Algebraic Programming Language

Sing Kuang Tan

November 15, 2022

Abstract

In this paper, I am going to propose a new programming language in mathematical algebraic form. Represent all algorithms in canonical form, that is easy to read, analyse and communicate with other people. Loop invariant, preconditions, post conditions are difficult to use. It can use to derive all properties of algorithms. It can be fed into computer to analyze and manipulate symbolically. Analyze an algorithm sequentially cannot see global pattern in the algorithms, and this is not a long term solution.

1 Introduction

I have always been interested in programming. I was once labeled as 'Super Programmer' by my colleagues. I have thought of developing my own programming language. But for many years, I did not know what aspect of programming language should I improve on and built into my programming language. Looking at the Python programming language, it has many features that make programming easier, such as garbage collection, matrix vector operations and highly readable codes without comment. My improvement to my programming language should follow the innovation techniques used to improve the Python programming language.

I thought the best programming language is fully mathematical and algebraic, so that complex nested while loops can be represented in 1 line of mathematical equation. Look at how a problem is formulated in convex optimization. It is very human readable and succinct that can represent the problem in a few lines of equations. In the past, the professor wrote the equations and the student wrote the codes to implement the equations. My ideal programming language is the programmer will write out the objective function in mathematical equations, the compiler will optimize the equations and convert it to a step by step algorithm, optimize the time complexity so that it can run in fewer operations (and space complexity by for example making using of the sparse matrix), compile it to machine codes and optimize the machine codes to distribute the execution tasks to multiple processors. From the equations, the compiler can derive the properties of the algorithm for verification and automatic analyze the properties for any faults or poor designs of algorithm. The compiler can rewrite the equation in canonical form (just like linear programming) for storing the equations and making them more readable. The compiler can also do codes obfuscation so that it is difficult to reverse engineer.

There are some existing algebraic approaches to represent an algorithm. In Fibonacci numbers

$$\begin{aligned} &F_0 = 0, F_1 = 1, \\ &F_n = F_{n-1} + F_{n-2}, \\ &\quad \text{for } n > 1, \end{aligned} \tag{1}$$

an algorithm is represented in recursion. An algorithm can also be represented in summation notations,

$$\sum_a \sum_b \sum_c \sum_d f(a,b,c,d) > 0. \tag{2}$$

This algorithm returns a 1 if the left side of the equation is greater than 0, otherwise it is a 0. An algorithm can also be represented in convex optimization,

$$\begin{aligned} &max_x f(x) \\ \text{such that} \\ &g(x) = 0 \\ &h(x) > 0. \end{aligned} \tag{3}$$

This is frequently used in machine learning in computer science to do dimension reduction, manifold learning and clustering. Markov random field, Boolean algebra, linear programming and many other representations can be used to represent an algorithm.

2 A new kind of programming language

An example of an algorithm is shown below (Algorithm 1). This algorithm shows how to implement it. However it is not in it's canonical form. Because we can swap line 7 and 8 of codes and get the same algorithm and result (is shown as Algorithm 2 below). Algorithmic representation has too many pseudo codes that represent same thing.

We need to represent algorithm in canonical form. One way is to use algebra to represent algorithm. Assume there is 4 elements in an array to be sorted. For simplicity, we can assume that all the elements in the array have distinct values. The equations can be easily modified to accommodate to non-distinct element values. Using Discrete Markov Random Field, a potential function can represent whether the l^{th} element is greater than the k^{th} element. $H(a_i = v_k, a_{i+1} = v_l) = 1$ if l^{th} element is greater than the k^{th} element, $H(a_i = v_k, a_{i+1} = v_l) = 0$ otherwise. For example a 4 elements array with items $v_1 = 3, v_2 = 16, v_3 = 11, v_4 = 7$, then $H(a_i = v_2, a_{i+1} = v_4) = 0$, $H(a_i = v_4, a_{i+1} = v_2) = 1$, $H(a_i = v_1, a_{i+1} = v_3) = 1$ and $H(a_i = v_3, a_{i+1} = v_1) = 0$.

