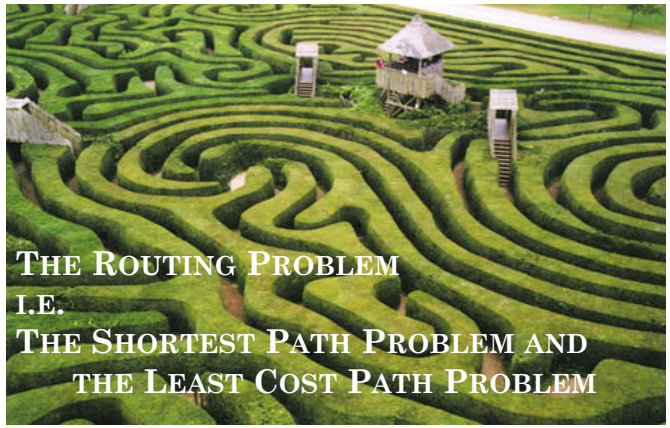
FIRST PART:
CABLE NETWORKS



THE ROUTING PROBLEM
I.E.
THE SHORTEST PATH PROBLEM AND THE LEAST COST PATH PROBLEM

Prof. Tiziana Calamoneri
Network Algorithms
A.A. 2013/14

THE ROUTING PROBLEM(1)

Given a network:

- When packets are sent from a computer to another one through the network, each computer has to route data on a path passing through intermediate computers.
- This is the very general routing problem.

THE ROUTING PROBLEM (2)

- Case 1. Not adaptive Routing. A routing algorithm could try to send packets through a network so that the length of the used path is minimized. Such length can be measured in terms of number of hops between pairs of computers.
- If the network is modeled as a graph (node = computer and edge = link), the problem reduces to the shortest path problem between two nodes.

The Routing Problem (3)

- Case 2. Adaptive Routing. It takes into account the traffic conditions: in order to decide next step, the traffic is estimated, so the packet is sent toward the zones of the network not affected by traffic.
- If the network is modeled by an edge-weighted graph (node = computer, edge= link and weight = dynamic value proportional to the traffic on the connection), the problem reduces to the (dynamic) least cost path problem.

The Routing Problem (4)

cases 1 and 2. Adaptive and non adaptive routings (cntd.)

- Non adaptive routing:
  - Good results with consistent topology and traffic
  - Poor performance if traffic volume or topologies change over time
  - Information about the entire network has to be available
  - Each packet is routed through an outgoing edge in a fixed way
  - Routing tables are **used**

The Routing Problem (5)

cases 1 and 2. Adaptive and non adaptive routings (cntd.)

- Adaptive routing:
  - Decisions are based on current network state
  - Packets follow dynamically computed routes
  - Routers are able to communicate
  - Re-calculations quite often necessary
  - Each router **creates** its own routing table

The Routing Problem (6)

- Case 3. Routing with faults. When the network is modeled as a graph, the length (weight) of an edge represents the probability of its failing (used, for instance, in networks of thelephone lines, or broadcasting systems in computer networks or in transportation routes). In all these cases, one is looking for the route having the highest probability for not failing.

  More precisely…

The Routing Problem (7)

case 3. Routing with faults (cntd.)

- Let $p(e)$ be the probability that edge e does not fail. Under the -not always realistic- assumption that failings of edges occur independently of each other, $p(e_1)*p(e_2)*\ldots*p(e_k)$ gives the probability that the path $P=(e_1,e_2,\ldots,e_k)$ can be used without any faults.
- We want to maximize this probability over all possible paths with starting point a and arrival point b…

The Routing Problem (8)

case 3. Routing with faults (cntd.)

- Note. Since function *log* is monotonic increasing, the maximum of the product $p(e_1)*p(e_2)*\ldots*p(e_k)$ is reached iff the logarithm of the product is maximum, i.e. iff:

  $\log p(e_1)+\log p(e_2)+\ldots+\log p(e_k)$ is maximum.
- $\log p(e) \leq 0$ for each e because $p(e) \leq 1$.
- Define $w(e)=-\log p(e)$, then $w(e)\geq 0$ for all e; furthermore, we have to find a path from a to b for which

  $w(e_1)+w(e_2)+\ldots+w(e_k)$ becomes minimum.
- Thus, our problem is reduced again to the least cost path problem.

The Routing Problem (9)

Why does routing matter?

- end-to-end performance
  - Quality of the path affects user performance (propagation delay, throughput, packet loss)
- Use of network resources
  - balance of the traffic over the routers and the links
  - Avoiding congestion by directing traffic to lightly-loaded links
- Transition
  - The periodical changes of the routing tables reduce the incidence of faults and increase load balancing
  - Limiting packet loss and delay during changes

The Shortest Path Problem and The Least Cost Path Problem

Shortest Paths (1)

- Let $G=(V,E)$ be a graph with length of each edge e $w(e)$.
- Many versions of the shortest path problem:
  - All to all
  - One to one
  - One to all
  - All to one
  - Lengths can be:
    - All equal (unit length)
    - Non negative
    - Possibly negative but without negative cycles
    - Possible negative cycles

Shortest Paths (2)

- Algorithm designed by Moore ['59] for the one-to-all shortest path problem and unit lenghts:

  … Breadth First Search (BFS) …
- TH. G is connected iff at the end of the BFS starting from node a, $dist(a,b) < \infty$ for each node b, where $dist$ is the distance in terms of number of edges
- Note. This claim is false if G is a digraph (indeed the notion of connected graph does not exist on digraphs: strong and weak connection)

Least Cost Paths (1)

Let $G=(V,E)$ be a graph or a digraph and let $w:E \rightarrow \mathbb{R}$ be a function.

