoberto uma vez que tenha sido visitado pela busca. O procedimento do algoritmo é realizado armazenando os vértices recém descobertos em uma fila, e explorando os mesmos seguindo a ordem em que foram adicionados. A exploração de um vértice v é feita pela busca de vértices diretamente alcançáveis a partir do vértice v que ainda não tenha sido descoberto. 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.

Para 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.

A implementação dessa técnica pode ser conferida 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.

3.1.2 Relaxamento

A técnica de relaxamento é um procedimento realizado na execução do algoritmo de Dijkstra, 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 $\varphi_V : V \to \mathbb{Z}_0^+$, que mapeia cada vértice $v' \in V$ a um valor inteiro não negativo que representa o limite superior da menor distância com que se alcança v' a partir de um vértice de origem o.

Algoritmo 1 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)
```

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 2.

Algoritmo 2 Relaxamento

```
1:  Função RELAXA-ARESTA(G, e)
2:      Se φ_V(δ_1(e)) > φ_V(δ_0(e)) + φ(e) Então
3:          φ_V(δ_1(e)) = φ_V(δ_0(e)) + φ(e)
4:          π_{δ_1(e)} = δ_0(e)
1:  Função RELAXA-VERTICE(G, u)
2:      Para todo e ∈ E | δ_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.

A implementação deste algoritmo pode ser observada no Algoritmo 3

Algoritmo 3 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 INICIA(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:     Retorna (L, π, φ)
1: Função DIJKSTRA(G, s)
2:     (L, π, φ) ← INICIA(G, s)
3:     Enquanto L não está vazia Faça
4:         cur ← PEGA-MENOR-VALOR(L)
5:         REMOVE(L, cur)
6:         Para todo e ∈ E | δ_0(e) == cur Faça
7:             Se φ_V(δ_1(e)) > φ_V(δ_0(e)) + φ(e) Então
8:                 INSERE(L, δ_1(e))
9:                 RELAXA-ARESTA(G, e)
```

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 notada a estagnação da capacidade destes programas (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 só 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 provar 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). A maioria dos conceitos desta seção foram baseadas em (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 teoria 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).

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 indutivos definidos não recursivamente (estruturas matemáticas como grafos, grupos e anéis), e `Inductive` e `Coinductive`, para tipos indutivos definidos recursivamente (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 exigência da linguagem *Gallina* das funções recursivas terem uma recursão estruturada.

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.

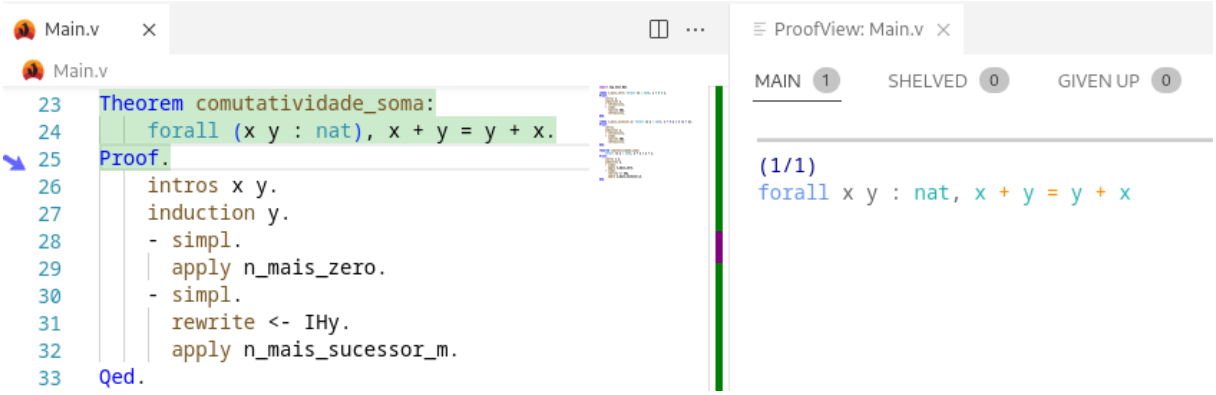

Figura 6 – Exemplo de um teorema provado no VSCoq
Fonte: O autor

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` ← {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).

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

5 FORMALIZAÇÃO E PROVA

Os algoritmos mostrados no Capítulo 3 seguem um paradigma imperativo na modelagem dos grafos e em suas respectivas implementações. O uso dessa lógica imperativa dificulta uma implementação destes algoritmos para o assistente de provas Coq, uma vez que sua linguagem de especificação, *gallina*, implementa um paradigma puramente funcional e, no caso de funções recursivas, exige uma condição de parada explícita.

Assim, as definições descritas no Capítulo 2 e as implementações propostas no Capítulo 3 serão adaptadas de modo que permitam seu desenvolvimento no assistente de provas Coq, sendo que estas adaptações serão discutidas no decorrer deste capítulo. O código completo da implementação e os *scripts* de prova podem ser encontrados em:

<https://github.com/joao-frohlich/dijkstra-spec>.

A fim de simplificar a implementação, durante este capítulo os pesos serão tradados como números naturais maiores que zero e os vértices serão tratados como números naturais.

5.1 MODELAGEM DE GRAFOS UTILIZANDO UMA DEFINIÇÃO INDUTIVA

Uma forma de modelar grafos no Coq seria utilizando tipos indutivos. Uma abordagem semelhante pode ser encontrada em (ERWIG, 2001), que modela um grafo de uma maneira semelhante a listas de adjacências. Esta abordagem foi feita para a linguagem de programação *Haskell* que é uma linguagem de programação de paradigma puramente funcional (MARLOW et al., 2010), assim como *gallina*. Portanto, este trabalho pode ser usado como base para uma implementação semelhante em Coq.

Para lidar com os pesos, foi criado um tipo auxiliar para representar, além dos números naturais, uma simulação do valor infinito, que é utilizado na implementação do algoritmo. Eles são representados como construtores diferentes de um tipo indutivo. Esse tipo é definido como:

```
Inductive NatInf :=
  | Nat : nat → NatInf
  | Infty : NatInf.
```

Onde o construtor `Nat` representa um número natural qualquer, e o construtor `Infty` representa um número infinito. Algumas operações sobre números naturais como soma, comparações e mínimo foram implementadas e podem ser acessadas no repositório deste trabalho.

As seguintes notações para esse tipo e as operações citadas acima foram definidas:

