Dijkstra's Algorithm

Read me first!
This module is designed for 620-261 students as a supplement to Chapter 9 of the lecture notes. There will be a question in the final exam on this topic. The module is a stand-alone, self-contained courseware. Please note that the tutors were instructed not to discuss any aspect of this module with students. All matters regarding this module should be communicated to the lecturer.

Warning:

If you are already familiar with this algorithm make sure that you comply with the rules and conventions deployed by this module. These could be different from the ones you used elsewhere.

1. Introduction

In 1959 a three pages long paper entitled *A Note on Two Problems in Connexion with Graphs* was published in the journal *Numerische Mathematik.* In this paper Edsger W. Dijkstra - then a twenty-nine-year-old computer scientist - proposed algorithms for the solution of two fundamental graph theoretic problems: the *minimum weight spanning tree problem* and the *shortest path problem.* Today Dijkstra's Algorithm for the shortest path problem is one of the most celebrated algorithms in computer science (CS) and a very popular algorithm in operations research (OR).

In the literature this algorithm is often described as a *greedy algorithm.* For example, the book *Algorithmics* (Brassard and Bratley [1988, pp. 87-92]) discusses it in the chapter entitled *Greedy Algorithms.* The *Encyclopedia of Operations Research and Management Science* (Gass and Harris [1996, pp. 166-167]) describes it as a "... node labelling greedy algorithm ... " and a greedy algorithm is described as "... a heuristic algorithm that at every step selects the best choice available at that step without regard to future consequences ..." (Gass and Harris [1996, p. 264]).

Although the algorithm is very popular in the OR/MS literature, it is generally regarded as a "computer science method". Apparently this is due to three factors: (a) its inventor was a computer scientist (b) its association with special data structures, and (c) there are competing OR/MS oriented algorithms for the shortest path problem. It is not surprising therefore that some well established OR/MS textbooks do not even mention this algorithm in their discussions on the shortest path problem (eg. Daellenbach et al [1983], Hillier and Lieberman [1990]) and those that do discuss it in that context present it in a stand-alone mode, that is, they do not relate it to standard OR/MS methods (eg. Markland and Sweigart [1987], Winston [2004]). For the same reason it is not surprising that the "Dijkstra's Algorithm" entry in the *Encyclopedia of Operations Research and Management Science* (Gass and Harris [1996, pp. 166-167]) does not have any reference whatsoever to dynamic programming.

One of the objectives of this module is to present a completely different portrayal of Dijkstra's Algorithm, its origin, its role in OR/MS and CS, and its relationship to other OR/MS methods and techniques. That is, we show that at the outset Dijkstra's Algorithm was inspired by *Bellman's Principle of Optimality* and that, not surprisingly, technically it should be viewed as a *dynamic programming successive approximation procedure.*

2. Shortest path problems

One of the main reasons for the popularity of Dijkstra's Algorithm is that it is one of the most important and useful algorithms available for generating (exact) optimal solutions to a large class of *shortest path problems.* The point being that this class of problems is extremely important theoretically, practically, as well as educationally.

Indeed, it is safe to say that the shortest path problem is one of the most important generic problem in such fields as OR/MS, CS and artificial intelligence (AI). One of the reasons for this is that essentially any combinatorial optimization problem can be formulated as a shortest path problem. Thus, this class of problems is extremely large and includes numerous practical problems that have nothing to do with actual ("genuine") shortest path problems.

New classes of genuine shortest path problem are becoming very important these days in connection with practical applications of Geographic Information Systems (GIS) such as on line computing of driving directions. It is not surprising therefore that, for example, Microsoft has a research project on algorithms for shortest path problems.

What may surprise a bit our students is how late in the history of applied mathematics this generic problem became the topic of extensive research work that ultimately produced efficient solution methods. However, in many respects it is not an accident at all that this generic problem is so intimately connected to the early days of OR/MS and electronic computers. The following quote is very indicative (Moore [1959, p. 292]):

> The problem was first solved in connection with Claude Shannon's maze-solving machine. When this machine was used with a maze which had more than one solution, a visitor asked why it had not built to always find the shortest path. Shannon and I each attempted to find economical methods of doing this by machine. He found several methods suitable for analog computation, and I obtained these algorithms. Months later the applicability of these ideas to practical problems in communication and transportation systems was suggested.

For the purposes of our discussion it is sufficient to consider only the "classical" version of the generic shortest path problem. There are many others.

Consider then the problem consisting of n > 1 cities {1,2,...,n} and a matrix D representing the length of the direct links between the cities, so that D(i,j) denotes the length of the *direct link* connecting city i to city j. This means that there is at most one direct link between any pair of cities. The distances are not assumed to be symmetric, so D(i,j) is not necessarily equal to D(j,i). The objective is to find the shortest path from a given city h, called *home,* to a given city d, called *destination.* The length of a path is assumed to be equal to the *sum* of the lengths of the links between consecutive cities on the path. With no loss of generality we assume that h=1 and d=n. So the basic question is: what is the shortest path from city 1 to city n?

To be able to cope easily with situations where the problem is not feasible (there is no path from city 1 to city n) we deploy the convention that if there is no direct link from city i to city j then D(i,j) is equal to *infinity.* Accordingly, should we conclude that the length of the shortest path from node i to node j is equal to infinity, the implication would be that there is no (feasible) path from node i to node j.

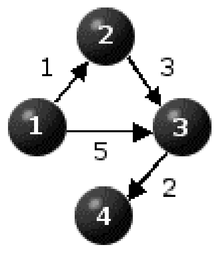
Observe that subject to these conventions, an instance of the shortest path problem is uniquely specified by its distance matrix D. Thus, this matrix can be regarded as a complete model of the problem.

As far as optimal solutions (paths) are concerned, we have to distinguish between three basic situations:

- An optimal solution exists.
- No optimal solution exits because there are no feasible solutions.
- No optimal solution exists because the length of feasible paths from city 1 to city n is unbounded from below.

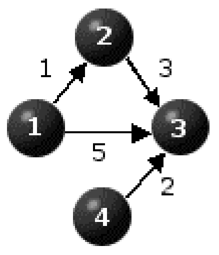
Ideally then, algorithms designed for the solution of shortest path problems should be capable of handling these three cases.

Figure 1 depicts three instances illustrating these cases. The cities are represented by the nodes and the distances are displayed on the directed arcs of the graphs. In all three cases n=4. The respective distance matrices are also provided. The symbol "*" represents infinity so the implication of D(i,j)="*" is that there is no direct link connecting city i to city j.



