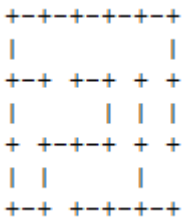
Shortest Paths

Sample Problem: Overfencing [Kolstad & Schrijvers, Spring 1999 USACO Open]

Farmer John created a huge maze of fences in a field. He omitted two fence segments on the edges, thus creating two ``exits'' for the maze. The maze is a `perfect' maze; you can find a way out of the maze from any point inside it.

Given the layout of the maze, calculate the number of steps required to exit the maze from the `worst' point in the maze (the point that is `farther' from either exit when walking optimally to the closest exit).

Here's what one particular W=5, H=3 maze looks like:



The Abstraction

Given:

- A directed graph with nonnegative weighted edges
- A path between two vertices of a graph is any sequence of adjacent edges joining them
- The shortest path between two vertices in a graph is the path which has minimal cost, where cost is the sum of the weights of edges in the path.

Problems often require only the cost of a shortest path not necessarily the path itself. This sample problem requires calculating only the costs of shortest paths between exits and interior points of the maze. Specifically, it requires the maximum of all of those various costs.

Dijkstra's algorithm to find shortest paths in a weighted graph

Given: lists of vertices, edges, and edge costs, this algorithm `visits' vertices in order of their distance from the source vertex.

- Start by setting the distance of all notes to infinity and the source's distance to 0.
- At each step, find the vertex u of minimum distance that hasn't been processed already. This vertex's distance is now frozen as the minimal cost of the shortest path to it from the source.
- Look at appending each neighbor v of vertex u to the shortest path to u. Check vertex v to see if this is a better path than the current known path to v. If so, update the best path information.

In determining the shortest path to a particular vertex, this algorithm determines all shorter paths from the source vertex as well since no more work is required to calculate *all* shortest paths from a single source to vertices in a graph.

Reference: Chapter 25 of [Cormen, Leiserson, Rivest]

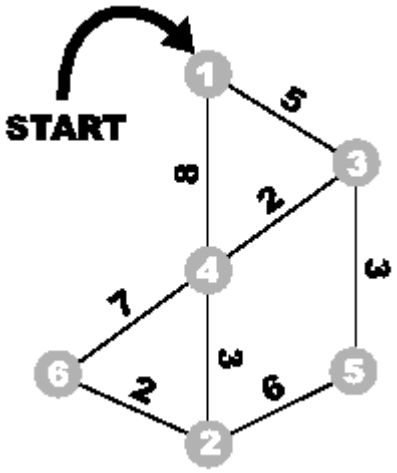
Pseudocode:

```
# distance(j) is distance from source vertex to vertex j
# parent(j) is the vertex that precedes vertex j in any shortest path
#                                (to reconstruct the path subsequently)

 1 For all nodes i
 2         distance(i) = infinity                 # not reachable yet
 3         visited(i) = False
 4         parent(i) = nil # no path to vertex yet

 5 distance(source) = 0 # source -> source is start of all paths
 6 parent(source) = nil
 7    8 while (nodesvisited < graphsize)
 9         find unvisited vertex with min distance to source; call it v
ertex i
10         assert (distance(i) != infinity, "Graph is not connected")

11         visited(i) = True # mark vertex i as visited

       # update distances of neighbors of i
12         For all neighbors j of vertex i
13                 if distance(i) + weight(i,j) < distance(j) then
14                         distance(j) = distance(i) + weight(i,j)
15                         parent(j) = i
```

Running time of this formulation is O(V^2). You can obtain O(E log V) (where E is the number of edges and V is the number of vertices) by using a heap to determine the next vertex to visit, but this is considerably more complex to code and only appreciably faster on large, sparse graphs.