```
Declare Scope natinf.
Notation "| n |" := (Nat n) (at level 60) : natinf.
Notation "a +i b" := (sum_inf a b) (at level 60) : natinf.
Notation "a <i b" := (Lt_inf a b) (at level 60) : natinf.
Notation "a =i b" := (Eq_inf a b) (at level 60) : natinf.
```

```
Notation "a <=i b" := (Le_inf a b) (at level 60) : natinf.
Notation "a <?i b" := (ltb_inf a b) (at level 60) : natinf.
Notation "a =?i b" := (eqb_inf a b) (at level 60) : natinf.
Open Scope natinf.
```

Por fim, esse tipo foi encapsulado por um módulo `NatInf`, para poder ser importado em outras partes do trabalho, ajudando na organização do código.

Para auxiliar na legibilidade do código, os tipos `Node` e `Weight` foram definidos como instâncias de números naturais e `NatInf` e representam, respectivamente, o tipo de um nó e o tipo de um peso de aresta.

```
Definition Node := nat.
Definition Weight := NatInf.
```

Também foi definido um tipo para representar uma lista de adjacência, que representam as arestas do grafo. Esta lista é composta por pares de nó e peso, que representam o vértice destino e o peso da aresta, respectivamente. O tipo `Adj` é uma instância dessa lista de adjacências. Outro tipo definido indutivamente, `Context`, é composto por um único construtor (`mkcontext`) que recebe um `Node` e um `Adj` e retorna um `Context`. Este tipo `Context` representa a lista de adjacência de um dado nó u, e esta lista de adjacência contém todas as arestas que têm u como origem.

```
Definition Adj := list (Node*Weight).
Inductive Context :=
  | mkcontext : Node → Adj → Context.
```

Por fim, um grafo é definido indutivamente por meio de dois construtores: o construtor de um grafo vazio `Empty` e o construtor de um grafo com um contexto extra `CGraph`, que recebe um `Context` e um `Graph`, retornando um `Graph`.

```
Inductive Graph :=
  | Empty : Graph
  | CGraph : Context → Graph → Graph.
```

Resumindo, um grafo é definido aqui como um conjunto de nós, representados pelos números naturais, que são associados a listas de adjacências, que são compostas por pares de nós e pesos, representados pelos números naturais, e que representam todas as arestas que têm esse nó associado como origem.

Para facilitar a modelagem de grafos, uma notação é definida para o construtor `mkcontext` e outra para o tipo `CGraph`, dentro de um escopo delimitado `graph_scope`.

```
Declare Scope graph_scope.
Notation "{ n , s }" := (mkcontext n s) (at level 60) : graph_scope.
Infix "&" := CGraph (at level 60, right associativity) : graph_scope.
Open Scope graph_scope.
```

Todas essas definições e notações foram, assim como com o tipo `NatInf`, encapsuladas em um módulo próprio para evitar sobrecarga de notações.

Para exemplificar essa definição, o grafo da Figura 3 é definido no código a seguir.

```
Definition grafo_exemplo :=
    {1, [(2,|4|);    (3,|3|)]}    &
    {2, [(1,|2|)]}    &
    {3, [(2,|1|)]}    &
    Empty.
```

Dado um nó, é possível extrair o contexto desse nó em um grafo a partir de uma função recursiva, que percorre o grafo em busca do contexto que represente esse nó.

```
Fixpoint get_node_context (g : Graph) (u : Node) : option Context :=
    match g with
    | Empty ⇒ None
    | {n, s} & g' ⇒
        if n =? u then
            Some {n, s}
        else
            get_node_context g' u
    end.
```

O retorno é do tipo `option Context` para permitir um retorno nulo, que acontecerá quando um vértice u não estiver contido no grafo g. Também é possível definir uma função recursiva para extrair todos os nós de um grafo.

```
Fixpoint nodes (g : Graph) : list Node :=
    match g with
    | Empty ⇒ []
    | {n, _} & g' ⇒ n :: (get_nodes g')
    end.
```

Uma função para verificar se o grafo é válido pode ser definida recursivamente, onde os dois construtores de um grafo são analisados. Um grafo será considerado válido se ele for vazio, ou se para cada lista de adjacências, os vértices adjacentes estão contidos no conjunto de nós de grafos, isto é, existe um contexto para cada um destes nós.

Para fazer a verificação de uma lista de adjacência, uma função auxiliar é definida, onde são recebidos como parâmetros uma lista de adjacências e uma lista de vértices, com retorno do tipo `Prop`.

```
Fixpoint Adj_In_Nodes (a : Adj) (nodes : list Node) : Prop :=
    match a with
    | [] ⇒ True
    | (n, _) :: a' ⇒ In n nodes ∧ Adj_In_Nodes a' nodes
    end.
```

A partir disso, a função de verificação de um grafo é definida por meio de uma função recursiva. Uma definição simplificada, que chama a função recursiva com os parâmetros necessários, também pode ser definida.

```
Fixpoint Valid_Graph' (g : Graph) (nodes : list Node) : Prop :=
    match g with
    | Empty ⇒ True
    | {_, s} & g' ⇒ Adj_In_Nodes s nodes ∧ Valid_Graph' g' nodes
    end.

Definition Valid_Graph (g : Graph) : Prop := Valid_Graph' g (get_nodes g).
```

Mesmo não estando no escopo deste trabalho, foram definidas algumas funções para permitir uma declaração de grafos dinâmicos. Uma função de adição de um nó a um grafo pode ser definida criando um contexto, que contém esse novo vértice e uma lista de sucessores vazia, caso o nó não esteja no grafo. Caso o nó já esteja presente no grafo, este não é alterado.

```
Definition add_node (g : Graph) (x : Node) :=
    match get_node_context g x with
    | None ⇒ {x, []} & g
    | Some _ ⇒ g
    end.
```

Uma função de adição de arestas a um grafo pode ser definida com o auxílio de uma função recursiva, que percorre o grafo em busca do contexto do nó de origem da aresta, e adiciona um par contendo o vértice destino e o peso da aresta ao começo da lista de sucessores deste contexto. A função principal é definida como a composição da função de adição de nós, tendo como parâmetros os vértices de origem e destino da aresta, com essa função auxiliar. Como a função de adição de nó só adiciona um vértice se esse não existe no grafo, não há risco de eliminação de arestas existentes durante a adição de uma aresta.

Note que, se uma aresta com mesma origem e destino já existe no grafo, existirão arestas paralelas no grafo, as quais podem ser ignoradas na definição das funções que percorrem as arestas.

