Formalization of Dijkstra's Shortest Path Algorithm

Jingchao Chen

Donghua University
Department of Communication
1882 Yan-an West Road, Shanghai, 200051, P. R. China
chen-jc@dhu.edu.cn

Abstract – This paper uses several functions in the Mizar library as basic operations to formalize Dijkstra's Shortest Path Algorithm, which has important applications in many fields, e.g., the Internet OSPF protocol. Although descriptions of this algorithm abound in the literature, many articles on the topic are short on discussions of the rationale of this algorithm due to its complexity. This paper attempts to adopt a formal approach to discussing the rationale behind the algorithm and gives a number of useful and interesting theorems related to it. Some results appear for the first time in this paper.

Keywords – shortest path, algorithm design and analysis, graph theory, computation models

1. Introduction

Given an edge-weighted graph, a source vertex *s*, and a target vertex *t*, the shortest path problem is to find an *s*-to-*t* path whose total length (cost) is minimum among all *s*-to-*t* paths. There is a wealth of literature on this problem and many solutions to the problem have already been presented. The classical solutions include the algorithms of Dijkstra, Bellman-Ford and Floyd-Warshall, etc. The most representative is Dijkstra's algorithm which is practical and has important applications in many fields, e.g., the Internet OSPF (Open Shortest Path First) protocol. Many textbooks introduce this algorithm, however with respect to its correctness, only the proof in the intuitive sense can be found. It is difficult to present its rigorous proof in natural language. Ref. [3] presents a rigorous proof in the Mizar language. Inspired by this result, the paper presents a formal definition of the algorithm. Because the justification of the correctness of the algorithm is elusive, we do not focus on its justification. Instead, we show several interesting and underlying theorems related to the algorithm. Some of these theorems are helpful for us to understand Dijkstra's algorithm.

2. The Review of Dijkstra's Algorithm

The underlying idea of Dijkstra's algorithm may be described as follows. The algorithm starts with a source *s*. It visits the vertices in order of increasing length and maintains a set *V* of visited vertices whose total length from the source has been computed and a tentative length *L*(*u*) to each unvisited vertex *u*. In fact, *L*(*u*) is the length of the shortest path from the source to *u* in the subgraph induced by $V \cup \{u\}$. Dijkstra's algorithm repeatedly searches the unvisited vertices for the vertex with minimum tentative length, adds it to the set *V* and modifies *L*-values by a procedure

Manuscript received January 31, 2005; revised May 13, 2005.

called Relax. Suppose the unvisited vertex with minimum tentative length is x, the procedure Relax replaces $L(u)$ with **min**$\{L(u), L(u)+\text{length}(x,u)\}$ where u is an unvisited vertex and length(x, u) is the length of edge (x, u). Below we outline this algorithm in PASCAL-like pseudo-code.

```
     Dijkstra_algorithm($G(V,E)$, $s$)
      begin
(1)        for each vertex $v \in V$[G] do
(2)                  $L[v]:=\infty$
(3)                  $P[v]:=NIL$
(4)        $L[s]:=0$
(5)        $S:=\phi$
(6)       while $S \neq V$ do
(7)                find $u_m \in V-S$ such that $L[u_m] = \min\{L[v] \mid v \in V-S\}$
(8)                $S := S+\{u_m\}$
(9)                for each outgoing edge of $u_m$,   $(u_m, v) \in E$ do
                               Relax($u_m$, $v$)
      end
```

```
Relax($u$, $v$)
begin
     if $L(v] > L[u]$+length($u$, $v$) then
             $L(v] := L[u]$+length($u$, $v$)
             $P[v]:= u$
end
```

Lines (1)-(5) initialize the relevant variables. Variable S keeps the set of visited vertices. The set of unvisited vertices can be computed by $V-S$. The initial value of S is empty. For each vertex $v \in V$, we maintain the variable $L[v]$, initially $+\infty$ ($v \neq u$), which is the length of the shortest path from the source s to u in the subgraph visited so far. The path is stored by the variable $P[v]$ which keeps the immediately preceding vertex (called the *predecessor*) of v on the path. At the end of the algorithm, we can recover the path by tracing the variable P starting from t through all intermediate vertices until reaching the source s.