Sample Algorithm Execution

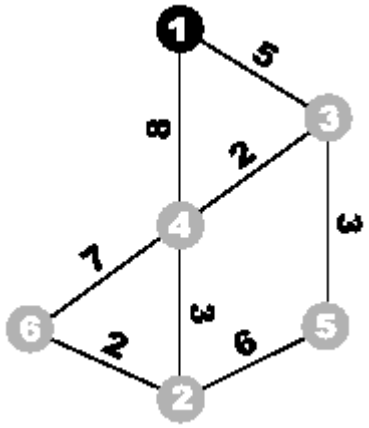
Consider the graph below, whose edge weights can be expressed two different ways:



Edge	Weight
(1, 3)	5
(1, 4)	8
(3, 4)	2
(3, 5)	3
(4, 2)	3
(4, 6)	7
(5, 2)	6
(2, 6)	2

	1	2	3	4	5	6
1	0	0	5	8	0	0
2	0	0	0	3	6	2
3	5	0	0	2	3	0
4	8	3	2	0	0	7
5	0	6	3	0	0	0
6	0	2	0	7	0	0

Here is the initial state of the program, both graphically and in a table:



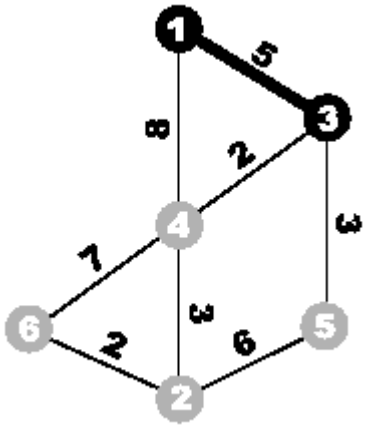
Node	Visited	Distance to Source	Parent
1	T	0	nil
2	F	infinity	nil
3	F	infinity	nil
4	F	infinity	nil
5	F	infinity	nil
6	F	infinity	nil

Updating the table, node 1's neighbors include nodes 3 and 4.

Node	Visited	Distance to Source	Parent
1	T	0	nil

2	F	infinity	nil
3	F	5	1
4	F	8	1
5	F	infinity	nil
6	F	infinity	nil

Node 3 is the closest unvisited node to the source node (smallest distance shown in column 3), so it is the next visited:

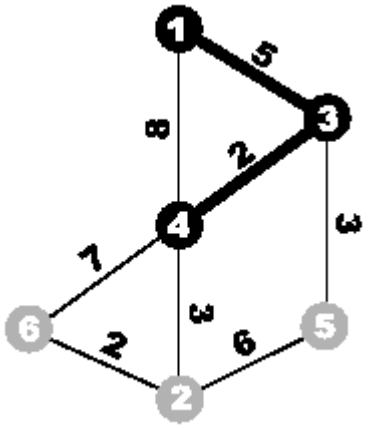


Node	Visited	Distance to Source	Parent
1	T	0	nil
2	F	infinity	nil
3	T	5	1
4	F	8	1
5	F	infinity	nil
6	F	infinity	nil

Node 3's neighbors are nodes 1, 4, and 5. Updating the unvisited neighbors yields:

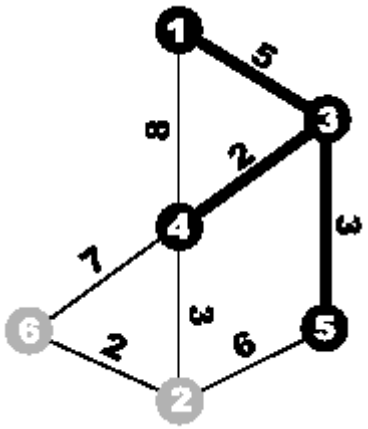
Node	Visited	Distance to Source	Parent
1	T	0	nil
2	F	infinity	nil
3	T	5	1
4	F	7	3
5	F	8	3
6	F	infinity	nil

Node 4 is the closest unvisited node to the source. Its neighbors are 1, 2, 3, and 6, of which only nodes 2 and 6 need be updated, since the others have already been visited:



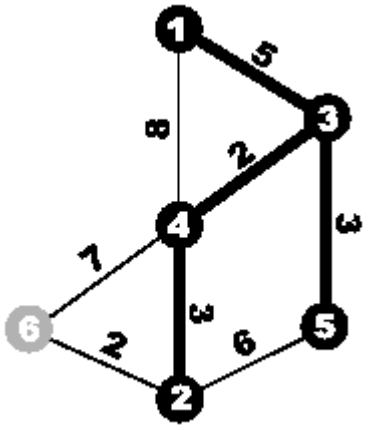
Node	Visited	Distance to Source	Parent
1	T	0	nil
2	F	**10**	**4**
3	T	5	1
4	**T**	7	3
5	F	8	3
6	F	**14**	**4**

Of the three remaining nodes (2, 5, and 6), node 5 is closest to the source and should be visited next. Its neighbors include nodes 3 and 2, of which only node 2 is unvisited. The distance to node 2 via node 5 is 14, which is longer than the already listed distance of 10 via node 4, so node 2 is not updated.



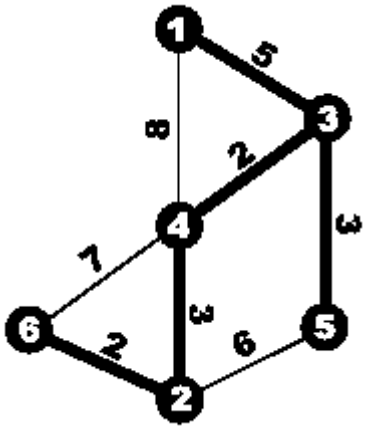
Node	Visited	Distance to Source	Parent
1	T	0	nil
2	F	10	4
3	T	5	1
4	T	7	3
5	**T**	8	3
6	F	14	4

The closest of the two remaining nodes is node 2, whose neighbors are nodes 4, 5, and 6, of which only node 6 is unvisited. Furthermore, node 6 is now closer, so its entry must be updated:



Node	Visited	Distance to Source	Parent
1	T	0	nil
2	T	10	4
3	T	5	1
4	T	7	3
5	T	8	3
6	F	12	2

Finally, only node 6 remains to be visited. All of its neighbors (indeed the entire graph) have now been visited:



Node	Visited	Distance to Source	Parent
1	T	0	nil
2	T	10	4
3	T	5	1
4	T	7	3
5	T	8	3
6	T	12	2

Sample Problem: Package Delivery

Given a set of locations, lengths of roads connecting them, and an ordered list of package dropoff locations. Find the length of the shortest route that visits each of the package dropoff locations in order.

Analysis: For each leg of the required path, run Dijkstra's algorithm to determine the shortest path connecting the two endpoints. If the number of legs in the journey exceeds N, instead of calculating each path, calculate the shortest path between all pairs of vertices, and then simply paste the shortest path for each leg of the journey together to get the entire journey.

Extended Problem: All Pairs, Shortest Paths

The extended problem is to determine a table a, where:
$a_{i,j}$ = length of shortest path between i and j, or infinity if i and j aren't connected.

This problem is usually solved as a subproblem of a larger problem, such as the package delivery problem.

Dijkstra's algorithm determines shortest paths for one source and all destinations in $O(N^2)$ time. We can run it for all N sources in $O(N^3)$ time.

If the paths do not need to be recreated, there's an even simpler solution that also runs in $O(N^3)$ time.

The Floyd-Warshall Algorithm

The Floyd-Warshall algorithm finds the length of the shortest paths between all pairs of vertices. It requires an adjacency matrix containing edge weights, the algorithm constructs optimal paths by piecing together optimal subpaths.

- Note that the single edge paths might not be optimal and this is okay.
- Start with all single edge paths. The distance between two vertices is the cost of the edge between them or infinity if there is no such edge.
- For each pair of vertices u and v, see if there is a vertex w such that the path from u to v through w is shorter than the current known path from to u to v. If so, update it.
- Miraculously, if ordered properly, the process requires only one iteration.
- For more information on why this works, consult Chapter 26 of [Cormen, Leiserson, Rivest].

