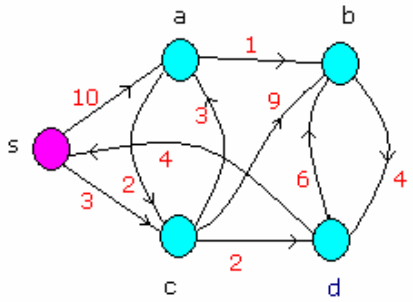
Dijkstra's Algorithm

As a Dynamic Programming strategy

Juan Chamero, spain@yahoo.es lectures

Discrete Structures & Algorithms, ds_006, as of April 2006



E weight: d(V)

[s a] - 10
[s c] - 3
[a b] - 1
[a c] - 2
[b d] - 4
[c a] - 3
[c b] - 9
[c d] - 2
[d b] - 6
[d s] - 4

V(G): [s a b c d]

Dijkstra's algorithm finds shortest paths along certain type of graphs (**Directed Graphs**). It belongs to the DP, **Dynamic Programming** family, and as such its logic rests within the Optimality criterion of *Richard Bellman*. The main aim of Dynamic Programming created by mathematician Richard Bellman in 1953 could be seen as a subtle tradeoff in computing, a tight compromise between computing process power and memory to solve problems efficiently by overlapping sub problems and optimal substructure. It means that optimal overall solutions for a given problem could be undertaken via optimal solutions of sub problems of it. In summary it's a Cartesian Strategy to solve complex problems, see below the rules of the **Descartes Method**

1. Accept as true only what is indubitable.
2. Divide every question into manageable parts.
3. Begin with the simplest issues and ascend to the more complex.
4. Review frequently enough to retain the whole argument at once.

Dynamic Programming strategy logic is

1. Break the problem into smaller sub problems.
2. Solve these problems optimally using this three-step process recursively.
3. Use these optimal solutions to construct an optimal solution for the original problem.

The sub problem search leveling has no limitations till found either the "atom" sub problem or a sub problem easy to solve. By overlapping sub problems we mean reuse of partial computations performed over low level sub problems. It saves time but as a counterpart it takes too much memory. Another characteristic of Dynamic Programming is "**memoization**" (see Peter Norvig AI works) that stands for keeping any computation potentially apt for ulterior reuse. Memoization requires high skill programming

Note: A directed graph or digraph G is an ordered pair $G:=(V, A)$ with a V set of vertices or nodes and a set of ordered pairs of vertices A, called directed edges, arcs, or arrows. An edge $e = (x, y)$ is considered to be directed from x to y; y is called the head and x is called the tail of the edge.

Algorithm work

The algorithm works from a source (s) by computing for each vertex *v* pertaining to V the cost *d*[*v*] of the shortest path found so far between *s* and *v*. Initially this value is set to 0 for the source vertex *s* (*d*[*s*]=0), and infinity for all other vertices, representing the fact that we do not know any path leading to those vertices (*d*[*v*]=∞ for every *v* in *V*, except *s*). When the algorithm finishes, *d*[*v*] should be the cost of the shortest path from *s* to *v* (or infinity, if no such path exists).

The basic operation of Dijkstra's algorithm rests on the essence of DP and is named "**edge relaxation**". Let's suppose that we are looking the shortest path that goes from s to v. If we know the shortest path from s to all possible u's connected to v and if there are edges from those u's to *v*, then the shortest known path from *s* to *u* (*d*[*u*]) can be obtained through a path (the best path) from *s* to *v* by adding edge (*u*,*v*) at the end. This path will have length *d*[*u*]+*w*(*u*,*v*). If this is less than the current *d*[*v*], we can replace the current value of *d*[*v*] with the new value. Edge relaxation is applied until all values *d*[*v*] represent the cost of the shortest path from *s* to *v*. The algorithm is organized so that each edge (*u*,*v*) is relaxed only once, when *d*[*u*] has reached its final value.