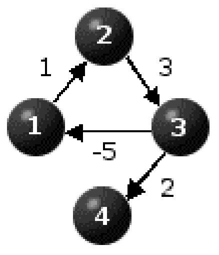
D(i,j)	1	2	3	4
1	*	1	5	*
2	*	*	3	*
3	*	*	*	2
4	*	*	*	*

a



D(i,j)	1	2	3	4
1	*	1	5	*
2	*	*	3	*
3	*	*	*	*
4	*	*	2	*

b



D(i,j)	1	2	3	4
1	*	1	*	*
2	*	*	3	*
3	-5	*	*	2
4	*	*	*	*

c

Figure 1

By inspection we see that the problem depicted in Figure 1(a) has a unique optimal path, that is x=(1,2,3,4), whose length is equal to 6. The problem depicted in Figure 1(b) does not have a feasible - hence optimal - solution. Figure 1(c) depicts a problem where there is no optimal solution because the length of a path from node 1 to node 4 can be made arbitrarily small by cycling through nodes 1,2 and 3. Every additional cycle will decrease the length of the path by 1.

Observe that if we require the feasible paths to be *simple,* namely not to include cycles, then the problem depicted in Figure 1(c) would be bounded. Indeed, it would have a unique optimal path x=(1,2,3,4) whose length is equal to 6. In our discussion we do not impose this condition on the problem formulation, namely we admit cyclic paths as feasible solutions provided that they satisfy the precedence constraints. Thus, x'=(1,2,3,1,2,3,4) and x"=(1,2,3,1,2,3,1,2,3,4) are feasible solutions for the problem depicted in Figure 1(c). This is why in the context of our discussion this problem does not have an optimal solution.

Let C={1,2,...,n} denote the set of cities and for each city j in C let P(j) denote the set of its *immediate predecessors,* and let S(j) denote the set of its *immediate successors,* namely set

P(j) = {k in C: D(k,j) < infinity} , j in C (1)

S(j) = {k in C: D(j,k) < infinity} , j in C (2)

Thus, for the problem depicted in Figure 1(a), P(1)={}, P(2)={1}, P(3)={1,2}, P(4)={3}, S(1)={2,3}, S(2)={3}, S(3)={4}, S(4)={}, where {} denotes the empty set.

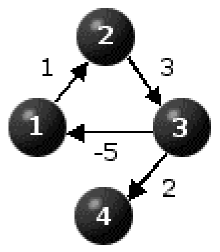
Also, let NP denote the set of cities that have no immediate predecessors, and let NS denote the set of cities that have no immediate successors, that is let

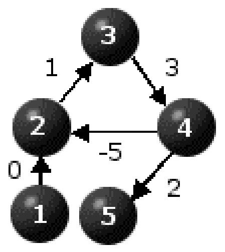
NP = {j in C: P(j) = {}} (3)

NS = {j in C: S(j) = {}} (4)

Thus, in the case of Figure 1(a), NP={1} and NS={4}. Obviously, if city 1 is in NS and/or city n is in NP then the problem is not feasible.

For technical reasons it is convenient to assume that P(1) = {}, namely that city 1 does not have any immediate predecessors. This is a mere formality because if this condition is not satisfied, we can simply introduce a dummy city and connect it to city 1 with a link of length 0. We can then assume that this dummy city - rather than city 1 - is the home city. This minor modelling issue is illustrated in Figure 2.





D(i,j)	1	2	3	4
1	*	1	*	*
2	*	*	3	*
3	-5	*	*	2
4	*	*	*	*

D(i,j)	1	2	3	4	5
1	*	0	*	*	*
2	*	*	1	*	*
3	*	*	*	3	*
4	*	-5	*	*	2
5	*	*	*	*	*

a b

Figure 2

The two problems are equivalent in the sense that the problem of finding an optimal path from city 1 to city 4 in Figure 2(a) is equivalent to the problem of finding an optimal path from city 1 to city 5 in Figure 2(b). There is a one to one correspondence between the feasible - therefore optimal - solutions to these two problems.

So with no loss of generality we assume that P(1)={}.

A variety of methods and algorithms are available for the solution of shortest path problems. In the next section we examine the DP approach to this generic OR/MS problem.

3. DP perspective

DP algorithms are inspired by the famous Bellman's [1957, p. 83] *Principle of Optimality:*

> An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute and optimal policy with regard to the state resulting from the first decision.

Guided by this principle, the first thing we do is generalize our shortest path problem (find an optimal path from city 1 to city n) by embedding it in a family of related problems (find an optimal path from city 1 to city j, for j=1,2,3,...,n). So in a typical dynamic programming manner we let

f(j) := length of the shortest path from node 1 to node j, , j=1,2,3,...,n. (5)

It is important to stress that our objective is to determine the value of f(n). The values of {f(j), j=1,2,...,n-1} are introduced not necessarily because we are interested in them, but first and foremost because this is the way dynamic programming works: in order to determine the value of f(n) it might be necessary to determine the values of f(j) for j=1,2,...,n-1 as well.

In any case, using the above definition of f(j), the following is one of the immediate implications of the principle in the context of the short path problem:

Corollary 1

f(j) = D(k,j) + f(k) , for some city k in P(j) (6)

for any city j such that P(j) != {}, where != denotes "not equal to".

Of course, algorithmically speaking, this result is not completely satisfactory because it does not identify precisely the value of k for which (6) holds. It merely guarantees that such a k exists and that it is an element of set P(j).

However, conceptually this is just a minor obstacle. After all, f(j) denotes the *shortest distance* from node 1 to node j and therefore as such it is obvious that the mysterious k on the right hand side of (6) can be identify by making the right hand side of (6) as small as possible. This leads to:

Corollary 2

f(j) = min {D(k,j) + f(k): k in P(j)} , if P(j) != {}. (!= means "not equal to") (7)

f(j) = Infinity , if P(j) = {} and j > 1. (8)

f(1) = 0 , (We assume that P(1)={}). (9)

This is the *dynamic programming functional equation* for the shortest path problem.

It should be stressed that, here and elsewhere, *the dynamic programming functional equation does **not** constitute an algorithm.* It merely stipulates certain properties that function f defined in (5) must satisfy. Indeed, in the context of Corollary 2 it constitutes a necessary optimality condition. This point is sometime not appreciated by students in their first encounter with dynamic programming. Apparently this is a reflection of the fact that in many 'textbook examples' the description of the algorithm used to solve the functional equation is almost a carbon copy of the equation itself.