Pseudocode:

```
# dist(i,j) is ″best″ distance so far from vertex i to vertex j

# Start with all single edge paths.
For i = 1 to n do
        For j = 1 to n do
                dist(i,j) = weight(i,j)

For k = 1 to n do # k is the `intermediate' vertex
        For i = 1 to n do
```

```
            For  j  =  1  to  n  do
                   if  (dist(i,k)  +  dist(k,j)  <  dist(i,j))  then
#  shorter  path?
                          dist(i,j)  =  dist(i,k)  +  dist(k,j)
```

This algorithm runs in O(V^3) time. It requires the adjacency matrix form of the graph.

It's very easy to code and get right (only a few lines).

Even if the solution requires only the single source shortest path, this algorithm is recommended, provided the time and memory and are available (chances are, if the adjacency matrix fits in available memory, there is enough time).

Problem Cues

If the problem wants an optimal path or the cost of a minimal route or journey, it is likely a shortest path problem. Even if a graph isn't obvious in a problem, if the problem wants the minimum cost of some process and there aren't many states, then it is usually easy to superimpose a graph on it. The big point here: shortest path = search for the minimal cost way of doing something.

Extensions

If the graph is unweighted, the shortest path contains a minimal number of edges. A breadth first search (BFS) will solve the problem in this case, using a queue to visit nodes in order of their distance from the source. If there are many vertices but few edges, this runs much faster than Dijkstra's algorithm (see Amazing Barn in Sample Problems).

If negative weight edges are allowed, Dijkstra's algorithm breaks down. Fortunately, the Floyd-Warshall algorithm isn't affected so long as there are no negative cycles in the graph (if there is a negative cycle, it can be traversed arbitrarily many times to get ever `shorter' paths). So, graphs must be checked for them before executing a shortest path algorithm.

It is possible to add additional conditions to the definition of shortest path (for example, in the event of a tie, the path with fewer edges is shorter). So long as the distance function can be augmented along with the comparison function, the problem remains the same. In the

example above, the distance function contains two values: weight and edge count. Both values would be compared if necessary.

Sample Problems

Graph diameter

Given: an undirected, unweighted, connected graph. Find two vertices which are the farthest apart.

Analysis: Find the length of shortest paths for all pairs and vertices, and calculate the maximum of these.

Knight moves

Given: Two squares on an NxN chessboard. Determine the shortest sequence of knight moves from one square to the other.

Analysis: Let the chessboard be a graph with 64 vertices. Each vertex has at most 8 edges, representing squares 1 knight move away.

Amazing Barn (abridged) [USACO Competition Round 1996]

Consider a very strange barn that consists of N stalls (N < 2500). Each stall has an ID number. From each stall you can reach 4 other stalls, but you can't necessarily come back the way you came.

Given the number of stalls and a formula for adjacent stalls, find any of the `most central' stalls. A stall is `most central' if it is among the stalls that yields the lowest average distance to other stalls using best paths.

Analysis: Compute all shortest paths from each vertex to determine its average distance. Any $O(N^3)$ algorithm for computing all-pairs shortest paths would be prohibitively expensive here since N=2500. However, there are very few edges (4 per vertex), making a BFS with queue ideal. A BFS runs in O(E) time, so to compute shortest paths for all sources takes O(VE) time - about:
$2500 \times 10{,}000 = 2.5 \times 10^6$ things, much more reasonable than $2500^3 = 1.56 \times 10^{10}$

Railroad Routing (abridged) [USACO Training Camp 1997, Contest 1]

Farmer John has decided to connect his dairy cows directly to the town pasteurizing plant by constructing his own personal railroad.

Farmer John's land is laid out as a grid of one kilometer squares specified as row and column.

The normal cost for laying a kilometer of track is $100. Track that must gain or lose elevation between squares is charged a per-kilometer cost of $100 + $3 x meters_of_change_in_elevation. If the track's direction changes 45 degrees within a square, costs rise an extra $25; a 90 degree turn costs $40. All other turns are not allowed.

Given the topographic map, and the location of both John's farm and the plan, calculate the cost of the cheapest track layout.

Analysis: This is almost a standard shortest path problem, with grid squares as vertices and rails as edges, except that the direction a square is entered limits the ways you can exit that square. The problem: it is not possible to specify which edges exist in advance (since the path matters).

