Paths, circuits and weighted graphs

Olli-Pekka Hämäläinen

Path, simple path and cycle

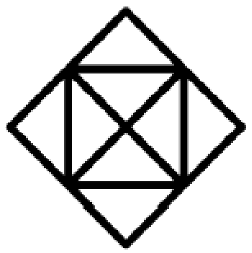
- Modelling of movements inside graphs is the key reason for graph analysis in many processes
- Definitions of some terms:
- *Path* = any route from node i to node j; allows wandering back and forth in the graph – nodes and edges may be repeated ad infinitum
- *Simple path* = efficient route from node i to node j; neither edges nor nodes can be repeated
- *Cycle* = path which ends in the same node where it began; neither edges nor nodes can be repeated

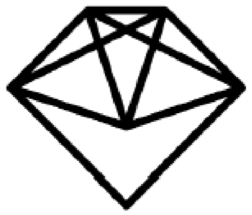
Euler path and circuit

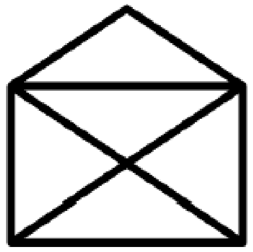
- A path that goes through all the **edges** in the graph exactly once is called a *Euler path*
- Typical in many optimization applications
  - Example: garbage truck goes through all the streets in town; optimal route = path where all streets are driven through just once
- If the initial and terminal nodes are the same node, this is a special case of Euler path called *Euler circuit*
- The existence of Euler path & circuit can be tested fairly simply by examining the degrees of nodes

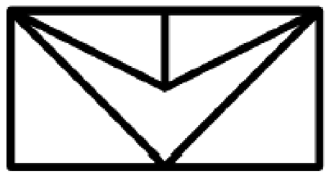
Euler path and circuit

- If the degrees of all nodes are even, a Euler circuit (and also path) can be found
- If the degree of all but 2 nodes are even, a Euler path can be found (but no circuit)
  - Initial and terminal nodes of the path are odd-degree nodes
- Is it possible to find a Euler path and/or circuit for the graphs shown below?


(a)

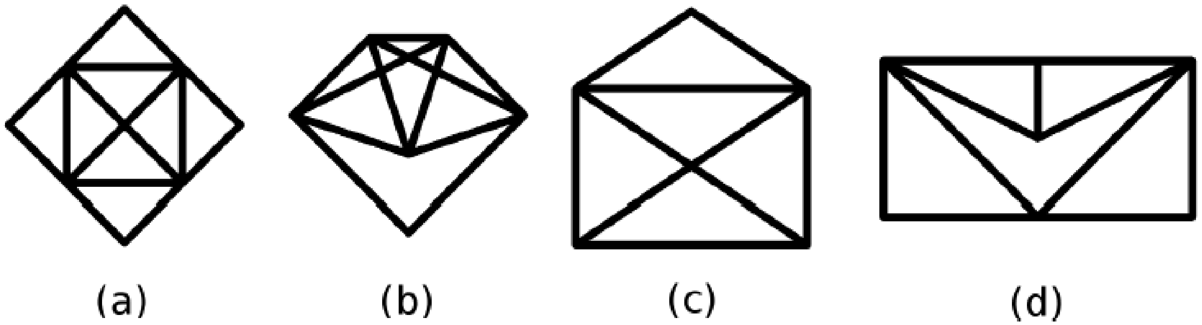

(b)


(c)


(d)

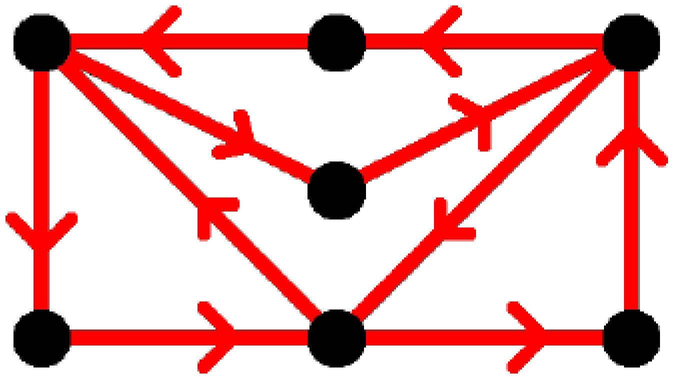
Euler path and circuit

- If the degrees of all nodes are even, a Euler circuit (and also path) can be found
- If the degree of all but 2 nodes are even, a Euler path can be found (but no circuit)
  - Initial and terminal nodes of the path are odd-degree nodes
- Is it possible to find a Euler path and/or circuit for the graphs shown below?
  - YES: b (circuit), c (path), d (path) NO: a



Euler path and circuit

- Concepts of Euler path and circuit are also valid for directed graphs; we can define clear rules which must be fulfilled in order for these to exist
- Euler circuit exists if and only if the indegree of each node matches its outdegree
- Euler path exists if and only if exactly one node has an indegree of 1 more than its outdegree and exactly one node has it the other way round
  - Initial and terminal nodes of the path are these 2 nodes
  - For all other nodes, outdegree = indegree

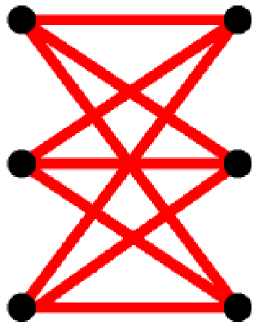


Hamiltonian path and circuit

- A path which goes through all the **nodes** in the graph exactly just once is called a *Hamiltonian path*
- Respectively, if the initial and terminal nodes are one and same node, the path is called a *Hamiltonian circuit*
- In practical optimization applications even more often of interest than Euler circuits:
  - Travelling Salesman Problems (TSP)
  - Optimization of logistics (postal packages, bus routes etc.)
  - Design of microchips
  - Error detection of CAN bus modules in automobiles

