A Neural Network for Shortest Path Computation

Filipe Araújo
Bernardete Ribeiro
Luís Rodrigues

DI–FCUL TR–00–2

April 6, 2000

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1700 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/biblioteca/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

A Neural Network for Shortest Path Computation

Filipe Araújo
Faculdade de Ciências
Universidade de Lisboa
Campo Grande, 1700 Lisboa, Portugal
filipius@di.fc.ul.pt

Bernardete Ribeiro
Centro de Informática e Sistemas
Universidade de Coimbra
Pinhal de Marrocos, 3030 Coimbra, Portugal
bribeiro@dei.uc.pt

Luís Rodrigues
Faculdade de Ciências
Universidade de Lisboa
Campo Grande, 1700 Lisboa, Portugal
ler@di.fc.ul.pt

April 6, 2000

Abstract

This paper presents a new neural network to solve the shortest path problem for internetwork routing. The proposed solution extends the traditional single-layer recurrent Hopfield architecture introducing a two-layer architecture that automatically guarantees an entire set of constraints held by any valid solution to the shortest path problem. This new method addresses some of the limitations of previous solutions, in particular the lack of reliability in what concerns succeeded and valid convergence. Experimental results show that a clear improvement in well-succeeded convergence can be achieved. Additionally, computation performance is also improved at the expense of slightly worse results.

1 Introduction

The problem of finding the shortest path (SP) from a single source to a single destination in a graph arises as a subproblem to many broader problems, including routing problems in computer networks. This problem has some well-known polynomial algorithmic solutions, namely Bellman-Ford's [2, 4] or Dijkstra's.

Problems that require multiple or extremely fast computations of SP, such as the *quasi-static bifurcated routing problem* in packet switched computer networks [7], can benefit from more efficient methods of finding the SP. Motivated by this class of problems, a number of attempts using neural networks (NN), namely Hopfield NN [3], were made to solve or provide an approximate solution to the SP problem faster than would be possible with any algorithmic solution, relying on the NN parallel architecture.

This idea was first presented by Rauch and Winarske [6]. Their method exhibits some important limitations, one of them being the need to know the number of hops of the SP in advance. To solve this problem an extension to this solution was introduced by Zhang and Thomopoulos [8] which made it possible to find a path with as many as N hops, N being the number of nodes in the graph which is also the highest number of hops the SP may have.

Ali and Kamoun [1] proposed a new method that aimed at NN adaptability to external varying conditions as it is the case of a computer network where links or nodes may go up and down easily. The idea was to apply graph arc costs and respective existence or non-existence in the neurons biases and to change these biases whenever the graph representing the computer network changed. However, this method has two major drawbacks: first, the NN fails to converge towards a valid

solution a considerable number of times and this problem worsens with an increasing number of nodes in the graph. This makes Ali and Kamoun's method nearly useless in certain classes of graphs, when the number of nodes approaches 40; second, the method finds poor solutions when compared to optimum solutions found by Dijkstra's algorithm.

An evolution of the Ali and Kamoun's method, thought for the multi-destination routing problem, was introduced by Park and Choi [5]. When used in a single-destination version it extends the range of operation of the former method, achieving noticeable improved solutions even with a bigger number of nodes. In spite of these advantages, Park and Choi's NN still fails to converge too many times and presents poorer behavior with increasing number of graph nodes in certain classes of graphs. All these solutions demand a number of neurons that squares the number of graph nodes.

This paper presents a new Hopfield NN that aims at improving the reliability of the solutions, where reliability stands for succeeded and valid convergence. To achieve this, a new architecture, named *Dependent Variables* (DV), which consists of a two layer Hopfield NN is presented. This architecture automatically guarantees an entire class of restrictions, thus considerably increasing the reliability of the method. At the same time, the number of neurons is equal to the number of arcs in the graph instead of being equal to the squared number of nodes as it is the case in Ali-Kamoun's and Park-Choi's NN. Thus, in general, the proposed architecture needs much less neurons and neurons' connections. Only in the worst-case scenario where all graph nodes are connected to each other the number of neurons is equal. The price to pay for this reduced number of neurons is that the NN is harder to adapt to external varying conditions, in particular to topology changes, but not to changes in arc costs however, which are, once again, coded in neurons biases.