Algorithm 1 A BubbleSort algorithm

```
procedure bubbleSort($A$ : list of sortable items)
    repeat
        $swapped \leftarrow False$
        for $i = 1$ to $n - 1$ inclusive do
            if $A[i-1] > A[i]$ then                ▷ if this pair is out of order
                                  ▷ swap them and remember something changed
                $swap(A[i-1], A[i])$
                $swapped \leftarrow True$
            end if
        end for
    until $swapped = False$
end procedure
```

Algorithm 2 A BubbleSort algorithm with 2 lines swapped

```
procedure bubbleSort($A$ : list of sortable items)
    repeat
        $swapped \leftarrow False$
        for $i = 1$ to $n - 1$ inclusive do
            if $A[i-1] > A[i]$ then                ▷ if this pair is out of order
                                  ▷ swap them and remember something changed
                $swapped \leftarrow True$
                $swap(A[i-1], A[i])$
            end if
        end for
    until $swapped = False$
end procedure
```

An equation to represent the algorithm is therefore

$$\sum_{i_1=1}^{4}\sum_{i_2=1}^{4}\sum_{i_3=1}^{4}\sum_{i_4=1}^{4} H(a_1 = v_{i_1}, a_2 = v_{i_2}) H(a_2 = v_{i_2}, a_3 = v_{i_3}) H(a_3 = v_{i_3}, a_4 = v_{i_4}) > 0. \tag{4}$$

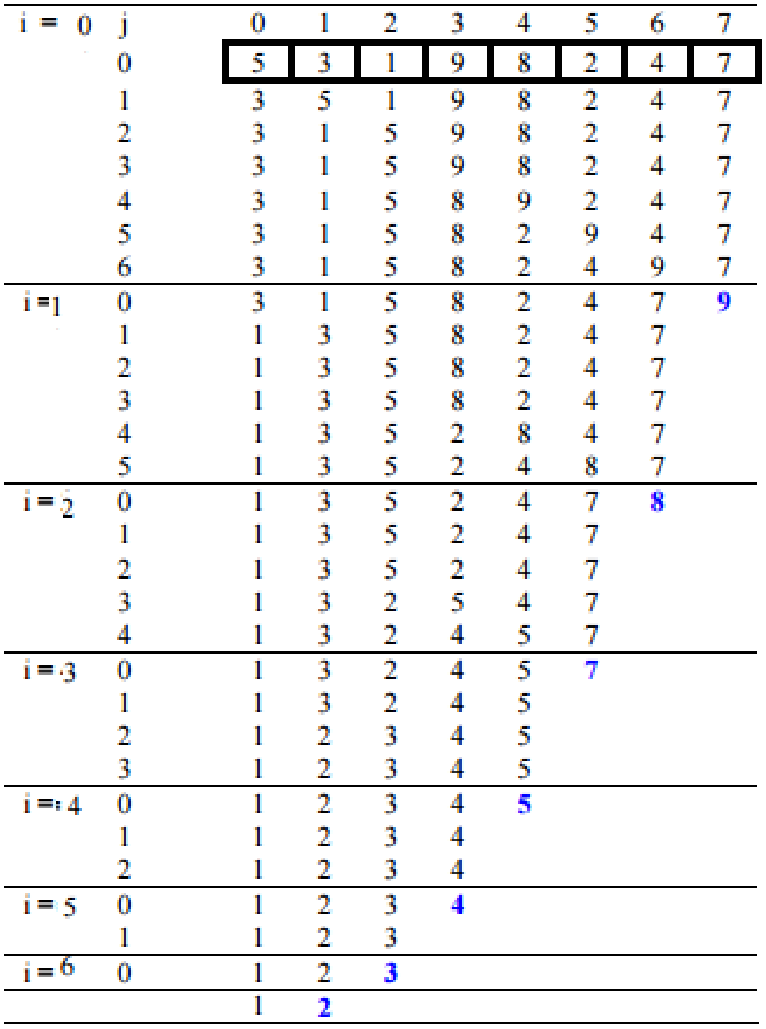
The variables $v_{i_1}, v_{i_2}, v_{i_3}, v_{i_4}$ that make the Markov Random Field greater than zero, represent the solution of the sorting problem. This is the naive way to compute it where all 256 operations (exponential number of computations with respect to 4 inputs). However if we can rewrite this algorithm by factoring the summations, then