Hamiltonian path and circuit

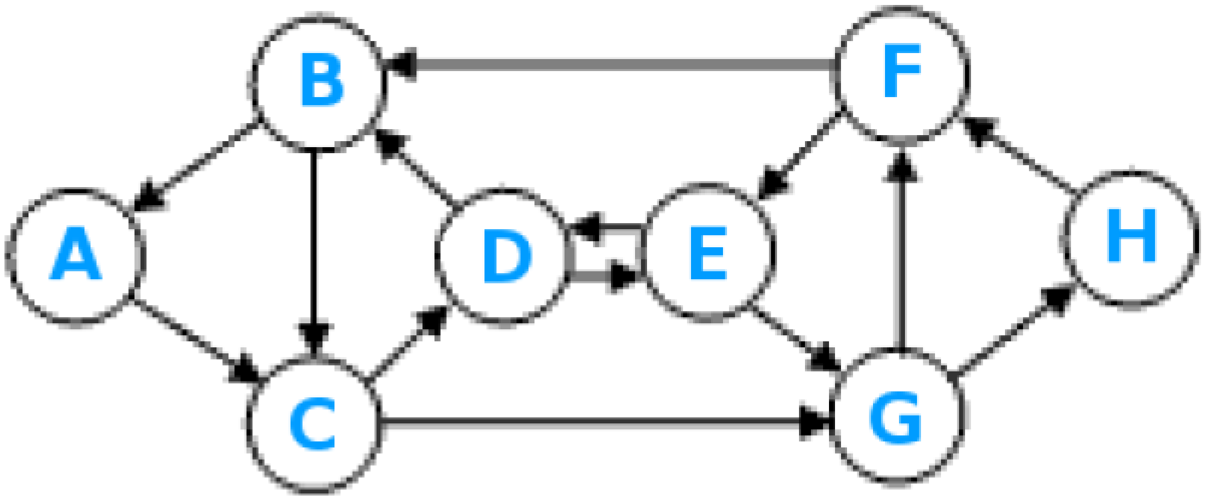
- Unlike in the case of Euler, there is no general rule for checking out if the graph has a Hamiltonian path/circuit
  - For this reason, many similar problems have risen to glory and are a topic of diligent research among mathematicians
- We know that if there are n nodes in the graph and each of them has a degree of at least $\frac{1}{2}n$, the graph must for sure have a Hamiltonian circuit
  - NOTE! This is not an "if and only if" -type of rule; the circuit can have a Hamiltonian circuit even if this fails



Example: this graph has to have a Hamiltonian circuit according to the rule above.

Hamiltonian path and circuit

- We can also look for Hamiltonian path/circuit in directed graphs
- A mathematical rule or algorithm which would prove the existence of a Hamiltonian path for directed graphs doesn’t exist (at least according to current knowledge), either

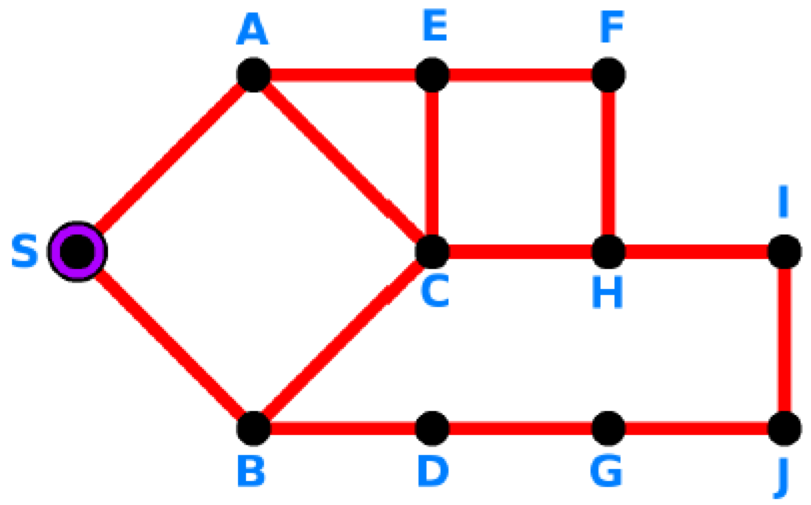


Shortest path

- If we want to move from node i to node j in the graph, there are often many (even simple) paths to choose from
- Optimization problem: which one of these is the *shortest*?
  - "Shortest" = contains least amount of edges
- There's a simple algorithm for finding it:
  1) Decide starting point
  2) Find the node(s) nearest to the starting point and label them based on their distance from starting point
  3) Save length of path until this node & information on its predecessor (= from which node we came from)
  4) Continue; now we can pass through previously labeled nodes
  5) Repeat steps 2-4 until all nodes are labeled
- Algorithm can be run either in graph or in matrix form