The solution: create eight vertices for each square, one for each direction you can enter that square. Now you can determine all of the edges in advance and solve the problem as a shortest path problem.

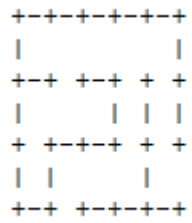
Shortest Paths

最短路径

译 by tim green

Sample Problem: Overfencing [Kolstad & Schrijvers, Spring 1999 USACO Open]

农夫 John 在外面的田野上搭建了一个巨大的用栅栏围成的迷宫。幸运的是，他在迷宫的边界上留出了两段栅栏作为迷宫的出口。更幸运的是，他所建造的迷宫是一个“完美的”迷宫：即你能从迷宫中的任意一点找到一条走出迷宫的路。
给定迷宫的宽 W(1<=W<=38)及长 H(1<=H<=100)。
2*H+1 行，每行 2*W+1 的字符以下面给出的格式表示一个迷宫。然后计算从迷宫中最“糟糕”的那一个点走出迷宫所需的步数。（即使从这一点以最优的方式走向最靠近的出口，它仍然需要最多的步数）当然了，牛们只会水平或垂直地在 X 或 Y 轴上移动，他们从来不走对角线。每移动到一个新的方格算作一步（包括移出迷宫的那一步）
这是一个 W=5,H=3 的迷宫：



如上图的例子，栅栏的柱子只出现在奇数行或奇数列。每个迷宫只有两个出口。

The Abstraction

给出:
一个边的权为非负的有向图

- 两个顶点间的一条路径是由图中相邻的边以任意的顺序连接而成的
- 两个顶点之间的最短路径是指图中连接这两个顶点的路径中代价最小的一条，一条路径的代价是指路径上所有边的权的和
- 题目常常只要求出最短路径的代价而不需要求出路径本身。在这个例子中只用求出迷宫内部的点和出口之间的最短距离(代价)。最别的，它需在求出所有代价中最大的一个。

用 Dijkstra 算法来求加权图中的最短路径

给出:所有的顶点、边、边的权，这个算法按离源点的距离从近到远的顺序来"访问"每个顶点。

- 开始时把所有不相邻顶点间的距离(边的权)标为无穷大, 自己到自己的距离标为 0。
- 每次, 找出一个到源点距离最近并且没有访问过的顶点 u。现在源点到这个顶点的距离就被确定为源点到这个顶点的最短距离。
- 考虑和 u 相邻的每个顶点 v 通过 u 到源点的距离。考查顶点 v, 看这条路径是不是更优于已知的 v 到源点的路径, 如果是, 更新 v 的最佳路径。

[In determining the shortest path to a particular vertex, this algorithm determines all shorter paths from the source vertex as well since no more work is required to calculate all shortest paths from a single source to vertices in a graph.]

参考: [Cormen, Leiserson, Rivest]的第 25 章

Pseudocode:

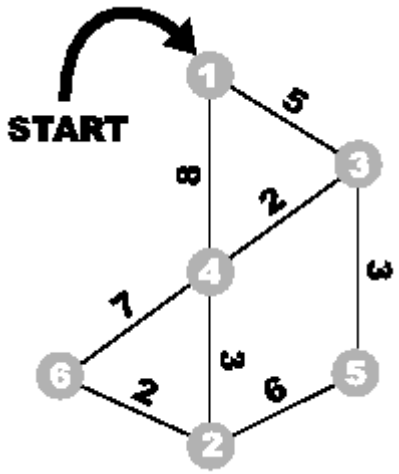
```
# distance(j) is distance from source vertex to vertex j
# parent(j) is the vertex that precedes vertex j in any shortest path
# (to reconstruct the path subsequently)
```

```
1 For all nodes i
2 distance(i) = infinity # not reachable yet
3 visited(i) = False
4 parent(i) = nil # no path to vertex yet

5 distance(source) = 0 # source -> source is start of all paths
6 parent(source) = nil
7 8 while (nodesvisited < graphsize)
9 find unvisited vertex with min distance to sourcd; call it vertex i
10 assert (distance(i) != infinity, "Graph is not connected")

11 visited(i) = True # mark vertex i as visited

# update distances of neighbors of i
12 For all neighbors j of vertex i
13 if distance(i) + weight(i,j) < distance(j) then
14 distance(j) = distance(i) + weight(i,j)
15 parent(j) = i
```