The paper is organized as follows. Section 2 defines the single source-destination SP problem. The new Dependent Variable Hopfield Neural Network (DVHNN) is decribed in Section 3 and its performance is evaluated in Section 4. Section 5 concludes the paper.

2 Problem Definition

Consider a directed graph $G = (V, A)$ composed of a set of N vertices — V — and a set of M directed arcs — A. Associated with each arc (r, s) is a nonnegative number C_{rs} that stands for the cost from node r to node s. Non-existing arc costs are set to infinite (∞). Often, for clarity of exposition, namely when referring to figures, we will label each existing arc with a unique index and denote its cost simply by C_i.

Let P_{sd} be a path from a source node s to a destination node d, defined as a set of consecutive nodes, connected by arcs in set A:

$$P_{sd} = \{s, n_1, n_2, \ldots, d\}$$

There is a cost associated with each path P_{sd} which consists of the sum of all partial arc costs participating in the path. The shortest path problem consists in finding the path connecting a given source-destination pair, (s, d), such that the cost associated with that path is minimum. Stated as an integer linear programming problem this is (here double indexes are used because (2) is thus easier to state):

$$\text{Minimize} \quad \sum_{i=1}^{N}\sum_{j=1}^{N} C_{ij} v_{ij} \tag{1}$$

$$\text{subject to} \quad \sum_{\substack{j=1 \\ j\neq i \\ (i,j) \text{ exists}}}^{N} v_{ij} \quad - \quad \sum_{\substack{j=1 \\ j\neq i \\ (j,i) \text{ exists}}}^{N} v_{ji} \quad = \phi_i \qquad i = 1, \ldots, N \tag{2}$$

$$\text{and } v_{ij} \in \{0, 1\} \tag{3}$$

- v_{ij} is the participation of the arc (i, j) in the path which can only be 0 or 1, i.e., the arc whether participates entirely or doesn't participate at all in the path. Non-existing arcs are not to be considered.
- $\phi_i = \begin{cases} 1 & \text{if } i = s \\ -1 & \text{if } i = d \\ 0 & \text{otherwise} \end{cases}$

The set of N equations (2) can be stated in matrix form, though only $N - 1$ equations are linearly independent (thus, one of them is omitted). In addition, the double indexes in v_{ij} are replaced by corresponding single indexes variables. The resulting equation is then stated as:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & & & \\ a_{N-1,1} & a_{N-1,2} & \cdots & a_{N-1,M} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_M \end{bmatrix} = \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{N-1} \end{bmatrix} \iff \tag{4}$$

$$\mathbf{A}\mathbf{v} = \phi$$

$$v_i \in \{0, 1\}$$

In this equation, $\mathbf{A}$ is a $(N - 1) \times M$ matrix which depends on graph configuration, ϕ is a vector with $(N - 1)$ elements, which enables path source and destination specification and $\mathbf{v}$ is a vector with M elements, where each one represents the participation of a single arc of the directed graph in the selected path.

The constraints just expressed in (4), that we will call Kirchoff's constraints, are of considerable importance, because they are held by any valid solution to the SP problem (though the reciprocal is not necessarily true).

3 Dependent Variables Hopfield Neural Network

3.1 Kirchoff's Constraints

To build a NN that guarantees constraints (4), this equation is first rewritten to the form presented in (5), using Gauss-Jordan elimination [1].

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_{N-1} \end{bmatrix} = \begin{bmatrix} \alpha_{1N} & \alpha_{1,N+1} & \cdots & \alpha_{1M} \\ \alpha_{2N} & \alpha_{2,N+1} & \cdots & \alpha_{2M} \\ \vdots & & & \\ \alpha_{N-1,N} & \alpha_{N-1,N+1} & \cdots & \alpha_{N-1,M} \end{bmatrix} \begin{bmatrix} v_N \\ v_{N+1} \\ \vdots \\ v_{M-1} \\ v_M \end{bmatrix} + \begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_{N-1} \end{bmatrix} \tag{5}$$

The variables $v_1, \ldots, v_{N-1}$ are dependent variables and will be represented by dependent neurons in the NN to be presented next, while variables $v_N, \ldots, v_M$ are independent variables, that will be represented by independent neurons.