- (G,w) is called network
- $w(e)$ is called length (though including meanings such as cost, capacity, weight, probability, …)
- For each path $P=(e_1, e_2, \ldots, e_k)$ (if G is a digraph, P is a dipath), the length of P is defined as $w(P)=w(e_1)+w(e_2)+ \ldots +w(e_k)$.
- Note. If $w(e)=1$ for each edge, the least cost path problem reduces to the shortest path problem.

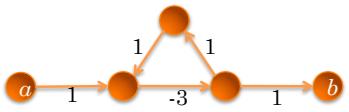
Least Cost Paths (2)

Given two nodes a and b, the distance $d(a,b)$ is defined as the minimum, over all the paths P connecting a and b, of $w(P)$.

Two problems arise:

- PR.1: b could be unreachable from a
- SOL.: define $d(a,b)=\infty$ if b is unreachable from a

- PR.2: the minimum could not exist (cycles of negative length)
- SOL: only networks without cycles of negative length are feasible



Least Cost Paths (3)

Example showing that negative lengths may occur:

- A ship travels from port *a* to port *b*, where the route (and possible intermediary ports) may be chosen freely.
- The length *w(x,y)* signifies the profit gained by going from *x* to *y*.
- For some edges, the ship might have to travel empty so that *w(e)* is negative for these edges: the profit is actually a loss.
- Replacing *w(e)* by *–w(e)* for all *e* in this network, the shortest path represents the route which yields the largest possible profit.

Least Cost Paths (4)

- In general, we can replace *w(e)* by *–w(e)* and look for largest weigth paths, but this could introduce cycles of negative weigth.
- There exist good algorithms that find minimum weigth paths even when *G* contains cycles of negative weigth

Least Cost Paths (5)

Observations concerning the solution

In any solution:

- Cycles having negative length cannot exist (we avoided them by hypthesis)
- Cycles having positive length cannot exist (by contradiction: if one of them is in the solution, the new solution without it has a lower cost)
- Cycles having null length do not exist without loss of generality: if one of them is in the solution, the new solution without it has the same cost and so is feasible, too
- So: our solution does not contains any cycles and hence it contains at most *n-1* edges

Least Cost Paths (6)

Observations concerning the solution

- In order to univocally determine a path from *a* to *b* it is enough, for each node in such path starting from *b* and coming back, to store its predecessor on the path.
- To do it: for each node *v* in *G* define a pointer *p(v)*, initially equal to NULL; at the end it points at the predecessor of *v* on the path.

Some Classical Algorithms

(already studied!)

Bellman-Ford Algorithm ['58]

- *G=(V,E)* directed with edge weights possibly negative
- It solves the problem of the shortest path from single source
- It outputs the distances from the source to each node
- Assume that *G* does not contains any cycles of negative length
- It is based on the principle of relaxation
- Time complexity: $O(nm)$

Dijkstra Algorithm ['59]

- *G=(V,E)* directed with non negative edge weights
- It solves the problem of the shortest path from single source
- It is based on the principle of relaxation
- Time complexity: either $O(n^2)$ or $O(m \log n)$
- The time complexity of the Dijkstra Algorithm is better than the time complexity of the Bellman-Ford Algorithm, but it is less versatile, as it requires non negative edge weight edges.

Floyd-Warshall Algorithm ['62]

- *G=(V,E)* directed with edge weights possibly negative
- It solves the problem of the all pairs shortest path
- Repeatedly applying the algs treated before, varying the source over all nodes in *V*:
  - Bellman-Ford: $n\ O(nm)=O(n^2m)$
  - Dijkstra: $n\ O(n^2) =O(n^3)$ o $n\ O(m \log n)=O(mn \log n)$
- Time complexity: $O(n^3)$ and negative edge weights are allowed

The Relaxation

- For each node v, let $d(v)$ be a function representing an estimate of the weight of the shortest path from s to v.
- At the beginning $d(v)=\infty$ for each v
- One relaxation step is performed as follows:
  - Given an edge (u,v)
  - If $d(u)+w(u,v)<d(v)$