这个算法的时间效率为 $O(V^2)$。你可以使用堆来确定下一个要访问的顶点来获得 O(E log V)的效率(这里的 E 是指边数 V 是指顶点数)，但这样做会使程序变得复杂，并且在一个大而稀疏的图中效率只能提高一点点。

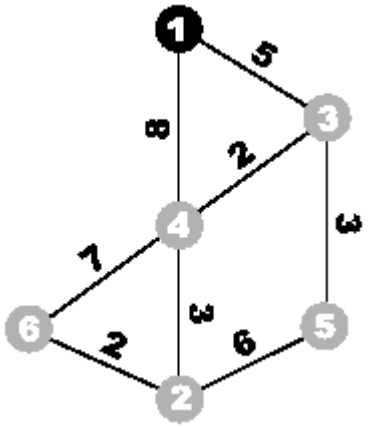
Sample Algorithm Execution

考虑下面的图，图中的边可以用两种不同的方式表示:



Edge	Weight	
(1, 3)	5	1 2 3 4 5 6
(1, 4)	8	1 0 0 5 8 0 0
(3, 4)	2	2 0 0 0 3 6 2
(3, 5)	3	3 5 0 0 2 3 0
(4, 2)	3	4 8 3 2 0 0 7
(4, 6)	7	5 0 6 3 0 0 0
(5, 2)	6	6 0 2 0 7 0 0
(2, 6)	2	

这是问题的初始状态:

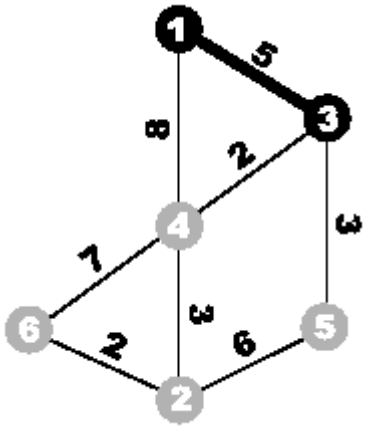


Node	Visited	Distance to Source	Parent
1	T	0	nil
2	F	infinity	nil
3	F	infinity	nil
4	F	infinity	nil
5	F	infinity	nil
6	F	infinity	nil

更新这个表格,和顶点 1 相邻的有顶点 3、4

Node	Visited	Distance to Source	Parent
1	T	0	nil
2	F	infinity	nil
3	F	**5**	**1**
4	F	**8**	**1**
5	F	infinity	nil
6	F	infinity	nil

现在还没被访问的顶点中顶点 3 到源点的距离最小,所以它是下一个要被访问的点:



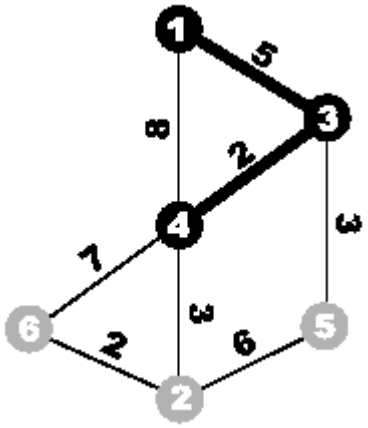
Node	Visited	Distance to Source	Parent
1	T	0	nil
2	F	infinity	nil
3	**T**	5	1
4	F	8	1
5	F	infinity	nil
6	F	infinity	nil

和顶点 3 相邻的顶点有 4、5 更新这些顶点:

Node	Visited	Distance to Source	Parent

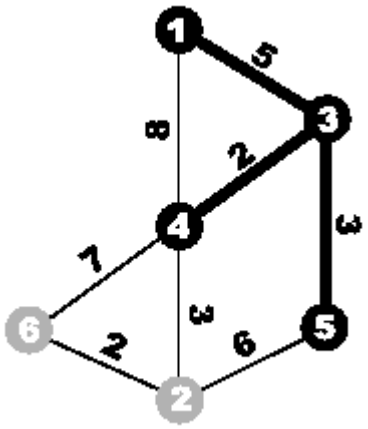
1	T	0	nil
2	F	infinity	nil
3	T	5	1
4	F	**7**	**3**
5	F	**8**	**3**
6	F	infinity	nil

现在还没被访问的顶点中顶点 4 到源点的距离最小，它的邻点有 1、2、3、6, 因为别的点都已经被访问过了，所以只有顶点 2、6 需要被更新:



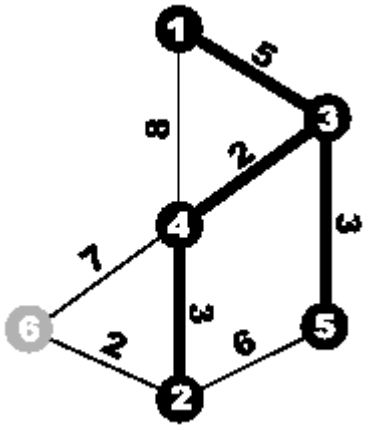
Node	Visited	Distance to Source	Parent
1	T	0	nil
2	F	**10**	**4**
3	T	5	1
4	**T**	7	3
5	F	8	3
6	F	**14**	**4**

在剩下的三个顶点(2, 5, 和 6)中, 顶点 5 最接近源点，所以要被访问。它的邻点有 2、3,只有顶点 2 没有被访问过。经顶点 5 再到顶点 2 的距离是 14, 比经顶点 4 的路径长(那条长为 10),所以顶点 2 不需要更新。



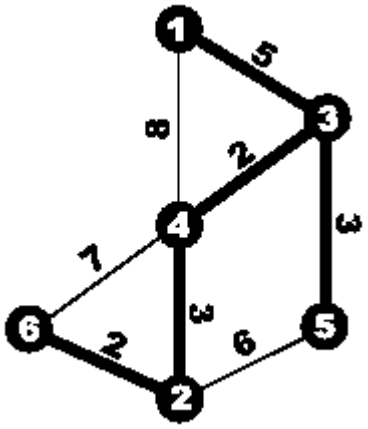
Node	Visited	Distance to Source	Parent
1	T	0	nil
2	F	10	4
3	T	5	1
4	T	7	3
5	**T**	8	3
6	F	14	4

在剩下的两个顶点中最接近的是顶点 2 ，它的邻点是顶点 4、5、6,只有顶点 6 没被访问过。另外顶点 6 要被更新:



Node	Visited	Distance to Source	Parent
1	T	0	nil
2	T	10	4
3	T	5	1
4	T	7	3
5	T	8	3
6	F	12	2

最后，只有顶点 6 没被访问过，它所有的邻点都被访问过了:



Node	Visited	Distance to Source	Parent
1	T	0	nil
2	T	10	4
3	T	5	1
4	T	7	3
5	T	8	3
6	T	12	2

Sample Problem: Package Delivery 投递包裹

给出一个地点的集合，和连接它们的道路的长，和一个按排好的包裹要投递的地点的清单。找出按顺序投递所有包裹的最短路程。

分析:对于每一次投递，使用 Dijkstra 算法来确定两个点间的最短路程。如果要投递的次数超过 N,我们就用计算每对点之间的最短距离来代替计算每投递的最短路程， 最后只用简单把每一次投递一路程相加就是答案。