Consider as an example the graph represented in Figure 1: arcs $(1, 3)$, $(2, 4)$, $(2, 5)$ and $(3, 5)$ may be represented by independent neurons while neurons representing arcs $(1, 2)$, $(2, 3)$, $(3, 4)$ and $(3, 5)$ will then depend linearly on former neurons values, for this selection of dependent and independent variables, as shown in (6).

$$\begin{array}{lll} (1) & v_{12} & = 1 - v_{13} \\ (2) & v_{23} & = 1 - v_{13} - v_{24} - v_{25} \\ (3) & v_{34} & = 1 - v_{24} - v_{25} - v_{35} \\ (4) & v_{45} & = 1 - v_{25} - v_{35} \end{array} \tag{6}$$

[1] The computational order of Gauss-Jordan elimination is not important and a method to achieve an equivalent result will be presented ahead in this paper.

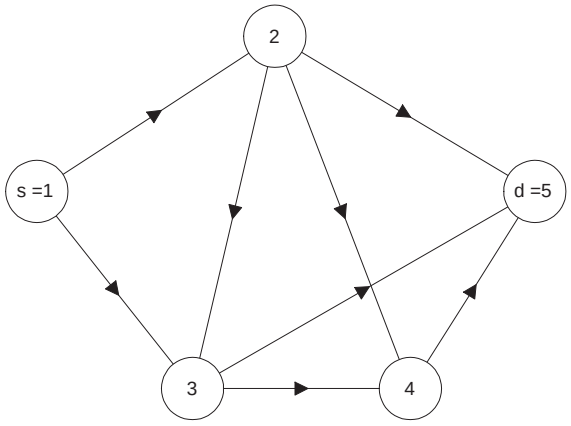

Figure 1: Graph example

3.2 Binary Outputs

Even if Kirchoff's constraints are held, as it is the case of the set of equations (6), it still is necessary to assure that $v_i \in \{0, 1\}$, for $i = 1, \ldots, M$. For that, a Liapunov energy function is defined for the NN:

$$E = \sum_{i=1}^{M} \rho_1 f(v_i) \tag{7}$$

In (7), ρ_1 is a positive constant and $f(x)$ is a function with zeros at $x = 0$ and $x = 1$. In (8) a function with these properties is presented and in addition to this, it is differentiable, which is rather important as will be seen shortly. $f'(x)$ is presented in (9). These functions are depicted in Figures 2 and 3. The energy function (7) will be minimum when all the v_i's have binary values of 0 or 1.

If (4) is met and all the variables, v_i, have binary output values, then the solution is valid, though not necessarily optimal.

$$f(x) = (x-0)^2 (x-1)^2 = x^4 - 2x^3 + x^2 \tag{8}$$

$$f'(x) = 4x^3 - 6x^2 + 2x \tag{9}$$

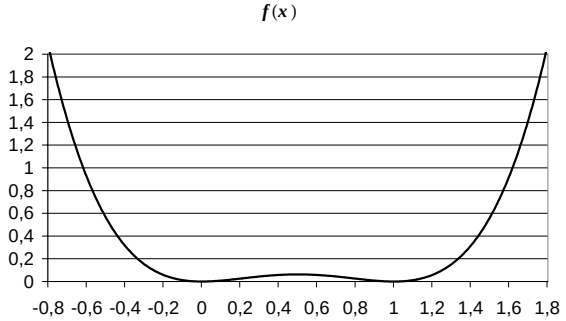

Figure 2: Neuron energy function

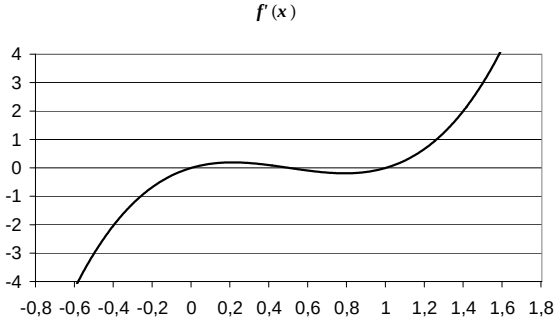

Figure 3: Neuron activation function (energy function derivative)

The dynamic properties of the DVHNN that make it capable of finding the shortest paths will be presented next.

3.3 Neuron Motion Equation

Only the motion equation of independent neurons (v_i, with $N \leq i \leq M$) needs to be defined since dependent neurons have a totally conditioned behavior. The neuron to be used is comprised of a summation unit and a capacitor, as depicted in Figure 4. The corresponding electrical equation is expressed in (10) and transformed to (11) without any loss of generality.