Shortest path

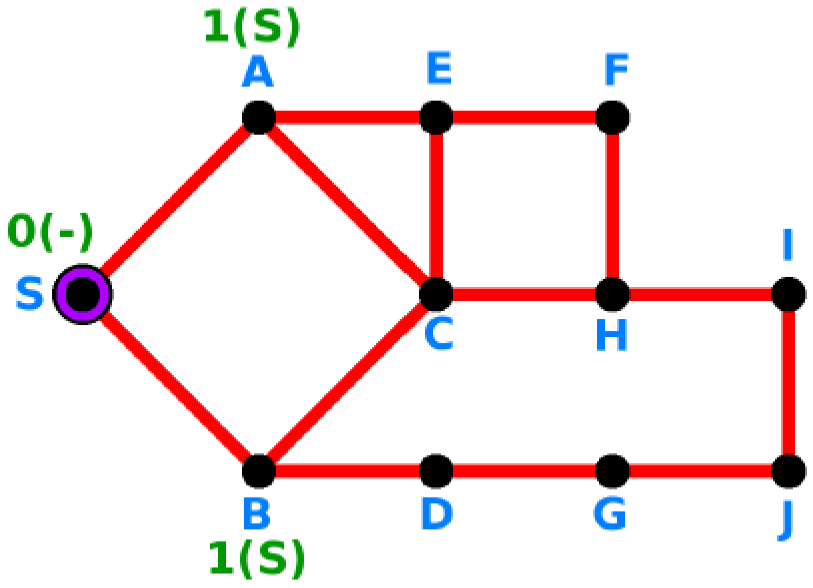
- Let’s consider the following graph and use S as a starting point



- Define the shortest paths from S to all other nodes

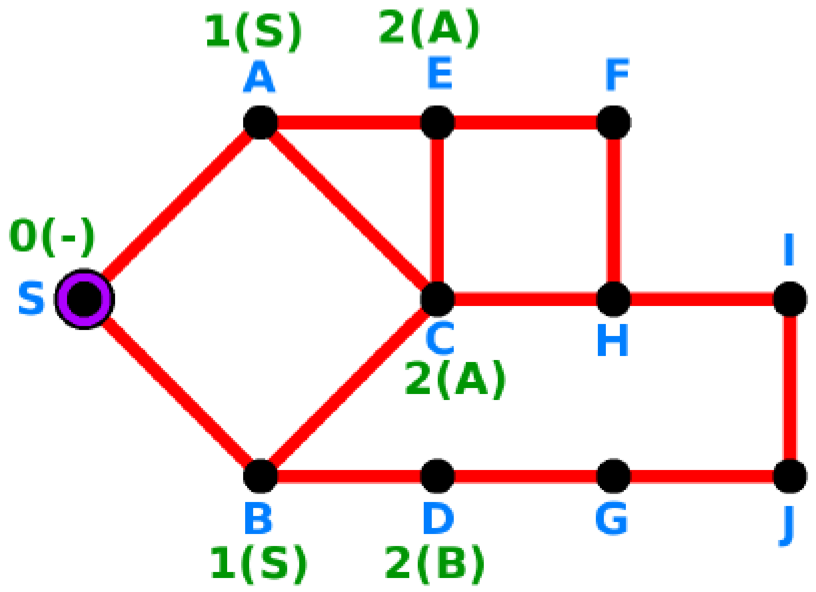
Shortest path

- We can get from S to either A or B with just 1 step
- Label both A and B. Mark total length of path and predecessor (in parentheses)



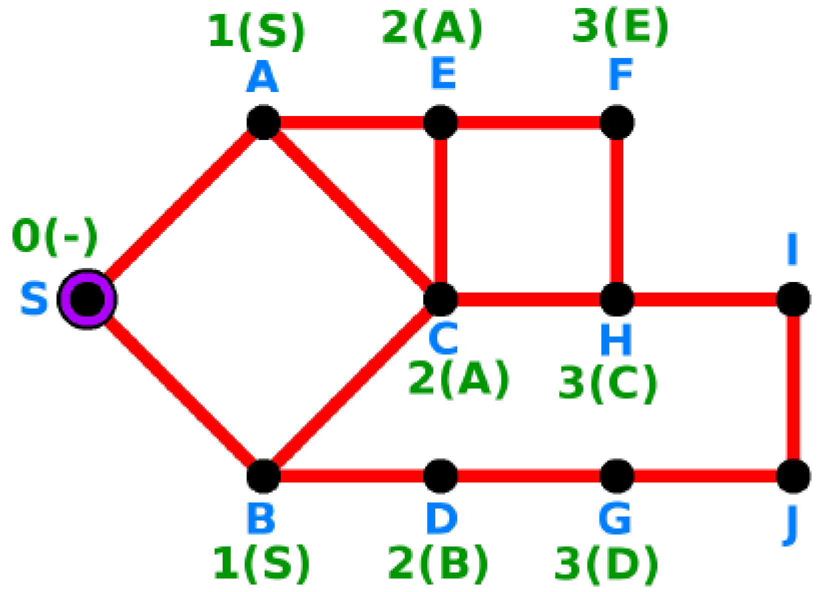
Shortest path

- From A we can go to either C or E and from B either to C or D. The length of these paths (from S) is 2.
- Label E, C and D.