$$\sum_{i_1=1}^{4}\sum_{i_2=1}^{4} H(a_1 = v_{i_1}, a_2 = v_{i_2}) \sum_{i_3=1}^{4} H(a_2 = v_{i_2}, a_3 = v_{i_3}) \sum_{i_4=1}^{4} H(a_3 = v_{i_3}, a_4 = v_{i_4}) > 0. \tag{5}$$

becomes a 16×3 operations algorithm (in algorithmic sense, it is n^3 time complexity where n is the number of input).

Using equations to represent algorithm make it easy to read analyze and communicate with other people. Equations can be used to do automatic program verification. Program verification using loop invariant, preconditions, post conditions on sequential algorithm is difficult to use. It can be used to derive all properties of algorithms, can be fed into computer to analyze and manipulate symbolically.

Sequential analysis is not a long term solution and cannot see global pattern in algorithms. An example from the internet shown in figure 1. The algorithm is analyzed sequentially with the change in variable values printed line by line and a next step is dependent on a previous step, which is a time consuming process. If algebraic equations (equations 4 and 5) are used, it can analyze the algorithm out of order, based solely on the constraints of the equations.



i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i = 1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	4				
	1	1	2	3					
i = 6	0	1	2	3					
		1	2						

Figure 1: Sequential analysis of a Bubble Sort algorithm

Representing an algorithm in canonical form has an advantage that it can detect many algorithms are actually the same in the same canonical form. We can borrow concepts (canonical form, sparsity, convergence, convergence rate, numerical stability, independence, eigenvector analysis, function approximation, relaxation, energy function, stochastic matrix, stochastic function and many other terms) from mathematics and use it in computer science. Express an algorithm in canonical form enables us to classify algorithms, and search for new algorithms that are different from existing algorithms that have different canonical form. Algebraic canonical form enables the use of mathematics techniques such as power series to analyze the algorithms.

NP time complexity is usually because the problem is discrete. There are a lot of existing work using algebra on non-discrete problem. Here we concentrate on discrete problem.

3 Algebra is the most Advance Form of Math

Algebra presentation in math is the most advance form math. Algebra represents the constraints and/or objective function of an algorithm, without specifying the steps to solve it, so it is the most general form of representation of algorithm, better than a step by step pseudo codes of an algorithm. Algebra representation has fewer lines of equations than a step by step pseudo codes algorithm.

Logic , language, terminology representation are difficult to understand than algebraic representation. Logic can also be used to represent an algorithm. However the number of lines in the logic is equivalent to the number of statement rules in the logic. For example, the logic statements

$$\begin{aligned} raining &\rightarrow cloudy \\ raining &\rightarrow dark, \end{aligned} \tag{6}$$

which means that when it is raining, the sky will be cloudy and dark. This can be rewritten in Discrete Markov Random Field format,

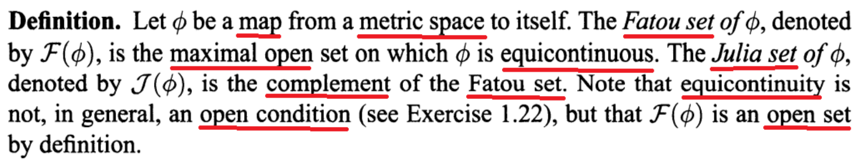
$$\sum_{a \in \{raining, \neg raining\}} \sum_{b \in \{cloudy, \neg cloudy\}} \sum_{c \in \{dark, \neg dark\}} h^{(1)}(a,b) h^{(2)}(a,c) > 0, \tag{7}$$

which is much more succinct and 1 line of equation instead of 2 lines of equations in logic. $h^{(1)}(a,b) = 1$ when (a=raining and b=cloudy) or a=not raining, otherwise $h^{(1)}(a,b) = 0$. $h^{(2)}(a,c) = 1$ when (a=raining and c=dark) or a=not raining, otherwise $h^{(2)}(a,c) = 0$. This Markov Random Field can be simplified to

$$\sum_{a \in \{raining, \neg raining\}} \sum_{b \in \{cloudy, \neg cloudy\}} h^{(1)}(a,b) \sum_{c \in \{dark, \neg dark\}} h^{(2)}(a,c) > 0, \tag{8}$$

which runs in fewer operations than the previous Markov Random Field representation.