3. Basic Theorems Related to Dijkstra's Algorithm

In an edge-weighted graph, even though there exists a directed path from vertex s to vertex t, we cannot ensure that a shortest s-to-t path exists in the following two cases.

(1) The number of vertices is infinite.
(2) The edge lengths (costs) are negative.

Below we give an example to show that no shortest path exists in case (1). In Fig.1, the set of vertices E equals $\{s, t\} \cup$ NAT, where NAT is the set of natural numbers excluding 0. With respect to the length of each edge, we have length(s, n) = length(n, t) =$1/n$ where n is a positive natural number. Clearly, we cannot find the shortest path from vertex s to vertex t because for $m > n$, we have length(s-n-t) = $2/n > 2/m$ = length(s-m-t), i.e., for any path $s \to n \to t$, we can always find a shorter path, $s \to \underline{m} \to t$.

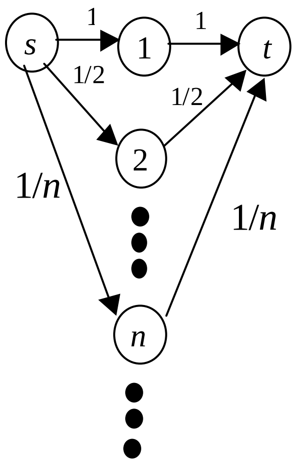

Figure 1. A graph in which there is no s-to-t shortest path.

In case (2), it is easy to find an example in which there is no shortest path. Assume that a graph has two edges where length(s, t) =length(t, s) = -1. Let $(s\text{-}t\text{-}s)^n$ denote n cycles, each of which starts with s, passes through t, and finally reaches s. Then, path $(s\text{-}t\text{-}s)^n\text{-}t$ has length of $-2n-1$. Hence, when $n \to \infty$, it tends towards $-\infty$. In other words, any s-t path with finite length is not a shortest path from s to t.

In this paper, we denote an edge-weighted graph by $G(V, E, W)$, where V is the set of vertices, E is the set of edges interconnecting them, and W is a weight function from E to R^+ (non-negative real numbers), i.e., W: $E \to R^+$. Below we define the cost of a path, the shortest path, and other related concepts.

Definition 3-1. Let $p = v_1 \to v_2 \to, \ldots, \to v_n$ be a directed path of $G(V, E, W)$ and $W(i, i+1)$ be the length of edge (v_i, v_{i+1}). The cost of p is defined as

$$cost(p) = \sum_{i=1}^{n-1} W(i, i+1).$$

Definition 3-2. Let p be a directed path of $G(V, E, W)$, s a source vertex, and t a target vertex. p is a shortest path from s to t *iff* for any directed path q from s to t, we have $cost(p) \le cost(q)$.

Definition 3-3. Let p be a directed path of $G(V, E, W)$, s a source vertex, and t a target vertex. $cost(p)$ is an upper bound on the shortest distance from s to any vertex of U in the subgraph induced by U *iff* for any t in U and any directed s-to-t path q in the subgraph induced by U , we have $cost(q) \le cost(p)$.

Definition 3-4. Let p be a directed path of $G(V, E, W)$ and s a source vertex of p. p is a shortest s-to-t path in the subgraph induced by $U \cup \{t\}$ *iff* for any directed s-to-t path q in the subgraph induced by $U \cup \{t\}$ path, we have $cost(p) \le cost(q)$.

In [2], we formally justified the following theorems. Because the focus of this paper is limited to the discussion of shortest paths, hereafter, without confusion, a graph refers to one that has finite vertices and non-negative edge lengths.

Theorem 3-1. For a graph with finite vertices and non-negative edge lengths, if there exists a directed path from vertex *s* to vertex *t*, then there exists a shortest *s*-to-*t* path.

Theorem 3-2. Given a graph *G*(*V*, *E*, *W*), vertices v_1 and v_2, a subset *U* of *V*, and a directed path *p*, if

(1) $v_1 \neq v_2$;

(2) *p* is a shortest v_1-to-v_2 path in the subgraph induced by $U \cup \{v_2\}$; and

(3) for any directed path *q* and any *v* in *V*, if *u* is not in *U* and *q* is a shortest v_1-to-*u* path in the subgraph induced by $U \cup \{u\}$, then $cost(p) \leq cost(q)$;

then *p* is a shortest v_1-to-v_2 path.