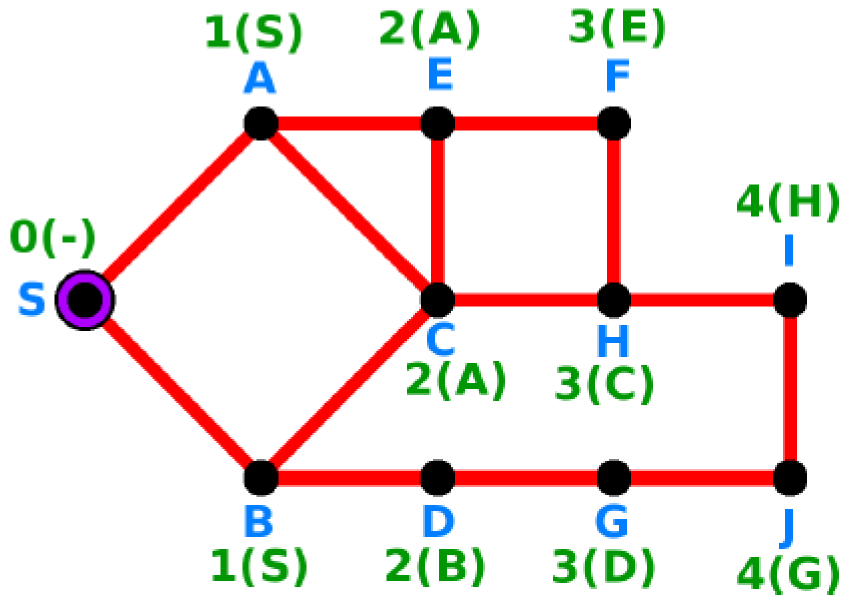
Shortest path

- From these nodes we can go forward to nodes F, H and G, where the total length of each path is 3.
- Label these nodes, too.



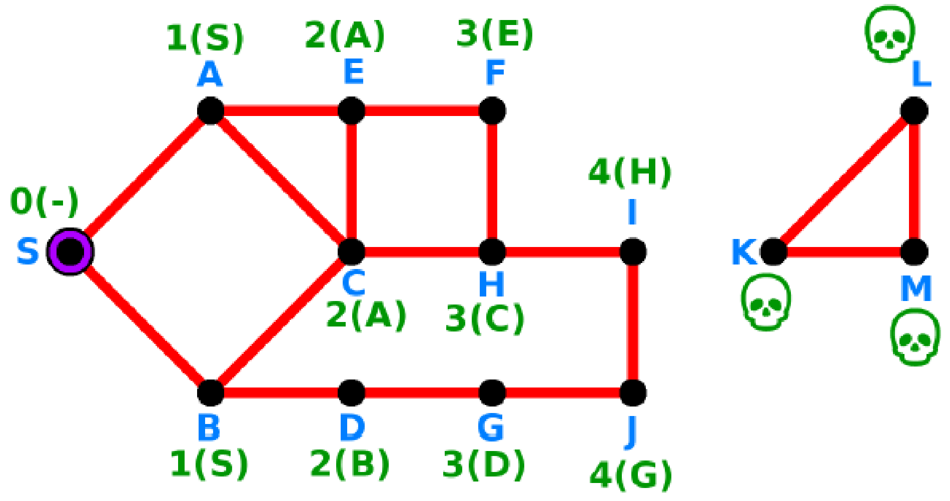
Shortest path

- Next step gets us to nodes I and J, where the total length of each path is 4.
- After labeling these nodes the algorithm ends, because we have no more nodes to label



Shortest path

- The algorithm continues forward until there are no more unlabeled nodes adjacent to labeled ones
- If the graph is not connected, this cutoff limit is reached as soon as the "island" where the starting point lies on has been completely labeled
- For example, in case below the nodes K, M and L would be deemed *unreachable*



Weighted graphs

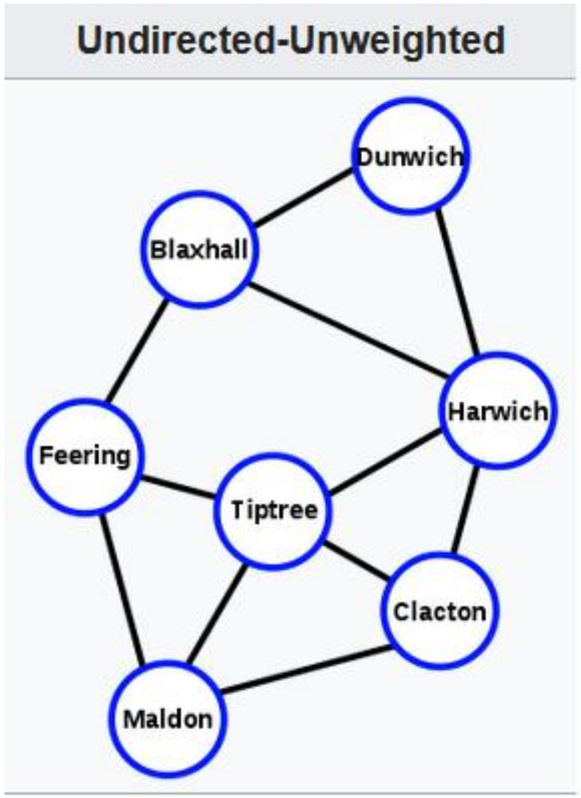
- The previous example might have looked stupid, because the graph was kind of a simple one – why do we need an algorithm in the first place?
- The rationality of the algorithm is lifted to a new level when we take into account that our graphs may be *weighted*
- Edges of the graph are given number values, which represent their “weight”
- Weight can be interpreted as length or price of the edge
  - The greater the weight, the more disadvantageous
  - …although in some cases we may aim to maximize the weight of the final path (example: bus line map; weights = assumed number of passengers living along certain roads)

Weighted graphs

- Weights are usually set as edge weights
- This doesn’t spark much changes in modelling the graph: instead of 1s, we just put the weights of edges in adjacency matrix and incidence matrix!
- Likewise, in place of 0s, we often set infinities (justification: going along a non-existing route requires infinite amount of work)
- Now one good side of the incidence matrix shows: we can separate possible parallel edges to have different weights (if that’s the case)
- …in reality this benefit is not huge, because for optimization purposes we want to simplify the graph by removing extra edges anyway (save the lightest one!)

