Formalize Floyd-Warshall algorithm in Coq

Junyi Guan
dept. computer science of University of Nottingham
psxjg12@exmail.nottingham.ac.uk

Abstract

This paper is about using a proof assistant called Coq to formalize Floyd-Warshall algorithm and proof the correctness of this algorithm.These two main tasks are divided into some sections.First,formalizing related data structures and introduce axioms of these data structures and prove some theorems or lemmas.Second, formalizing Floyd-Warshal algorithm using the existing data structures.Third formalizing some predicate in order to build propositions.Fourth, formalize some theorems,lemmas and axioms which represent correctness of this algorithm.Finally proof all of these.The result of this paper is that by introducing some axioms,Floyd-Warshall algorithm is successfully verified.

1 Introduction

1.1 Theorem Prover

Why we need to use theorem prover to proof a theorem? Why we don't simply write the proof in natural language ?There are reasons for this.First of all,writing a proof using natural language is too difficult to check by computer although natural language processing is developing very quickly.But computer is able to run a program.By Curry-Howard correspondence(which will be discussed later),proving a mathematical theorem can be convert to writing a program.So to check if a proof of a proposition is correct can be fully done by computer.That is quiet useful and convenient,since people do not need to spend a long time to understand and check the proof.Also,when human writing a proof on paper using some informal language,it is very likely to make mistakes such as assuming some false statements are correct or forgetting to consider some special cases.By contrast,Theorem prover not allow the user to use any assumptions until they have been proved them and it can consider all different cases the user need to consider.So if a proposition is proved by a theorem prover,the proposition is really correct.

1.2 Shortest Path Algorithm

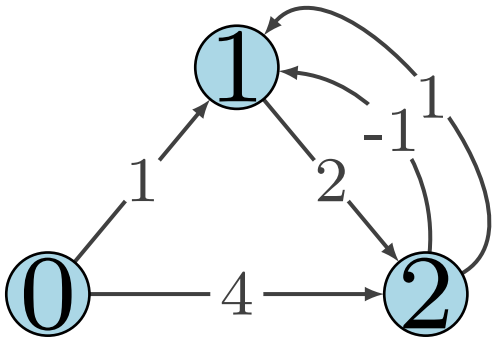
Finding shortest paths is a very classical problem in graph theory and also has a lot of applications in real world.For example,when we want to travel from one place to other places,in order to saving the travelling time,it is useful to find the shortest paths and distances from the starting point to these destinations.There are two main types of shortest paths problems.The first type of that is called Single-Source Shortest Path problem (SSSP).It need to find all shortest path and distance from a given vertex (source) to all other vertices in a directed weighted graph.The second type is called All-Pairs Shortest Paths (APSP) problem.It aim to find all the shortest path and distance between any pairs of vertices.So If we can solve APSP, we will able to find all the solutions of SSSP for all different sources. For SSSP,there are two famous algorithms to solve it.Dijkstra's and Bellman–Ford algorithm.[1]Dijkstra algorithm is much faster than the Bellman ford algorithm.But Bellman ford algorithm also can handle the situation when the graph contains some negative edges.However,this paper is not going to discuss these two algorithms.The core algorithm for this paper is Floyd-Warshall algorithm which is able to solve APSP and the paper is mainly focus on how to find the minimum distance (minimum path length) for any given vertices rather than finding

the exact minimum path.It is also worth to mention that Floyd-Warshall algorithm also can used to find the transitive closure of a graph and detect if a graph contains of any negative cycles.

1.3 Floyd-Warshall algorithm

1.3.1 Intuition and Ideas

Before introducing this algorithm,it is helpful to illustrate the intuition and the main idea.Let's solve APSP for the following graph.



Before solving this question,let's introduce the following terms.

Intermediate vertex:v is an intermediate vertex locate in a path from i to j means v appear in a path but not the starting and ending vertex.For example,vertex1 is a intermediate vertex of the path $0-1-2$ which goes from 0 to 2.The path $0-1-2-0$ uses 2 intermediate vertices ,which are vertex 1 and 2.The path $0-1$ uses no intermediate vertex.

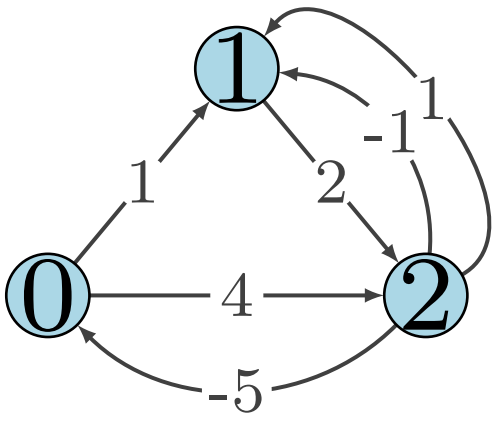
Constrained minimum distance: Constrained minimum distance $d(i,j,C)$ from i to j is not the real minimum distance $d(i,j)$ in the graph.$d(i,j,C)$ are defined as the minimum distance from i to j if we only allow to use some intermediate vertices in the vertex set C.**Notice**:$d(i,j,C)$ equal to $d(i,j)$ if and only if C is equal to the whole vertex set V. For example:$4 = d(0,2,\emptyset)$,$3 = d(0,2,\{0,1,2\}) = d(0,2)$.
According to the definition of intermediate vertex and constrained Minimum distance.We can set a very strict initial constraint to the problem and gradually loosen the constraint until there is no any constraint in the problem.This idea is called relaxation.To be more exact,we can find $d(i,j,\emptyset)$,$d(i,j,\{0\})$,$d(i,j,\{0,1\})$ and $d(i,j,\{0,1,2\})$ inductively.In order to record the weight clearly,we can use a matrix to record the constrained minimum distance from i to j.If there is no path from i to j under some constrain,we set $d(i,j,C) = \infty$.from i to i under some constrain,we set $d(i,j,C) = 0$.Since there is no negative cycle in the graph,which means we don't need any intermediate vertices from itself to itself ,so the minimum distance from a vertex to itself is 0 for any constrains.The following matrices represent the relaxation process of problem and solution in different degree of relaxation.

$$\begin{pmatrix} 0 & 1 & 4 \\ \infty & 0 & 2 \\ \infty & -1 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 & 4 \\ \infty & 0 & 2 \\ \infty & -1 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 & 3 \\ \infty & 0 & 2 \\ \infty & -1 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 & 3 \\ \infty & 0 & 2 \\ \infty & -1 & 0 \end{pmatrix}$$

It is clear that $d(i, j, \emptyset)$ is an edge with minimum wight from i to j.So it will be easy to construct the first matrix.For the rest of the matrix, we can use the following formula to construct them.

$$\forall k \in V, d(i, j, C \cup k) \leq min(d(i, j, C), d(i, k, C) + d(k, j, C)).$$

It will be explained in more clear latter on.
Next,let's consider how can we find the minimum distance in the following graph.



There is a negative cycle $0 - 2 - 0$ in the graph.Which means the minimum distance do not guarantee to be exist between any 2 vertices.Why?Since we can go into this negative cycle an infinite number of times,so the minimum distance from 0 to 2 is not well defined.

1.3.2 Imperative Floyd-Warshall algorithm

According to the idea of relaxation.The following is some set up and pseudo code of Floyd-Warshall algorithm written in imperative style.
Let (G, w) be a network not containing any cycles of negative length, and assume $V = 1, ..., n$. w_{ij} is the minimum edge length from i to j.Let $w_{ij} = \infty$ if $i - j$ is not an edge in G.M is used to record the minimum distance under different constrains for the relaxation process.
There are two main parts of the algorithm.The first part is initializing the minimum distance matrix M for no intermediate vertex is allowed to use.The second part is finding the minimum distance matrix M by using more and more intermediate vertices.The outermost for loop is responsible for introducing the intermediate vertices in turn.The rest of the for two loops are used to update the minimum distance matrix M when introducing the first $k - th$ vertices.

Algorithm 1 Floyd-Warshall algorithm[4]

1: **procedure** FLOYD-WARSHALL($G, w; M$)
2: **for** `i = 1 to n` **do** ▷ Initialize the minimum distance matrix M for no use intermediate vertex
3: **for** `j = 1 to n` **do**
4: **if** i == j **then**
5: $M_{ij} \leftarrow 0$
6: **else**
7: $M_{ij} \leftarrow w_{ij}$
8: **end if**
9: **end for**
10: **end for**
11: **for** `k = 1 to n` **do** ▷ Using the idea of relaxation to find the minimum distance
12: **for** `i = 1 to n` **do**
13: **for** `j = 1 to n` **do**
14: $M_{ij} \leftarrow min(M_{ij}, M_{ik} + M_{kj})$
15: **end for**
16: **end for**
17: **end for**
18: **return** M ▷ M is the minimum distance matrix for G
19: **end procedure**

1.3.3 Informal proofs

Let M_0 denotes the matrix generate in step 10. M_k denotes the matrix generated by the k-th iterations in step 14. $D_{k_{ij}}$ is the minimum distance from i to j use the intermediate vertices in set $\{1, \ldots, k\}$.$D_{0_{ij}}$ is the minimum distance from i to j use no intermediate vertices.