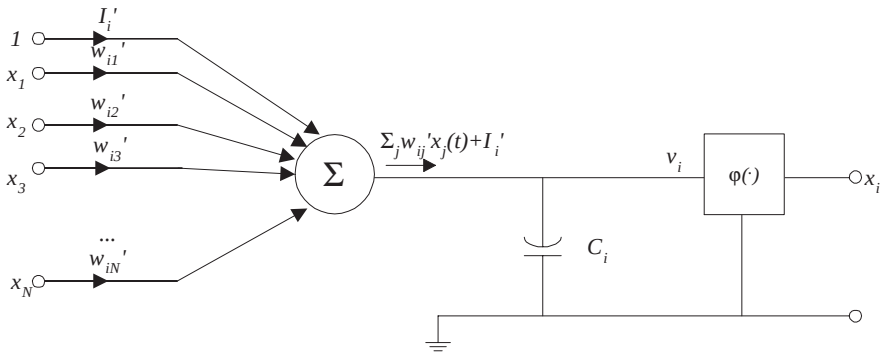

Figure 4: Neuron model

$$C_i \frac{dv_i(t)}{dt} = \sum_{j=1}^{N} w'_{ij} x_j(t) + I'_i \tag{10}$$

$$\frac{dv_i(t)}{dt} = \sum_{j=1}^{N} w_{ij} x_j(t) + I_i \tag{11}$$

For the NN to follow a gradient-descent of the energy function, the neurons' motion equation is defined as in an Hopfield NN, except for the fact that, in (12), no activation function is considered. In this equation k is a positive constant.

$$\frac{dv_i}{dt} = -k \frac{\partial E}{\partial v_i} \tag{12}$$

If (12) holds, then

$$\frac{dE}{dt} = \sum_{i=N}^{M} \frac{\partial E}{\partial v_i} \frac{dv_i}{dt} = \\ = -k \sum_{i=N}^{M} \left(\frac{\partial E}{\partial v_i} \right)^2 \leq 0$$

and as $E \geq 0$ the DVHNN should evolve to an energy minimum (though, this does not guarantee by itself that the NN may not cycle between equivalent energy states with $E > 0$).

Now, calculating $\partial E / \partial v_i$ and from the fact that $\partial v_j / \partial v_i = 0$, $i, j = N, \ldots, M$ (i.e., v_i, v_j are both independent neurons) and $i \neq j$:

$$\frac{\partial E}{\partial v_i} = \rho_1 \sum_{j=1}^{M} \frac{\partial f(v_j)}{\partial v_i}$$
$$= \rho_1 f'(v_i) + \rho_1 \sum_{j=1}^{N-1} f'(v_j) \frac{\partial v_j}{\partial v_i}$$

Since $\partial v_j / \partial v_i = \alpha_{ji}$, for $j = 1, \ldots, N$; $i = N, \ldots, M$, (i.e., v_j is a dependent neuron and v_i independent) it yields (13) and then (14) where $\mu_1 = k\rho_1$.

$$\frac{\partial E}{\partial v_i} = \rho_1 f'(v_i) + \rho_1 \sum_{j=1}^{N-1} \alpha_{ji} f'(v_j) \tag{13}$$

$$\frac{dv_i(t)}{dt} = -\mu_1 f'(v_i(t)) - \mu_1 \sum_{j=1}^{N-1} \alpha_{ji} f'(v_j(t)) \tag{14}$$

Comparing (11) with (14) it yields, for any neuron l and for an independent neuron i:

$$x_l = f'(v_l) \tag{15}$$

$$w_{il} = \begin{cases} -\mu_1 & \text{if } l = i \\ 0 & \text{if } l \neq i,\ l = N, \ldots, M \\ -\mu_1 \alpha_{li} & \text{if } l = 1, \ldots, N \end{cases} \tag{16}$$

$$I_i = 0 \tag{17}$$

A relevant fact that stems from this result is that feedback to the NN is done through independent neurons with neuron outputs affected by individual derivative functions $f'(x)$ which play the role of activation functions.