    $d(v)=d(u)+w(u,w)$

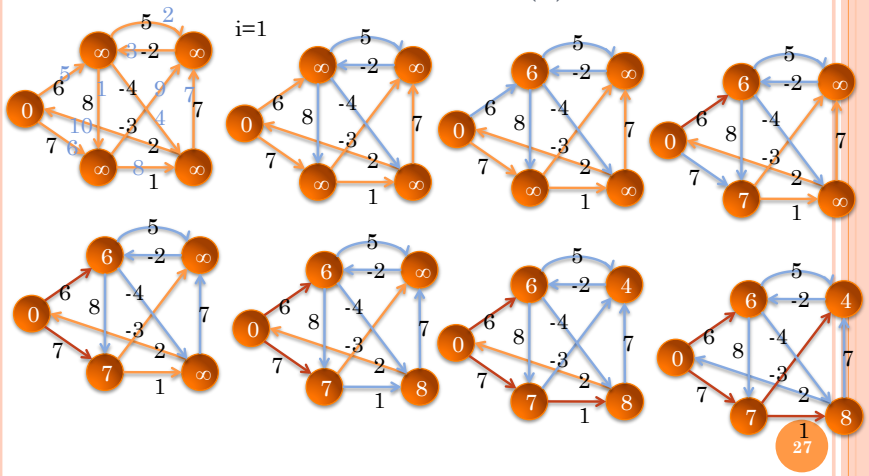
    $p(v)=u$
- Time complexity of one relaxation step: $O(1)$

Bellman-Ford Algorithm (1)

- Assume that G does not contains any cycles of negative length
- For each v initialize $d(v)$ and $p(v)$ — $O(n)$
- For $i=1$ to $n-1$ do — $n-1$ times
  - For each (u,v) relax v wrt (u,v) — $mO(1)$
- Time Complexity: $O(nm)$

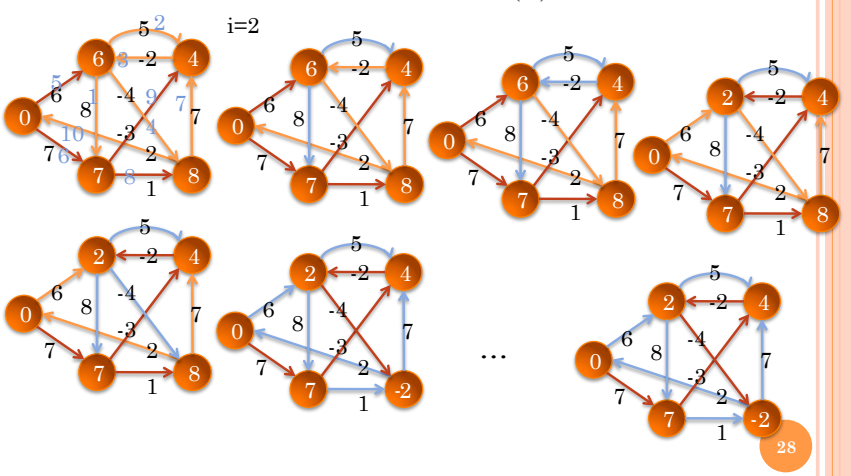
Bellman-Ford Algorithm (2)

x: order of the edges
: visited edges
: predecessor

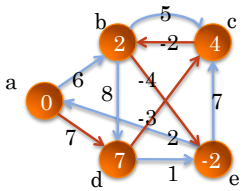


Bellman-Ford Algorithm (3)

x: order of the edges
: visited edges
: predecessor



Bellman-Ford Algorithm (4)



Forwarding table of *a*:

b *(a,d)* as the shortest path is *a-d-c-b*
c *(a,d)* as the shortest path is *a-d-c*
d *(a,d)*
e *(a,d)* as the shortest path is *a-d-c-b-e*

Dijkstra Algorithm (1)

- *G=(V,E)* directed with non negative edge weights
- It partitions the nodes of *G:* nodes whose shortest path from *s* has already been found (*S*) and all the other nodes (*V-S*)
- Greedy algorithm

- At each step, let *u* be the node in *V-S* with minimum value of *d*; add *u* to *S* and relax all the edges outcoming from *u*
- Keep *V-S* in a priority queue (e.g. min heap)

Dijkstra Algorithm (2)

- For each *v* initialize *d(v)* and *p(v)*
- *S*=empty set
- Q=*V*
- While Q is not empty
  - *u*=ExtractMin(Q)
  - *S*=*S U {u}*
  - For each edge *(u,v)*
    outcoming from *u*
    relax *v* wrt *(u,v)*
    Update Q

The time complexity depends on the data structure used to implement Q:

Queue: $O(n^2)$
Heap: $O(m \log n)$

Dijkstra Algorithm (3)

- For each *v* initialize *d(v)* and *p(v)*
- *S*=empty set
- Q=*V*
- While Q is not empty
  - *u*=ExtractMin(Q)
  - *S*=*S U {u}*
  - For each edge *(u,v)*
    outcoming from *u*
    relax *v* wrt *(u,v)*
    Update Q