1.3.3.1 Theorem

$\forall k \in \{0, \ldots, n\}, \forall i, j \in \{1,\ldots,\text{n}\}, D_{k_{ij}} = M_{k_{ij}}$

1.3.3.2 Proof by induction

Base case:

According to step 2 to 10 in the algorithm and the definition of w_{ij}. $M_{0_{ij}} = D_{0_{ij}}$.

Inductive case:

Assume $\forall i, j, k, M_{k_{ij}} = D_{k_{ij}}$. The shortest path from i to j by using the intermediate vertices in the set $\{1, \ldots, k, k+1\}$ only has 2 possibilities.Either including vertex $k+1$ as an intermediate vertex or not including k.If the first situation is the minimum path,then $D_{(k+1)_{ij}} = D_{k_{ij}}$.If the second situation is the minimum path,the minimum path can be regarded as two part. The shortest path from i to $k+1$ and the shortest path from $k+1$ to j.By using the inductive hypothesis,we know $M_{k_{i(k+1)}} = D_{k_{i(k+1)}}$ and $M_{k_{(k+1)j}} = D_{k_{(k+1)j}}$.So,$D_{(k+1)_{ij}} = D_{k_{i(k+1)}} + D_{k_{(k+1)j}}$.According to the case analysis for vertex $k+1$.We get $M_{(k+1)_{ij}} = min(D_{k_{ij}}, D_{k_{i(k+1)}} + D_{k_{(k+1)j}})$.Which is equivalent to the step 14 in the algorithm.So after this iteration,we get $M_{(k+1)_{ij}} = D_{(k+1)_{ij}}$.

□

After showing the correctness of the theorem, we can let $k = n$,which will lead to $\forall i, j \in \{1,\ldots,\text{n}\}, D_{n_{ij}} = M_{n_{ij}}$.After the finial iteration,matrix D contains all the minimum distance from i to j without any

contains.So D is the true minimum distance matrix.

1.3.3.3 Discussion on the proof

This on paper proof is easy for people to understand.However,there are quiet a lot of things does not formalize.For example, minimum distances, shortest path,negative cycle.In order to verify the correctness of the algorithm,we need to formalize every things in the algorithm.

2 Related Work

[5]Previously, Paulin-Mohring used Coq to verify that the algorithm can find the transitive closure of a directed graph. However, she did not verify that the algorithm works for APSP problem. Wimmer and Lammich used functional implementation to verify the APSP problem for the algorithm and developed an efficient imperative implementation using the Imperative refinement Framework. But all of these are verified in Isabelle.I can not find some one use Coq to verify Floyd-Warshall algorithm solves APSP problem before and the idea of the algorithm is very elegant,so that is the reason why I write this paper.

3 Coq Basic

Coq is not only a proof assistant,but also a pure functional programming language.In this section,the reader can learn some basic programming syntax of Coq as well as how to use it proof some theorems.If the reader has already known Coq, feel free to skip this section.In addition,reading the first 8 chapters of Software Foundations Volume 1[6] and the first 2[2] chapters of Volume 3 or reading the first 8 chapters of Coq'art[3] is sufficient to understand this paper.

3.1 Coq installation

Here is the link to install Coq:`https://github.com/coq/platform/releases/tag/2022.04.1`
Or you can use Coq online:`https://jscoq.github.io/scratchpad.html`

3.2 Types In Coq,every well-defined terms must be have a type.Terms normally are identifies and expressions.For example,the natural number 0 has type **nat** and expression **0 + 0** is also has type **nat**.In Coq,if we want to check a type a term, we can use the keyword **Check**.For example:**Check 0** will return **nat**.When successfully install Coq ide,the symbol '0' still not defined.Which means there is no any built-in data types.But,Coq allows users to build any data types,writing some function and proving some properties related to this data types.The following are some examples of some important data type define in Coq library.If you want to use it you still need to import it.

3.2.1 bool

```
Inductive bool : Set :=
    true : bool | false : bool.
```

Here is a very simple definition of a type called **bool**.In order to definde a type in Coq we need to used a keyword **Inductive** and followed by a type name.**bool : Set** in the code also means the identifier **bool** also has a type,which actually is the type of a type and that is a **Set**.We also can let **bool : Type** or ignore to write a type for it.Since Coq is able to infer what the type should be.The contents after **:=** are the definition of a type.We know **bool** has 2 elements inside it.So we can use *true* and $false$ to construct **bool**.So we can say *true* and $false$ are the **constructors** of **bool**.In this definition,since we can list all the elements in this type,we called it as **enumerate type.**

3.2.2 nat

```
Inductive nat : Set :=
    O : nat | S : nat -> nat.
```

We also can define nat in Coq according to the Peano axioms.In this definition,there are two constructors. O and S. O correspond to 0 and S which is a function map a natural number to another natural number which we call it successor.For example,$O, S(O), S(S(O)), S(S(S(O)))$ are correspond to natural number $0, 1, 2, 3$.S also tell us **nat** actually defines in terms of itself.So we called this sort of type is a **recursive type.**

3.2.3 nat list

```
Inductive natlist : Type :=
  | nil
  | cons (n : nat) (l : natlist).
```

Here is the definition of nat list[6].The definition is very similar to **nat**.It also has a base case *nil* which corresponds to the empty list.The constructor *cons* is used to construct a longer list by adding a new natural number into a given list.For example,**cons 0(cons 1 nil)** correspond to **[0,1]**.In order to make the list more readable,we can define the some notation for **natlist**.

```
Notation "x :: l" := (cons x l)
                     (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).
```

3.2.4 Polymorphic Lists

```
Inductive list (A : Type) : Type :=
    nil : list A
  | cons : A -> list A -> list A
```

After defining nat list,we might consider define bool list or even some lists containing others type of elements.However the structure of the definition are the same.Hence it is useful to define a polymorphic lists which can define any sort of list with the same type element inside it.For the definition of polymorphic Lists.We can see that *list* is a function map a type to a type.So **list nat** and **list bool**

can be easily defined by function *list*.We also can define some instant of these type.For example **nil nat**,**nil bool** and **(cons nat 0 (cons nat 1 (nil nat)))** are the empty list of nat list,empty list of bool list and [0,1].However the code for [0,1] is too long the 'nat' term is redundant.Since coq has type inference system,according to 1 and 2 ,Coq knows that the list type must be **list nat**.So we can use the following code to make the type argument in *nil* and *cons* becomes implicit arguments.

```
Arguments nil {X}.
Arguments cons {X}.
```

After using the implicit arguments definition,we can use **(cons 0 (cons 1 nil))** to represent [0,1] in Coq.However coq is not able to infer the type of **nil** since,there is no element in the list.In order to represent nil in **nat list**.We need to use **@nil nat**,instead of **nil nat**.Since after defining implicit arguments nil take no input.Using symbol '@' can make the implicit arguments becomes explicit.

3.3 Function

In a functional programming language,the most important block is function.The following will show how to define a recursive and non-recursive function in Coq.After we defining a function,we can use keyword **Compute** to compute the output of a function apply to an input. For example,**Compute 1+1.** will return 2.Notice:'+' is also a function,take two natural number as input and return the sum of it.

3.3.1 negb

```
Definition negb (b:bool) : bool :=
  match b with
  | true => false
  | false => true
  end.
```

negb is a negation operator in bool.In coq,we can use **Definition** followed by a function name to define a function.**(b:bool) : bool** means the function takes a **bool** called b as input and return **bool** as output.After the ':=' symbol,it is the definition of the function.'match b with' is signifying we are going to analysis difference cases of input 'b'(pattern matching).For different case of 'b' we need to use '|' to separate them.The left hand side of '=>' is the input pattern of b and the right hand side of '=>' means what is the output for these given pattern on b.

3.3.2 plus

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

plus is a function takes 2 natural number as inputs and return the sum of them.According to the pattern matching on the first input n.There are 2 brunches.The first brunch is that the first number

is equal to 0.The other brunch is the first number is a predecessor of a natural number n'.By Peano arithmetic,for the second case of the first input n.We need to apply **plus** recursively.When defining a recursive function in Coq,we need to write the **Fixpoint** instead of using **Definition** to write a function.

3.4 Proposition After explaining how to defining a data type and function,we also need to know how to define propositions related to some data types and functions.For example,'The list is sorted' is a proposition about the a list.

3.4.1 Proposition as Types,Proof as Program

Like we mention before,every well-defined term must be have a type.Proposition also has a type which is **Prop** in Coq.And the evidences can proof the proposition also has a type which is the proposition itself.That is 'Proposition as Types'. However,What is the point of consider proposition as type?Let's consider a following proposition.

$P \implies (P \implies Q) \implies P$,for all P, Q are proposition.