Exercise 1 Solve the dynamic programming functional equation associated with the shortest path problem depicted in Figure 3. For your convenience the system of equations is given provided:

f(1) = 0
f(2) = 4 + f(4)

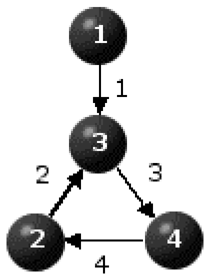


Figure 3

f(3) = min {2 + f(2), 1 + f(1)}
f(4) = 3 + f(3)

Note that the purpose of the exercise is not merely to find a solution to this system. After all, it is obvious (by inspection) that f(1)=0; f(2)=8; f(3)=1;f(4)=3 is a solution. Rather, the purpose is to highlight the fact that the system itself does not constitute a procedure for determining the values of {f(j): j=1,2,3,4} associated with this system.

And this brings us back to the algorithmic aspects of the shortest path problem: how do we solve (7)-(9)? What kind of algorithms are available for this purpose?

In summary, dynamic programming algorithms for the shortest path problem are procedures designed to determine the values of {f(j)} defined in (6) by solving the dynamic programming functional equation (7)-(9).

Remark: It is possible to formulate a linear programming model for the shortest path problem that is very similar to the dynamic programming functional equation (7)-(9). Hint: See Lawler [1976, pp. 78-82].

Our mission being what it is, the first question that comes to mind at this stage is obviously: Is Dijkstra's Algorithm a dynamic programming inspired algorithm? A linear programming inspired algorithm? Or perhaps an algorithm of another type altogether?

The following quote from an IBM advertisement of their OSL package may serve as a clue to the answer:

> Ants don't need Dijkstra's algorithm to find the shortest path to the picnic. They always find their way, selecting the fastest route, overcoming obstacles, maximizing return with minimum effort. Near perfect utilization of resources with maximum benefit; completing herculean tasks with unparalleled cooperation and teamwork. Wouldn't it be nice if your software could do the same?
>
> OR/MS Today, 29(1), p. 37, 2002

As we indicated at the outset, in this module we take the view that Dijkstra's Algorithm is a DP algorithm par excellence. The question therefore arises: what is so special about this algorithm, say compared to the DP algorithm discussed in Chapter 9 of the lecure notes?

The quick answer to this question is that Dijkstra's Algorithm is capable of handling cyclic shortest path problems whereas the algorithm in Chapter 9 cannot. On the other hand, the algorithm in Chapter 9 can handle negative distances, whereas Dijkstra's Algorithm cannot.

With this in mind let us now have a lok at this famous algorithm.

4. Dijkstra's Algorithm

From a purely technical point of view Dijkstra's Algorithm can be described as an iterative procedure inspired by (6) that repeatedly attempts to improve an initial **approximation** {F(j)} of the (exact) values of {f(j)}. The initial approximation is simply F(1)=0 and F(j)=infinity for j=2,...,n. The details are as follows.

Each city (node) is *processed* exactly once according to an order to be specified below. City 1 is processed first. A record (set) is kept of the cities that are yet to be processed, call it U. So initially U = C = {1,...,n}. When city k is processed the following task is performed:

Update F: set F(j) = min{F(j),D(k,j) + F(k)}, for all j in U/\S(k) (10)

where A/\B denotes the intersection of sets A and B. Recall that S(j) denotes the set of immediate successors of city j. Thus, when city k is processed, the {F(j)} values of its immediate successors that have not yet been processed are updated in accordance with (10).

To complete the informal description of the algorithm it is only necessary to specify the order in which the cities are processed. This is not difficult: the next city to be processed is one whose F(j) value is the smallest over all the unprocessed cities:

Update k: k = arg min {F(j): j in U} (11)

recalling that U denotes the set of unprocessed cities. Thus, initially U = {1,...,n} and then after city k is processed it is immediately deleted from U. Note that "arg min {F(j): j in U}" denotes the value of j in U, call it k, such that F(k) = min {F(j): j in U}. That is, the new value of k is an element of U such that F(k) = min {F(j): j in U}.

Formally then

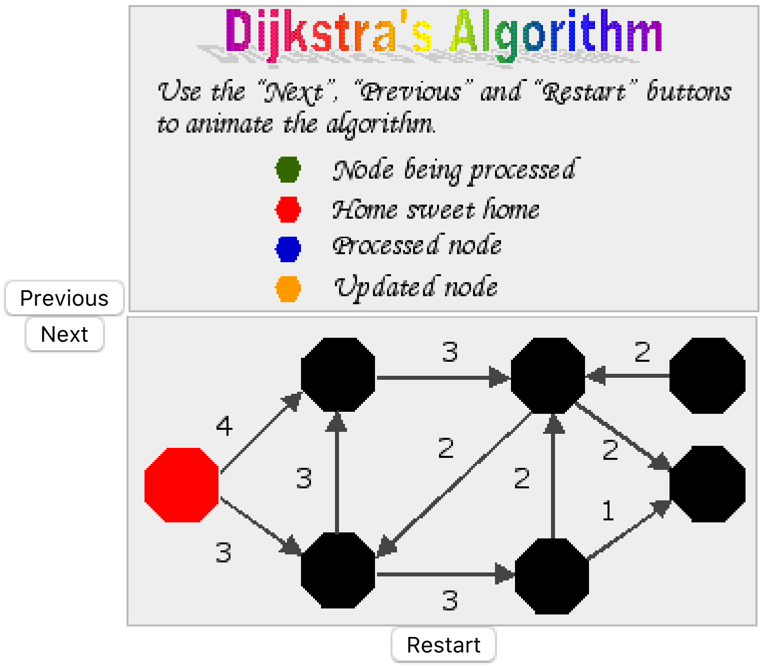
Update U: U = U\{k} (12)

We note in passing that the reason that Dijkstra's Algorithm is regarded as a Greedy method lies in the rule it deploys, (11), to select the next city to be processed: the next city to be processed is the one that is nearest to city 1 among all cities that are yet to be processed.

The stopping rule is simple: Stop when the destination city - in our case city n - is about to be processed. If the objective is to find the shortest path from node 1 to all other cities, then we stop when all the cities have been processed.

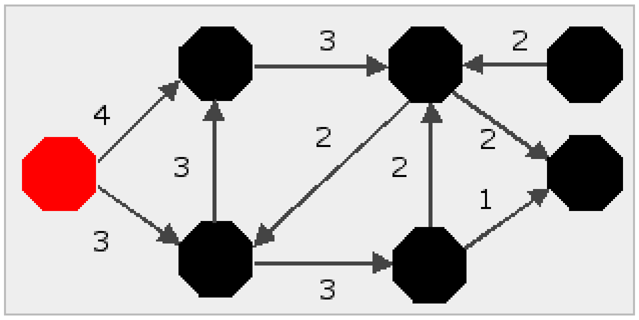
Example 1

The following module was designed to illustrate how the algorithm works. The objective is to find the shortest paths from the red node (origin) to all other nodes. Observe that the updated {F(j)} values are displayed inside the respective nodes. A color scheme is used to indicate which nodes have been processed, which node is being processed and so on. As the nodes are being processed, the optimal arcs are identified and highlighted.