The figure 2 shows an example of a mathematical definition.



Definition. Let ϕ be a map from a metric space to itself. The *Fatou set of* ϕ, denoted by $\mathcal{F}(\phi)$, is the maximal open set on which ϕ is equicontinuous. The *Julia set of* ϕ, denoted by $\mathcal{J}(\phi)$, is the complement of the Fatou set. Note that equicontinuity is not, in general, an open condition (see Exercise 1.22), but that $\mathcal{F}(\phi)$ is an open set by definition.

Figure 2: An example of a mathematical definition with many terminologies (underlined)

Instead of using English language to describe a mathematical definition, we can convert it to fully logic or algebraic equations (without any English description) that is more succinct, less ambiguous, more mathematical, more systematic and human readable, easier to communicate between people, and can be algebraically processed for simplification with properties extraction for analysis.

4 Example of How to Represent An Algorithm in Canonical Form

Shortest path problem is a problem in graph theory. Given two vertices, its goal is to find the shortest path between the two vertices such that the sum of all weights of the edges of the shortest path between the two vertices are minimized. Note that a path is a sequence of vertices and the length of the path is the sum of all weights of all edges that join each vertex to the next vertex in the sequence.

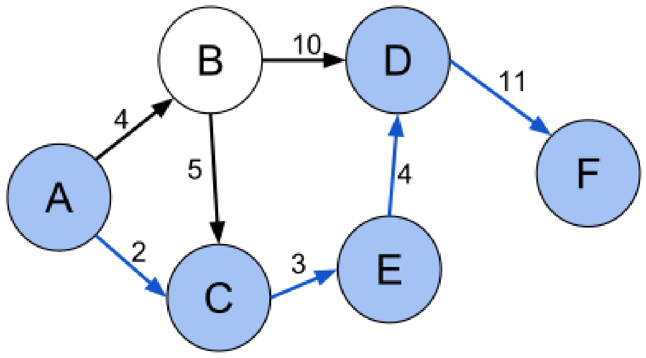


Figure 3: An example of a graph with directed edges and weights. The shortest path in this graph is shown in blue.

The MRF used to represent the shortest path algorithm is shown below.

$$\text{dist}(v_{i_1}, v_{i_4}) \quad = \min_{i_2=1}^{4} H(a_1 = v_{i_1}, a_2 = v_{i_2}) \min_{i_3=1}^{4} H(a_2 = v_{i_2}, a_3 = v_{i_3}) H(a_3 = v_{i_3}, a_4 = v_{i_4}) H(a_4 = v_{i_4}). \tag{9}$$

We assume that this graph has only 4 vertices. The values v_{i_4}, v_{i_3}, v_{i_2} and v_{i_1} of a_4, a_3, a_2 and a_1 represent the shortest path sequence of vertices from origin v_{i_4} to destination v_{i_1}.

The equation

$$H(a_i = k, a_{i+1} = l) = e^{c(k,l)} \tag{10}$$

represents the cost of traversing from vertex k to vertex l (or vertex l to vertex k) where $c(k,l)$ is the edge cost of traversing from vertex k to vertex l. The number e is Euler's number, a constant equals to 2.71828. Note that $c(k,k)$ is 0 if we are traversing from the destination vertex to the destination vertex, otherwise $c(k,k)$ is ∞ of vertex k other than destination vertex because traveling from a vertex to itself is not allowed. To allow traversing from destination vertex to destination vertex so that our algebraic algorithm are able to compute shortest path that are shorter than 4 vertices in length. $c(k,l)$ is ∞ if there does not exist an edge that connects vertex k to vertex l. This equation is valid for $H(a_1,a_2)$, $H(a_2,a_3)$ and $H(a_3,a_4)$.