We can proof this proposition into ways.The first way is that we can view each proposition variable takes 2 value, either true or false.By using truth table we can find out that if the proposition always true for all difference combination of P and Q.The second way to proof this problem is much more intuitive.We can consider the original proposition like:If we know how to proof P and also if we know how to proof Q if we have already proof P,then We can proof Q.It is clear from this point of view why this proposition is true.Since we have know the proof of P,and by using $(P \implies Q)$ we will know Q is also true.This is really like using function application.In fact we can define $H0 : P, H1 : P -> Q$.H0 is the evidence or hypothesis that P is true.H1 a function takes the evidence of P is true as input and return Q is also true.So the term H1(H0) has type Q,which means the program H1(H0) is the proof of Q.That is 'Proof as program'.Combining 'Proposition as types'we also called that Curry-Howard correspondence.

3.4.2 Proposition construction

Like some data types we define before,we also can define our proposition in Coq.It has types **Prop**.For example $1 = 1; \forall n \in nat, n+1 > n$ are two proposition.The following materials will cover some examples to construct some basic and useful proposition.

3.4.3 True[6]

```
Inductive True : Prop :=
  | I : True.
```

Here is definition of the True proposition.Notice:the proposition 'True' is not the same 'true'.'True' has type **Prop**,but true has type **type**.We can define a proposition like normal data types.But,the constructor of it now is the evidence.The left hand side of ':' is the evidence name I.The right hand side is what the evidence actually is.For this example,to proof proposition True is simple,since True itself is a evidence of the proof.**Notice**:If the reader think this is difficult to understand,you can skip this and return back after understand some of the following propositions first.

3.4.4 False[6]

```
Inductive False : Prop :=.
```

This is the definition of False proposition means a proposition is false.If a proposition is false,which means we can not find any kind of evidences to proof it,so there is no constructors inside the definition of False proposition.

3.4.5 not[6]

```
Definition not (P:Prop) := P -> False.
```

We also can define function which can deal with proposition.The function *not* takes arbitrary proposition as input and return the proposition False.When we want to formalize a proposition which is not true,we can a apply *not* to this proposition.

3.4.5 sorted[6]

```
Inductive sorted: list nat -> Prop :=
| sorted_nil:
    sorted nil
| sorted_1: forall x,
    sorted (x::nil)
| sorted_cons: forall x y l,
   x <= y -> sorted (y::l) -> sorted (x::y::l).
```

sorted is a predicate which can state the property of the input.If there are two inputs we can called the proposition as a relation.For example '=' is a relation between two object have the same type.From the definition of **sorted**,we know that,if we want to proof a nat list is sorted.There are 3 evidences for us to seeking for.First of all,if the input list is an empty list,it is sorted.Second,if the input has only one element,it is sorted.The final evidence state that if the fist element of the list is less then or equal to the second and the sub list not containing the first element is sorted,then we known the original input list is also sorted.

3.4.6 Logical connectives

We can using more logical connectives to build more complex proposition.

```
/\      (*and*)
\/      (*or*)
not     (*not*)
-> (*implies*)
<->   (*if and only if*)
forall  (*for rall*)
exists  (*exists*)
```

3.5 Theorem proving To build a proposition we are going to proof in Coq,we can use the following keyword to specifying a proposition:'Theorem','Lemma' and 'Example'.If we want to state a Axiom we can use keyword 'Axiom'.The following are some examples.

```
Theorem associate:forall (p q r:nat),(p + q) + r = p + (q + r).
Example one_plus_one: 1 + 1 = 2.
```

3.5.1 Proof system After defining theorem we can start to proof it use a keyword **Proof** and Coq we illustrate a window containing the goal we need to proof.

```
Theorem T1: negb(negb true) = true.
Proof.
```

```
----------------------------
1 subgoal
______________________________________(1/1)
negb (negb true) = true
```

3.5.2 Tactics

Coq provides a lot of tactics for user to solve the goals.The following are some examples of how to some important tactics.

simpl

simpl is used for simplify the goals.for example,return a result for function application.

```
Theorem T1: negb(negb true) = true.
Proof.
simpl.
----------------------------
1 subgoal
______________________________________(1/1)
true = true
```

when using **simpl**,the term **negb(negb true)** will simplify to true.

reflexivity

We can used tactic **reflexivity** to proof the goal if the goals is $a = a$.If we solve all the goals,we need to use **Qed** to end the proof.

```
Theorem T1: negb(negb true) = true.
Proof.
simpl. reflexivity.
Qed.
```

auto

If a theorem can use a sequence of **simpl** and **reflexivity** to proof, we can use tactic **auto** to proof it.

intros

When there is universal quantifiers in the goals,We can use tactic **intros** to move the universal quantifiers into the current context.

```
Theorem T2: forall n:nat,plus O n =n.
Proof.
----------------------------------------------
1 subgoal
______________________________________(1/1)
forall n : nat, plus O n = n

----------------------------------------------
```

```
Theorem T2: forall n:nat,plus O n =n.
Proof.
intros.
--------------------------------------------
1 subgoal
n : nat
--------------------------------------(1/1)
plus O n = n
--------------------------------------------
Theorem T2: forall n:nat,plus O n =n.
Proof.
intros.
simpl. reflexivity.
Qed.
```

intros also allow to move the assumption in the goals to the current context.

```
Theorem T5: forall P:Prop,
P -> P .
Proof. intros. assumption.
Qed.
```

assumption
Tactic **assumption** is used to solve a goal if it has been in the current context, which means it has already proof.
destruct
simpl can not work all the time.Since the outputs of some functions depend on the input patterns.So before applying **simpl** we need to do case analysis on the input variable by **destruct**.**destruct b** means case analysis variable b.

```
Theorem T3: forall b:bool,negb(negb b) = b.
Proof.
intros.
destruct b.
2 subgoals
-------------------------------------(1/2)
negb (negb true) = true
-------------------------------------(2/2)
negb (negb false) = false
```

When we use **destruct b**,we will need to proof 2 sub goals instead of 1 goal.We can use + to focus on the the first goal or using **Focus 2** to solve the second goal first.Alternatively,we can use curly brackets to focus our goal.If we want to proof the whole theorem,we need to solve all the sub goals.

```
Theorem T3: forall b:bool,negb(negb b) = b.
Proof.
```

```
intros.
destruct b.
+ simpl. reflexivity.
+ simpl. reflexivity.
Qed.
```

destruct also can use to decompose a hypothesis containing 'AND' into 2 assumptions.Finally, **destruct** can case analysis on some assumptions which contains 'OR'.

left/right

If 'OR' is in our goals,we can use **left** or **right** to fit our evidence in the goal.The following is two examples.

```
Theorem T6: forall P Q:Prop,
P -> P \/ Q.
Proof.
intros. left.
assumption.
Qed.
Theorem T7: forall P Q:Prop,
Q -> P \/ Q.
Proof.
intros. right.
assumption.
Qed.
```

split

When the goal have 'AND',we can split the goal into two sub goals using tactic **split**. **apply**

apply is used to apply some hypothesis to solve the goal.We can use **apply** in the following 2 different ways.

```
Theorem T7: forall P Q:Prop,
P -> (P->Q) -> Q.
Proof.
intros. apply H0.
assumption.
Qed.
------------------------------
Theorem T7: forall P Q:Prop,
P -> (P->Q) -> Q.
Proof.
intros. apply H0 in H.
assumption.
Qed.
```

The first proof is assuming the goals is correct under some assumptions and proofing if the assumptions are correct.

The second proof is just simply apply the function $H0$ to H to find the evidence that the goal is correct.
We also can apply some proved theorem to proof a new theorem by **apply**.

```
Theorem T8: forall P Q:Prop,
P -> (P->Q) -> Q.
Proof.
intros.
apply (T7 P Q). assumption. assumption.
Qed.
```

Finally,we also can apply some evidence of the inductive definition of propositions.

```
Theorem T9: sorted [1;2].
Proof.
apply sorted_cons.
+ lia.
+ apply sorted_1.
Qed.
```

lia
Tactic **lia** can used to proof some simple proposition such as $1 \leq 2$.
rewrite/unfold
rewrite $\rightarrow$ **H** is rewrite the goals in assumption H.**unfold f**
is unfold the definition of a function f.
induction
We also can proof theorem by induction on inductive types.When we use this tactics.The original theorem will decompose into 2 goals.The first goal is the base case and the second goal is the inductive case.The following proof is how we use **induction** to proof the associate laws in natural number.

```
Theorem associate:forall (p q r:nat),(p + q) + r = p + (q + r).
Proof.
intros. induction p.
+ simpl. reflexivity.
+ simpl. rewrite -> IHp . reflexivity.
Qed.
```

4 Coq Library

The first step of the project is loading the following library into the current environment.

```
From Coq Require Import Arith.Arith.
From Coq Require Import Bool.Bool.
```

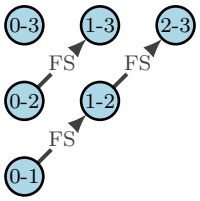
```
From Coq Require Import Logic.FunctionalExtensionality.
From Coq Require Import Lists.List.
Import ListNotations.
Require Export Coq.Vectors.Fin.
Require Export Coq.ZArith.ZArith.
From Coq Require Import Lia.
```

5 Data structure

Node
Nodes(Vertices) are the most basic elements of a edge and graph. In order to formalize edge and graph,we need to formalize a type called **Node** first.Since there are finite nodes in the graph,we can use a finite set to formalize type **Node**.'Library Coq.Vectors.Fin' has already defined the type of size n finite set.But,it is worth to explain the definition of it.
Node.definition

```
Inductive t : nat -> Set :=
|F1 : forall {n}, t (S n)
|FS : forall {n}, t n -> t (S n).
```

t is a type depends on a nat.t n means a set contains n elements.i.e. $\{0, \ldots, n-1\}$.**t 0** means an empty set.It might be useful to use a graph to illustrate how to construct element according to this inductive definition.