The algorithm maintains two sets of vertices *S* and *Q*. Set *S* contains all vertices for which we know that the value *d*[*v*] is already the cost of the shortest path and set *Q* contains all other vertices. Set *S* starts empty, and in each step one vertex is moved from *Q* to *S*. This vertex is chosen as the vertex with lowest value of *d*[*u*]. When a vertex *u* is moved to *S*, the algorithm relaxes every outgoing edge (*u*,*v*).

Note: Relaxation is a general procedure to solve "contracture problems" in physics and mathematics. For instance in physics we may have potential surfaces supposedly in equilibrium, satisfying a set of differential equations. We may guess its equilibrium assigning discrete values of potential to a grid of points over the surface. Initially the surface will look like with "contractures" not evenly smoothed. We may imagine then a convergent procedure to "relax" the surface, make contractures disappear progressively. Relaxation proceeds form point to point navigating three dimensionally as follows:

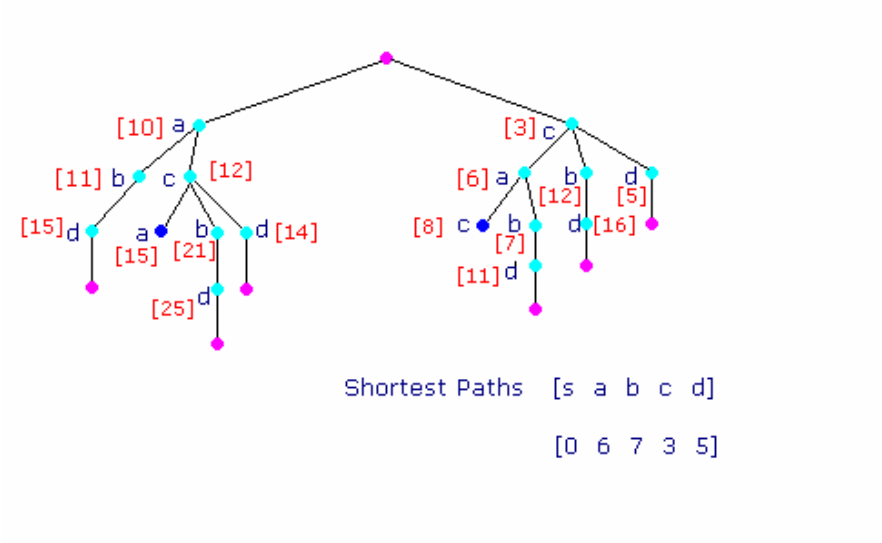
$$V(i, j, k) = W1(V(i-1, j, k) + V(i+1, j, k)) + W2(V(i, j-1, k) + V(i,j+1, k)) + W3(V(i, j, k-1) + V(I,,j,k+1)) + WF(i, j, k)$$

In discrete approaches relaxation stands for a similar procedure, "relaxing" distances as in Dynamic Programming and Linear Programming algorithms.

DIJKSTRA (G, w, s) pseudo code

1. INITIALIZE SINGLE-SOURCE (G, s)
2. S <= { } // S will ultimately contains vertices of final shortest-path weights from s
3. Initialize priority queue Q i.e., Q <= V[G]
4. while priority queue Q is not empty do
5. *u* <= EXTRACT_MIN(Q) // Pull out new vertex
6. S <= S**U**{*u*} // Perform relaxation for each vertex *v* adjacent to u
7. for each vertex v in Adj[u] do
8. Relax (*u*, *v*, *w*)