For the graph presented in Figure 1 the DVHNN depicted in Figure 5 would be used, where independent neurons are in the bottom, while dependent neurons are on the top. Recurrent connections (top to down in the picture) are all affected by the activation function $f'(x)$.

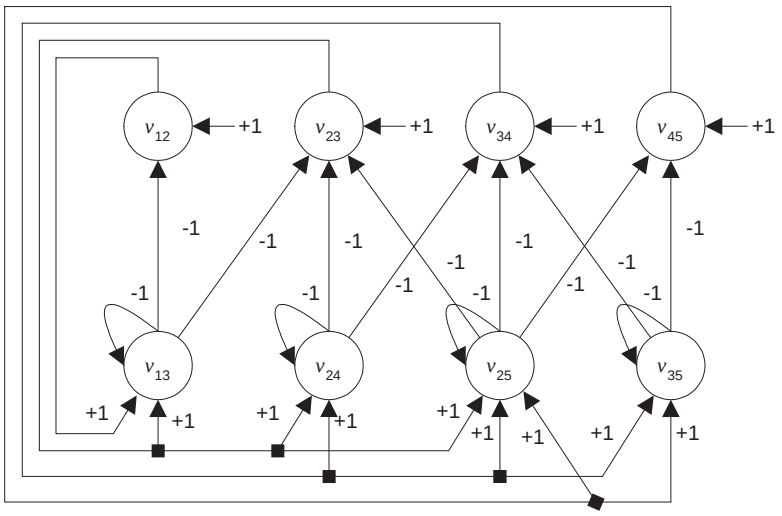

Figure 5: Example of Dependent Variables Hopfield Neural Network

3.4 Cost Consideration

The NN presented until this point considers only solution validity but ignores arc costs. In order to take arc costs into consideration, the following method is used:

- change the energy function to consider the costs;
- start NN and let it converge to a final result which may not be valid but that considers arc costs;
- from this point of convergence, eliminate the costs from energy function and let NN converge again, this time to a valid solution.

Cost consideration is easy to implement because costs will be introduced in neurons biases so it is easy to consider or not the costs just by controlling neurons biases.

The energy function presented in (18) differs from (7) in the term affected by ρ_2. C_i stands for the cost of arc i (here a sequential index number is given to each one of the arcs of the graph unlike (1) where a source and destination node identified the arc).

$$E = \sum_{i=1}^{M} \left(\rho_1 f\left(v_i\right) + \rho_2 C_i v_i \right) \tag{18}$$

Using exactly the same reasoning presented in the previous section it yields:

$$\begin{aligned} \frac{dv_i}{dt} = & -\mu_1 f'(v_i) - \mu_1 \sum_{j=1}^{N-1} \alpha_{ji} f'(v_j) - \\ & - \mu_2 C_i - \mu_2 \sum_{j=1}^{N-1} \alpha_{ji} C_j \end{aligned} \tag{19}$$

From comparing (19) and (11), equations (15) and (16) still hold, but equation (17) must be replaced by (20), for $i = N, \ldots, M$.

$$I_i = -\mu_2 C_i - \mu_2 \sum_{j=1}^{N-1} \alpha_{ji} C_j \tag{20}$$

So, it is very simple to consider or not the costs: just make I_i as in (20) or $I_i = 0$, respectively, as in (17).

It is important to note that dependent variables cost is thus coded in independent variables which are the only neurons that can vary freely.

3.5 NN Convergence

One aspect that affects the convergence of the algorithm is the existence of equivalent cost paths. These may lead the NN to an indefinite final state where it cannot decide for one or the other. To solve this, some random noise is added to the independent neurons biases. If b was the bias to apply to some neuron, the real bias should be randomly selected in the interval $[(1 - \gamma/2)\, b, (1 + \gamma/2)\, b]$ around b. The experimental results shown in section 4 consider $\gamma = 5\%$. This procedure enables symmetry breaking.

Alternative techniques that first consider and then eliminate costs can also be used to improve convergence. For instance, independent neurons biases could be halved each time the NN converges, instead of being totally eliminated as it was proposed before in this text. This corresponds to an exponential reduction and would have the same effect as reducing the constant μ_2, used above to calculate the biases. Although more accurate, this method consumes much more time, because instead of converging only twice, the NN must converge a predefined number of times, say

N times (this however does not mean that the total calculation is N times slower than a single convergence). Thus, here too, there is a trade-off between convergence time and solution quality.