$m-n$ means the element m in **t n**.We can construct 0 in **t (S n)** by using construct $F1$.And other element by using FS and a node in type **t n**.For example,if we want to represent $2-3$ we need to use FS and node $1-2$.So the arrow form the graph illustrate how to construct them.The codes below are all the 3 elements in the set **t 3**.

```
Check @F1 2. (*  0 in t 3  *)
Check FS (@F1 1). (* 1 in t 3  *)
Check FS( FS (@F1 0)). (* 2 in t 3  *)
```

In order to make type **t** more readable,I rename it as **node**.

```
Notation "'node'" := (t).
Check FS( FS (@F1 0)). (* node 3  *)
```

Node.

Node.function

```
Fixpoint NodeLst (n:nat):list (node n):=
(*
 NodeLst n can return a list which contains all
 elements in node n.
*)
match n with
| O => nil
| S O =>  [@F1 0]
| S (S n') => @F1 (S n')
      :: map FS ((@F1  n') ::(map FS (NodeLst  n')))
end.
```

The function takes a natural number n as input and return a list which contains all the elements in **t n**. **map** is a function in coq ,takes a map f and a list l as input,and return a new list by mapping l by using f.
Here are some output by using**NodeLst**:

```
Compute NodeLst O.
Compute NodeLst 1.
Compute NodeLst 2.
Compute NodeLst 3.

(*Output*)
         = []
     : list (node 0)
         = [F1]
     : list (node 1)
         = [F1; FS F1]
     : list (node 2)
         = [F1; FS F1; FS (FS F1)]
     : list (node 3)
```

Node.axiom

NodeLst-Correctness
The following axiom is about the properties of the function **NodeLst**.

```
Axiom  NodeLst_Correctness:
forall (n:nat)(x:node n),(1 <= n)%nat -> In x (NodeLst n).
```

NodeLst-Correctness says every element in **node n** must be in **NodeLst n**.

optionZ
After formalizing the data type **node**,we can use them to build a new type **edge**.But,before we build the new type,we also need to use a data structure to represent the weight of an edge.In Floyd-Warshall algorithm,the edge's weight can be any real number R,For simplicity,I use the integer Z.However data type **Z** in coq is still not enough to represent ∞.So it is necessary to define a new data type include the ∞ symbol in **Z**.The new data type is called **optionZ**.

optionZ.definition

```
Inductive optionZ := inf | Some (z :Z).
```

There are 2 constructors for this data type.inf represent ∞ and $Some$ can map any integers to this data type.

optionZ.function

oZadd

```
Definition oZadd (n m:optionZ) : optionZ :=
  match n,m with
  | inf,_ => inf
  | _,inf => inf
  | Some n',Some m' => Some (n'+m')
end.
Notation "x + y" := (oZadd x y) : optionZ_scope.
```

The definition of addition in **optionZ** is called $oZadd$.If one input of the function is inf,the output is still inf.Ortherwise, just add this two inputs like type **Z**.Finally define notation '+' for $oZadd$.

```
Open Scope optionZ_scope.
Compute  inf + (Some (-100000000000)).
Compute  inf + (Some 100000000000).
Compute  (Some 10) + (Some 10).
(*Output*)
         = inf
     : optionZ
         = inf
     : optionZ
         = Some 20
     : optionZ
```

oZleb

```
Definition oZleb (n m:optionZ) : bool :=
  match n,m with
  | _,inf => true
  | inf,_ => false
  | Some n',Some m' => n' <=? m'
end.
```

oZleb is a function takes two element form **optionZ** and return a **bool**.The definition actually is less than or equal to relation for **optionZ**. In **optionZ**,any elements are less than or equal to ∞.if non of the inputs are ∞,we can just compare the integer part of this two inputs.

```
Notation "x <=? y" := (oZleb x y) (at level 70) : optionZ_scope.
Compute inf <=? inf.
Compute  Some 0  <=?  Some 0 .
Compute Some (-1) <=? Some 1.
Compute Some 1 <=? Some (-1).
```

oZmin

```
Definition oZmin (x y : optionZ) : optionZ:=
  if x <=? y then x else y.
```

oZmin is simply return the minimum varaible from two varable are **optionZ**.

```
Compute oZmin (Some (-1))(Some (-2)).
(*output*)
         = Some (-2)
  : optionZ
```

oZle

```
Inductive oZle : optionZ -> optionZ -> Prop :=
  | oZle_1inf (n:optionZ) : oZle n inf
  | oZle_nm (n m:Z): n <= m -> oZle (Some n) (Some m).
Notation "n <= m" := (oZle n m).
```

oZle is a predicates and actually it is the relation '$\leq$' **optionZ**.there are 2 evidences which can poof the proposition. *oZle-1inf* means every element is less than or equal to inf. *oZle-nm* means if the integer part of two elements from **oZle** n, m satisfy $n \leq m$,the relation *oZle* is also satisfy for the original two inputs.

optionZ.theorem

oZleb-reflect

```
Theorem oZleb_reflect : forall (x y:optionZ), reflect (x <= y) (x <=? y).
```

The proof of **oZleb-reflect** contains two part:The first part shows that **oZleb** imples **oZle**.The second shows **oZle** imples **oZleb**.The main ideas of the 2 proofs are using **destruct** and **apply**.Since **optionZ** is built by **Z**,so these poofs will finally end up with showing the following.

```
forall n m : Z, (n <=? m)%Z = true <-> n <= m
```

This is a theorem in **Z.leb-le**.So we can apply this to finish the sub proofs.

oZmin-Correctness

```
Theorem oZmin_Correctness: forall (x y : optionZ),
oZmin x y <= x /\ oZmin x y <= y.
Proof.
  intros. split.
  + unfold oZmin.
-----------------------------------
x, y: optionZ
-----------------------------------
1/1
(if x <=? y then x else y) <= x
-----------------------------------
```

That is the **if-then-else** block we need to work at.we can use **oZleb-reflect** to decompose this block.After that,the main goal will be split into two different goals,one is for the then branch the other is for the else branch.

```
assert (R: reflect (x <= y) (x <=? y)).
apply oZleb_reflect. destruct R.
```

```
Goal:
x, y: optionZ
o: x <= y
-----------------------------------
1/2
x <= x
2/2
y <= x
```

optionZ.axiom

oZleb-Add-zero

```
Axiom oZleb_Add_zero: forall (x:optionZ) ,x =  x + Some 0.
```

Axiom oZleb-Add-zero is similar to $z = z + 0$ for $z \in Z$.

ozle-a1,ozle-a2,ozle-a3

```
Axiom ozle_a1: forall (a b c : optionZ),
not (b <= a) -> (b <= c) -> (a <= c).

Axiom ozle_a2: forall (a b c d e: optionZ),
(a <= b) -> (c <= d)-> (e <= a + c) -> (e <= b + d).

Axiom ozle_a3: forall (a b c d : optionZ),
(a <= b) -> (c <= d)-> (a + c <= b + d).
```

These axioms are easy understand by regards the type **optionZ** as **Z**.

edge

edge.definition

```
Inductive edge (n:nat) : Type :=
  Edge (x:node n)(y :node n) (z:optionZ).
Arguments Edge {n}.
```

edge is a dependent type and dependent on a natural number n.It only allows to connect 2 nodes in the same node sets.An edge contains 3 parts,the first 2 parts are the two node linked by the edge.The third part is the weight of the edge and it has the type **optionZ**.

```
Check Edge  (@F1 2) (FS (@F1 1)) (Some(-1)).
Fail Check Edge (@F1 1)(@F1 2) (Some 3).

(*output*)
Edge F1 (FS F1) inf
         : edge

The term "F1" has type "node 3"
while it is expected to have type
 "node 2".
```

The first example denotes an edge connecting 0,2 in node set $\{0, 1, 2\}$ with weight -1.The second example is not well typed since by definition of edge, we only allow to connect two nodes in the

same node set.
edge.function

fst,snd,trd

```
Definition fst {n:nat} (e:edge n):=
  match e with
  | Edge x _ _  => x
  end.

Definition snd {n:nat} (e:edge n):=
  match e with
  | Edge _ y _  => y
  end.

Definition trd {n:nat} (e:edge n):=
  match e with
  | Edge _ _ z  => z
  end.
```

These 3 functions fst,snd and trd take an edge as input an return the first,second and third component respectively.

```
Compute trd (Edge  (@F1 2) (FS (@F1 1)) (Some (-100))).
(*output*)
         = Some (-100)
     : optionZ
```