Theorem 3-3. Given a graph *G*(*V*, *E*, *W*), vertices v_1 and v_2, subsets U_1 and U_2 of *V*, and a directed path *p*, if

(1) $v_1 \neq v_2$ and $U_1 \subseteq U_2$;

(2) *p* is a shortest v_1-to-v_2 path in the subgraph induced by $U_1 \cup \{v_2\}$; and

(3) for any directed path *q* and any *v* in *V*, if *u* is not in U_1 and *q* is a shortest v_1-to-*u* path in the subgraph induced by $U_1 \cup \{u\}$, then $cost(p) \leq cost(q)$;

then *p* is a shortest v_1-to-v_2 path in the subgraph induced by $U_2 \cup \{v_2\}$.

Theorem 3-4. Given a graph *G*(*V*, *E*, *W*), vertices v_1, v_2, v_3, a subset *U* of *V*, and directed paths *p*, *q* and *r*, if

(1) $v_1 \neq v_2$ and $v_1 \neq v_3$;

(2) *p* is a non-empty shortest v_1-to-v_2 path in the subgraph induced by $U \cup \{v_2\}$;

(3) *q* is a shortest v_1-to-v_3 path in the subgraph induced by $U \cup \{v_3\}$;

(4) *cost*(*p*) is an upper bound on the shortest distance from v_1 to any vertex of *U* in the subgraph induced by *U*; and

(5) Edge(v_2, v_3) is in *E* and $r = p\hat{}\ v_3$, where $p\hat{}\ v_3$ denotes a path extending *p* to v_3;

then we can conclude:

(A) $cost(q) \leq cost(r)$ implies *q* is a shortest v_1-to-v_3 path in the subgraph induced by $U \cup \{v_3\}$;

(B) $cost(q) \geq cost(r)$ implies *r* is a shortest v_1-to-v_3 path in the subgraph induced by $U \cup \{v_3\}$.

In [3], we formally justified Theorems 3-5 to 3-8 below.

Theorem 3-5. Given a graph *G*(*V*, *E*, *W*), vertices v_1, v_2, v_3, a subset *U* of *V*, and directed paths *p* and *q*, if

(1) $v_1 \neq v_2$ and $v_1 \neq v_3$;

(2) *p* is a shortest v_1-to-v_2 path in the subgraph induced by $U \cup \{v_2\}$;

(3) *q* is a shortest v_1-to-v_3 path in the subgraph induced by $U \cup \{v_3\}$;

(4) Edge(v_2,v_3) is not in *E*; and

(5) *cost*(*p*) is an upper bound on the shortest distance from v_1 to any vertex of *U* in the subgraph induced by *U*;

then *q* is a shortest v_1-to-v_3 path in the subgraph induced by $U \cup \{v_2, v_3\}$.

Theorem 3-6. Given a graph *G*(*V*, *E*, *W*) and vertices v_1 and v_2, if Edge(v_1,v_2) is in *E*, then Edge(v_1,v_2) is a shortest v_1-to-v_2 path in the subgraph induced by $\{v_1, v_2\}$.

Theorem 3-7. Given a graph $G(V, E, W)$, vertices v_1, v_2, v_3, a subset U of V, and a directed path p, if

(1) $v_1 \neq v_3$;
(2) p is a shortest v_1-to-v_2 path in the subgraph induced by $U \cup \{v_2\}$;
(3) Edge(v_2, v_3) is in E; and
(4) for any v in U, Edge(v,v_3) is not in E;

then p^v_3 is a shortest v_1-to-v_3 path in the subgraph induced by $U \cup \{v_2,v_3\}$.

Theorem 3-8. Given a graph $G(V, E, W)$, vertices v_1 and v_2, subsets U_1 and U_2 of V, and a directed path p, if

(1) $V = U_1 \cup U_2$;
(2) v_1 in U_1; and
(3) for any u in U_1 and any v in U_2, Edge(u, v) is not in E;

then p is a shortest v_1-to-v_2 path in the subgraph induced by $U_1 \cup \{v_2\}$ *iff* p is a shortest v_1-to-v_2 path.