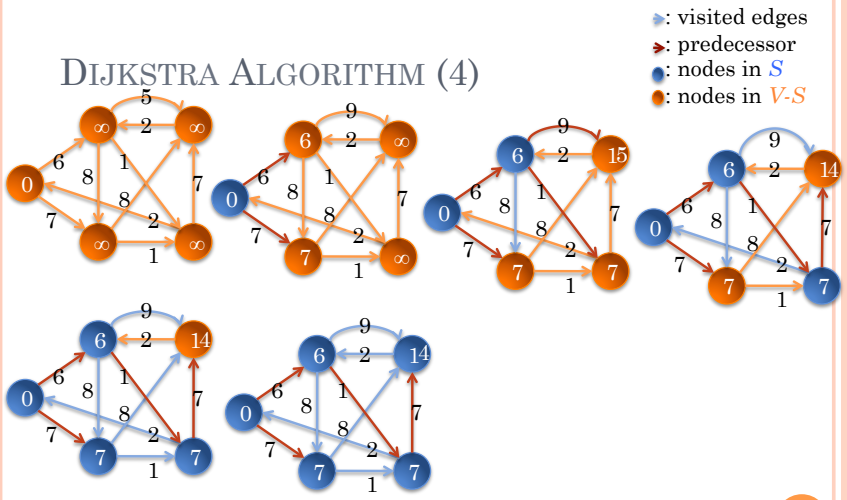
Use heap:
O(n)
O(1)
O(n)

N times
O(log n)
O(1)

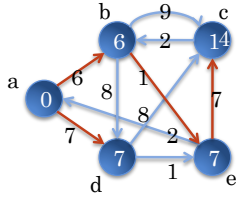
O(deg u) times
O(1)
O(log n)

tot.
$O(n \log n + m \log n) = O(m \log n)$

Dijkstra Algorithm (4)



Dijkstra Algorithm (5)



Forwarding table of a:

b	*(a,b)*
c	*(a,b)* as the shortest path is *a-b-e-c*
d	*(a,d)*
e	*(a,b)* as the shortest path is *a-b-e*

Dijkstra Algorithm (6)

Dijkstra Algorithm is used for dynamic routing protocols.

- Each router:
  - Keeps trace of its incident links
    - Whether the link is up or down
    - The cost on the link (varying in time)
  - Broadcasts the link state (flooding)
    - So, every router has a complete view of the graph
  - Runs Dijkstra Algorithm
    - To compute the shortest paths…
    - …and construct the forwarding table
- An exemple: Open Shortest Path First (OSPF) used in the networks with Internet Protocol (IP)

Floyd-Warshall Algorithm (1)

Sometimes it is not enough to calculate the distances wrt a certain node *s*: we need to know the distances between all pairs of nodes.

- *G=(V,E)* directed with any edge weight.
- Algorithm for the all-to-all shortest path problem
- Trick 1: all edges are in; the non-existing ones have $w=\infty$
- Trick 2: in order to go from *i* to *j* you can either go directly or passing to a third node *k*
- dynamic programming

Floyd-Warshall Algorithm (2)

Algorithm:

- For each node i, initialize $d(i,i)=0$ — $O(n)$
- For each edge (i,j) initialize $d(i,j)=w(i,j)$ — $O(m)$
- For each node k — n times
  - For each node i — n times
    - For each node j — n times

      $dist(i,j)=min\{dist(i,j), dist(i,k)+dist(k,j)\}$ — $O(1)$

- Time Complexity $O(n^3)$

Another Application (1)

Difference Constraints:

- Let be given some tasks with precedence constraints and running lengths, and an unlimited (or limited by n=number of tasks) number of processors :
  - Each task i has:
    - Starting time s_i
    - Time to complete $b_i>0$
    - Constraint $s_j+b_j \le s_i$ if task i can be started after that task j has been completed
  - First task can start at time 0
  - When can we finish last task?

Another Application (2)

This is the Shortest Path Problem on directed acyclic graphs:

- Define a graph having a node for each task
- Insert a dummy node v_0 (that models time 0)
- Insert an arc (v_0,i) for each task node i and let 0 be its weight
- For each precedence constraint $s_j+b_j \le s_i$ insert an arc (j,i) with weight b_j
- Optimal Solution: start each task i at time equal to the length L_i of the longest path from v_0 to i. All the tasks are surely completed within time

  $max_i\ (L_i+b_i)$

Another Application (3)

To transform to the shortest path problem, multiply all lengths and times by -1:

$b_1=2$

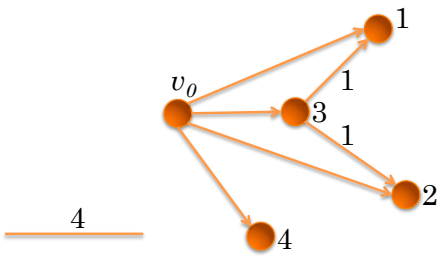
$b_2=3$

$b_3=1$

$b_4=2$

$s_3+b_3 \le s_2$

$s_3+b_3 \le s_1$



Packet-Routing Algorithms on Interconnection Topologies

Packet-Routing on Interconnection Topolgies (1)

- Up to now, in the routing problem we have considered the network as a graph unknown to the nodes and variable in time (faults, varying traffic, etc.)
- Nevertheless, when the network is an interconnection topology (and connects, for example, processors), it is known and fixed in time. Furthermore, efficiency is a primary issue.
- Solutions having stronger properties than the simple shortest path are required.

Packet-Routing on Interconnection Topolgies (2)

Many different types of routing models.

- Here, we will focus on the store-and-forward model (also known as the packet-switching model):
  - Each packet is maintained as an entity that is passed from node to node as it moves through the network
  - A single packet can cross each edge during each step of the routing
  - Depending on the algorithm, we may or may not allow packets to pile up in queues located at each node. When queues are allowed, effort to keep them short.