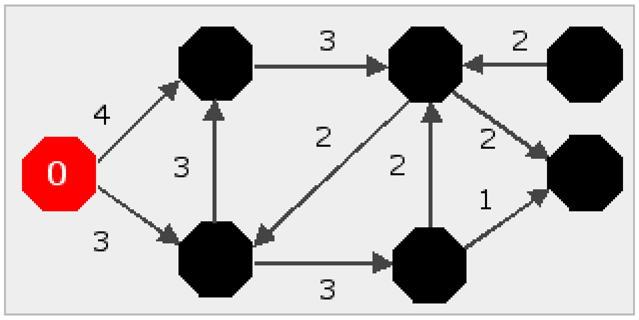


Module 1

A full explanation is provided below. Note that the nodes are not labeled so it is tempting to display the F(j) values on the nodes themselves. If the nodes are labeled, their F(j) values can be displayed nearby, say just above or just below the nodes.

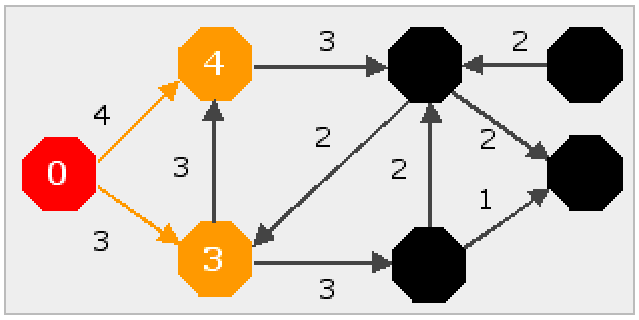


Unprocessed network.

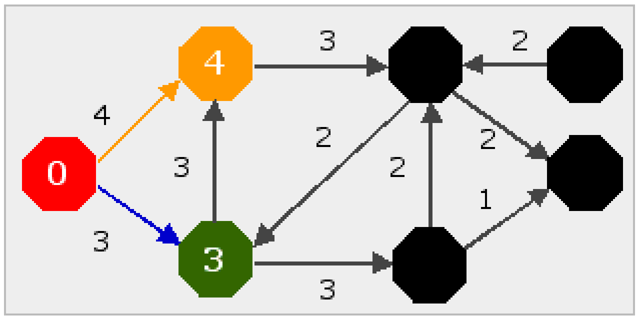


Initialization: U = set of all nodes except the origin, F(origin)=0, F(j)=infinity for all other nodes.

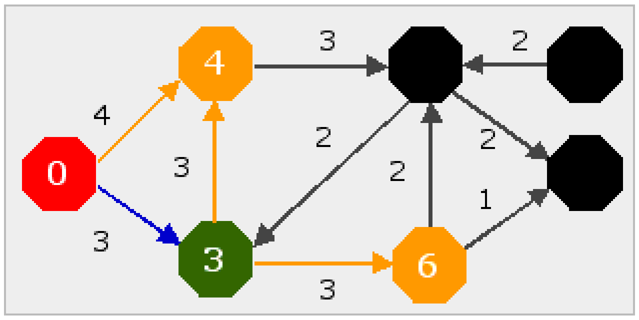
Update F: The origin city is processed



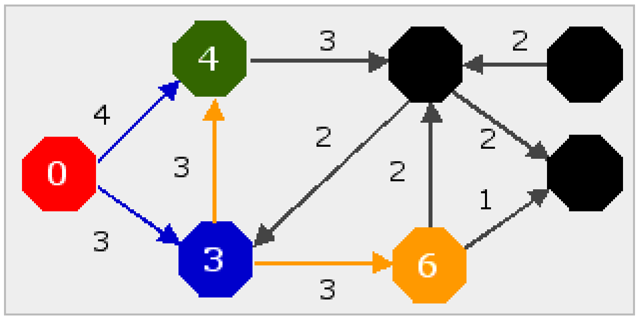
so the F(j) values of its two immediate successors (gold nodes) are updated according to (10).



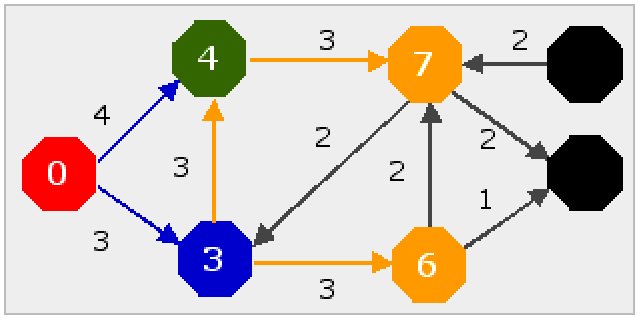
Update k: The next nearest node to the origin (green node) is identified according to (11) for processing.



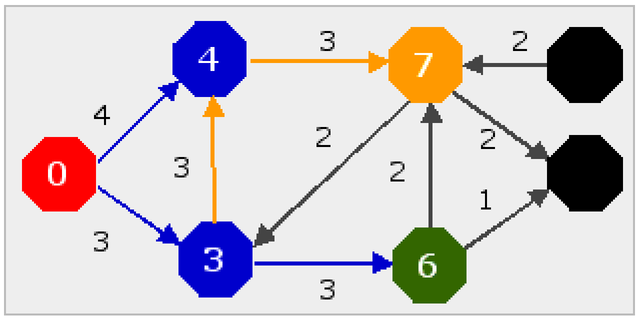
Update F: The green node is being processed, so the F(j) values of its immediate successors (gold nodes) are being updated according to (10).



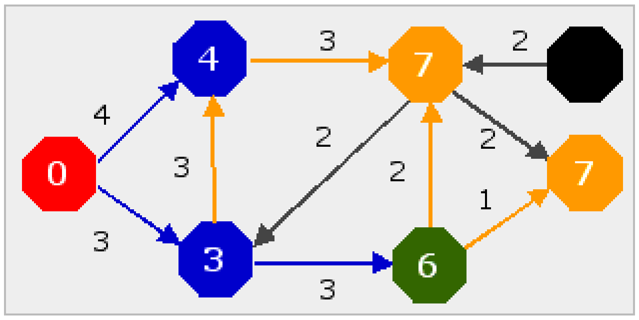
Update k: The next nearest node to the origin (green node) is identified according to (11) for processing.