4. Data Structure for Dijkstra's Algorithm

As shown in Section 2, the main procedure of Dijkstra's algorithm needs to maintain three kinds of data: a set S of visited vertices whose distances from the source have been computed, a tentative (or final) distance $L(v)$ to each vertex v, and the *predecessor* $P[v]$ of v. Procedure Relax needs to employ the length information of each edge, denoted by length[u,v]. In addition, in line (7), u_m in the formula $L[u_m]$=**min** $\{L[v] \mid v \in V - S\}$ requires a workspace. We use the concept of finite sequences [4] in the Mizar library to store these variables and arrays. Suppose that the set of vertices is the natural numbers from 1 to n and a finite sequence f is of length n^2+3n+1. Fig. 2 shows how to use the sequence f to store the variables and arrays. The first n items are used to store S, i.e., $S[1]=f[1],\ldots,S[n]=f[n]$. The next n items store $P[v]$, i.e., $P[1]=f[n+1],\ldots,P[n]=f[2n]$. Then, L[v] is stored, i.e., $L[1]=f[2n+1],\ldots,L[n]=f[3n]$. Finally, length[$u$,$v$] is stored, i.e., length[1,1]=$f[3n+1]$,…,length[1,n]=$f[4n]$,…,length[n, n]=$f[n^2+3n]$. The last item stores u_m where $L[\underline{u_m}]$=**min** $\{L[v] \mid v \in V - S\}$, i.e., $u_m = f[n^2+3n+1]$.

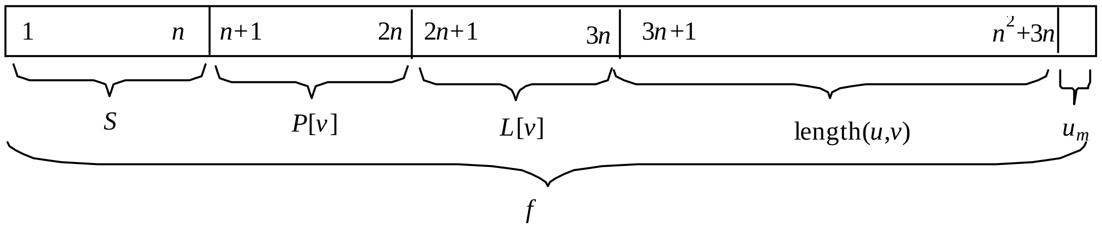

Figure 2. Data structure for Dijkstra's algorithm.

The value of $S[i]$ is set as follows. $S[i] = -1$ if the i-th vertex is visited and $S[i] = 1$ otherwise. The initial value of $P[i]$ is set to -1, except for $P[1]=0$. The initial value of $L[i]$ is set to 0. In the Mizar Library, there is no concept of ∞ (how to formalize ∞ is still an open problem in Mizar). Consequently, when Edge[u, v] is not in E(G), we set length[u, v] $=-1$, instead of $-\infty$, since we assume that when Edge[u, v] is in E(G), length[u, v] ≥ 0. In [3], we initialize the above data structure formally as follows:

Seg n=the Vertices of G &

```
         (for i st 1 <= i & i <= n holds f.i=1 & f.(2*n+i)=0) &
         f.(n+1)=0 & (for i st 2 <= i & i <= n holds f.(n+i)=-1) &
         (for i,j being Vertex of G,k,m st k=i & m=j holds f.(2*n+n*k+m)=Weight(i,j,W))
```

where *i*, *j*, *k*, and *n* are natural numbers.