Graph

Graph.definition

```
Notation "'Graph' n" := (list (edge n))(at level 100).
```

The type **Graph n** are a set of graphs which have exactly n nodes and a graph is a list of element of type **edge n**.This graph definition can represent many types of graph.Such as multigraph,directed graph or graphs containing self-loop.

```
Check @nil (edge 3).

Definition mygraph :=
[Edge  (@F1 2) (FS (@F1 1)) (Some 1);
        Edge  (@F1 2) (FS(FS (@F1 0))) (Some 4);
        Edge   (FS (@F1 1)) (FS(FS (@F1 0))) (Some 2);
```

```
       Edge   (FS(FS (@F1 0))) (FS (@F1 1)) (Some 1);
        Edge   (FS(FS (@F1 0))) (FS (@F1 1)) (Some(-1))].

Fail Check [Edge  (@F1 2) (FS (@F1 1)) (Some 3);
        Edge  (@F1 2) (FS(FS (@F1 0))) (Some 4);
        Edge  (@F1 1) (FS (@F1 0))  (Some(-5))].

(*output*)
[]
         : Graph 3

The term
 "[Edge F1 (FS F1) (Some (-5))]"
has type "Graph 2"
while it is expected to have type
 "Graph 3".
graph is defined as a list of edge and Graph n means a graph
```

The first example is a graph has 3 nodes but no edges. The second example is the graph in **intuition and ideas** part in this paper. The third example is not a valid graph,since the edges are not from the same graph.

Graph.function

Find-weight

```
Fixpoint Find_weight {n:nat} (i j : node n) (g : Graph n) (current:optionZ) : optionZ :=
if eqb i j then Some 0
else
match g with
| nil => current
| h :: t => if eqb (fst h) i && eqb (snd h) j && (trd h <=? current)
            then Find_weight i j t (trd h)
            else Find_weight i j t current
end.
```

Find-weight is used to find the minimum edge length from i to j in order to build the adjacency matrix latter.The idea of this function is simple,If we want to find the minimum edge from 2 nodes ,we can initialize the minimum edge length as ∞ and search the the minimum length in the edge list.There is two special cases in this function. The minimum edge from a node to itself is defined as 0 and if there is no direct edge between two nodes,the function will return ∞.

```
Compute Find_weight (FS(FS (@F1 0))) (FS (@F1 1))  mygraph inf.
(*output*)
```

```
        = Some (-1)
    : optionZ
```

Graph.axiom

Find-weight-a1

```
Axiom Find-weight-a1: forall (n:nat) (i :node n)(g : Graph n),
Find-weight i i g inf = Some 0.
```

Axiom Find-weight-a1 claims function **Find-weight** will return that 0 when searching the minimum edge from a node to itself.
Find-weight-a2

```
Axiom Find_weight_a2: forall (n:nat) (i j:node n)(g : Graph n)(e:edge n),
In e g -> fst e = i -> snd e = j -> Find_weight i j g inf <= trd e.
```

Axiom Find-weight-a2 claims the weight return by function **Find-weight** are indeed the minimum edge weight from 2 different nodes.

Matrix

```
Definition Matrix n := (node n) -> (node n) -> optionZ.
```

Matrix is the final data structure for this paper.Unlike defining matrices as list of list in some programming language.In functional programming language like **Coq**,we can define matrices using the idea of mapping.**Matrix n** takes two nodes as inputs and return a value of type **optionZ**.This definition is used to build the adjacency matrix.

6 Algorithm

adj-Matrix,Update and Floyd-warshall

```
Definition adj_Matrix {n:nat} (g : Graph n) : Matrix n :=
   fun i j =>  Find_weight i j g inf.

Definition Update {n:nat}(m:Matrix n)(v:node n):(Matrix n):=
fun i j => if m i j <=?  (m i v) + (m v j) then  m i j else  (m i v) + (m v j).


Fixpoint Floyd_warshall {n:nat}(m:Matrix n)(vlist:list (node n)) : (Matrix n):=
```

```
  match vlist with
|  nil => m
|  h::t =>  Floyd_warshall (Update m  h) t
end.
```

This there functions are functional implementation of Floyd-warshall algorithm.
The first function **adj-Matrix** is finding the adjacency matrix in a graph.The entries of the adjacency matrix are defined as the minimum edge length of a graph,since the definition of the graph also allow to define multigraph.
The second function *Update* takes a constrain minimum distance matrix and a node as input and returning a new minimum distance matrix by consider adding a new node v.
The third function takes a distance matrix m and a list of intermediate nodes *vlist* as input and updates the matrix m using the nodes in *vlist* one by one.The function will stop and return the current minimum distance matrix when all the nodes in *vlist* are considered.i.e vlist is a empty list.

```
Compute (Floyd_warshall (adj_Matrix mygraph) (NodeLst 3)) F1 F1.
Compute (Floyd_warshall (adj_Matrix mygraph) (NodeLst 3)) F1 (FS F1).
Compute (Floyd_warshall (adj_Matrix mygraph) (NodeLst 3)) F1 (FS(FS F1)).
(*output*)
         = Some 0
     : optionZ
         = Some 1
     : optionZ
         = Some 3
     : optionZ
```

There 3 examples are some little experiments show the algorithm return what we expect.These outputs are the first row of minimum distance matrix for *mygraph* and they are the same as **page 2**.It is also worth to mention that,the input matrix of the Floyd-warshall algorithm is the adjacency matrix of the objective graph.

7 Predicate,Proposition,Function

IsPath

```
Inductive IsPath (n:nat): (Graph n) -> Prop :=
  | IsPath0:forall (e:edge n), IsPath n [e]
  | IsPath1:forall (h m:edge n) (t:Graph n) ,
    IsPath n (m::t) -> snd h = fst m -> IsPath n (h::m::t).
Implicit Arguments IsPath [n].
```

IsPath is a predicate, it takes a list of edges g as input and return a proposition.**Notice**: a list of **edge n** also has type **Graph n**.This inductive definition *IsPath* g is considered to be a path if it

has only one edge or all the edges are end to end.For example a list containing a single edge is a path;[(1 2 3);(2 6 5);(6 3 4)] is a path $1-2-6$;[(1 2 5);(3 4 5)] is not a path since the two edge do not connect to each other.

PathLength

```
Fixpoint PathLength{n:nat} (p:Graph n): optionZ:=
match p with
| nil => Some 0
| h::t =>  oZadd (trd h)  (PathLength t)
end.
```

PathLength is used to calculate the path length of a path or the overall weight sum of a given graph.

PathLength-a1

```
Axiom PathLength_a1:forall (n:nat)(p p': Graph n),
PathLength p + PathLength p' = PathLength ( p ++ p').
```

Axiom **PathLength-a1** says that the path length of connecting two paths together is equal to the sum of these two path lengths.

InGraph

```
Fixpoint InGraph {n:nat} (p:Graph n) (g:Graph n): Prop :=
match p with
| nil => True
| h :: t =>
(In h g \/
trd h = inf
\/ (trd h = Some 0 /\ fst h = snd h))
/\ InGraph t g
end.
```

InGraph is a predicate and it used to define a the proposition that all the edges from a given graph p is from the original graph g.This is similar to the definition as subgraph in graph theory,but there are some differences here.In the original graph g,there are actually 2 types of invisible edges in it even there are not in the original edge list.The first type is that an edge has weight 0 and connect from a node to itself.The second type is that an edge connect to any 2 nodes which has infinity weight length.

FromI2J

```
Fixpoint FromI2J {n:nat} (i j: node n) (p:Graph n): Prop :=
match p with
```

```
| nil => False
| e :: nil => (fst e = i)  /\ (snd e = j)
| e :: l => (fst e = i) /\  snd( last l (Edge  i j (Some 0)) ) = j
end.
```

FromI2J is a predicate for a list of edge p and two nodes i, j.It is used to return the proposition that the fist element of the edge in p is i and the final edge of g is j.If p is a path and p also satisfy **FromI2J i j p**,then the p actually is a path from node i to j.

UseIntermediate

```
Fixpoint UseIntermediate {n:nat} (p:Graph n) (l: list (node n)) : Prop :=
match p with
| nil => True
| e::nil => True
| h :: t => In (snd h) l  /\  UseIntermediate  t l
end.
```

UseIntermediate is a predicate for a list of edge p as well as a list of intermediate nodes l and returning a proposition describing p only used intermediate nodes.For example, a single edge [(1,2,3)] and a path [(1,2,3);(2,3,5)] use intermediate node list [2].

ValidPath

```
Definition ValidPath {n:nat}
(p: Graph n) (g:Graph n) (i j: node n) (l:list (node n)): Prop :=
  (IsPath p ) /\ (InGraph p g) /\ (FromI2J i j p) /\ (UseIntermediate p l).
```

After defining the **IsPath, InGraph, FromI2J and UseIntermediate**, we can use these to construct a **very important** proposition in the paper.**ValidPath p g i j l** means p is a path from i to j in graph g and only use intermediate nodes in node list l.So if a edge list p satisfy this proposition,we can simply simply say p is a valid path.

```
Example ValidPathE1:
ValidPath [Edge (@F1 2)  (FS(@F1 1)) (Some 1);
           Edge (FS(@F1 1)) (FS (FS (@F1 0))) (Some 2)]
           mygraph (@F1 2) (FS (FS (@F1 0))) [(FS(@F1 1))].
```

This is an example about [(0,1,1);(1;2;0)] is a valid path in *mygraph* by using node 1 as intermediate node.The proof actually is not hard,we can use the tatic **split** to proof **IsPath, InGraph, FromI2J and UseIntermediate** separately.