Packet-Routing on Interconnection Topolgies (3)

- Global controller to precompute routing paths not allowed
- Problem handled using only local control
- A routing problem is called one-to-one if at most one packet must be addressed to every node and each packet has a different destination.
- In contrast, one-to-many and many-to-one

Butterfly Network (1)

Def. Let $N=2^n$ (hence $n=\log N$); the *n*-dimensional Butterfly is a layered graph with:

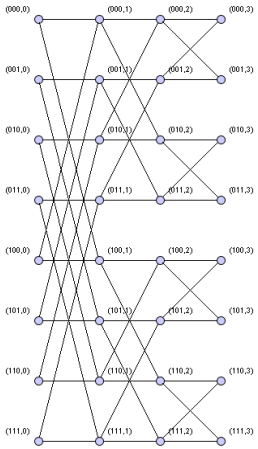
- $N(n+1)$ nodes ($n+1$ layers with 2^n nodes each) and
- $2Nn$ edges.

Nodes:

nodes correspond to pairs (w, i), where:

- i is the *layer* of the node
- w is an n-bit *binary number* that denotes the *row* of the node.

…



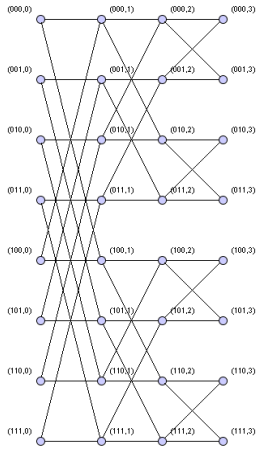
Butterfly Network (2)

def. of n-dimensional butterfly (cntd.)

Edges:

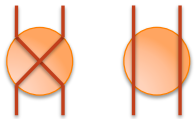
Two nodes (w, i) e (w', i') are linked by an edge iff $i'=i+1$ and either:

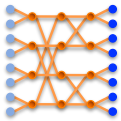
- $w=w'$ (*straight edge*) or
- w and w' differ in precisely the i-th bit (*cross edge*).



Butterfly Network (3)

- The nodes of the Butterfly are *crossbar switches,* i.e. switches with two input and two output values and can assume two states, cross and bar.
- Hence, the butterfly can be seen as a switching network connecting $2N$ ($N=2^n$) input units to $2N$ output units trough a $\log N+1$ layered network, having N nodes each.
- Input and output devices are usually processors and are often omitted in the graphical representations for the sake of simplicity.

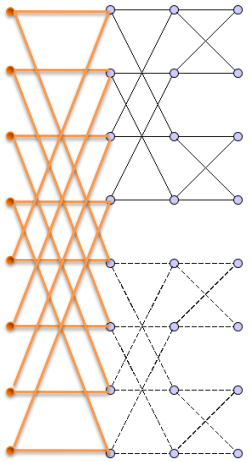




Butterfly Network (4)

- The butterfly has a simple recursive structure:

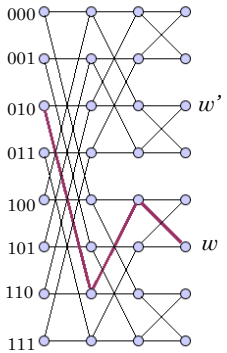
  one n-dim. butterfly contains two (n-1)-dim. butterflies as subgraphs (just remove either the layer *0* nodes or the layer n nodes of the n-dim. butterfly to get two *(n-1)*-dim. butterflies).



Butterfly Network (5)

for each pair of rows w and w', there exists a unique path of length n (known as greedy path) from $(w,0)$ to (w', n);

this path passes through each layer exactly once, using a cross-edge from layer i to layer i+1 (i=0, …,n) iff w and w' differ in the i-th bit



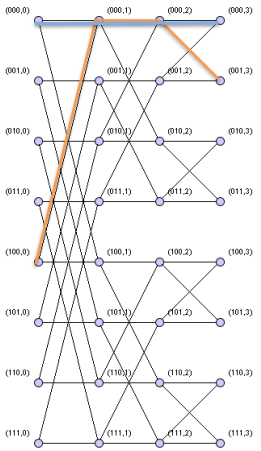
Routing on the Butterfly (1)

Problem of routing N packets from layer 0 to layer n in an n-dimensional butterfly:

- Each node $(u,0)$ on layer 0 of the butterfly contains a packet that is destined for node $(\pi(u), n)$ on layer n, where $\pi:[1, N]\rightarrow[1,N]$ is a permutation.
- In the greedy routing algorithm, each packet is constrained to follow its greedy path.
- When there is only one packet to route, the greedy algorithm performs very well.
- Trouble can arise when many packets have to be routed in parallel, however. …

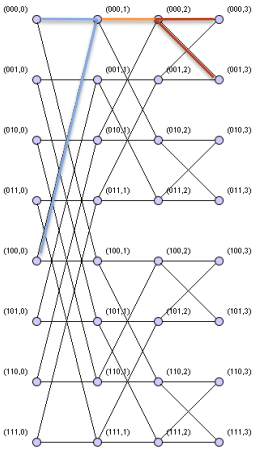
Routing on the Butterfly (2)