Adjacency matrix of an unweighted & undirected graph

- Example: Road network - unweighted

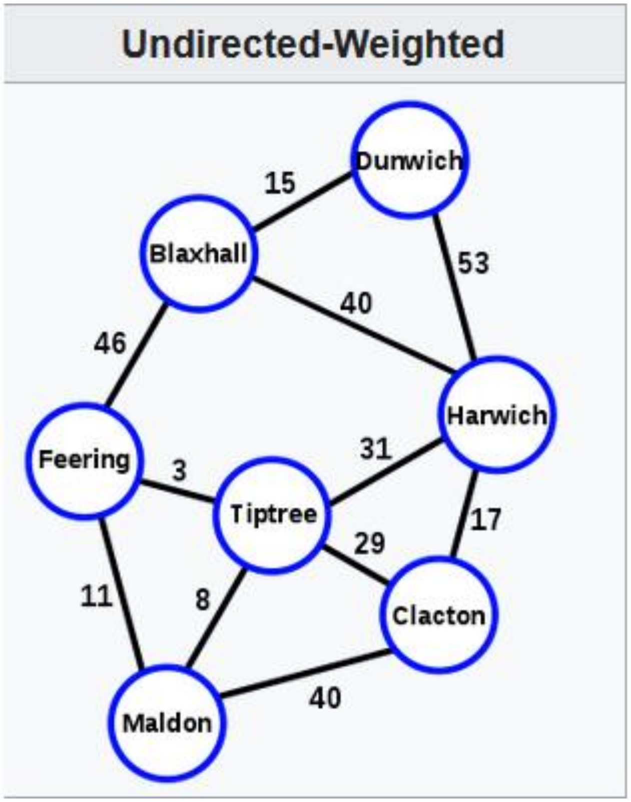


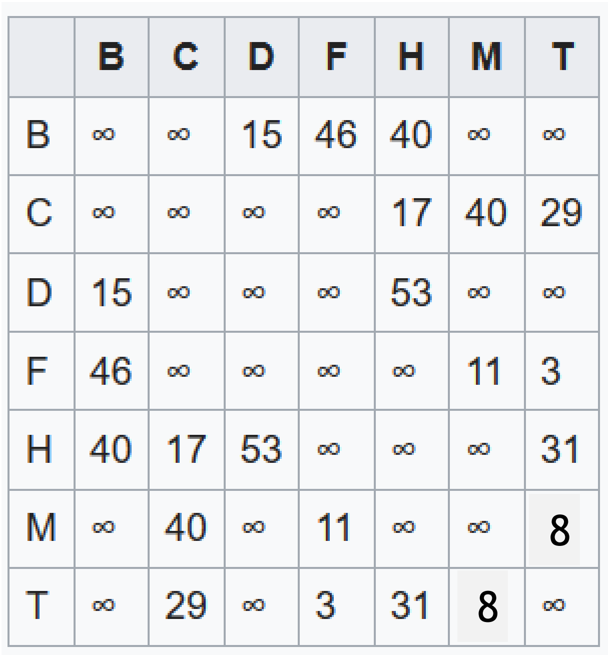
	B	C	D	F	H	M	T
B	0	0	1	1	1	0	0
C	0	0	0	0	1	1	1
D	1	0	0	0	1	0	0
F	1	0	0	0	0	1	1
H	1	1	1	0	0	0	1
M	0	1	0	1	0	0	1
T	0	1	0	1	1	1	0

Just ones and zeros; symmetric

Adjacency matrix of a weighted & undirected graph

- Example: Road network - weighted



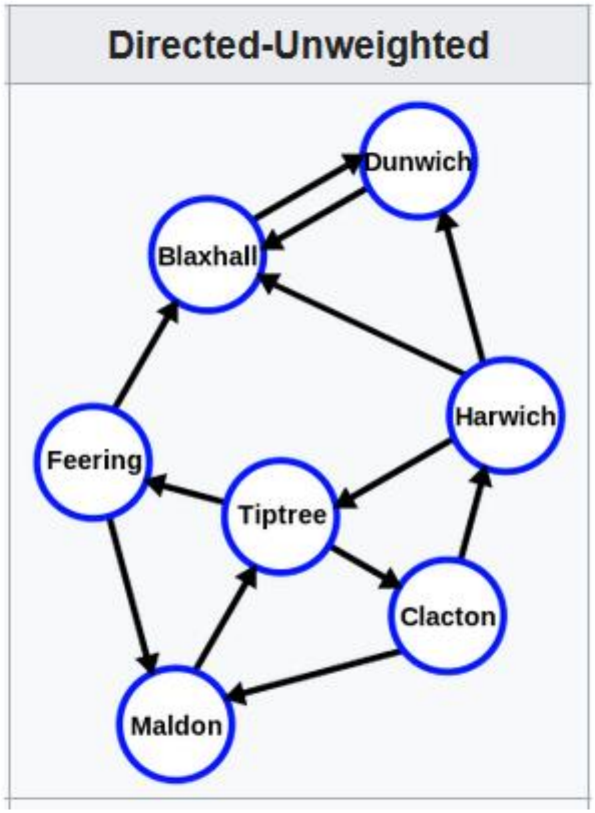


	B	C	D	F	H	M	T
B	∞	∞	15	46	40	∞	∞
C	∞	∞	∞	∞	17	40	29
D	15	∞	∞	∞	53	∞	∞
F	46	∞	∞	∞	∞	11	3
H	40	17	53	∞	∞	∞	31
M	∞	40	∞	11	∞	∞	8
T	∞	29	∞	3	31	8	∞