NoNegCyc

```
Definition NoNegCyc {n:nat} (g:Graph n) : Prop :=
forall (p:Graph n)(i:node n)(l :list (node n)),
ValidPath p g i i l -> Some 0 <= PathLength p.
```

NoNegCyc g is a proposition which says a graph g has no negative cycle.By the definition,it actually means for all valid path p in original graph g.If p is a cycle, the length of it must be larger and equal to 0.

```
Definition MinDistance {n:nat}
(d:optionZ ) (i j: node n)(g: Graph n)(l : list (node n))
 : Prop :=
 (exists (p: Graph n), (ValidPath p g i j l) /\ (d = PathLength p)  /\
 (forall (p': Graph n), (ValidPath p' g i j l) -> d <= PathLength p'))
```

MinDistance d i j g l is a proposition saying that d is the minimum distance from i to j in graph g by using the intermediate vertices in l.The definition can be explianed by the following.There must be exist a valid path p from i to j by intermediate nodes l and the path length must be equal to the distance d.In addition,d must be less than or equal to all valid paths by using intermediate nodes l.

SelectSP

```
Definition SelectSP {n:nat} (p p':Graph n) : Graph n:=
  if PathLength p <=? PathLength p' then  p  else p'.
```

SelectSP is a function takes two path as input and returning the shortest path.

SelectSP-a1

```
Axiom SelectSP_a1: forall (n:nat)(i j a :node n)(g x x0 x1 :Graph n) (l: list (node n)),
ValidPath x g i j l ->
ValidPath x0 g i a l ->
ValidPath x1 g a j l ->
ValidPath (SelectSP x (x0 ++ x1)) g i j (a :: l).
```

SelectSP-a1 is an axiom which says that by using intermediate nodes l,if x is a valid path from i to j, $x0$ is a valid path from using i to a and $x1$ is a valid path from a to j.The shorter path for these two path x and $x0 + +x1$ is also a valid path by using larger intermediate node set $(a :: l)$.

8 Axiom,Lemma,Theorem and Proof

In this part,we are mainly focus on the In this part,we are mainly focus on formalizing and proofing the correctness of Floyd-Warshall algorithm.

Cyc-is-zero

```
Lemma Cyc_is_zero:
forall (n:nat) (g p:Graph n) (i:node n),
NoNegCyc g ->
MinDistance (Some 0) i i g [].
```

Cyc-is-zero says if a graph does not contains any negative cycle,the minimum distance from a node to itself without using intermediate nodes is 0.The proof of it actually is simple we can use the path [(i,i,0)] to handle the **exsist** part in the goal.And using the fact that the graph has no negative cycle,we can conclude that the [(i,i,0)] is minimum.

Base-Case-Correctness-l1

```
Axiom Base_Case_Correctness_l1:
forall (n:nat) (g :Graph n) (i j : node n),
InGraph [Edge i j (adj_Matrix g i j)] g.
```

The axiom simply means any edge with the minimum edge weight is still in the graph.
in-a1

```
Axiom in_a1: forall (n: nat)(i j :node n)(g p: Graph n),
i <> j -> ValidPath p g i j  [] -> (In (Edge i j (PathLength p)) g) \/(PathLength p = inf).
```

Axiom in-a1 says that if an edge is a valid path and it is not a self ring ($i \neq j$).It must in the original edge list g or the weight of the edge is equal to ∞.

Floyd-Warshal-a1

```
Axiom Floyd_Warshal_a1:forall (n:nat)(i j a: node n)(g: Graph n)(l : list (node n)),
Floyd_warshall (adj_Matrix g) (a :: l) i j =
 oZmin (Floyd_warshall (adj_Matrix g) l i j)
  ((Floyd_warshall (adj_Matrix g) l i a) +
  (Floyd_warshall (adj_Matrix g) l a j)).
```

Axiom **Floyd-Warshal-a1** is exactly how **Floyd-Warshal-a1** define.

Three-case-a1

```
Axiom Three_case_a1: forall (n:nat)(i j a:node n)(g p':Graph n)(l:list(node n)),
ValidPath p' g i j (a :: l) ->
ValidPath p' g i j l \/
(exists (p1' p2' : Graph n), ValidPath p1' g i a l /\ ValidPath p2' g a j l
/\ p' = p1' ++ p2')
\/
(exists (p1' p2' p3' : Graph n), (ValidPath p1' g i a l) /\  (ValidPath p2' g a a l) /\
(ValidPath p3' g a j l) /\ (p' = p1' ++ p2' ++ p3')).
```

Axiom **Three-case-a1** says that if p is a valid path use intermediate node list $a :: l$.Let a appear k times in l.Path p' has 3 different cases.the first case is p' only use l as intermediate nodes.The second case is that p' uses a as intermediate node and uses a exactly $k+1$ times.More specifically,p' is built by two paths.By using l,they are the paths from i to a and from a to j.The third case is that p' uses a more than $k+1$ times.So p' is built by three paths.By using l,they are the paths from i to a,from a to a and from a to j.

Repeat-node-not-shortest

```
Axiom Repeat_node_not_shortest:
forall (n:nat)(i j a: node n)(g x2 x3 x4: Graph n)(l : list (node n)),
NoNegCyc g ->
ValidPath x2 g i a l ->
ValidPath x3 g a a l ->
ValidPath x4 g a j l ->
oZmin (Floyd_warshall (adj_Matrix g) l i j)
  (Floyd_warshall (adj_Matrix g) l i a +
   Floyd_warshall (adj_Matrix g) l a j) <=
PathLength (x2 ++ x3 ++ x4).


forall (n:nat)(i j: node n)(g: Graph n),
NoNegCyc g->
MinDistance ((adj_Matrix g) i j) i j g nil.
```

Repeat-node-not-shortest says if a graph does not contains any negative cycles,if a path repeating using one node as intermediate node.Then,it can not be shortest path.It larger than **min(Floyd-warshall (adj-Matrix g) l i j), (Floyd-warshall (adj-Matrix g) l i a + Floyd-warshall (adj-Matrix g) l a j)**.

Base-Case-Correctness

```
Theorem Base_Case_Correctness:
forall (n:nat)(i j: node n)(g: Graph n),
NoNegCyc g->
MinDistance ((adj_Matrix g) i j) i j g nil.
```

The method to proof the correctness of the algorithm is induction on the intermediate node list.**Base-Case-Correctness** is the theorem of the base case.The theorem says that if a graph does not have any negative cycle,the minimum distance from i to j by using no intermediate nodes is the corresponding entry in the adjacency matrix.We can split this whole proof into two goals.
The first goal only consider the case when $i = j$.In this situation,we can just solve it by lemma **Cyc-is-zero**.
The second goal is consider when $i \neq j$.First of all we can use [Edge i j ((adj-Matrix g) i j)] as the minimum path to handle **exists** in the goal.The first part we need to show is that the path is a valid path.The key to proof this is use **Base-Case-Correctness-l1**.After that we also need to proof the path is shorter then other valid paths.So if we use no any intermediate nodes.We can use only two types of path to describe all paths.The first type of path are edges which are in the original graph g,the other are those invisible edges(edge with $\infty weight$).The rest of proofs are not difficult,the key to proof them is using axiom **Find-weight-a2**.

Inductive-Case-Correctness

```
Theorem Inductive-Case-Correctness:
forall (n:nat)(i j a: node n)(g: Graph n)(l : list (node n)),
 NoNegCyc g
 -> (MinDistance (Floyd_warshall (adj_Matrix g) l i j) i j g l)
 -> (MinDistance (Floyd_warshall (adj_Matrix g) l i a) i a g l)
 -> (MinDistance (Floyd_warshall (adj_Matrix g) l a j) a j g l)
 -> MinDistance (Floyd_warshall (adj_Matrix g) (a::l) i j) i j g (a::l).
```

This theorem is the inductive case of the induction.It says that if a graph dose not contains any negative cycle and by using function **Floyd-warshall**,we can correctly found the minimum distance from i to j, i to a and a to j by using intermediate node list l and i, j, a are arbitrary nodes in the graph.Then we function **Floyd-warshall** can also correctly find the minimum distance from i to j by using intermediate node list $a :: l$.
The proof is the most difficult part in this paper.Like before,we need to find the shortest path from i to j to move **exists** from the goal to the context.**p=(SelectSP x (x0 ++ x1))** is the path.The key to proof p is a valid path,we can apply **SelectSP-a1** to proof p is a valid path.
Next we need to proof that the path length of p must be the same as the **Floyd-Warshall** output.We can rewrite **PathLength-a1** and **Floyd-Warshal-a1** to solve this goal.
The final goal is to proof that p is shorter then all other valid paths p'.Since the intermediate node list is $a :: l$,we can think p' has 3 different types.if a appear k times is l. the first case is p' does use a exactly k times as intermediate node.The second case is that p' uses a as intermediate node and uses a exactly $k + 1$ time.The third case is that p' uses a more than $k + 1$ time.We can proof the first 2 cases by using the first 3 hypotheses.The second case must be shorter than third case,So We can proof the third case by axiom **Repeat-node-not-shortest**.