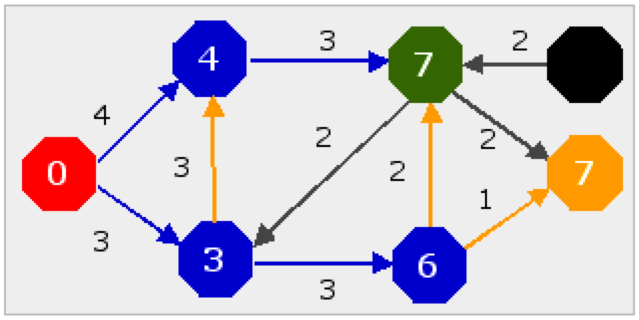
Update F: The green node is being processed, so the F(j) values of its immediate successors (gold nodes) are being updated according to (10).



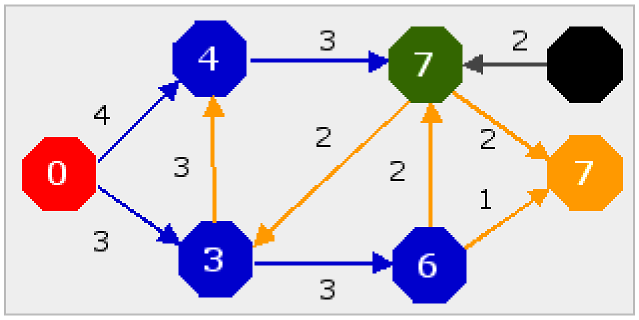
Update k: The next nearest node to the origin (green node) is identified according to (11) for processing.



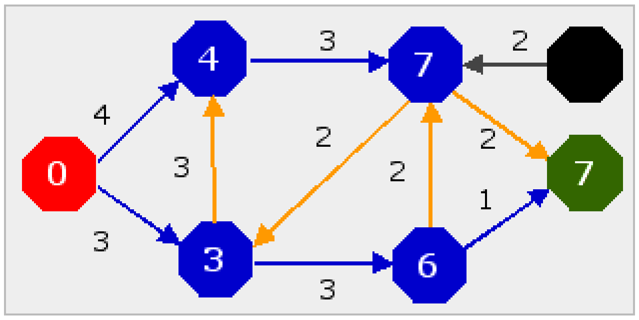
Update F: The green node is being processed, so the F(j) values of its immediate successors (gold nodes) are being updated according to (10).



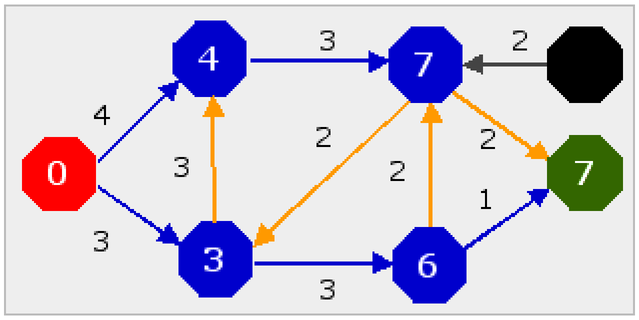
Update k: The next nearest node to the origin (green node) is identified according to (11) for processing.



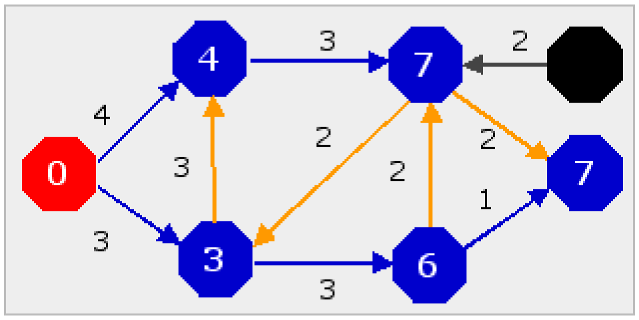
Update F: The green node is being processed, so the F(j) values of its immediate successors (gold nodes) are being updated according to (10). Note that the F(j) value of nodes that have already been processed (blue nodes) do not have to be updated.



Update k: The next nearest node to the origin (green node) is identified according to (11) for processing.



Update F: The green node is being processed, so the F(j) values of its immediate successors (gold nodes) have to be updated according to (10). Note that the F(j) value of nodes that have already been processed (blue nodes) do not have to be updated. So in this case there is nothing to update.



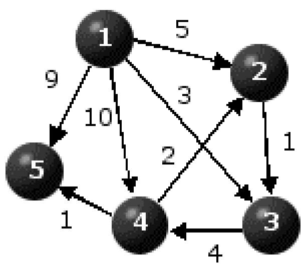
Stopping Rule:
only one node is left unprocessed, so we stop. Since this node has not been updated, it means that it cannot be reached from the origin. Formally its F(j) value is equal to infinity.

Because all the arc lengths are non-negative, upon termination the F(j) values of the nodes are equal to the f(j) values. The blue arcs specify the shortest paths to the nodes.

The following example illustrate how the intermediate results generated by Dijkstra's Algorithm can be displayed in a tabular rather than graphical manner.

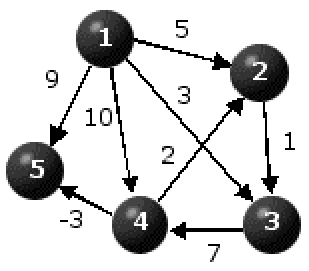
Example 2

Table 1 displays the results generated by Dijkstra's Algorithm for the two problems depicted in Figure 4. The objective is to find the shortest distance from node 1 to node 5. The values in the k, U, F columns are the values of the respective objects at the **end** of the respective iteration. Iteration 0 represents the initialization step. The underlined F(j) entries represent nodes that have already been processed.



D(i,j)	1	2	3	4	5
1	*	5	3	10	9
2	*	*	1	*	*
3	*	*	*	4	*
4	*	2	*	*	1
5	*	*	*	*	*

a



D(i,j)	1	2	3	4	5
1	*	5	3	10	9
2	*	*	1	*	*
3	*	*	*	7	*
4	*	2	*	*	-3
5	*	*	*	*	*

b

Figure 4

Iteration	k	U	F
0	--	{1,2,3,4,5}	(0,*,*,*,*)
1	1	{2,3,4,5}	$(\underline{0},5,3,10,9)$
2	3	{2,4,5}	$(\underline{0},5,\underline{3},7,9)$
3	2	{4,5}	$(\underline{0},\underline{5},\underline{3},7,9)$
4	4	{5}	$(\underline{0},\underline{5},\underline{3},\underline{7},8)$
5	5	{}	$(\underline{0},\underline{5},\underline{3},\underline{7},\underline{8})$

a

Iteration	k	U	F
0	--	{1,2,3,4,5}	(0,*,*,*,*)
1	1	{2,3,4,5}	$(\underline{0},5,3,10,9)$
2	3	{2,4,5}	$(\underline{0},5,\underline{3},10,9)$
3	2	{4,5}	$(\underline{0},\underline{5},\underline{3},10,9)$
4	5	{4}	$(\underline{0},\underline{5},\underline{3},10,\underline{9})$

b

Table 1

Note that in case (b) the algorithm failed to generate an optimal solution: F(n) = F(5) = 9 > f(n) = f(5) = 7. This is due to the fact that there is an arc whose length is negative. In case (a) all the distances are non-negative hence the algorithm generates the optimal solution, thus we have f(1)=0; f(2)=5; f(3)=3; f(4)=7; f(5)=8.