Weights in place of 1s, infinities in place of 0s; symmetric

Adjacency matrix of an unweighted & directed graph

- Example: Road network – directed & unweighted



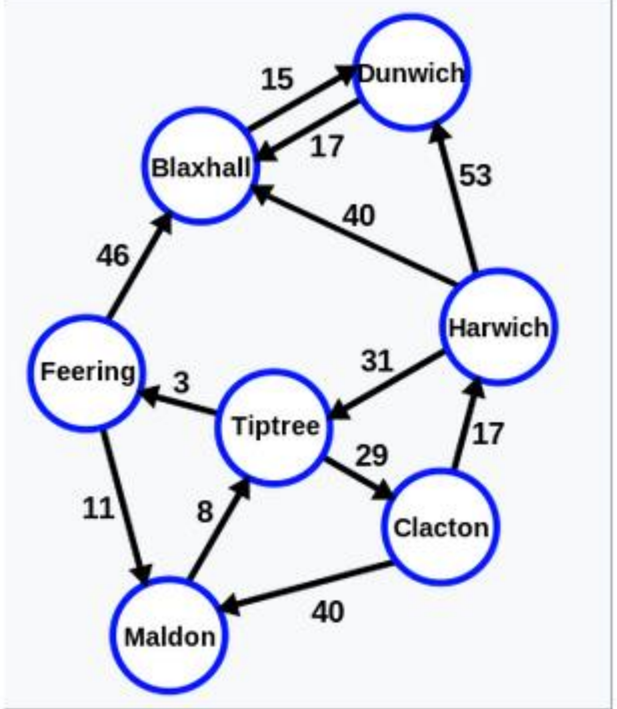
	B	C	D	F	H	M	T
B	0	0	1	0	0	0	0
C	0	0	0	0	1	1	0
D	1	0	0	0	0	0	0
F	1	0	0	0	0	1	0
H	1	0	1	0	0	0	1
M	0	0	0	0	0	0	1
T	0	1	0	1	0	0	0

Just ones and zeros;
not symmetric

Adjacency matrix of a weighted & directed graph

- Example: Road network – directed & weighted

Directed-Weighted



	B	C	D	F	H	M	T
B	∞	∞	15	∞	∞	∞	∞
C	∞	∞	∞	∞	17	40	∞
D	17	∞	∞	∞	∞	∞	∞
F	46	∞	∞	∞	∞	11	∞
H	40	∞	53	∞	∞	∞	31
M	∞	∞	∞	∞	∞	∞	8
T	∞	29	∞	3	∞	∞	∞

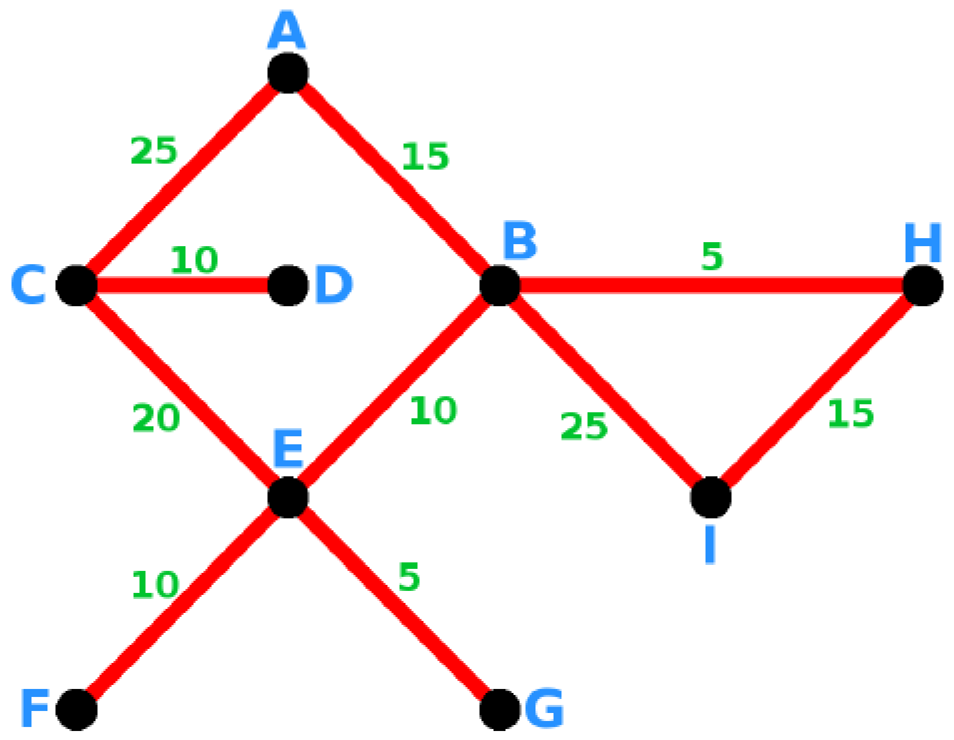
Weights in place of 1s, infinities in place of 0s; not symmetric