The equations

$$\begin{aligned} H(a_4 = k) &= e^0 = 1 \text{ if k is the origin vertex} \\ H(a_4 = k) &= e^{\infty} \text{ otherwise} \end{aligned} \tag{11}$$

are to eliminate shortest path solution where the source vertex is not the origin.

The Dijkstra algorithm in summation notation is

$$\text{dist}(v_{i_1}, v_{i_4}) = \min_{i_2=1}^{4}\Big(c(v_{i_1}, v_{i_2}) + \min_{i_3=1}^{4}\big(c(v_{i_2}, v_{i_3}) + c(v_{i_3}, v_{i_4}) + c(v_{i_4})\big)\Big). \tag{12}$$

$c(v_{i_4})$ is $\exp(0)$ if v_{i_4} is the origin vertex, otherwise it is $\exp(\infty)$.

This algebra above can be easily converted to Markov Random Field notation using

$$e^{\text{dist}(v_{i_1}, v_{i_4})} = \min_{i_2=1}^{4}\Big(e^{c(v_{i_1}, v_{i_2})} \min_{i_3=1}^{4}\big(e^{c(v_{i_2}, v_{i_3})} e^{c(v_{i_3}, v_{i_4})} e^{c(v_{i_4})}\big)\Big), \tag{13}$$

which can transformed to

$$e^{\text{dist}(v_{i_1}, v_{i_4})} = e^{\min_{i_2=1}^{4}\Big(c(v_{i_1}, v_{i_2}) + \min_{i_3=1}^{4}\big(c(v_{i_2}, v_{i_3}) + c(v_{i_3}, v_{i_4}) + c(v_{i_4})\big)\Big)} \tag{14}$$

where it is similar to the distance sum form in Equation 12.

$H(a_1 = k, a_2 = k) = H(a_2 = k, a_3 = k) = H(a_3 = k, a_4 = k) = e^0$ if k is the destination vertex of the shortest path algorithm, otherwise it is e^{∞}. This is to ensure the convergence of the shortest path algorithm when the shortest path is 2 or 1 edge away from the origin vertex.

The algorithm 3 below is the Dijkstra shortest path algorithm.

The precondition, loop invariant and post condition can be derived algebraically from the MRF equation.

Since shortest path problem is a minimization problem, dist[v] is initialized to infinity such that dist[v] $>=$ dist(origin, v). Since dist(origin, origin) $= 0$, dist[origin] $=$ dist(origin, origin) $= 0$. These are the dist[v] for the preconditions.

Algorithm 3 A Dijkstra algorithm

```
procedure DIJKSTRA($A$ : list of sortable items)
    for each vertex v in Graph.Vertices do
        $dist[v] \leftarrow INFINITY$    ▷ precondition d[v]=$\infty$ if v is not equal to source
        $prev[v] \leftarrow UNDEFINED$
        add v to Q
    end for
    $dist[source] \leftarrow 0$    ▷ precondition d[source]=0 for source vertex
    while Q is not empty do
        $u \leftarrow$ vertex in Q with min $dist[u]$
        remove u from Q
        for each neighbor v of u still in Q do
            $alt \leftarrow dist[u] + Graph.Edges(u, v)$
            if alt < dist[v] then
                $dist[v] \leftarrow alt$    ▷ loop invariant dist[v]>=dist(source,v)
                $prev[v] \leftarrow u$    ▷ dist(a,b) is the shortest path between vertex a and vertex b
            end if
        end for
    end while
    return dist[], prev[]    ▷ post condition dist[v]=dist(source,v) shortest path is found for all vertices to the origin
end procedure
```

The equation below represents the preconditions.