Important Remarks:

1. In the exam it will be up to you to decide what format to use to display the intermediate results (tabular or graphical).

2. In either case, you'll have to indicate clearly what are the f(j) values and what are the shortest paths from the origin to the required destination(s) nodes.

3. Winston's book (on reserve in the Maths Library) has a discussion on Dijkstra's Algorithm.

4. In the Appendix we provide some additional interactive modules related to Dijkstra's Algorithm.

4. The Appendix also includes a more formal description of the algorithm and some facts about it.

Appendix

The following is a more formal description of Dijkstra's Algorithm for the case where there are n cities and the objective is to go from city 1 to city n.

Dijkstra's Algorithm: Version 1
(Determine the value of f(n))
Cycles are welcome, but negative distances are not allowed)

```
Initialization: k=1; F(1) = 0 ; F(j) = Infinity, j=2,...,n ; U = {1,...,n}.
     Iteration: While ((k !=n) and (F(k) < infinity)) Do:
                    Update U: U = U\{k}
                    Update F: F(j) = min {F(j),D(k,j) + F(k)}, j in U/\S(k)
                    Update k: k = arg min {F(j): j in U}
                End Do.
```

This means that the algorithm terminates if either the destination city n is next to be processed (k = n) or F(k) = infinity. The latter implies that at this iteration F(j) = infinity for all cities j that have yet to be processed and the conclusion is therefore that these cities cannot be reached from city 1.

Note that the above formulation of the algorithm does not explain how tours are constructed. It only updates the **length** of tours. The construction of tours can be easily incorporated in this formulation by recording the *best immediate predecessor* of the city being processed. This predecessor is updated on the fly as F(j) is being updated.

The main result concerning this version of the algorithm is as follows:

Theorem 1

Dijkstra's Algorithm terminates after at most n-1 iterations. If the distance matrix D is not negative and the algorithm terminates because k=n, then upon termination F(n)=f(n) and furthermore F(j)=f(j) for all j in C for which f(j)<f(n). If termination occurs because F(k) = infinity, then F(j)=f(j) for all j in C for which f(j) < Infinity and f(j)=infinity for all other cities.

In short, if the inter-city distances are not negative the algorithm yields the desired results.

Sometime it is required to determine the shortest distance from city 1 to every other city j, j=2,3,...,n. In this case the "While (...)" clause in Version 1 of the algorithm can be changed to "While (|U| > 1 and F(k) < Infinity)". This yields the more familiar formulation of the algorithm:

Dijkstra's Algorithm: Version 2
(Determine the value of f(j), j=1,...,n)
Cycles are welcome, but negative distances are not allowed

```
Initialize: F(1) = 0; F(j) = Infinity, j=2,...,n
            U = {1,2,3,...,n}
   Iterate: While (|U| > 1 and F(k) < infinity) Do:
              U = U\{k}
              F(j) = min {F(j),D(k,j) + F(k)}, j in U/\S(k)
              k = arg min {F(i): i in U}
            End Do.
```

You may now wish to experiment with this algorithm using Module 2. Note that the module updates the values of F(j) in two steps. First, the values of G(k,j) = D(k,j) + F(k) are computed for j in U/\S(k) and then the respective values of F(j)=min {F(j),G(k,j)} are computed.

Start | Next ... | Clear | Help

```
Initialize: F(1) = 0; F(j) = Infinity, j=2,...,n
            U = {1,2,3,...,n}
   Iterate: While (|U|>1 & F(k)<infinity) Do:
              U = U\{k}
              G(k,j) = D(k,j) + F(k) , j in U/\S(k)
              F(j) = min {F(j),G(k,j)}
              k = arg min {F(i): i in U}
            End Do.
```

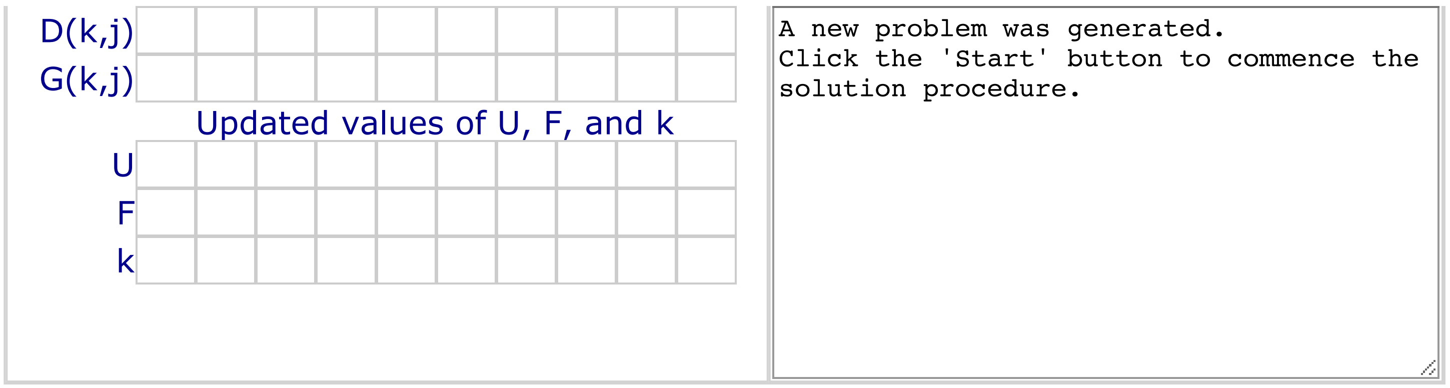
Iteration No.

j	1	2	3	4	5	6	7	8	9	10
k										
U										
F										

Fill-in | Example | Clear

D(i,j)	1	2	3	4	5	6	7	8	9	10
1		2	1	2	2	10	1	8	3	9
2			9	9	5	4	1	4	5	9
3		6		4	8	9	4	8	11	2
4		4	9		6	3	1	9	3	10
5		10	7	1		8	4	4	4	9
6		9	1	6	5		6	4	8	8
7		7	6	2	7	6		6	6	5
8		3	5	5	2	2	6		1	9
9		2	7	6	6	7	8	9		6
10		5	5	5	6	5	10	3	6	

Sparsity | Cyclic



D(k,j)										
G(k,j)										