Lightest path

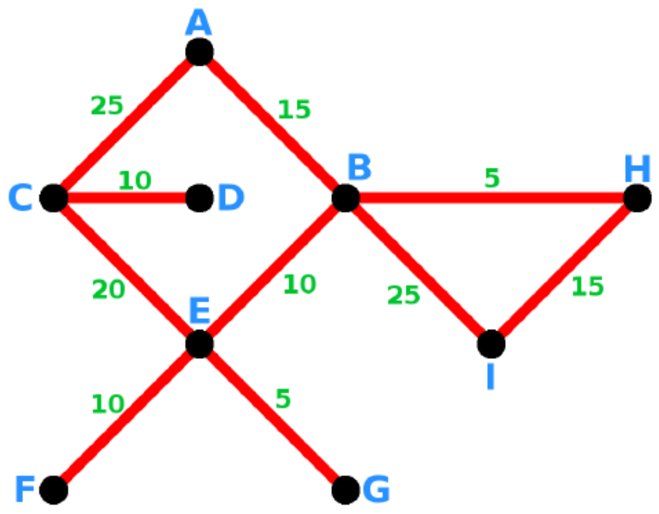
- Now the lengths of paths can be thought as weights
- The shortest path is the one that is lightest – so, has the lowest total weight
- This kind of modelling is again one step closer to network problems of the real world
- For example, route suggestions of Google maps are based on similar optimization (slightly more complex when it comes to algorithms, but the general idea):
  - Shortest route = weights are distances in kilometers
  - Fastest route = weights are estimated travel times (speed limits and possible traffic situations have an effect)
- For weighted graphs, a corresponding shortest path algorithm is called *Dijkstra's algorithm*
  - Let's learn to do this in matrix form!

Dijkstra’s algorithm

- Let’s go through the graph below, starting from A.
- Find out the adjacency matrix and extend it with two more columns: sum column and predecessor column
  - Labeled nodes = rows that have green



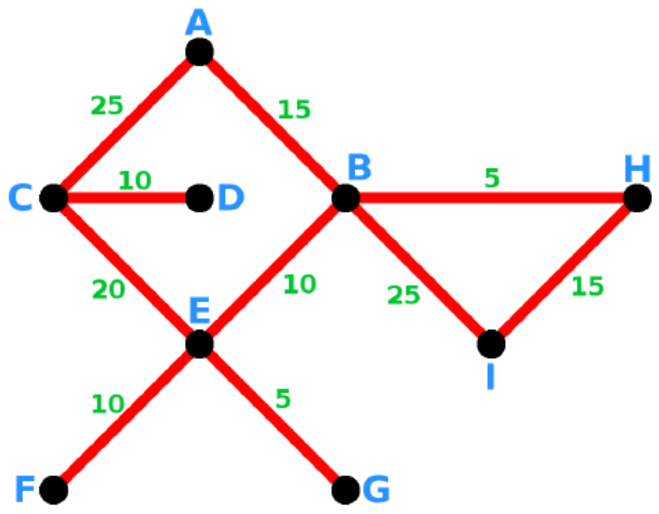
Dijkstra's algorithm



	A	B	C	D	E	F	G	H	I	∑	P
A	🟥	15	25							0	-
B	15	🟥			10			5	25		
C	25		🟥	10	20						
D			10	🟥							
E		10	20		🟥	10	5				
F					10	🟥					
G					5		🟥				
H		5						🟥	15		
I		25						15	🟥		

Starting from A, we can go to B or C. Distance to B is shorter, so label B. Sum = 15 and predecessor A.

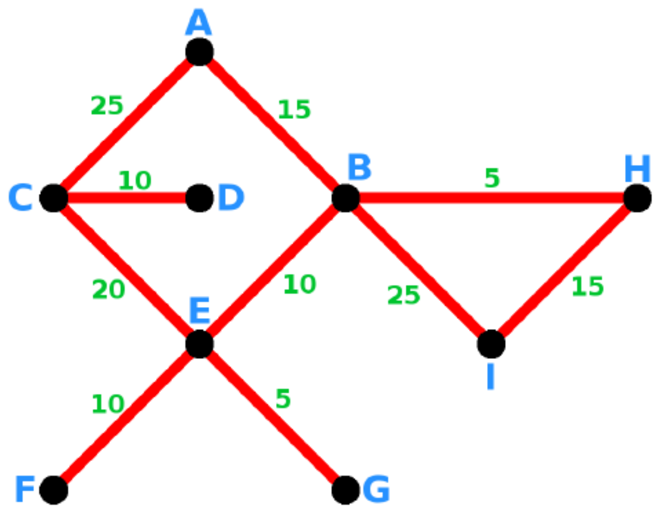
Dijkstra's algorithm



	A	B	C	D	E	F	G	H	I	∑	P
A	🟥	🟩 15	25							0	-
B	🟩 15	🟥			10			5	25	15	A
C	25		🟥	10	20						
D			10	🟥							
E		10	20		🟥	10	5				
F					10	🟥					
G					5		🟥				
H		5						🟥	15		
I		25						15	🟥		

Now the shortest path is ABH, because its length is 15+5 = 20.
So, we label H.
Sum = 20 and predecessor is B.