Extended Problem: All Pairs, Shortest Paths 每对点之间的最短路程

这个扩展问题就是确定一个表格 a：

这里 $a_{i,j}$= 顶点 i 和 j 之间的最短距离，或者无穷大如果顶点 i 和 j 无法连通。

这个问题常常是其它大的问题的子问题，如投递包裹。

Dijkstra 算法确定单源最短路径的效率为 $O(N^2)$。我们在每个顶点上执行一次的

效率为 $O(N^3)$。
如果不要求输出路径，还有一个更简单的算法效率也是 $O(N^3)$

The Floyd-Warshall Algorithm

Floyd-Warshall 算法用来找出每对点之间的最短距离。它需要用邻接矩阵来储存边，这个算法通过考虑最佳子路径来得到最佳路径。

注意单独一条边的路径也不一定是最佳路径。

- 从任意一条单边路径开始。一个两点之间的距离是边的权，或者无穷大如呆两点之间没有国相连。
- 对于每一对顶点 u 和 v，看看是否存在一个顶点 w 使得从 u 到 w 再到 v 比已知的路径更短。如果是更新它。
- 不可思议的是，只要按排适当，就能得到结果。
- 想知道更多关于这个算法是如何工作的，请参考[Cormen, Leiserson, Rivest] 第 26 章。

Pseudocode:

```
# dist(i,j) is "best" distance so far from vertex i to ver
tex j

# Start with all single edge paths.
For i = 1 to n do
        For j = 1 to n do
                dist(i,j) = weight(i,j)

For k = 1 to n do # k is the `intermediate' vertex
        For i = 1 to n do
                For j = 1 to n do
                        if (dist(i,k) + dist(k,j) < dist(i,j))
then # shorter path?
                                dist(i,j) = dist(i,k) + dist(k,j)
```

这个算法的效率是 $O(V^3)$。它需要邻接矩阵来储存图。
这个算法很容易实现，只要几行。

即使问题是求单源最短路径，还是推荐使用这个算法，如果时间和空间允许(只要有放的下邻接矩阵的空间，时间上就没问题)。

Problem Cues

如果问题要求最佳路径或最短路程，这当然是一个最短路径问题。即使问题中没有给出一个图，如果问题要求最小的代价并且这里没有很多的状态，这时常常可以构造一个图。最大的一点要注意的是:最短路径 = 搜索做某件事的最小代价。

Extensions

如果图是不加权的，最短路径包括最少的边。这种情况下用宽优搜索(BFS)就可以了。如果图中有很多顶点而边很少，这样会比 **Dijkstra** 算法要快(看例子 Amazing Barn)

如果充许负权边，那么 **Dijkstra** 算法就错了。幸运的是只要图中没有负权回路 **Floyd-Warshall** 算法就不受影响(如果有负权回路，就可以多走几圈得到更小的代价)。所以在执行算法之前必须要检查一下图。

可以再增加一些条件来确定最短路径(如，边少的路径比较短)。只要用从距离函数中得到比较函数，问题就是一样的。在上面的例子中距离函数包括两个值 代价和边数。两个值都要比较如果必要的话。

Sample Problems

Graph diameter

图的直径

给出:一个无向无权的连通图,找出两点距离最远的顶点。
分析:找出所有点之间的最短距离，再找出最大的一个。

Knight moves

爵士的移动

给出:两个 N*N 的棋盘.确定一个爵士从一个棋盘移动到另一个所需的最少步数。
分析:把棋盘变成一个 64 个顶点的图。每个顶点有 8 条边表示爵士的移动。

Amazing Barn (abridged) [USACO Competition Round 1996]

考虑一个非常大的谷仓由 N(N < 2500)个畜栏组成，每个畜栏都 有一个 ID 号。从每个畜栏你可以到达另外 4 个，但你不能走回头路。
给出畜栏的数目和计算相邻关系的公式，找出"最中心的畜栏",一个畜栏到其它的平均距离最小则被认为是"最中心的畜栏"的。
分析:计算每对顶点间的最短距离来确定平均距离。任何 $O(N^3)$的算法在 N=2500 的情况下都是不行的。注意到图中的边比较少(一个 4 条边)，用 BFS 加上队列是可行的，一次 BFS 的较率为 O(E)，所以 $2500 \times 10,000 = 2.5 \times 10^6$ 比 $2500^3 = 1.56 \times 10^{10}$ 要好的多。