- Many greedy paths might pass through a single node or edge.
- Since only one of these packets can use the edge at a time, one of them must be delayed before crossing the edge.
- The butterfly is not able to route each permutation without delays, i.e. is a blocking network
- The congestion problem arising in this example is not overly serious. When N is larger, however, the problem can be much serious. In fact…



Routing on the Butterfly (3)

- Assume for simplicity N odd (but similar results hold when N is even), and consider the edge e=((00…0, (n-1)/2), (00…0,(n+1)/2))
- The node (00…0, (n-1)/2) is the root of a complete binary tree extending to the left having $2^{(n-1)/2}$ leaves
- Analogously to the right
- …



Routing on the Butterfly (4)

- …
- The permutation can be such that each greedy path from a leaf of the left tree arrives to a leaf of the right tree traversing e
- There are $2^{(n-1)/2}=\sqrt{N}/2$ possible such paths, and thus $2^{(n-1)/2}=\sqrt{N}/2$ packets must traverse e. So at least one of them will be delayed by $\sqrt{N}/2-1$ steps.
- It takes at least $n=\log N$ steps to traverse the whole networks and to route a packet to its destination.
- In general, the greedy algorithm can take $\sqrt{N}/2+\log N-1$ steps to route a permutation.

Routing on the Butterfly (5)

Th. *Given any routing problem on an n-dimensional butterfly for which at most one packet starts at each layer 0 node and at most one packet is destined for each layer n node, the greedy algorithm will route all the packets to their destinations in $O(\sqrt{N})$ steps.*

Dim. For simplicity, assume n odd (but the case n even is similar)

- Let e be any edge in layer i, $0<i\leq n$, and define n_i to be the number of greedy paths that traverse e
- $n_i \leq 2^{i-1}$ (left tree) and, similarly, $n_i \leq 2^{n-i}$ (right tree)
- Any packet crossing e can only be delayed by the other n_i-1 packets that want to cross the edge.
- …

Routing on the Butterfly (6)

- …
- As this packet traverses layers 1, 2, …, n, the total delay encountered can be at most:

$$\sum_{i=1}^{n}(n_i-1)=\sum_{i=1}^{(n+1)/2}(n_i-1)+\sum_{i=(n+3)/2}^{n}(n_i-1)\leq \sum_{i=1}^{(n+1)/2}(2^{i-1}-1)+\sum_{i=(n+3)/2}^{n}(2^{n-i}-1)\leq$$

$=(n+1)/2+1$

$=\sum_{j=0}^{(n+1)/2-2}(2^j-1)$

$=\sum_{j=0}^{(n-3)/2}(2^j-1)$

$$\leq 2^{(n+1)/2}+2^{(n-1)/2}-n=O(\sqrt{N})-n=O(\sqrt{N})$$ CVD

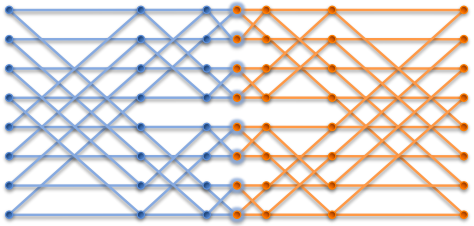
ricordando che $\sum_{j=0}^{k}2k=2^{k+1}-1$

Routing on the Butterfly (7)

- Despite the fact that the greedy routing algorithm performs poorly in the worst case, the greedy algorithm is very useful.
- For many useful classes of permutations, the greedy algorithm runs in n steps, which is optimal and, for most permutations, the greedy algorithm runs in $n + o(n)$ steps.
- As a consequence, the greedy algorithm is widely used in practice.

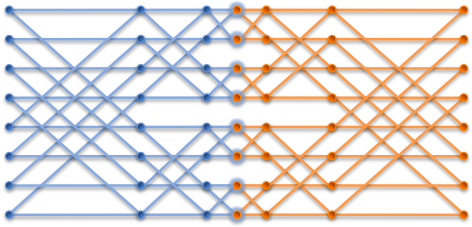
Beneš Network (1)

- A possibility to avoid a routing with delays is providing a non blocking topology.
- Beneš network has this property
- It consists of two back-to-back butterflies



Beneš Network (2)

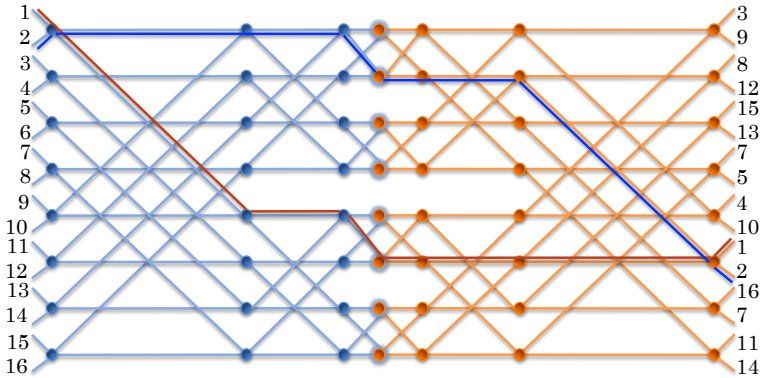
- The n-dimensional Beneš network has $2n+1$ layers, each with 2^n nodes.
- The first and last $n+1$ layers in the network form an n-dimensional Butterfly (the middle layer is shared by these butterflies).
- Not surprisingly, the Beneš network is very similar to the butterfly, in terms of both its computational power and its network structure.