Updated values of U, F, and k

U										
F										
k										

```
A new problem was generated.
Click the 'Start' button to commence the
solution procedure.
```

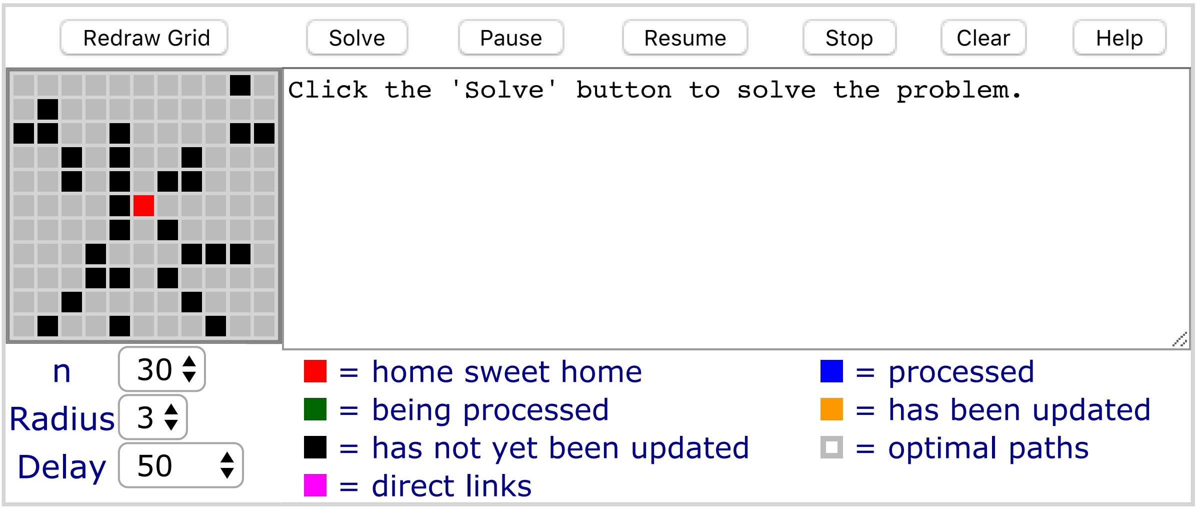
Module 2

Note that the algorithm terminates either when only one city remains to be processed (|U| = 1) and/or when F(j) is equal to infinity for all j in U. The latter indicates that all the cities yet to be processed are not reachable from city 1.

Example 3

Consider the shortest path problem depicted on the grid of Module 3. The cities are represented by black squares. The red square in the center of the grid represents the home city. The distances between cities are Euclidean, rounded-off to integer values, except that if the Euclidean distance is greater than some critical value (called **radius**), the distance is set to be equal to infinity, meaning that the two cities are not connected by a direct link. Also, for simplicity the distance between a city to itself, D(j,j), is set to infinity. The distance matrix is therefore symmetric, hence the graph representing the problem is cyclic and the distances are positive. To highlight the cities connected to a given city by direct links, place the mouse over the city (it is not necessary to click the mouse).

As the algorithm runs, the color of the squares representing the cities changes from black to blue as the cities are being processed. Cities that have been updated but have not yet been processed are colored gold. The pattern to observe is that the blue squares (processed cities) are spreading from the center of the grid outward.



```
Click the 'Solve' button to solve the problem.
```

n 30

Radius 3

Delay 50

🟥 = home sweet home

🟩 = being processed

⬛ = has not yet been updated

🟪 = direct links

🟦 = processed

🟧 = has been updated

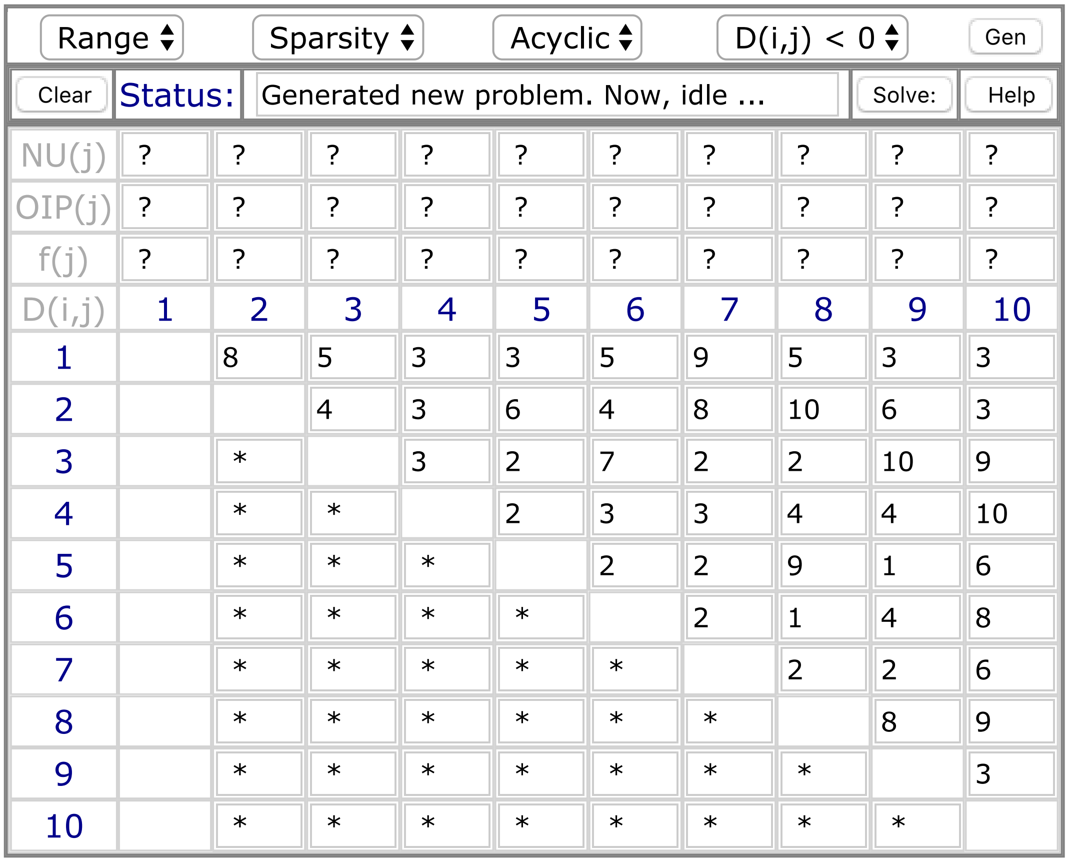
⬜ = optimal paths

Module 3

Note that in this particular example the distances are non-negative hence Dijkstra's Algorithm generates the optimal solution. Cities that are not reachable from the home city are also identified (black squares on termination).