```
Fixpoint add_edge' (g : Graph) (o d : Node) (p : Weight) :=
    match g with
    | Empty ⇒ Empty
    | {n, s} & g' ⇒
        if o =? n then {n, ((d,p) :: s)} & g'
        else ({n, s}) & add_edge' g' o d p
    end.

Definition add_edge (g : Graph) (o d : Node) (p : Weight) :=
    add_edge' (add_node (add_node g o) d) o d p.
```

Também foi implementada uma função para adicionar uma lista de arestas, a fim de facilitar a definição de um grafo.

```
Fixpoint add_edges (g : Graph) (l : list (Node*Node*Weight)) :=
    match l with
    | [] ⇒ g
    | (o,d,p) :: l' ⇒ add_edges (add_edge g o d p) l'
    end.
```

Para exemplificar o uso das funções acima, o grafo da Figura 3 é definido novamente, agora por meio destas funções.

```
Definition grafo_exemplo_add_edges :=
    add_edges Empty [(1,2,|4|);  (1,3,|3|);  (3,2,|1|);  (2,1,|2|)].
```

O grafo da Figura 2 também pode ser definido como:

```
    Definition grafo_exemplo_2 :=
        add_edges Empty [(1,2,|3|);  (1,3,|1|);  (1,4,|5|);  (2,3,|5|);  (3,2,|1|)].
```

A prova da corretude dessas implementações de adição de nós e arestas não foram desenvolvidas, mas consistem em provar os seguintes teoremas:

```
Lemma add_node_correct : forall (g : Graph) (n : Node) :=
    Valid_Graph (add_node g n).


Lemma add_edge_correct : forall (g : Graph) (o d : Node) (w : Weight) :=
    Valid_Graph (add_edge g o d w).
```

A partir dessa modelagem, é possível definir uma implementação da função de Dijkstra usando a tática `Program` em sua definição, pois essa tática permite delimitar qual será o argumento decrescente de uma função recursiva, para garantir que a recursão é estruturada.

Os parâmetros da função são um grafo, dois nós `u` e `d` que representam, respectivamente, o vértice atual e o vértice de destino da busca, um peso para representar um valor infinito, uma lista de nós, que contém os nós que ainda não foram visitados pela busca e uma lista de pares de vértices e pesos, que representa a menor distância até cada nó do grafo.

Na implementação, diversas funções auxiliares foram definidas, sendo algumas delas:

- `get_edges_in_list`: Retorna as arestas cujo vértice de destino esteja contido na lista de vértices;
- `set_nat_head`: Move um elemento para a frente de uma lista;
- `get_node_dist`: Retorna a menor distância encontrada até um dado vértice;
- `update_node_dist`: Atualiza a menor distância até um dado vértice;

- `next_node`: Retorna o vértice que possui a menor distância até ele e que não foi visitado ainda;

Outras funções auxiliares foram implementadas para facilitar na definição das funções acima. Todo o código pode ser encontrado no repositório deste trabalho.

Por fim, a função de Dijkstra é implementada como segue:

```
Program Fixpoint dijkstra' (g : Graph) (u d : Node) (to_vis : list Node)
    (dist : list (Node*Weight)) {measure (length to_vis)} : Weight :=
    if u =? d then (get_node_dist dist d)
    else
      (* 1 *)
      let to_vis' :=
        set_nat_head to_vis u
      in
      (* 2 *)
      let suc := match (get_node_context g u) with
        | None => []
        | Some ({_, s}) => get_edges_in_list s to_vis
      end in
      (* 3 *)
      let relax (dist : list (Node*Weight)) (n : (Node*Weight)) : list (Node*Weight) :=
        let v := (fst n) in
        let w := (snd n) in
        let new_dist :=
          (get_node_dist dist u) +i w
        in
        if (new_dist) <?i (get_node_dist dist v) then
          update_node_dist dist v new_dist
        else
          dist
      in
      (* 4 *)
      let new_dist_list : list (Node*Weight) :=
        fold_left (relax) suc dist
      in
      (* 5 *)
      match to_vis' with
        | [] => (get_node_dist dist d)
        | h :: t => match (next_node new_dist_list t) with
          | None => (get_node_dist dist d)
          | Some v => dijkstra' g v d t new_dist_list
```

```
    end
  end.
```

A notação {measure (length to_vis)} indica que o argumento decrescente é o tamanho da lista to_vis.

Alguns ponteiros serão fornecidos para tentar ajudar no entendimento da implementação.

No let que sucede o comentário (* 1 *), o vértice atual é movido para a frente da lista de vértices a serem visitados para que ele possa ser consumido num *pattern matching* sobre essa lista.

No let que sucede o comentário (* 2 *), é construída a lista de sucessores do vértice atual a serem relaxados, que consiste em todos os vértices presentes na lista de sucessores do nó que ainda não foram visitados.

No let que sucede o comentário (* 3 *), é implementada uma função de relaxamento de arestas, utilizando os conceitos discutidos na Seção 3.1.2.

No let que sucede o comentário (* 4 *), é definida uma chamada da função de relaxamento (3), que é aplicada a todos os sucessores (2) a partir de um *folding*.

Por fim, no *pattern matching* que sucede o comentário (* 5 *), algumas verificações *dummy* são realizadas, por uma necessidade do assistente de provas, e a função é recursivamente chamada excluindo o nó atual (u) da lista de nós, passando o nó com menor distância que ainda não foi visitado como novo nó u, e com a lista de distâncias relaxadas em (4).

Para simplificar a chamada da função de Dijkstra, uma definição pode ser implementada da seguinte forma:

```
Definition dijkstra (g : Graph) (o d : Node) : Weight :=
    let dist :=
        (combine (get_nodes g) (repeat Infty (length (get_nodes g))))
    in
    dijkstra' g o d (get_nodes g) (update_node_dist dist o (|0|)).
```

Dessa forma, fazendo a chamada da função dijkstra grafo_exemplo 1 2, o retorno será o peso 3. Contudo, não é possível afirmar que o resultado está certo. Dessa forma, é necessário especificar alguma propriedade que permita mostrar que esse algoritmo está correto. De maneira geral, dizemos que um algoritmo de menor caminho é correto se, para quaisquer dois nós em um grafo, ele é capaz de sempre retornar o menor caminho.

Portanto, uma forma de testar a corretude dessa implementação é gerar todos os caminhos entre dois vértices e comparar o peso de cada um desses caminhos com o resultado do algoritmo. A formalização dessa propriedade, pode ser definida como:

```
Fixpoint Min_Weight (g: Graph) (w : Weight) (paths : list Path) :=
    match paths with
    | [] ⇒ True
    | path :: paths' ⇒ w <=i (get_path_weight g path) ∧ Min_Weight g w paths'
    end.
```

Onde `get_path_weight` é uma função que retorna o peso de um caminho `path` de um grafo `g` específico. A especificação usando a função de Dijkstra é definida pela seguinte função:

```
Definition Dijkstra_Min_Weight (g : Graph) (o d : Node) :=
    Min_Weight g (dijkstra g o d) (get_paths g o d).
```

A função `get_paths` é uma função que retorna todos os caminhos sem laços entre dois vértices em um grafo. Assim como a função `dijkstra`, esta definição é uma simplificação de uma outra função, que é implementada usando `Program Fixpoint`.

```
Program Fixpoint get_paths' (g : Graph) (u d :  Node)
    (to_vis :  list Node) {measure (length to_vis)} : list Path :=
    if u =? d then [[u]]
    else
        let to_vis' :=
            set_nat_head to_vis u
        in
        let suc := match (get_node_context g u) with
        |  None ⇒ []
        |  Some (mkcontext _ s) ⇒ get_elem_in_list s to_vis
        end in
        let aux (x :  Node) :=
            match to_vis' with
            |  []  ⇒ []
            |  h ::  t ⇒ map (cons h) (get_paths' g x d t)
            end
        in
        fold_left append_nat_lists (map aux suc) [].
```

Note que a principal diferença consiste na forma como a função `fold_left` é utilizada nas duas funções. Enquanto na função `dijkstra'` o *folding* é realizado com a operação de relaxamento sobre todos os sucessores, na função `get_paths'` o *folding* aplica uma concatenação de listas de caminhos, que é obtida por meio do mapeamento da chamada recursiva da função `get_paths'` em todos os sucessores do vértice atual.

A função `get_paths` é definida omitindo o parâmetro da lista de vértices a serem visitados:

```
Definition get_paths (g : Graph) (o d :  Node) :=
    get_paths' g o d (get_nodes g).
```

Como a função `get_paths` é a base para a especificação da função `dijkstra`, é importante que essa também seja verificada. Para não estender este trabalho, será apresentada a ideia geral por trás da especificação dessa função, e a implementação dessa ideia pode ser verificada no repositório.

A função `get_paths` pode ser considerada correta se ela gerar todos os caminhos sem laços entre dois vértices. Já que um caminho é definido como uma lista de vértices, tem-se que

um caminho sem repetição é uma lista de vértices sem vértices duplicados. Portanto, todos os caminhos sem repetição de um grafo podem ser representados como todas as combinações de 1 a N vértices, sendo N a quantidade de vértices do grafo, que representem um caminho válido.

Um caminho é dito válido se, para qualquer par de vértices consecutivos no caminho a_i e a_{i+1}, a_{i+1} seja um vértice sucessor do vértice a_i. Além disso, como o objetivo é gerar todos os caminhos entre dois vértices, serão selecionados apenas as sequências de vértices que tenham um determinado vértice inicial e um determinado vértice final. No código do repositório, essa função possui o nome `generate_all_valid_graph_paths_from_o_to_d`.

Por fim, a verificação da função `get_paths` consiste em testar se todo caminho retornado pela função `get_paths` está contido nos caminhos gerados na função `generate_all_valid_graph_paths_from_o_to_d` e vice-versa.

Com as funções `dijkstra` e `get_paths` definidas, bem como a especificação `Dijkstra_Min_Weight`, é possível provar que a implementação de `dijkstra` é correta para um dado grafo e par de vértices. Para exemplificar, serão utilizados os grafos `grafo_exemplo` e `grafo_exemplo_2`, e alguns pares de vértices interessantes para realizar os testes. O teste com todos os pares possíveis de vértices em ambos os grafos estão disponíveis no repositório deste trabalho.

Os seguintes exemplos foram provados:

```
Example grafo_exemplo_dijkstra_1_2 : (Dijkstra_Min_Weight grafo_exemplo 1 2).

Example grafo_exemplo_dijkstra_2_3 : (Dijkstra_Min_Weight grafo_exemplo 2 3).

Example grafo_exemplo_2_dijkstra_1_2 : (Dijkstra_Min_Weight grafo_exemplo_2 1 2).

Example grafo_exemplo_2_dijkstra_1_3 : (Dijkstra_Min_Weight grafo_exemplo_2 1 3).
```

Durante a implementação das provas, notou-se que todas elas podem ser resolvidas utilizando apenas 2 táticas: `compute` e `intuition`.

5.2 IMPLEMENTAÇÃO UTILIZANDO A REPRESENTAÇÃO POR GRAFO DE FLUXO DE CONTROLE

Apesar de implementada, a definição da função de Dijkstra na seção anterior ficou muito complexa e não permitiu o desenvolvimento de uma prova mais geral do algoritmo. Dessa forma, para se tentar especificar formalmente o algoritmo e provar sua corretude para qualquer grafo, se faz necessário pensar em como simplificar a implementação.

Uma forma de simplificar este código pode ser feita utilizando uma técnica inspirada no desenvolvimento de compiladores para linguagens imperativas. Nesses compiladores, o *Static Single-Assignment* (SSA) é uma linguagem de representação intermediária onde um conjunto de instruções são representados por meio de um grafo de fluxo de controle (CFG), e cada variável utilizada no código só pode receber uma única atribuição de valor. Nele, as instruções de um

código são divididas em pequenos blocos, que representam os vértices deste CFG, e o fluxo do programa, isto é, qual bloco deve ser executado após o processamento de um conjunto de instruções, representa as arestas deste grafo. Essa transformação permite a extração de algoritmos imperativos para algoritmos funcionais (APPEL, 1998).

Portanto, seja o seguinte trecho do algoritmo de Dijkstra, apresentado no Algoritmo 3, com algumas modificações no nome das funções:

Algoritmo 4 Algoritmo de Dijkstra

1: **Enquanto** Q não está vazia **Faça**
2: $u \leftarrow$ vértice em Q com menor $d(u)$
3: REMOVE(Q, u)
4: **Para todo** vizinho v de u que esteja presente em Q **Faça**
5: $alt \leftarrow d(u) + edges(u,v)$
6: **Se** $alt < d(v)$ **Então**
7: $d(v) \leftarrow alt$
8: $p(v) \leftarrow u$
Retorne d, p

Usando a mesma ideia do SSA, de maneira similar à utilizada em (RIGON; TORRENS; VASCONCELLOS, 2020), este código pode ser representado por meio de 6 blocos básicos, como apresentado na Figura 7. Por mais que o código não esteja na forma SSA, ele apresenta uma forma simplificada de visualizar a função de Dijkstra, que pode ser implementada em Coq. Inicialmente, é definida uma função recursiva para representar o fluxo de controle dos blocos b2, b3 e b4, bem como o retorno de b2 para o bloco b0:

```
(* b2 *)
Fixpoint b2 N (d: distance) u :=
    match N with
        (* retorno para b0 *)
        | [] ⇒
            d
        (* b3 *)
        | v :: N' ⇒
            let alt := get d u 0 + get_weight u v in
            if alt <? get d v (1 + alt) then
                (* b4 *)
                let d' := set d v alt in
                (* retorno para b2 *)
                b2 N' d' u
            else
                (* retorno para b2 *)
                b2 N' d u
    end.
```

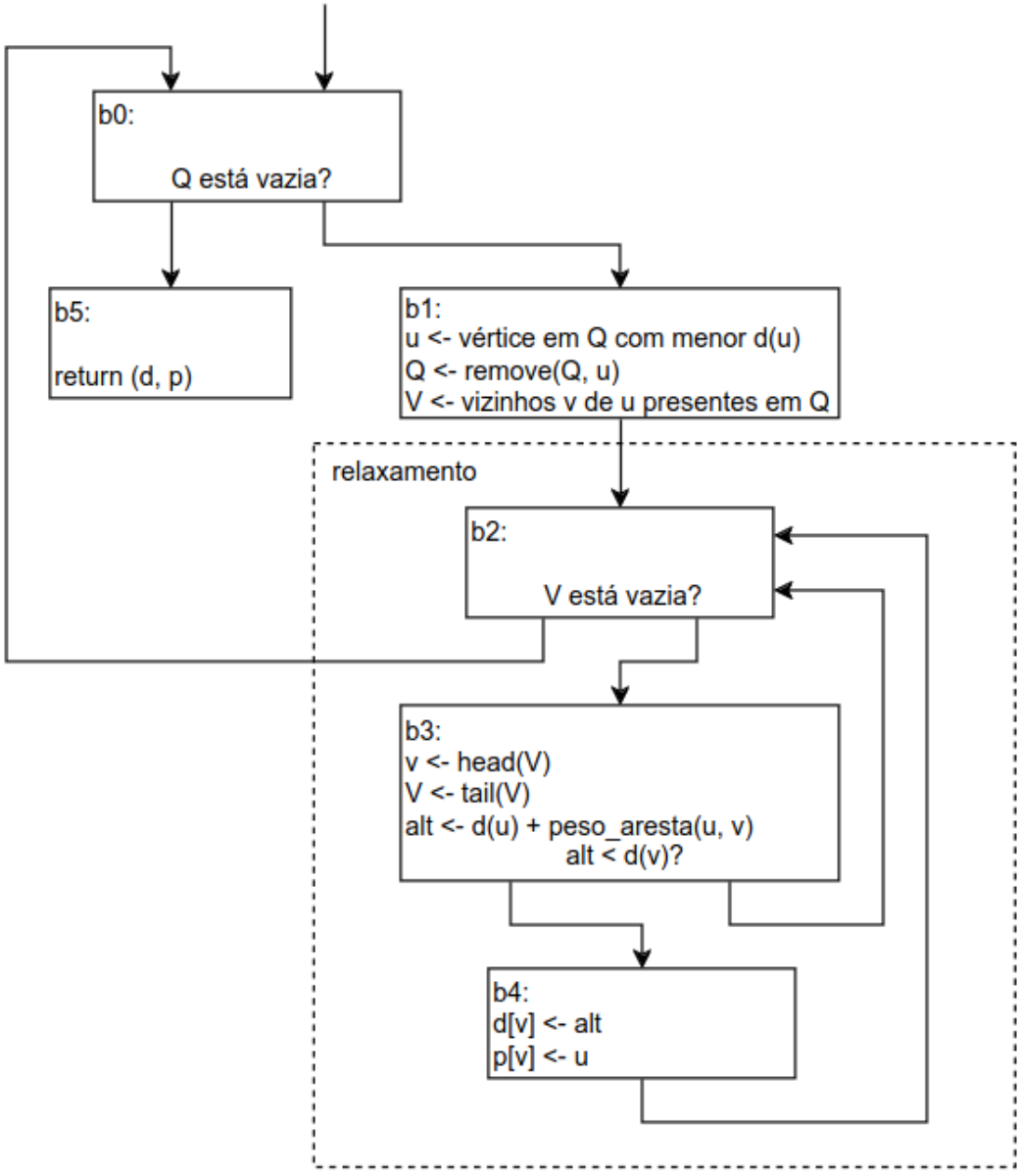

Figura 7 – Grafo do fluxo de controle do Algoritmo 4
Fonte: O autor

A partir da definição de b2, é possível definir os blocos b0, b1 e b5, utilizando `Program Fixpoint`:

```
(* b0 *)
Program Fixpoint b0 Q d { measure (length Q) } :=
    match Q with
    | [] ⇒
        (* b5 *)
        d
    | _ ⇒
        (* b1 *)
        let u := take_shortest Q d _ in
        let Q' := remove Q u in
        let N := neighbors Q' u in
        (* chamada de b2 *)
        let d' := b2 N d u in
        (* retorno para b0 *)
        b0 Q' d'.
```

A obrigação gerada pelo `Program Fixpoint` pode ser resolvida a partir da prova que a função `remove` reduz o tamanho de Q, se u estiver presente em Q, seguida pela prova que a função `take_shortest` garante que u esteja presente em Q.

Por fim, basta definir as funções `get_weight`, `neighbors`, `remove`, `take_shortest`, `get` e `set` para que essa implementação funcione.

Para a modelagem dos grafos utilizados no algoritmo, o código foi encapsulado em uma `Section`, para permitir a declaração de `Variables` e `Hypothesis` que parametrizam implicitamente o código dentro desta `Section`, o que deixa o código mais organizado, pois tira a necessidade de passar uma mesma variável como parâmetro em todas as funções, como por exemplo o vértice de origem da busca. É declarado que um `vertex` pode assumir um tipo qualquer, e representa o tipo dos vértices, `V` é uma lista de `vertex` e representa o conjunto de vértices do grafo e `E` é uma função que recebe dois `vertex` e retorna um `option nat`, que representa as arestas de um grafo, com o valor `None` representando a inexistência de uma aresta entre dois nós e `Some x` indicando que existe uma aresta com peso `x` entre dois vértices. O uso de `option nat` se dá para contornar o fato de que `E` é uma função parcial, visto que todas as funções no Coq são totais. Assim, o construtor `None` pode ser atribuído a elementos não definidos na função parcial. Além disso, é definido o vértice de origem do Dijkstra em uma variável `s` e uma hipótese afirmando que a igualdade do tipo `vertex` é decidível.

As distâncias serão definidas por meio de uma lista associativa, que associa um vértice a um peso, e são o retorno da função `b0`. A função `dijkstra` pode ser definida como uma simplificação da chamada de `b0` e, por conta da definição prévia de `V`, `E` e `s`, ela não recebe nenhum parâmetro.

```
Definition dijkstra := b0 V [(s, 0)].
```

5.2.1 Prova informal do algoritmo de Dijkstra

O objetivo de um assistente de provas, como o Coq, é verificar formalmente em uma teoria uma prova matemática, as quais muitas vezes já possuem provas informais feitas com papel e caneta. Portanto, será apresentada aqui a ideia geral por trás da prova do algoritmo de Dijkstra. A prova apresentada aqui é baseada em (Wikipedia contributors, 2023).

A prova do algoritmo consiste em provar que o resultado do procedimento gera o menor caminho do vértice de origem para qualquer outro vértice, e é construída por indução no número de vértices visitados, ou, alternativamente, no número de vértices que ainda devem ser visitados. Em uma prova de algoritmo por indução, uma hipótese invariante é uma proposição que se mantém verdadeira durante a execução e, nesse caso, é verdadeira após o algoritmo visitar cada um dos vértices.

Hipótese invariante: para cada vértice visitado v, $\varphi_V(v)$ é a menor distância da origem s até v, e para cada vértice não visitado u, $\varphi_V(u)$ será a menor distância de s até u, considerando apenas caminhos que passem por vértices já visitados, ou infinito caso não exista tal caminho.

O caso base da indução ocorre quando tem apenas um vértice visitado, que é o vértice de origem s. Nesse caso, $\varphi_V(s) = 0$, e esta é a menor distância possível, portanto a hipótese é verdadeira para o caso base.

Passo indutivo: seja a hipótese verdadeira para $k-1$ vértices visitados, e seja u o próximo nó a ser visitado pelo algoritmo. O objetivo é provar que $\varphi_V(u)$ é a menor distância entre s e u.

A prova segue por contradição, ou seja, suponha que exista um caminho π mais curto entre s e u. Existem dois casos possíveis: existe um outro vértice não visitado em π ou não existe outro vértice não visitado em π.

Primeiro caso: seja w o primeiro nó não visitado em π. Pela hipótese invariante, o menor caminho entre s e u e entre s e w são, respectivamente, $\varphi_V(u)$ e $\varphi_V(w)$. Isto significa que o custo para ir de s até u passando por w terá um custo de, pelo menos, $\varphi_V(w)+x$, tal que $x \in \mathbb{Z}^+$ por definição e x representa o menor caminho entre w e u. Além disso, tem-se que $\varphi_V(u) \leq \varphi_V(w)$, pois o algoritmo escolheu u ao invés de w. Portanto, chega-se à contradição $\varphi_V(u)(\leq \varphi_V(w)+x) < \varphi_V(u) \equiv \varphi_V(u) < \varphi_V(u)$.

Segundo caso: seja w o penúltimo vértice do menor caminho entre s e u. Isto significa que $\varphi_V(w) + \varphi(w,u) < \varphi_V(u)$. Isto é falso pois, no processo de relaxamento de w, $\varphi_V(u)$ já teria recebido o valor $\varphi_V(w) + \varphi(w,u)$, o que geraria a contradição $\varphi_V(w) + \varphi(w,u) < \varphi_V(w) + \varphi(w,u)$.

Portanto, é provado que uma iteração do algoritmo não falsifica a hipótese invariante e, portanto, o algoritmo é correto.

5.2.2 Prova do algoritmo de Dijkstra em Coq

Para aplicar as ideias da prova matemática no Coq, algumas propriedades precisam ser definidas. A primeira delas é a propriedade de caminhos ponderados. A definição desta propriedade checa se dados dois vértices v e $u \in V$, existe um caminho de peso n entre eles. A existência de um caminho ponderado pode ser verificada em três casos distintos: no primeiro caso, se $v = u$, então existe um caminho de peso 0. No segundo caso, se existe uma aresta ligando diretamente v a u, com peso n, então existe um caminho de v a u com peso n. Por fim, sejam v, u e w vértices distintos, n e m pesos não necessariamente distintos, se existe um caminho de v a u com peso n, e um caminho de u a w com peso m, então existe um caminho de v a w com peso $n+m$. Essa propriedade pode ser definida em Coq de maneira indutiva como segue:

```
Inductive path: vertex → vertex → nat → Prop :=
  | path_here:
    forall v,
    In v V →
    path v v 0
  | path_step:
    forall v u n,
    In v V →
    In u V →
    E v u = Some n →
    path v u n
  | path_trans:
    forall v u w n m,
    path v u n →
    path u w m →
    path v w (n + m).
```

A segunda propriedade é a de menor caminho, onde a definição de caminho pode ser utilizada. Sejam v e u vértices quaisquer, e n um peso qualquer, dizemos que eles formam um menor caminho se existe um caminho de v para u com peso n, e se para qualquer peso m, se existir um caminho de v para u com peso m, então $n \leq m$. Em Coq, isso é representado pela seguinte função:

```
Definition shortest_path v u n: Prop :=
  path v u n ∧ (forall m, path v u m → n <= m).
```

Outra propriedade é a sanidade da lista de distâncias, isto é, um par de vértice e peso só está presente na lista de distâncias se existe um caminho da origem para o vértice com esse peso. Nesta propriedade, seja d uma lista associativa de distâncias, para todo vértice v e peso x, se o par $(v,x) \in d$, então d é sã se existe um caminho do vértice de origem s para o vértice x com peso d. Em Coq, isto é definido como:

```
Definition sane d: Prop :=
```

```
    forall v x,
    In (v, x) d → path s v x.
```

Além disso, outra propriedade definida é sobre a alcançabilidade entre dois vértices, ou seja, que um vértice u é alcançável a partir de um vértice v, isto é, existe um caminho de u para v. De forma trivial, é possível demonstrar, em papel e caneta, que se existe um caminho, então também existe um menor caminho, portanto essa propriedade será tratada na segunda forma, pois a implementação de provas da lógica clássica em Coq demandam o uso de axiomas adicionais. Sejam v e u vértices, eles são alcançáveis se existe um peso w tal que exista um caminho entre v e u com peso w. Em Coq, a propriedade de alcançabilidade é definida como:

```
Definition reachable v u: Prop :=
    exists w,
    shortest_path v u w.
```

Por fim, a última propriedade a ser definida para poder desenvolver a prova é a hipótese de invariância. Nesta propriedade, seja Q uma lista de vértices a serem visitados e d uma lista associativa de distâncias, para todo vértice $v \in V$, ou esse vértice ainda não foi visitado, ou se o vértice v é alcançável a partir de s, então ele já foi visitado e a distância definida em d para v é a menor distância de s para v. Em Coq, essa propriedade pode ser definida como:

```
Definition invariant (Q: list vertex) (d: distance): Prop :=
    forall v,
    In v V →
        In v Q ∨
        (reachable s v → exists2 w, In (v, w) d & shortest_path s v w).
```

A prova da corretude do algoritmo pode ser declarada como "para todo vértice v e peso w, se o menor caminho de s para v tem peso w, então o par (v,w) está presente na lista associativa retornada pela função `dijkstra`". Isso é representado em Coq como:

```
Theorem correctness:
    forall v w,
    shortest_path s v w →
    In (v, w) dijkstra.
```

Como discutido na Subseção 5.2.1, a ideia da prova desse teorema segue por indução no número de vértices visitados. Para auxiliar na construção dessa ideia, é possível provar o lema auxiliar que diz que "para toda lista de vértices não visitados Q, lista de distâncias d e peso w, se a invariante é preservada para Q e d, então para todo v, se o menor caminho de s para v tem peso w, então o par (v,w) está presente na lista associativa retornada pela aplicação da função b0 com os parâmetros Q e d, ou seja, após a execução do laço principal do algoritmo". Lembre que a função `dijkstra` é apenas uma chamada da função `b0` com o conjunto de vértices V como Q e a lista $[(s,0)]$ como d. Em Coq, esse lema auxiliar é representado como:

```
Lemma step_correctness:
```

```
    forall Q d w,
    invariant Q d →
    forall v,
    shortest_path s v w →
    In (v, w) (b0 Q d).
```

Outro lema auxiliar importante é a preservação da hipótese de invariância pela aplicação da função `b0`. Por definição, esse lema é equivalente a dizer que a aplicação da função `b2`, que faz o processo de relaxamento de arestas, preserva a hipótese de invariância. Em Coq, isso pode ser definido como:

```
Lemma invariance_preservation:
    forall Q d,
    invariant Q d →
    forall v w,
    shortest_path s v w →
    forall H u,
    u = take_shortest Q d H →
    forall Q',
    Q' = remove Q u →
    forall N,
    N = neighbors Q' u →
    forall d',
    d' = b2 N d u →
    invariant Q' d'.
```

Por fim, é preciso provar também que a função de relaxamento `b2` mantém a propriedade de sanidade. Este lema é declarado de maneira similar ao lema da preservação da invariância:

```
Lemma sane_preservation:
    forall Q d,
    sane d →
    forall H u,
    u = take_shortest Q d H →
    forall Q',
    Q' = remove Q u →
    forall N,
    N = neighbors Q' u →
    forall d',
    d' = b2 N d u →
    sane d'.
```

Todos os *scripts* de prova podem ser encontrados no repositório deste trabalho. Os teoremas `step_correctness` e `correctness` foram provados, porém os lemas `sane_preservation` e `invariance_preservation` estão parcialmente provados, devido às manipulações que precisam

ser realizadas para conseguir terminar as provas. Contudo, como elas seguem as ideias da prova informal, é possível dizer, com certa confiabilidade, que estes lemas poderão ter suas provas terminadas e aumentam a confiança sobre o código produzido. Ainda assim, para permitir a utilização dos lemas da preservação da sanidade e da invariância nos lemas principais, foi utilizado o comando `Admitted` para encerrar estas provas incompletas.

5.2.3 Exemplo e extração do código

Para mostrar o funcionamento dessa implementação em um exemplo, será novamente utilizado o grafo da Figura 3. O grafo é definido pelo conjunto de vértices e pelo conjunto de arestas:

```
Definition V :=
    [1;  2;  3].

Definition E: nat → nat → option nat :=
    fun v u ⇒
        match (v,  u) with
        | (1,  2) ⇒ Some 4
        | (1,  3) ⇒ Some 3
        | (2,  1) ⇒ Some 2
        | (3,  2) ⇒ Some 1
        | _ ⇒ None
        end.
```

A chamada da função de Dijkstra pode ser definida para qualquer vértice presente no conjunto V, para exemplificar, será utilizado o vértice 1:

```
Definition example := dijkstra V E 1 Nat.eq_dec.
```

Por fim, é possível extrair a função `exemplo`, juntamente com a função `dijkstra` e todas as funções e tipos dos quais ela depende, para a linguagem de programação *Haskell* com os seguintes comandos:

```
Require Import Extraction.
Extraction Language Haskell.