In the experimental results presented in this paper, the first method described in this section (eliminate the biases at once) is used.

3.6 Dependent Variable Selection

As stated before, to find an equation in form (5) as needed by a Dependent Variable NN, it is not efficient to use Gauss-Jordan elimination each time a new SP problem between different source-destination pair arises.

To solve the problem, equation (5) is first extended to let dependent variables depend on other dependent variables. This still enables NN to converge as long as there are not closed cycles dependencies (as we shall insure below), i.e., given dependent variables v_A and v_B, v_A cannot depend on v_B if v_B previously depended on v_A. Note that v_A is also said to depend on v_B if v_A depends on v_C and v_C depends on v_B, being this definition recurrent. (22) shows dependent variable v_A depending on dependent variable v_B defined in (21).

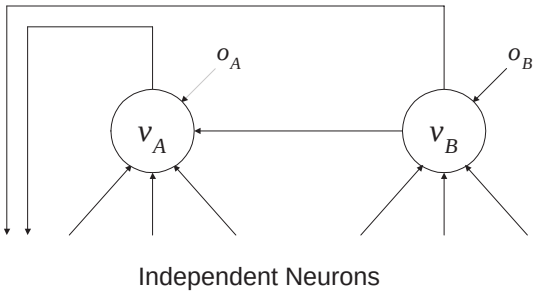

Figure 6: Dependent variable v_A depends on another dependent variable, v_B

$$v_B = \sum_{k=N}^{M} \alpha_{Bk} v_k + o_B \tag{21}$$

$$v_A = \alpha_{AB} \left(\sum_{k=N}^{M} \alpha_{Bk} v_k + o_B \right) + \sum_{k=N}^{M} \alpha_{Ak} v_k + o_A \tag{22}$$

The connection between these two neurons is represented in Figure 6. The advantage of doing this is that the o_i's, from (5) and represented again in (21) and (22) (o_A and o_B) should correspond to ϕ_i in (2) thus making it possible to switch source-destination pair instantaneously. How to do this will be seen shortly.

To find all the dependent variable equations such as (22) (without any cycles between DV) an algorithm is presented next. The algorithm keeps three sets of nodes: T — treated nodes; V — neighbors of treated nodes; N — all other nodes. In the beginning T and V are empty and N contains all the nodes. The algorithm is presented in Figure 7.

In the end, all nodes will belong to T, though all but the first one which enters there directly must pass in V before entering T. When a node, say s, passes from V to T, the respective restriction equation (2) is used and a dependent variable is chosen from this equation. A node enters V whenever one of its neighbors enters T. So, for s to be in V, there must be at least one neighbor, say r which is already in T.

Then, the DV to choose is exactly rs if it exists or sr if not (one of them must exist, otherwise r and s would not be neighbors). Nor rs, neither sr may participate in any other equation to be written in a further step of the algorithm since s is added to T and r has been already added to

```
Initialize set N to contain all the nodes
Initialize two empty sets: V (neighbors) and T (treated)
For  number of nodes - 1 times
    if T = ∅ then
        Select any node s from N
        N ← N\{s} ; T ← T ∪ {s}
        Forall n; n is neighbor of s
            N ← N\{n}; V ← V ∪ {n}
    else
        if V = ∅ then
            Stop (with an error if N is not empty)
        else
            Select any node s from V
            Select r: r is neighbor of s and r ∈ T
            if arc rs exists then
                rs is the dependent variable
            else
                sr is the dependent variable
            Write the restriction equation of node s taking into consideration the source
                and destination of the path
            Forall n; n is neighbor of s
                N ← N\{n}; V ← V ∪ {n}
            V ← V\{s}; T ← T ∪ {s}
```

Figure 7: Algorithm that determines equations representing graph restrictions

T in a previous step. Now, rs/sr (which one was used doesn't matter) may depend on another dependent variable to be chosen ahead but not on a dependent variable already chosen and the same applies to each one of the other dependent variables. Thus cycles cannot occur.

With the algorithm presented above, the o_i's are easily determined to be equal to ϕ_i, for any node i (0, if the path is nor started neither finished in the node, 1 if the path starts in the node, -1 if it ends there). This is possible because all the node equations are written directly without any algebraic transformation.