5. Formalization of Dijkstra's Algorithm

In the Mizar library, there are several computer models, e.g., SCMFSA [5] and SCMPDS [6], etc. However, it is extremely difficult to use these models to formalize Dijkstra's algorithm. On the other hand, these computer models are based on some formalized functions in the Mizar library. Each instruction in the models corresponds to a function which transforms a memory state into another memory state. Assuming that the set of states is S, an instruction can be regarded as $F : S \to S$. In theory, it is possible to implement the formalization of algorithms. In fact, in high-level programming languages, there are also similar examples. For instance, Lisp is a functional programming language. Furthermore, the Mizar library has a wealth of formalized functions. Consequently, we decided to use Mizar functions to formalize Dijkstra's algorithm. The advantage of this formalization is the improvement of readability, since one will see that functions used in this formalization look like pseudo-codes. Below we formalize Dijkstra's algorithm in the Mizar language.

```
definition   let n be Nat;
   func DijkstraAlgorithm(n) -> Element of Funcs(REAL*,REAL*)   equals
           while_do(Relax(n)*findmin(n),n);
end;
```

In this definition, **findmin** corresponds to lines (7) and (8) of Dijkstra_algorithm. **Relax** corresponds to procedure Relax. **REAL*** is the set of finite real sequences, which can be used to denote the set of *f*'s defined in the previous section. Functor **while_do**, which corresponds to the "while do" statement of Pascal, is defined as follows.

```
definition let f be Element of Funcs(REAL*,REAL*), n be Nat;
   func while_do(f, n) -> Element of Funcs(REAL*,REAL*) means
       dom it=REAL* & for h being Element of REAL* holds
         it.h=(repeat f).LifeSpan(f, h, n).h;
end;

definition let X be set, f be Element of Funcs(X,X);
   func repeat(f) -> Function of NAT,Funcs(X,X) means
    it.0 = id X &
    for i being Nat, x being Element of Funcs(X,X) st x = it.i holds   it.(i+1)=f*x;
end;
```

```
 definition
   let f be Element of Funcs(REAL*,REAL*), g be Element of REAL*, n be Nat;
  assume   ex i st OuterVx((repeat f).i.g,n) = {};
  func LifeSpan(f,g,n) -> Nat means
    OuterVx((repeat f).it.g,n) = {} &
    for k being Nat st OuterVx((repeat f).k.g,n) = {} holds it <= k;
end;

definition let f be Element of REAL*, n be Nat;
  func OuterVx(f, n) -> Subset of NAT equals
   {i: i in dom f & 1 <= i & i <= n & f.i <> -1 & f.(n+i) <> -1};
end;
```

Here **OuterVx** denotes the set of unvisited vertices, each of which has an edge connecting it with some already visited vertex. Namely, for $v \in$ **OuterVx**, there exists $u \in S$ such that edge(u, v) $\in E(G)$. The following is the definition of **findmin**.

```
definition let n be Nat;
  func findmin(n)  ->  Element of Funcs(REAL*,REAL*) means
   dom it = REAL* & for f be Element of REAL* holds it.f=
    (f,  n*n+3*n+1) := (Argmin(OuterVx(f,n), f, n), -1);
end;

definition let x,y be set,f be Function;
  func (f,x):=y -> Function means
     dom it = dom f & (for z st z in dom f & z <> x holds it.z=f.z) &
    (x in dom f implies it.x=y);
end;

definition let i,k be Nat,f be FinSequence of REAL,r be Real;
  func (f,i):=(k,r) -> FinSequence of REAL equals
  ((f,i):=k,k):=r;
end;
```

Here functor **:=** is similar to the assignment statement in high-level programming languages. **Argmin** is responsible for finding $u_m \in X$ such that $L[u_m]$= **min** $\{L[v] \mid v \in X\}$, which is defined as follows.

```
definition let X be finite Subset of NAT, f be Element of REAL*,n;
  func Argmin(X, f, n) -> Nat means
    (X<>{} implies ex i st i=it & i in X &
    (for k st k in X holds f/.(2*n+i) <= f/.(2*n+k)) &
    (for k st k in X & f/.(2*n+i) = f/.(2*n+k) holds i <= k)) & (X={} implies it=0);
end;
```