Beneš Network (3)

- The reason for defining the Beneš network is that it is an excellent example of a rearrangeable network.
- **Def.** A network with N inputs and N outputs is said to be rearrangeable if for any one-to-one mapping π of the inputs to the outputs (i.e. for any permutation), we can construct edge-disjoint paths in the network linking the i-th input to the $\pi(i)$-th output for $1 \le i \le N$.
- In the case of the n-dimensional Beneš network, we can have *two* inputs for each layer 0 node and *two* outputs for every layer $2n$ node, and still connect every permutation of inputs to outputs with edge-disjoint paths.
- Hence, in this case, $N=2^{n+1}$

Beneš Network (4)



Beneš Network (5)

It seems extraordinary that we can find edge-disjoint paths for any permutation. Nevertheless, the result is true, and it is even fairly easy to prove, as we show in the following:

Th. *Given any one-to-one mapping π of 2^{n+1} inputs to 2^{n+1} outputs on an r-dimensional Beneš network, there is a set of edge- disjoint paths from the inputs to the outputs connecting input i to output π(i) for $1 \le i \le 2^{n+1}$.*

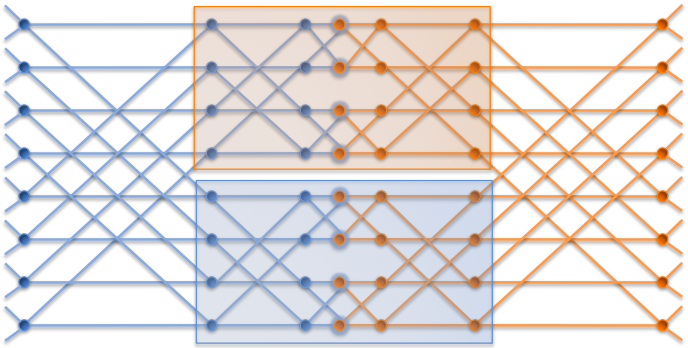
Proof. By induction on *n*.

- If *n=1*, the Beneš network consists of a single node (i.e. a single 2x2 switch) and the result is obvious.
- Induction: assume that the result is true for an *(n-1)*-dimensional Beneš network
- …

Beneš Network (6)

Proof of the rearangeability of the Beneš network (cntd.)

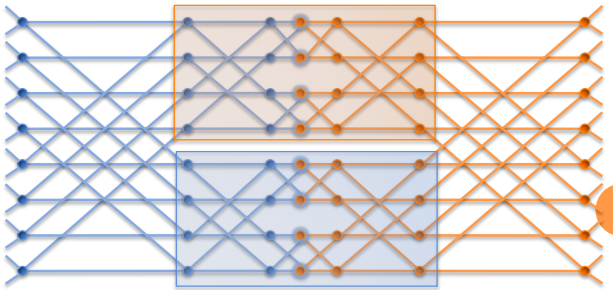
- Key observation: the middle *2n-1* layers of an *n*-dimensional Beneš network comprise two *(n-1)*-dimensional Beneš networks



Beneš Network (7)

Proof of the rearangeability of the Beneš network (cntd.)

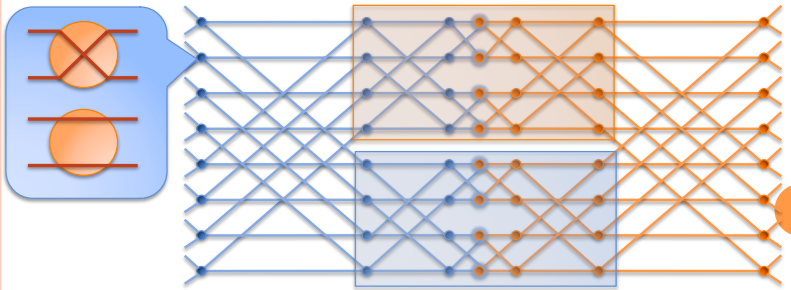
- Hence, for each path, it will be sufficient to decide whether it is to be routed through the upper sub-Beneš network or through the lower sub-Beneš network.



Beneš Network (8)

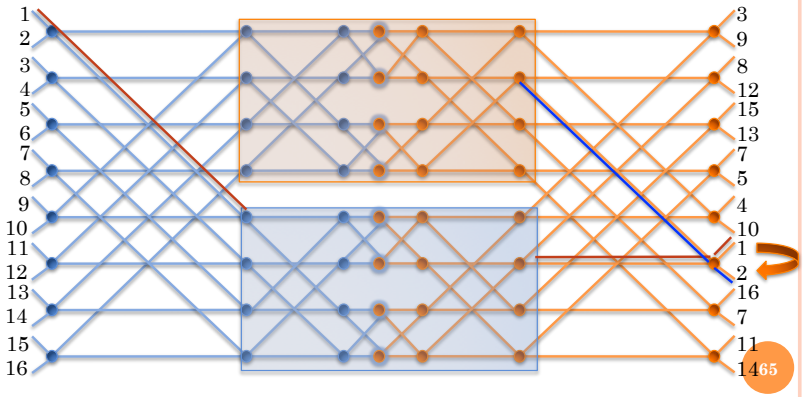
Proof of the rearangeability of the Beneš network (cntd.)