4 Experimental Results

In this section an attempt is made to determine the behavior of the DVHNN when compared with other methods of finding SP, namely with Dijstra's algorithm and with Park and Choi's NN (PCNN) [5].

The three algorithms were repeatedly applyed to two 40-node graphs. The two graphs have exactly the same configuration. The first graph has arcs with different costs, ranging from 4 to 48. Table 1 represents existing arcs and respective costs for Graph 1. Graph 2 has the same arcs, but their costs are all set equal to 1. For each graph, 40 random source-destination pairs were applied to each one of the three routing methods (DVHNN, PCNN, Dijkstra's algorithm)[2].

To run the experiments, some numeric constant values had to be assigned. The Park and Choi NN constants where assigned with the values proposed by its authors [5]. The respective energy function is restated here for self containment:

[2]Specifically, the following pairs were chosen for the simulations: (10-32), (39-16), (21-17), (34-15), (33-14), (27-28), (1-29), (2-35), (29-34), (22-33), (1-8), (37-34), (22-8), (0-31), (10-12), (32-35), (0-24), (2-22), (3-35), (7-34), (35-6), (20-14), (34-38), (11-6), (3-19), (3-18), (11-23), (29-0), (0-17), (14-22), (39-37), (22-17), (22-32), (4-39), (39-5), (10-7), (19-6), (31-30), (10-21), (7-32).

Table 1: Graph 1 Topology

Arc (s, d)	Cost	Arc (s, d)	Cost	Arc (s, d)	Cost
(0, 33)	18	(0, 24)	24	(0, 19)	16
(0, 9)	28	(1, 33)	28	(2, 25)	12
(2, 12)	22	(3, 27)	32	(3, 14)	25
(3, 4)	17	(5, 39)	24	(5, 26)	4
(5, 18)	18	(5, 6)	16	(6, 20)	25
(6, 11)	16	(7, 26)	26	(8, 20)	22
(8, 17)	18	(9, 39)	22	(9, 36)	35
(9, 31)	14	(9, 25)	18	(9, 23)	20
(9, 13)	48	(10, 39)	21	(10, 30)	4
(10, 26)	19	(12, 29)	8	(12, 25)	19
(13, 19)	12	(14, 28)	22	(14, 18)	9
(15, 36)	26	(16, 31)	16	(16, 22)	23
(16, 21)	16	(17, 22)	27	(19, 36)	12
(19, 32)	16	(21, 31)	31	(23, 37)	22
(23, 36)	37	(23, 26)	19	(26, 34)	15
(28, 39)	31	(29, 35)	34	(29, 31)	16
(30, 38)	34	(34, 39)	32	(38, 39)	37

$$
\begin{aligned}
E =& \frac{A}{2} \sum_{m=1}^{n} \sum_{\substack{i=1 \\ i \neq m}}^{n} C_{mi} x_{mi} + \frac{B}{2} \sum_{m=1}^{n} \sum_{\substack{i=1 \\ i \neq m}}^{n} \rho_{mi} x_{mi} + \\
& \frac{C}{2} \sum_{m=1}^{n} \left\{ \sum_{\substack{i=1 \\ i \neq m}}^{n} x_{mi} - \sum_{\substack{i=1 \\ i \neq m}}^{n} x_{im} - \phi_m \right\}^2 + \frac{D}{2} \sum_{m=1}^{n} \sum_{\substack{i=1 \\ i \neq m}}^{n} x_{mi} \left(1 - x_{mi}\right) + \\
& \frac{F}{2} \sum_{m=1}^{n} \sum_{\substack{i=1 \\ i \neq m}}^{n} x_{mi} x_{im}
\end{aligned}
$$

The constants are $A = 550$, $B = 2550$, $C = 2150$, $D = 250$ and $F = 1350$.

In the DVHNN the constant values assigned where $\mu_1 = 500$ and $\mu_2 = 80$. These values mean that more emphasis is put on the convergence success (bigger μ_1) rather than on solution quality (smaller μ_2). Increasing μ_1 and μ_2 proportionally, results in faster convergence but worse solutions.

To simulate by software the two NN the fourth-order Runge-Kutta method was applied as described in [1] and [5]. The time step considered was $\Delta t = 10^{-5}$s for PCNN and $\Delta t = 10^{-5}$s for DVHNN in graph 1 and $\Delta t = 10^{-6}$ for DVHNN in graph 2. This NN is very sensitive to this parameter and may diverge if the step is chosen too high. This problem would not, of course, occur in a real physical system.