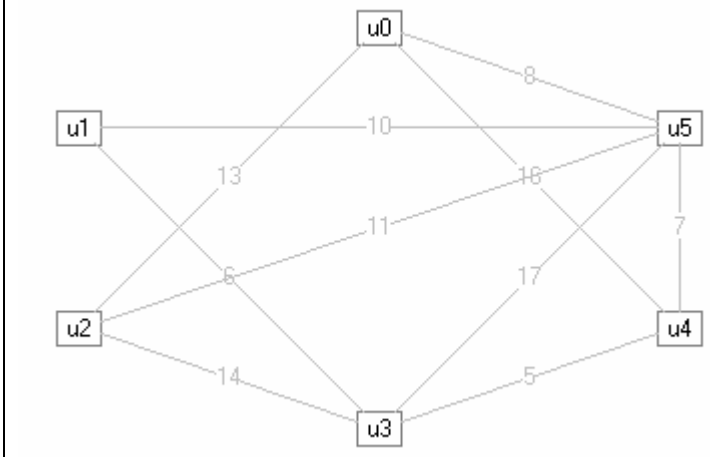
```
1  Function Dijkstra(G, w, s)
2    for each vertex v in V[G]   // Initialization
3      do d[v] := infinity
4         previous[v] := undefined
5    d[s] := 0
6    S := empty set
7    Q := set of all vertices
8    while Q is not an empty set
9      do u := Extract-Min(Q)
10        S := S union {u}
11        for each edge (u,v) outgoing from u
12          do if d[v] > d[u] + w(u,v) //Relax( u,v)
13            then d[v] := d[u] + w(u,v)
14               previous[v] := u
```

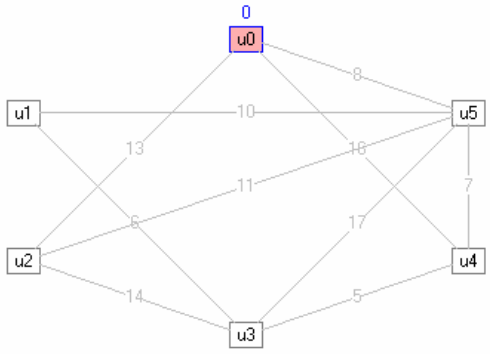


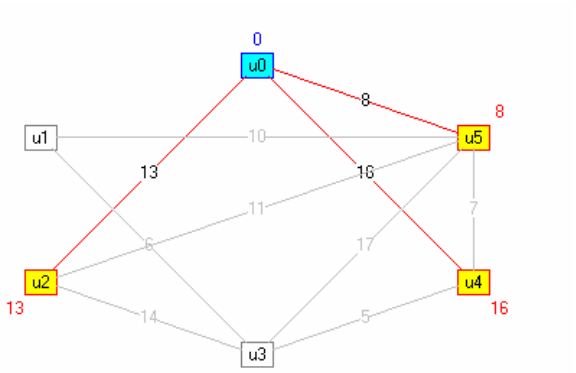
Another Example

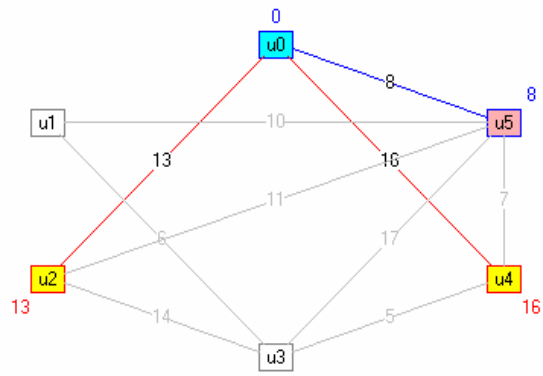
Iteration per Iteration

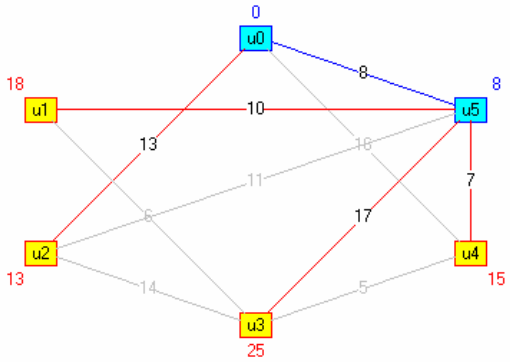
u0 u2 13
u0 u4 16
u0 u5 8
u1 u3 6
u1 u5 10
u2 u3 14
u2 u5 11
u3 u4 5
u3 u5 17
u4 u5 7