Recursive Extraction example.
```

Dessa forma, ao compilar o código utilizando a aplicação *coqc*, o resultado será o algoritmo convertido para *Haskell*. Contudo, para poder testar o código gerado, é necessário remover a modularização (**module** Main **where**) e declarar uma função que transforma os tipos Nat, Prod e **List** em strings (**deriving** **Show**). Para uma visualização efetiva, seria interessante modificar os tipos do código gerado por tipos presentes nas bibliotecas do *Haskell*.

6 CONSIDERAÇÕES FINAIS

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 essa escassez de implementação e formalização de algoritmos e teoremas já provados no assistente de provas Coq, percebe-se a necessidade de criar uma base sólida para a prova de novos teoremas para a teoria de grafos com o uso deste assistente de prova. Sendo assim, uma das primeiras contribuições deste trabalho pode ser percebida na implementação e especificação formal sobre o algoritmo de Dijkstra, onde o único trabalho encontrado sobre isso foca apenas na prova de uma implementação na linguagem C, usando o compilador CompCert C, porém não foram encontrados trabalhos que fizessem tanto a implementação quanto a prova do algoritmo no Coq.

Neste trabalho foram implementadas duas abordagens distintas para modelar grafos no Coq e implementar os algoritmos, uma utilizando uma definição de grafos indutivos e outra utilizando funções de mapeamento. Na definição de grafos indutivos, mesmo conseguindo implementar o algoritmo de Dijkstra, só foi possível desenvolver especificações para provar o funcionamento do algoritmo usando instâncias de grafos e pares de nós definidos. Já na definição por funções de mapeamento, o algoritmo foi desenvolvido utilizando uma técnica para modelar o funcionamento do código por meio de um grafo de fluxo de controle antes da implementação, o que permitiu a simplificação desta. Nessa definição, foi possível elaborar uma prova da corretude do algoritmo seguindo uma prova matemática. Apesar de não ter sido terminada, a prova parcial indica uma grande probabilidade de que o código implementado está correto. Além disso, foi possível extrair o código para a linguagem *Haskell* e testar a execução do código nesta linguagem.

Durante o desenvolvimento do trabalho, os principais problemas encontrados foram relacionados à complexidade de implementação de códigos pensados para um paradigma imperativo na linguagem *Gallina*. Além disso, foram encontradas muitas dificuldades na manipulação de grafos em Coq, especialmente na terminação das provas. Isto se dá porque, de maneira geral, grafos não são definidos utilizando um critério estrutural, o que dificulta a prova por indução mecanizada sobre grafos e, como a linguagem *Gallina* exige a prova de terminação de todo

código escrito nela, é necessário um trabalho adicional para demonstrar a terminação, algo que é feito implicitamente e de maneira informal nas provas tradicionais feitas no papel.

Para trabalhos futuros, inicialmente sugere-se a conclusão da prova de corretude do algoritmo. Além disso, sugere-se tentar uma outra forma de modelar grafos, como por exemplo uma definição algébrica descrita em (MOKHOV, 2017). Também sugere-se realizar a implementação e prova da corretude de outros algoritmos da teoria de grafos, bem como de outros problemas, como árvore geradora mínima, fluxo máximo, ordenação topológica, entre outros.

REFERÊNCIAS

APPEL, Andrew W. Ssa is functional programming. **Acm Sigplan Notices**, ACM New York, NY, USA, v. 33, n. 4, p. 17–20, 1998. Citado na página 38.

APPEL, K; HAKEN, W. Every planar map is four colorable. **American Mathematical Society**, v. 82, n. 5, 1976. Citado na página 10.

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 10.

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 20.

DOCZKAL, Christian. A variant of wagner's theorem based on combinatorial hypermaps. 2021. Citado na página 11.

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 na página 11.

DOCZKAL, Christian; POUS, Damien. Graph theory in coq: Minors, treewidth, and isomorphisms. **Journal of Automated Reasoning**, Springer, v. 64, p. 795–825, 2020. Citado na página 11.

ERWIG, Martin. Inductive graphs and functional graph algorithms. **Journal of Functional Programming**, Cambridge University Press, v. 11, n. 5, p. 467–492, 2001. Citado na página 29.

GEUVERS, Herman. Proof assistants: History, ideas and future. **Sadhana**, Springer, v. 34, p. 3–25, 2009. Citado 2 vezes nas páginas 10 e 23.

GONTHIER, Georges. **A computer-checked proof of the four colour theorem**. 2005. Citado 2 vezes nas páginas 11 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 10.

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 2 vezes nas páginas 11 e 24.

MANGE, Robin; KUHN, Jonathan. Verifying dijkstra ' s algorithm in jahob. In: . [s.n.], 2007. Disponível em: <https://api.semanticscholar.org/CorpusID:202722804>. Citado na página 12.

MARLOW, Simon et al. Haskell 2010 language report. 2010. Citado na página 29.

MOHAN, Anshuman; WANG, Shengyi; HOBOR, Aquinas. A machine-checked c implementation of dijkstra's shortest path algorithm. In: . [s.n.], 2020. Disponível em: <https://api.semanticscholar.org/CorpusID:211549446>. Citado na página 12.

MOKHOV, Andrey. Algebraic graphs with class (functional pearl). **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 52, n. 10, p. 2–13, 2017. Citado na página 47.

MOORE, J. Strother; ZHANG, Qiang. Proof pearl: Dijkstra's shortest path algorithm verified with acl2. In: **International Conference on Theorem Proving in Higher Order Logics**. [s.n.], 2005. Disponível em: <https://api.semanticscholar.org/CorpusID:19329531>. Citado na página 12.

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 2 vezes nas páginas 11 e 12.

RIGON, Leonardo Filipe; TORRENS, Paulo; VASCONCELLOS, Cristiano. Inferring types and effects via static single assignment. In: **Proceedings of the 35th Annual ACM Symposium on Applied Computing**. New York, NY, USA: Association for Computing Machinery, 2020. (SAC '20), p. 1314–1321. ISBN 9781450368667. Disponível em: <https://doi.org/10.1145/3341105.3373888>. Citado na página 38.

SILVA, Rafael Castro Goncalves. **Uma certificação em COQ do algoritmo W monádico. 2019. 78 p**. Dissertação (Mestrado) — Universidade do Estado de Santa Catarina, Programa de Pós Graduação em Computação Aplicada, 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 11.

Wikipedia contributors. **Dijkstra's algorithm — Wikipedia, The Free Encyclopedia**. 2023. [Online; accessed 15-November-2023]. Disponível em: <https://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=1183855305>. Citado na página 41.