To improve the results achieved by both NN, graph costs were bounded between 0 and 2 for DVHNN and between 0 and 0.6 for PCNN. To do that, a linear transformation (23) was applied to each arc cost, with f defined as C_{MN}/C_M, being C_{MN} the desired maximum arc cost and C_M the existing maximum arc cost. $cost_N$ is the new normalized cost, while $cost$ is simply the original arc cost.

$$cost_N = f \times cost \tag{23}$$

Symmetry breaking was introduced in DVHNN in the neurons biases representing the costs,

as explained in Section 3.5. In the PCNN symmetry breaking was introduced in neurons initial activity, randomly set between 0 and 0.0002.

The experimental results are shown in Table 2. A failure is a try (i.e., a source-destination pair) for which the method is unable to find a solution. As should be apparent, this case just happens with NN, but never with Dijkstra's algorithm (unless no path exists), when the NN gets locked in a local minimum which doesn't lead to a valid solution whether optimum or not.

Table 2: Simulation results

Graph 1			
	Dijkstra	PC	DV
Total cost (average)	2603 (72.3)	2613 (72.6)	2702 (75.1)
Failures	0	4	0
Graph 2			
	Dijkstra	PC	DV
Total cost (average)	94 (3.1)	94 (3.1)	100 (3.3)
Failures	0	10	0

As can be seen in graph 1 results, the PCNN performs slightly better (3% to 6% smaller costs) than DVHNN, whenever it reaches a succeeded convergence, but DVHNN successfully converges many more times, proving itself to be a much more reliable solution, achieving 100% reliability with these graphs.

An interesting fact is that the PCNN performs worse with equal arc costs (failing 25% of the times) while the DVHNN performs better.

Regarding the number of iterations needed, DVHNN shows the best results, needing only 22% of the iterations, which means, 22% of the time in Graph 1 and 15% of the time in Graph 2[3], as can be seen in table 3.

Table 3: Number of iterations and time average

	PC	DV
Iterations avg. (graph 1)	7489	1684
Iterations avg. (graph2)	8485	1286
Time avg. (graph 1)	74.89 ms	16.84 ms
Time avg. (graph 2)	84.85 ms	12.86 ms

5 Conclusions

A new method to solve the shortest path problem was proposed using a two-layer Hopfield Neural Network. This solution aims to achieve an increased number of succeeded and valid convergences, which is one of the main limitations of previous solutions based on Neural Networks. Additionally, in general, it requires less neurons.

The experimental results show that the main goal of the architecture is accomplished making it almost totally reliable (i.e., it achieves succeeded and valid convergences almost always) while considerably improving computational performance at the expense of the results, which are, only slightly worse.

Two open issues deserve further work: first, the convergence to sub-optimal results; second, the adaptability to external varying conditions, in particular, to different graph topologies, which may not be trivial to achieve with the proposed architecture.

[3]Note that an iteration with step $\Delta t = 10^{-5}$ is worth for 10 iterations with step $\Delta t = 10^{-6}$.

References

[1] Mustafa K. Mehmet Ali and Faouzi Kamoun. Neural networks for shortest path computation and routing in computer networks. *IEEE Transactions on Neural Networks*, 4(5):941–953, November 1993.

[2] R.E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, N.J., 1957.

[3] J.J. Hopfield and D.W. Tank. "Neural" computation of decisions in optimization problems. *Biological Cybernetics*, pages 533–541, 1986.

[4] L.R. Ford Jr. and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, N.J., 1962.

[5] Dong-Chul Park and Seung-Eok Choi. A neural network based multi-destination routing algorithm for communication network. *IEEE*, pages 1673–1678, 1998.

[6] Herbert E. Rauch and Theo Winarske. Neural networks for routing communication traffic. *IEEE Control Systems Magazine*, pages 26–31, April 1988.

[7] Mischa Scharwtz. *Telecommunication Networks*. Addison-Wesley Publishing Company, 1987.

[8] L. Zhang and S. C. A. Thomopoulos. Neural network implementation of the shortrest path algorithm for traffic routing in communication networks. In *Proceedings of International Conference Neural Networks*, page 591, 1989.

List of Figures