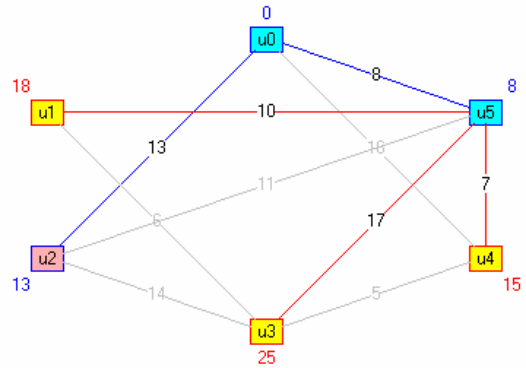


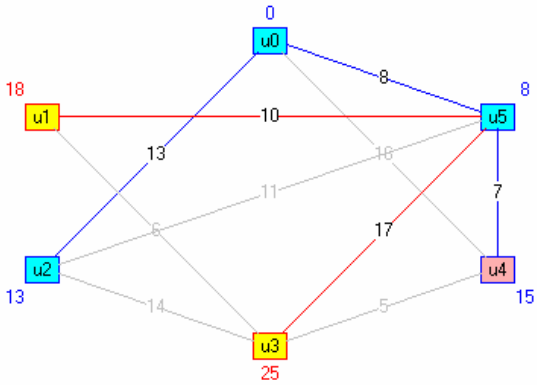


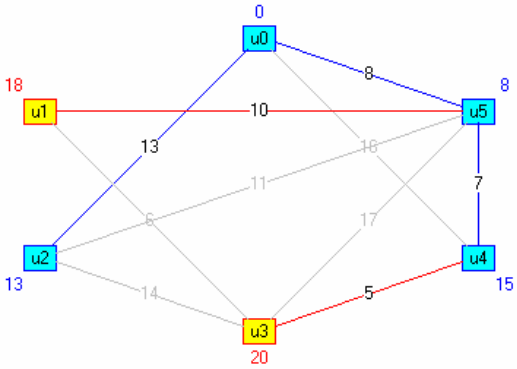


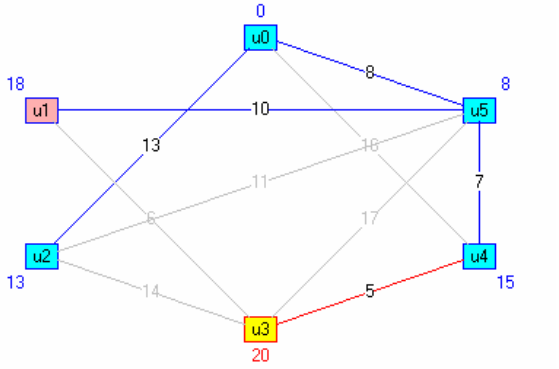

0
u0
8
u1
10
u5
8
13
16
11
7
6
17
u2
u4
13
16
14
5
u3

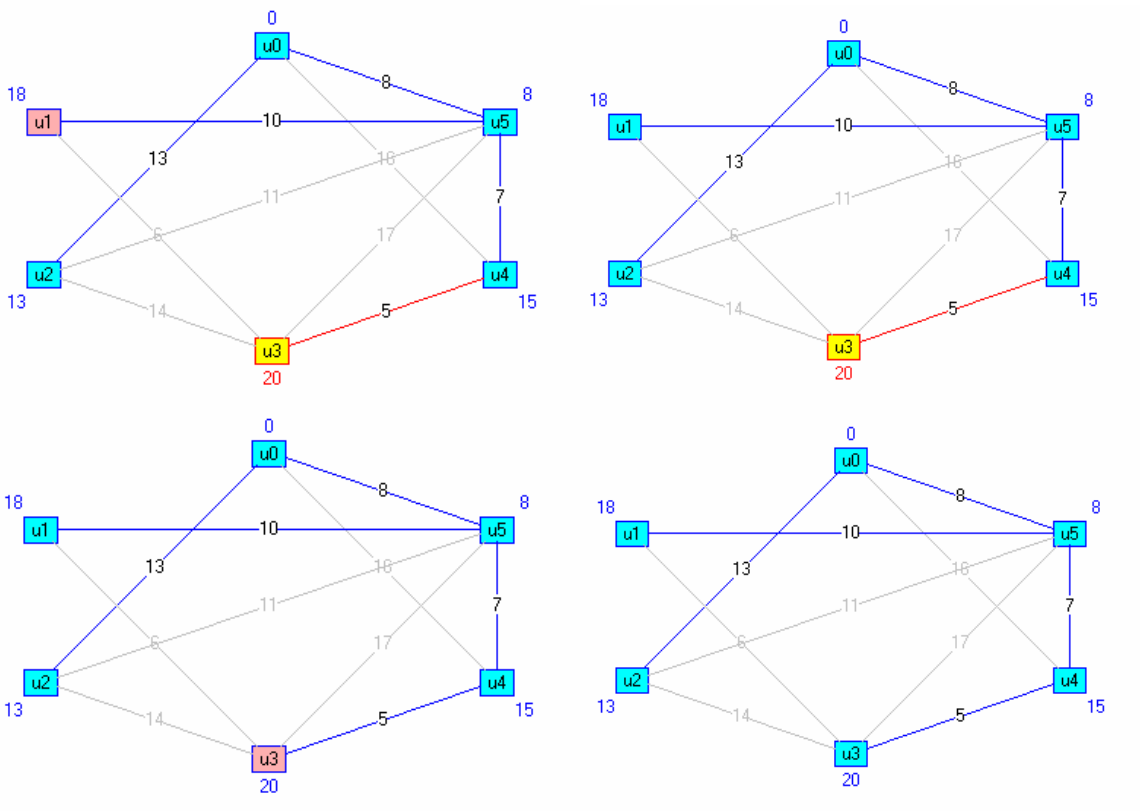

0
u0
8
18
u1
10
u5
8
13
16
11
7
6
17
u2
u4
13
15
14
5
u3
25

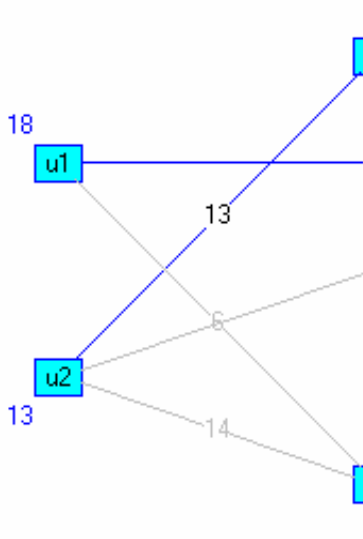

0
u0
8
18
u1
10
u5
8
13
16
11
7
6
17
u2
u4
13
15
14
5
u3
25


0
u0
8
18
u1
10
u5
8
13
16
11
7
6
17
u2
u4
13
15
14
5
u3
25


0
u0
8
18
u1
10
u5
8
13
16
11
7
6
17
u2
u4
13
15
14
5
u3
20


0
u0
8
18
u1
10
u5
8
13
16
11
7
6
17
u2
u4
13
15
14
5
u3
20


0
u0
8
18
u1
10
8
u5
13
16
11
7
6
17
u2
u4
13
14
5
15
u3
20


0
u0
8
18
u1
10
8
u5
13
16
11
7
6
17
u2
u4
13
14
5
15
u3
20