$$\begin{aligned}\infty &\geq \min_{i_1=1}^{4} \min_{i_2=1}^{4} H(a_1 = v_{i_1}, a_2 = v_{i_2}) \min_{i_3=1}^{4} H(a_2 = v_{i_2}, a_3 = v_{i_3}) \min_{i_4=1}^{4} H(a_3 = v_{i_3}, a_4 = v_{i_4}) H(a_4 = v_{i_4})\\ &\geq \min_{i_2=1}^{4} \min_{i_3=1}^{4} H(a_2 = v_{i_2}, a_3 = v_{i_3}) \min_{i_4=1}^{4} H(a_3 = v_{i_3}, a_4 = v_{i_4}) H(a_4 = v_{i_4})\\ &\geq \min_{i_3=1}^{4} \min_{i_4=1}^{4} H(a_3 = v_{i_3}, a_4 = v_{i_4}) H(a_4 = v_{i_4})\\ &\geq \min_{i_4=1}^{4} H(a_4 = v_{i_4}). \end{aligned} \tag{15}$$

Since $H(a_4 = v_{\text{origin}}) = \exp(0)$ if the a_4 variable value represents the origin vertex,

$$\exp(0) = \min_{i_4=1}^{4} H(a_4 = v_{i_4}). \tag{16}$$

Since during computation of the Dijkstra algorithm, the solution dist[v] is always greater or equal to the solution of shortest distance to vertex v (loop invariant),

$$\exp(\text{dist}[v_{i_1}]) \geq \min_{i_2=1}^{4} H(a_1 = v_{i_1}, a_2 = v_{i_2}) \min_{i_3=1}^{4} H(a_2 = v_{i_2}, a_3 = v_{i_3}) \min_{i_4=1}^{4} H(a_3 = v_{i_3}, a_4 = v_{i_4}) H(a_4 = v_{i_4}). \tag{17}$$

After the Dijkstra algorithm has finished computing its solution, dist[v] is the shortest distance between vertex v and origin vertex (post conditions),

$$\exp(\text{dist}[v_{i_1}]) = \min_{i_2=1}^{4} H(a_1 = v_{i_1}, a_2 = v_{i_2}) \min_{i_3=1}^{4} H(a_2 = v_{i_2}, a_3 = v_{i_3}) \min_{i_4=1}^{4} H(a_3 = v_{i_3}, a_4 = v_{i_4}) H(a_4 = v_{i_4}). \tag{18}$$

So using my algebraic representation of the shortest path algorithm (using MRF equation), we are able to derive the preconditions, loop invariant and post conditions of the Dijkstra algorithm. So my algebraic approach can derive from the objective of the algorithm to the step by step algorithm solution to the problem, then derive the preconditions, loop invariant and post conditions of the algorithm.

The solution of the Dijkstra algorithm can be represented by a spanning tree of the shortest path to the origin.

$$\min_{i_1=1}^{4} \min_{i_2=1}^{4} H(a_1 = v_{i_1}, a_2 = v_{i_2}) \min_{i_3=1}^{4} H(a_2 = v_{i_2}, a_3 = v_{i_3}) \min_{i_4=1}^{4} H(a_3 = v_{i_3}, a_4 = v_{i_4}) H(a_4 = v_{i_4}) > 0 \tag{19}$$

can be rewritten as

$$\begin{aligned}
H^{(1)}(a_1) &= \min_{i_1=1}^{4} \min_{i_2=1}^{4} H(a_1 = v_{i_1}, a_2 = v_{i_2}) H^{(2)}(a_2) \\
H^{(2)}(a_2) &= \min_{i_3=1}^{4} H(a_2 = v_{i_2}, a_3 = v_{i_3}) H^{(3)}(a_3) \\
H^{(3)}(a_3) &= \min_{i_4=1}^{4} H(a_3 = v_{i_3}, a_4 = v_{i_4}) H^{(4)}(a_4 = v_{i_4}) \\
H^{(4)}(a_4) &= H(a_4 = v_{i_4}) \qquad (20)
\end{aligned}$$

where we compute step $H^4(a_4)$ first, followed by $H^3(a_3)$, $H^2(a_2)$ and lastly $H^1(a_1)$. Since $H^i(a_i)$ is computed from the minimum of $H(a_i, a_{i+1})H^{(i+1)}(a_{i+1})$, every value in a_i is connected to 1 value in a_{i+1}. So the shortest path solution to the origin vertex can be represented by a minimum spanning tree to the origin vertex.