The grid in this module was made deliberately small (11x11) so that the article could be loaded quickly over the network. As a result, the illustration itself is not as effective as it should be. For this reason we provide a stand-alone module that can generate much larger grids (up to 40x40) and thus can do a much better job illustrating how Dijkstra's Algorithm expands the set of processed cities in an outward direction from the home city. Although this module is very light, it can easily generate and solve problems with hundreds of cities. However, before you attempt to solve large problems, experiment with small problems to check that your computer can cope with larger problems.

With all due respect to Dijkstra's Algorithm, it is not the only algorithm in town. In fact, there is a variety of methods and algorithms for the solution of shortest path problems. You can use the following on-line module to experiment with a general purpose algorithm based on a generic successive approximation method.



Status: Generated new problem. Now, idle ...

NU(j)	?	?	?	?	?	?	?	?	?	?
OIP(j)	?	?	?	?	?	?	?	?	?	?
f(j)	?	?	?	?	?	?	?	?	?	?
D(i,j)	1	2	3	4	5	6	7	8	9	10
1		8	5	3	3	5	9	5	3	3
2			4	3	6	4	8	10	6	3
3		*		3	2	7	2	2	10	9
4		*	*		2	3	3	4	4	10
5		*	*	*		2	2	9	1	6
6		*	*	*	*		2	1	4	8
7		*	*	*	*	*		2	2	6
8		*	*	*	*	*	*		8	9
9		*	*	*	*	*	*	*		3
10		*	*	*	*	*	*	*	*	

Module 4

In relation to our discussion on Dijkstra's Algorithm, we note that this module computes the lengths of the shortest paths from node 1 to all other nodes. It can handle cyclic and acyclic problems and it does not care much whether the distances are negative or non-negative. This flexibility and versatility is not due to an algorithmic breakthrough but merely a reflection of the fact that, as stated above, we do not exclude cyclic paths. In other words, we do not restrict the feasible paths to be *simple:* we do allow feasible path to include cycles. This means that even though there are only finitely many cities, if the distances are allowed to be negative and there is a cycle whose length is negative, then the length of the shortest path could be unbounded from below. The module detects such cases.

This is perhaps an appropriate place to say something about the seemingly odd notion of **negative distances.** Firstly, it must be stressed that this is not an odd notion at all because the term *distance* is used here as a metaphor. There are many practical problems where the term *distance* represents costs and/or benefits, and in this context it is very natural to talk about positive and negative distances within the framework of the same problem.

Secondly, it should be stressed that the term *shortest path problem* itself is a metaphor. Many optimization problems that have nothing to do with 'real' shortest path problems. For example, the classical knapsack problem, equipment replacement problem, production lot sizing problem, and investment planning problem all have very simple and natural shortest path representations (see for instance Evans and Minieka [1992]). In fact, as was already mentioned above, any combinatorial optimization problem having finitely many feasible solutions can be formulated as a shortest path problem. In this extremely general context the notion of negative distances is not odd at all.

Thirdly, there are cases where the original problem we attempt to solve possesses only non-negative distances but the method used to solve it generates and solves (on the fly) other (related) shortest path problems whose distances can be negative. This is the case, for example, in networks problems where the objective function is the ratio of two functions (eg. see Lawler [1976, pp. 94-97]).

Finally, there are many practical situations where we are interested in the **longest** rather than shortest path. For example, the Critical Path Method (CPM) is based on the detection of longest paths in project networks (Daellenbach et al [1983], Markland and Sweigart [1987], Hillier and Lieberman [1990], Winston [2004]). If we apply the old trick of multiplying the objective function by -1 to transform a maximization problem into an equivalent minimization problem, then we would multiply all the distances by -1. Thus if originally the distances are all positive, after the transformation they will all be negative. The equivalent minimization problem would then have negative distances.

For these reasons, allowing the inter-city distances to be negative is not an eccentric idea. Rather it is a reflection of the fact that many important shortest path problems do possess this feature. Therefore, the interest in algorithms capable of solving shortest path problems whose distances are allowed to be negative is not merely academic in nature.

References

Ahuja, R.K., Magnanti, T.L. and Orlin, J.B. (1993), *Network Flow Theory, Algorithms, and Applications,* Prentice-Hall, Englewood-Cliffs, NJ.

Bellman, R. (1957), *Dynamic Programming,* Princeton University Press, Princeton, NJ.

Brassard, G. and Bratley, P. (1988) *Algorithmics,*Prentice-Hall, Englewood Cliffs, NJ.

Daellenbach, H.G., George, J.A. and D.C. McNickle, (1983), *Introduction to Operations Research Techniques,* 2nd Edition, Allyn and Bacon, Boston.

Dantzig, G.B., (1963), *Linear Programming and Extensions,* Princeton University Press, Princeton, NJ.

Dantzig, G.B. and N.M Thapa, (2003), *Linear Programming 2: Theory and Extensions,* Springer Verlag, Berlin.

Denardo, E.V. (2003), *Dynamic Programming,* Dover, Mineola, NY.

Dijkstra, E.W. (1959), A note on Two Problems in Connexion with Graphs, *Numerische mathematik,* 1, 269-271.

Dreyfus, S. (1969), An appraisal of some shortest-path algorithms *Operations Research,* 17, 395-412.

Evans, J.R. and Minieka, E. (1992), *Optimization Algorithms for Networks and Graphs,* Marcel Dekker, NY.

Gass, S.I. and Harris, C.M. (1996), *Encyclopedia of Operations Research and management Science,* Kluwer, Boston, Mass.

Hillier, F.S. and Lieberman, G.J. (1990) *Introduction to Operations Research,* 5th Edition, Holden -Day, Oakland, CA.

Lawler, E.L. (1976), *Combinatorial Optimization: Networks and Matroids,* Holt, Rinehart and Whinston, NY.

Markland, R.E. and J.R. Sweigart, (1987), *Quantitative Methods: Applications to Managerial Decision Making,* John Wiley, NY.

Microsoft Shortest Path Algorithms Project: research.microsoft.com/research/sv/SPA/ex.html.

Moore, E.F. (1959), The shortest path through a maze, pp. 285-292 in *Proceedings of an International Synposium on the Theory of Switching (Cambridge, Massachusetts, 2-5 April, 1957),* Harvard University Press, Cambridge.

Smith, D.K. (2003), *Networks and Graphs: Techniques and Computational Methods,* Horwood Publishing Limited, Chichester, UK.

Sniedovich, M. (1992), *Dynamic Programming,* Marcel Dekker, NY.

Winston, W.L. (2004), *Operations Research Applications and Algorithms,* Fourth Edition, Brooks/Cole, Belmont, CA.