Floyd-warshall-lemma

```
Fixpoint Floyd_warshall_lemma (n:nat)(i j: node n)(g: Graph n)(l : list (node n)) :
  NoNegCyc g -> MinDistance ((Floyd_warshall (adj_Matrix g) l) i j) i j g l.
```

The **Floyd-warshall-lemma** says that Floyd-warshall algorithm can correctly return minimum

distance from i to j by any intermediate node list l.The proof of it is using induction.The base case and inductive case have been proven.We just need to simply apply them to proof the lemma.

Floyd-warshall-Correctness

```
Theorem Floyd_warshall_Correctness:
forall (n:nat)(i j: node n)(g: Graph n), NoNegCyc g ->
MinDistance ((Floyd_warshall (adj_Matrix g) (NodeLst n)) i j) i j g (NodeLst n).
```

Here is one of the main theorem in this paper.It says that if a graph has no negative cycle,if we use the intermediate node list **(NodeLst n)**(all nodes in the graph),Floyd-warshall algorithm will give the minimum distance from any 2 nodes by **(NodeLst n)**.The proof is easy,we just need to apply **Floyd-warshall-lemma**.
Notice:after proving this theorem,we still can not say Floyd-warshall algorithm solves the all pairs of shortest path problem,although **(NodeLst n)** considers all nodes as intermediate nodes.Let's see the theorem describe the APSP problem.
Floyd-warshall-Solved-APSP

```
Theorem  Floyd_warshall_Solved_APSP:
forall (n:nat)(i j: node n)(g p: Graph n)(l:list (node n)),
(1 <= n)%nat -> NoNegCyc g
-> (ValidPath p g i j l)
-> ((Floyd_warshall (adj_Matrix g) (NodeLst n)) i j) <= PathLength p.
```

Floyd-warshall-Solved-APSP theorem says that if a graph has at least one node and has no negative cycle.The distance from i to j return by the Floyd-warshall algorithm is less than any path from i to j.The reader might ask why this definition can consider all valid path? The reason is a path is defined by an intermediate node list.And the theorem consider all the intermediate node lists,so it will includes all the path.In order to proof it,we need to introduce some new theorems.So we will come back to the proof later.

sublist

```
Fixpoint sublist {n :nat} (l:list (node n))(v:list (node n)) : Prop :=
match l with
| nil => True
| h :: t => (In h v) /\ (sublist t v)
end.
```

sublist l v is a proposition which says if l is a sub list of v,every element of l must be in v or l is an empty list.For example [0;1] is a sub list of [0;1;2] and [0;0;0;0] is also a sub list of [0;1;2].

NodeLst-include-All-list

```
Theorem NodeLst_include_All_list:
```

```
forall (n:nat)(l :list (node n)),
(1 <= n)%nat  -> sublist l (NodeLst n).
```

NodeLst-include-All-list says any node list has type **list (node n)**,is the sub list of **NodeLst n**.The proof is simple,we just need to use axiom **NodeLst-Correctness** which is defined in data structure part.

path-still-valid

```
Axiom path_still_valid:
forall (n:nat)(i j :node n)(l v :list (node n))(g p:Graph n),
 (1 <= n)%nat
 -> sublist l v
 -> ValidPath p g i j l
 -> ValidPath p g i j v.
```

Axiom **path-still-valid** says if path p is a valid path by intermediate node list l and l is a sublist of $v.p$ is also a valid path for intermediate node list including v.

proof of Floyd-warshall-Solved-APSP

```
Theorem  Floyd_warshall_Solved_APSP:
forall (n:nat)(i j: node n)(g p: Graph n)(l:list (node n)),
(1 <= n)%nat -> NoNegCyc g
-> (ValidPath p g i j l)
-> ((Floyd_warshall (adj_Matrix g) (NodeLst n)) i j) <= PathLength p.
```

Let's come back to the proof of final theorem in this paper **Floyd-warshall-Solved-APSP**.By apply theorem **NodeLst-include-All-list** and **path-still-valid**.We will get the result that any paths in the graph is a valid path by using **NodeLst n** as intermediate node list.Finally,by applying **Floyd-warshall-Correctness** we can complete the whole proof.

9 Conclusion

In this paper,I build some important data types and predicates to formalize Floyd-Warshall algorithm and by using some auxiliary axioms, Floyd-Warshall algorithm has been verfied in Coq.

References

[1] Samah WG AbuSalim, Rosziati Ibrahim, Mohd Zainuri Saringat, Sapiee Jamel, and Jahari Abdul Wahab. Comparative analysis between dijkstra and bellman-ford algorithms in shortest

path optimization. In *IOP Conference Series: Materials Science and Engineering*, volume 917, page 012077. IOP Publishing, 2020.

[2] Andrew W. Appel. *Verified Functional Algorithms*, volume 3 of *Software Foundations*. Electronic textbook, 2022. Version 1.5.2, http://softwarefoundations.cis.upenn.edu.

[3] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[4] Dieter Jungnickel and D Jungnickel. *Graphs, networks and algorithms*, volume 3. Springer, 2005.

[5] Tobias Nipkow, Manuel Eberl, and Maximilian PL Haslbeck. Verified textbook algorithms. In *International Symposium on Automated Technology for Verification and Analysis*, pages 25–53. Springer, 2020.

[6] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*, volume 1 of *Software Foundations*. Electronic textbook, 2022. Version 6.2, http://softwarefoundations.cis.upenn.edu.

Appendix

oZleb-reflect

```
Theorem oZleb_reflect : forall (x y:optionZ), reflect (x <= y) (x <=? y).
Proof.
  intros x y. apply iff_reflect.
  split.
  +  intro E. destruct E.
    - destruct n.
      * reflexivity .
      * reflexivity .
    - simpl. apply Z.leb_le. assumption.
  +
    - intro E.  destruct x.
      Focus 2.  destruct y.
      *  apply oZle_1inf.
      Focus 2. apply oZle_nm. simpl in E. apply Z.leb_le. assumption.
      * destruct y. apply oZle_1inf. simpl in E. congruence.
Qed.
```

oZmin-Correctness

```
Theorem oZmin_Correctness: forall (x y : optionZ),oZmin x y <= x /\ oZmin x y <= y.
Proof.
```

```
  intros. split.
  + unfold oZmin. assert (R: reflect (x <= y) (x <=? y)).
    apply oZleb_reflect. destruct R.
    - destruct x. apply  oZle_1inf. apply oZle_nm. lia.
    -  destruct x.
      *  destruct y. apply oZle_1inf. apply oZle_1inf.
      * destruct y.
          **assert (R: Some z <= inf). apply  oZle_1inf. congruence.
          **  apply oZle_nm. unfold not in n.
              assert (H1: not(z <= z0)%Z ). unfold not. intros.
              assert (H2:  Some z <= Some z0 ). apply oZle_nm. assumption.
              apply n. assumption. lia.
  + unfold oZmin. assert (R: reflect (x <= y) (x <=? y)).
  apply oZleb_reflect. destruct R.
  - destruct y. apply  oZle_1inf.  assumption.
  - destruct y. apply oZle_1inf. apply oZle_nm. lia.
  Qed.
```

ValidPathE1

```
Example ValidPathE1:
ValidPath [Edge (@F1 2)  (FS(@F1 1)) (Some 1);
           Edge (FS(@F1 1)) (FS (FS (@F1 0))) (Some 2)]
           mygraph (@F1 2) (FS (FS (@F1 0))) [(FS(@F1 1))].

Proof.
unfold ValidPath.

split.
+ apply IsPath1.
  - apply IsPath0.
  - simpl. reflexivity.
+ split.
  - simpl. split. left. left. reflexivity.
    split. left. right. right. left. reflexivity.
    reflexivity.
  - split.
    * simpl. split. reflexivity. reflexivity.
  * simpl. split. left. reflexivity.
reflexivity.
Qed.
```

Cyc-is-zero

```
Lemma Cyc_is_zero:
forall (n:nat) (g p:Graph n) (i:node n),
```

```
NoNegCyc g ->
MinDistance (Some 0) i i g [].
Proof.
intros.
exists [Edge i i (Some 0)].
split.
{
  split.
  {
    apply IsPath0.
  }
  split.
  {
    simpl. split. right. right. reflexivity. reflexivity.
  }
  {
    simpl. split. apply FromI2JE0. reflexivity.
  }

}
split.
{
  simpl. reflexivity.
}
intros. apply H in H0. assumption.
Qed.
```