- The only constraints we have to consider to decide whether paths use the upper or lower subnetworks are that paths from inputs $2i - 1$ and $2i$ must use different subnetworks for $1 \le i \le 2n$, and that paths to outputs $2i - 1$ and $2i$ must use different subnetworks.
- …easy…



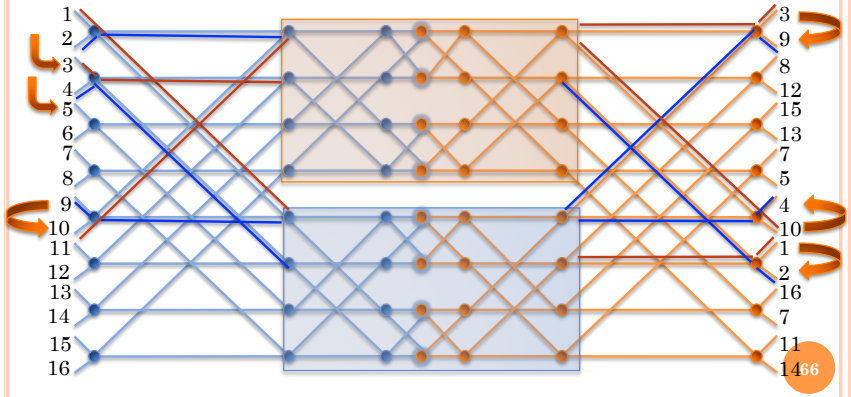
Beneš Network (9)

Proof of the rearangeability of the Beneš network (cntd.)



Beneš Network (10)

Proof of the rearangeability of the Beneš network (cntd.)



And so on…

Beneš Network (11)

Proof of the rearangeability of the Beneš network (cntd.)

- We start by routing the first path through the upper sub-network.
- We next satisfy the constraint generated at the output by routing the corresponding path through the lower sub-network.
- We keep on going back and forth through the network, satisfying constraints at the inputs by routing through the upper sub-network and satisfying constraints at the outputs by routing through the lower sub-network.
- …

Beneš Network (12)

Proof of the rearangeability of the Beneš network (cntd.)

- Eventually, we will close the loop by routing a path through the lower sub-network (in response to an output constraint) that shares an input switch with the first path that was routed.
- If any additional paths needs to be routed, we continue as before, starting over again with an arbitrary unrouted path.
- In this way, all paths can be assigned to the upper or lower sub-networks without conflict.

Beneš Network (13)

Proof of the rearangeability of the Beneš network (cntd.)

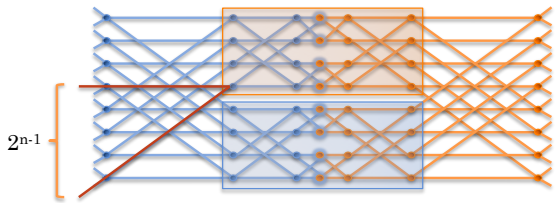
- This algorithm is called looping algorithm.
- It is easy to see that all paths can be assigned to the upper or lower sub-networks without conflict:
- By construction, if we start going to the upper sub-network, we will arrive to the corresponding output in the upper sub-network and we will leave it to the lower sub-network, and so on.
- For parity reason, when a loop is close, we will correctly arrive from the right sub-network.
- The remainder of the path routing and switch setting is handled by induction in the sub-networks.

Beneš Network (14)

- In the case that each layer 0 node of the n-dim. *Beneš* network has just one input and each layer $2n$ node has just one output, then the paths from the inputs to the outputs can be constructed so as to be node- disjoint (instead of only edge-disjoint):
- …

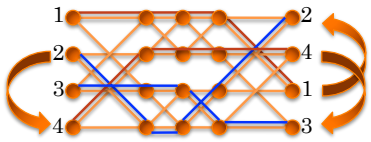
Beneš Network (15)

- Th. *Given any one-to-one mapping of* π *of* 2^n *inputs to* 2^n *outputs in an n-dim. Beneš network, there is a set of node-disjoint paths from the inputs to the outputs connecting input* i *to output* $\pi(i)$ *for* $1 \le i \le 2^n$.
- Proof. Identical to the previous one, but the paths needing to use different Beneš networks are now i and $i+2^{n-1}$, $1 \le i \le 2^{n-1}$ (and not $2i-1$ and $2i$).



Beneš Network (16)

- Exemple: n=2, hence 2^{n-1}=2



BENEŠ NETWORK (17)

- The only drawback to these theorems is that we do not know how to set the switches on-line. In other words, each switch needs to be told what to do by a **global control** that has knowledge of the permutation being routed.
- There exist numerous methods for overcoming this difficulty (not studied here).