The figure below shows that the shortest path to all vertices in a graph from an origin vertex can be represented by a shortest path spanning tree.

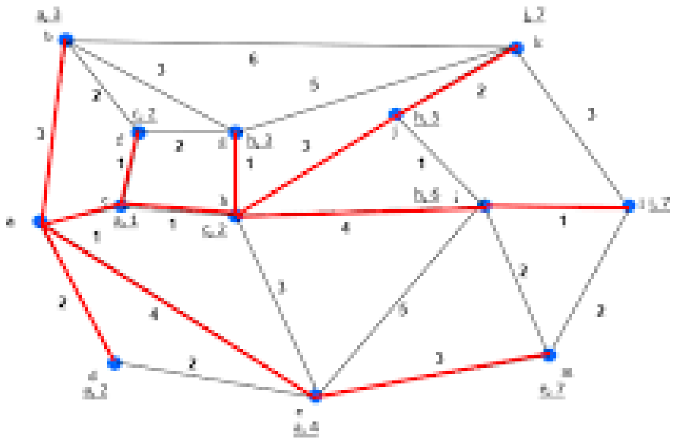


Figure 4: The shortest path to all vertices in a graph from an origin vertex can be represented by a shortest path spanning tree.

This allows very efficient storage of all possible shortest paths from an origin vertex. The storage space is equal to the number of vertices in a graph. Using my algebraic representation of Dijkstra algorithm (using MRF), we are able to prove the property of shortest path minimum spanning tree using algebraic derivation.

5 Represent MRF as Indexing Notation

We can also represent Markov Random Field as indexing notation instead of potential notation. The equations

$$\sum_{i_1=1}^{4}\sum_{i_2=1}^{4}\sum_{i_3=1}^{4}\sum_{i_4=1}^{4} H(a_1 = v_{i_1}, a_2 = v_{i_2})H(a_2 = v_{i_2}, a_3 = v_{i_3})H(a_3 = v_{i_3}, a_4 = v_{i_4}) > 0 \tag{21}$$

and

$$\sum_{i_1=1}^{4}\sum_{i_2=1}^{4} H(a_1 = v_{i_1}, a_2 = v_{i_2}) \sum_{i_3=1}^{4} H(a_2 = v_{i_2}, a_3 = v_{i_3}) \sum_{i_4=1}^{4} H(a_3 = v_{i_3}, a_4 = v_{i_4}) > 0, \tag{22}$$

can be rewritten as

$$\sum_{i_1=1}^{4}\sum_{i_2=1}^{4}\sum_{i_3=1}^{4}\sum_{i_4=1}^{4} h^{(1,2)}_{v_{i_1},v_{i_2}} h^{(2,3)}_{v_{i_2},v_{i_3}} h^{(3,4)}_{v_{i_3},v_{i_4}} > 0 \tag{23}$$

and

$$\sum_{i_1=1}^{4}\sum_{i_2=1}^{4} h^{(1,2)}_{v_{i_1},v_{i_2}} \sum_{i_3=1}^{4} h^{(2,3)}_{v_{i_2},v_{i_3}} \sum_{i_4=1}^{4} h^{(3,4)}_{v_{i_3},v_{i_4}} > 0. \tag{24}$$

This indexing notation is shorter and more readable. Those who are not mathematically inclined will feel very frightened seeing many brackets in the original representation.

6 Conclusion

I have developed a new programming language. This programming language can represent in objective format (goals of the algorithm), then convert to a sequential algebraic form that can be computed by a computer. The properties of the algorithm such as pre-conditions, loop invariant, post conditions and inductive properties can be automatically derived. The user can feed in to the computer the objective format of the algorithm in algebraic form, the computer will analyse the algorithm and generate the sequential algebraic form of the algorithm to compile into machine codes. The properties of the algorithm can also be generated to analyze the correctness of the algorithm. The algorithm can also be rewritten into standard form (canonical form) or more readable form so that it can be recorded into an online user manual that can be reused later.