Base-Case-Correctness

```
Theorem Base_Case_Correctness:
forall (n:nat)(i j: node n)(g: Graph n),
NoNegCyc g->
MinDistance ((adj_Matrix g) i j) i j g nil.

Proof.
intros. unfold adj_Matrix. fold (if eqb i j
then Some 0
else Find_weight i j g inf).
assert (R: reflect (i = j) (eqb i j) ).
apply  iff_reflect. symmetry. apply eqb_eq. destruct R.
{

rewrite e. rewrite Find_weight_a1.  apply Cyc_is_zero. assumption. assumption.
}

  unfold MinDistance. exists [Edge i j ((adj_Matrix g) i j)]. split.
  {
```

```
    split. apply IsPath0.
    split.
    apply Base_Case_Correctness_l1.

    split.
    destruct i. simpl. auto. simpl. auto. simpl. auto.

  }
  split.
  {
    simpl.
    assert (R: reflect (i = j) (eqb i j) ).
    apply  iff_reflect. symmetry. apply eqb_eq. destruct R.
    unfold adj_Matrix.  apply oZleb_Add_zero.
    unfold adj_Matrix. apply oZleb_Add_zero.
  }
  {
    intros.
    assert ((In (Edge  i j (PathLength p')) g ) \/ (PathLength p' = inf)) .
    apply in_a1. assumption. assumption.
    assert ( PathLength p' = trd (Edge i j (PathLength p'))). simpl. reflexivity.
    destruct H1.
    {
      rewrite H2.
      apply Find_weight_a2. assumption. reflexivity.  reflexivity.

    }
    {
      rewrite H1.
      destruct (Find_weight i j g inf).
      assert (R: reflect (inf <= inf) (inf <=? inf) ).
      apply oZleb_reflect.  apply reflect_iff in R. apply R.
      auto.
      assert (R: reflect (Some z <= inf) (Some z <=? inf) ).
      apply oZleb_reflect.  apply reflect_iff in R. apply R. auto.

    }
  }
  Qed.
```

Inductive-Case-Correctness

```
Theorem Inductive_Case_Correctness:
forall (n:nat)(i j a: node n)(g: Graph n)(l : list (node n)),
 NoNegCyc g
 -> (MinDistance (Floyd_warshall (adj_Matrix g) l i j) i j g l)
 -> (MinDistance (Floyd_warshall (adj_Matrix g) l i a) i a g l)
```

```
 -> (MinDistance (Floyd_warshall (adj_Matrix g) l a j) a j g l)
 -> MinDistance (Floyd_warshall (adj_Matrix g) (a::l) i j) i j g (a::l).
Proof.
 intros.
 destruct H0. destruct H1. destruct H2.
 unfold MinDistance.
 exists (SelectSP x (x0 ++ x1)).
 split.
 {
   apply SelectSP_a1.
   destruct H0.
   assumption. destruct H1.  assumption. destruct H2. assumption.
 }
 split.
 {    rewrite Floyd_Warshal_a1.
 unfold SelectSP.
 destruct H0. destruct H3.
 rewrite H3.
 destruct H1. destruct H5.
 rewrite H5.
 destruct H2. destruct H7.
 rewrite  H7.
 assert (R: reflect (PathLength x <= PathLength (x0 ++ x1))
 (PathLength x <=? PathLength (x0 ++ x1))).
  apply oZleb_reflect. destruct R.
  + unfold oZmin.
  assert (R: reflect (PathLength x <= PathLength (x0 ++ x1))
  (PathLength x <=? PathLength (x0 ++ x1))). apply oZleb_reflect.
  rewrite PathLength_a1.  destruct R.
  -
   reflexivity.
  - congruence.
  + unfold oZmin.
  rewrite PathLength_a1.
  assert (R: reflect (PathLength x <= PathLength (x0 ++ x1))
  (PathLength x <=? PathLength (x0 ++ x1))). apply oZleb_reflect. destruct R.
  - congruence.
  - reflexivity.

}
  {
  intros. rewrite Floyd_Warshal_a1.
  apply Three_case_a1 in H3. destruct H3.
  {
    unfold oZmin.
    assert (R: reflect (Floyd_warshall (adj_Matrix g) l i j <=
    Floyd_warshall (adj_Matrix g) l i a +
    Floyd_warshall (adj_Matrix g) l a j)
```

```
(Floyd_warshall (adj_Matrix g) l i j <=?
Floyd_warshall (adj_Matrix g) l i a +
Floyd_warshall (adj_Matrix g) l a j)). apply oZleb_reflect. destruct R.
destruct H0. destruct H4.
{
  apply H5. apply H3.
}
{
  destruct H0. destruct H4.
  apply H5 in H3.
  apply (ozle_a1 (Floyd_warshall (adj_Matrix g) l i a +
  Floyd_warshall (adj_Matrix g) l a j)
  (Floyd_warshall (adj_Matrix g) l i j) (PathLength p')).
  assumption. assumption.
}
}
destruct H3.
{
  destruct H3. destruct H3. destruct H3. destruct H4.
  rewrite H5. Print PathLength_a1.
  rewrite <- PathLength_a1. unfold oZmin.
  assert (R: reflect ( Floyd_warshall (adj_Matrix g) l i j <=
  Floyd_warshall (adj_Matrix g) l i a +
  Floyd_warshall (adj_Matrix g) l a j)
 ( Floyd_warshall (adj_Matrix g) l i j <=?
 Floyd_warshall (adj_Matrix g) l i a +
 Floyd_warshall (adj_Matrix g) l a j)). apply oZleb_reflect. destruct R.
 + destruct H1. destruct H6. apply H7 in H3.
  destruct H2. destruct H8. apply H9 in H4.
  apply (ozle_a2 (Floyd_warshall (adj_Matrix g) l i a) (PathLength x2)
  (Floyd_warshall (adj_Matrix g) l a j)(PathLength x3)
  (Floyd_warshall (adj_Matrix g) l i j)).
  assumption. assumption.  assumption.
  + destruct H1. destruct H6. apply H7 in H3.
  destruct H2. destruct H8. apply H9 in H4.
  apply (ozle_a3 (Floyd_warshall (adj_Matrix g) l i a)(PathLength x2)
  (Floyd_warshall (adj_Matrix g) l a j)(PathLength x3)).
  assumption. assumption.
}
{


  destruct H3. destruct H3. destruct H3.
  destruct H3. destruct H4. destruct H5. rewrite H6.
  apply (Repeat_node_not_shortest n i j a g x2 x3 x4 l).
  assumption.  assumption. assumption. assumption.
```

```
    }

  }

Qed.
```

Floyd-warshall-lemma

```
Fixpoint Floyd_warshall_lemma (n:nat)(i j: node n)(g: Graph n)(l : list (node n)) :
  NoNegCyc g -> MinDistance ((Floyd_warshall (adj_Matrix g) l) i j) i j g l.
Proof.
  intros.
  induction l.
  + simpl. apply Base_Case_Correctness. assumption.
  + apply Inductive_Case_Correctness. assumption. assumption.
  apply Floyd_warshall_lemma . assumption.
  apply Floyd_warshall_lemma. assumption.
Qed.
```

Floyd-warshall-Correctness

```
Theorem Floyd_warshall_Correctness:
forall (n:nat)(i j: node n)(g: Graph n), NoNegCyc g ->
MinDistance ((Floyd_warshall (adj_Matrix g) (NodeLst n)) i j) i j g (NodeLst n).
Proof. intros. apply  Floyd_warshall_lemma. assumption.
Qed.
```

NodeLst-include-All-list

```
Theorem NodeLst_include_All_list:
forall (n:nat)(l :list (node n)),
(1 <= n)%nat  -> sublist l (NodeLst n).
Proof.
intros. induction l.
+ destruct n.
  - simpl.  reflexivity .
  - simpl.  reflexivity .
+ simpl. split.
 Focus 2. assumption.
apply  NodeLst_Correctness. assumption.
Qed.
```

Floyd-warshall-Solved-APSP

```
*)
Theorem  Floyd_warshall_Solved_APSP:

(*
Use:
Floyd_warshall_Correctness.
NodeLst_include_All_list.
path_still_valid
*)
forall (n:nat)(i j: node n)(g p: Graph n)(l:list (node n)),
(1 <= n)%nat -> NoNegCyc g
-> (ValidPath p g i j l)
-> ((Floyd_warshall (adj_Matrix g) (NodeLst n)) i j) <= PathLength p.

Proof.
intros.
assert
(R1: MinDistance ((Floyd_warshall (adj_Matrix g) (NodeLst n)) i j) i j g (NodeLst n)).
apply Floyd_warshall_Correctness.
assert
(R2: sublist l (NodeLst n)). apply  NodeLst_include_All_list. assumption.
assert
(R3:ValidPath p g i j  (NodeLst n)).
apply ( path_still_valid n i j l (NodeLst n) g p).
assumption. assumption. assumption.
assert
(R4:MinDistance(Floyd_warshall(adj_Matrix g)
   (NodeLst n) i j) i j g
(NodeLst n)).
apply Floyd_warshall_Correctness. assumption. assumption.
destruct R1. destruct H2. destruct H3. apply H4.
apply (path_still_valid n i j l (NodeLst n) g p).
assumption. apply (NodeLst_include_All_list n l).
assumption. assumption. Qed.
```