The following is the definition of **Relax.**

```
definition let n be Nat;
  func Relax(n) -> Element of Funcs(REAL*,REAL*) means
   dom it = REAL* & for f be Element of REAL* holds it.f=Relax(f,n);
end;



definition let f be Element of REAL*,n be Nat;
  func Relax(f,n) -> Element of REAL* means
    dom it = dom f &  for k be Nat st k in dom f holds
    (n<k & k <= 2*n implies
          (f hasBetterPathAt n,(k-'n) implies it.k=f/.(n*n+3*n+1)) &
          (not f hasBetterPathAt n,(k-'n)   implies it.k=f.k)) &
    (2*n <k & k <=3*n implies
          (f hasBetterPathAt n,(k-'2*n) implies it.k=newpathcost(f,n,k-'2*n)) &
          (not f hasBetterPathAt n,(k-'2*n)   implies it.k=f.k)) &
    (k<=n or k > 3*n implies it.k=f.k);
end;

definition let n,k be Nat,f be Element of REAL*;
  pred f hasBetterPathAt n,k means
  (f.(n+k)=-1 or f/.(2*n+k) > newpathcost(f,n,k)) &
    f/.(2*n+n*(f/.(n*n+3*n+1))+k) >= 0 & f.k <> -1;
end;
```

It is easy to see that each function above can be implemented by a regular computer. Therefore, the definitions above are of practical significance. Ref. [3] proved the following theorem which shows the meaning of Dijkstra's algorithm.

Theorem 5-1. Given a graph $G(V, E, W)$, vertices v_1, v_2, and a natural number n, let f be a finite real sequence which is initialized according to initial data structure defined in Section 4 (i.e., each item of f is of the value shown in Section 4) and g a finite real sequence which stores the final result. If $V = \{i \mid 1\leq i \leq n\}$, $v_1 = 1$, and $g =$ (DijkstraAlgorithm(n)).f; then

(A) $v_2 \in V$ and $f.v_2 = -1$ (i.e., v_2 is a visited vertex) implies $\exists_p\, p$ is a v_1-to-v_2 shortest path, $cost(p)= g.(2n+v_2)$ and p can be computed by tracing the value stored in g starting from v_2, through all intermediate vertices until reaching the source v_1, i.e., $p[m] = v_2$, $p[m\text{-}1] = g.(n+ p[m])$, $p[m\text{-}2] = g.(n+ p[m\text{-}1])$,…, $p[1] = v_1$, where m is the length of p and $p[i]$ $(1\leq i \leq n)$ is the i-th vertex of p; and

(B) $v_2 \in V$ and $f.v_2 = 1$ (i.e., v_2 is an unvisited vertex) implies not $\exists_p\, p$ is a v_1-to-v_2 directed path.

In this theorem, f can usually be regarded as the input of Dijkstra's algorithm and g the output of Dijkstra's algorithm.

6. Conclusion

Dijkstra's shortest path algorithm is a common and practical algorithm. It is very simple to implement it with C or PASCAL. However, it is difficult to rigorously justify its correctness in natural language. Ref. [3] achieved this goal in the Mizar language. This paper first re-reviewed the algorithm in a Pascal-like pseudo-coded fashion, then re-described the formal definition of the algorithm given in [3]. We also presented several theorems used to prove the correctness of the algorithm. When formalizing algorithms, we found that formalizing the time and space complexity of an algorithm is an extremely difficult task even if it is a very simple algorithm. This will be an open problem.

References

[1] E. W. Dijkstra, *A note on two problems in connexion with graphs*, In Numer. Math.,1,

1959, pp.260-271.

[2] J. C. Chen, Y. Nakamura, *The Underlying Principle of Dijkstra's Shortest Path Algorithm*, Journal of Formalized Mathematics, Vol.15, 2003.

[3] J. C. Chen, *Dijkstra's Shortest Path Algorithm*, Formalized Mathematics, Vol. 11(3), 2003, pp.237-247.

[4] G. Bancerek, K. Hryniewiecki, *Segments of Natural Numbers and Finite Sequences*, Formalized Mathematics, 1(1),1990, pp.107-114.

[5] A. Trybulec, Y. Nakamura, and P. Rudnicki, *An Extension of SCM*, Formalized Mathematics, Vol. 5(4), 1996, pp.507-512, http://mizar.org//JFM/Vol8/scmfsa_3.html.

[6] J. C. Chen, *A Small Computer Model with Push-Down Stack*, Formalized Mathematics, Vol. 8(1), 1999, pp.175-182, http://mizar.org//JFM/Vol11/scmpds_1.html.