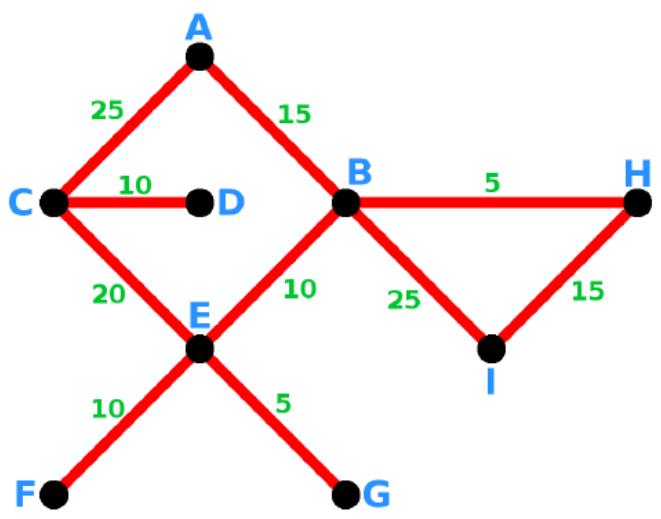
Dijkstra’s algorithm



	A	B	C	D	E	F	G	H	I	∑	P
A	🟥	15	25							0	-
B	15	🟥			10			5	25	15	A
C	25		🟥	10	20						
D			10	🟥							
E		10	20		🟥	10	5				
F					10	🟥					
G					5		🟥				
H		5						🟥	15	20	B
I		25						15	🟥		

Next the shortest paths are AC and ABE (25 and 15+10 = 25). Label C (Sum = 25, predecessor A) and E (Sum = 25, predecessor B).

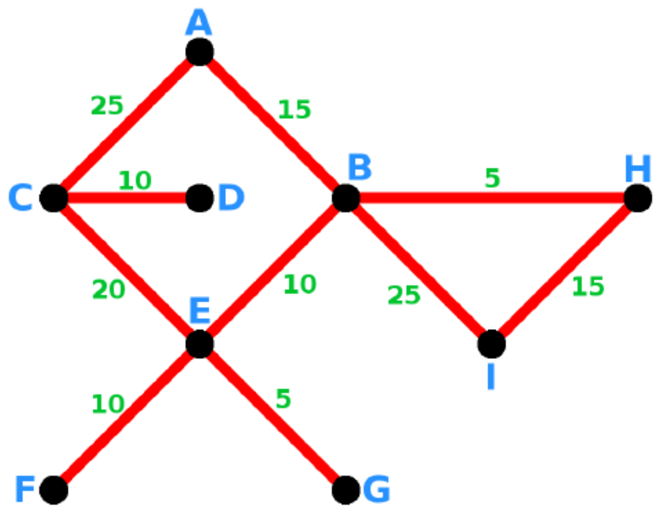
Dijkstra's algorithm



	A	B	C	D	E	F	G	H	I	∑	P
A	🟥	15	25							0	-
B	15	🟥			10			5	25	15	A
C	25		🟥	10	20					25	A
D			10	🟥							
E		10	20		🟥	10	5			25	B
F					10	🟥					
G					5		🟥				
H		5						🟥	15	20	B
I		25						15	🟥		

Next shortest path is ABEG (25+5 = 30). Label G. Sum = 30, predecessor is E.

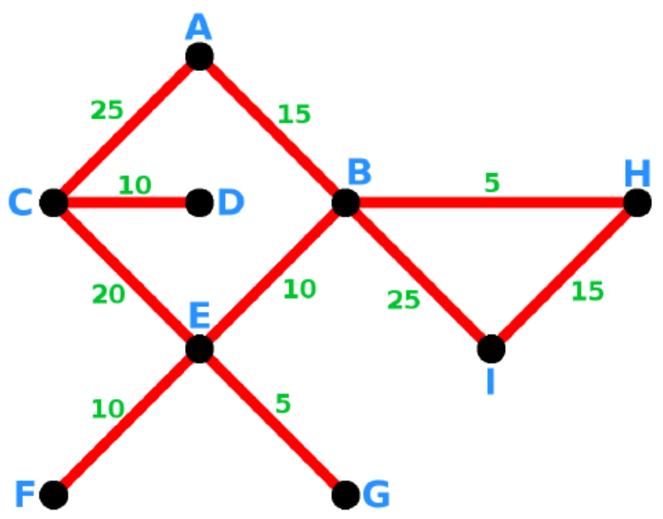
Dijkstra’s algorithm



	A	B	C	D	E	F	G	H	I	∑	P
A	🟥	15	25							0	-
B	15	🟥			10			5	25	15	A
C	25		🟥	10	20					25	A
D			10	🟥							
E		10	20		🟥	10	5			25	B
F					10	🟥					
G					5		🟥			30	E
H		5						🟥	15	20	B
I		25						15	🟥		

Now we only have nodes D, F and I left. D can be only reached via C (25+10 = 35), so label D with Sum = 35 and predecessor C. Likewise, F can only be reached via E (25+10) = 35. Label F, Sum = 35, predecessor = E.

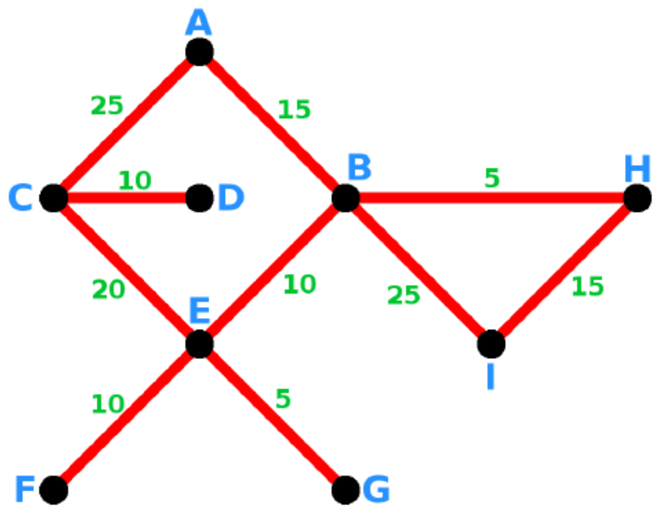
Dijkstra's algorithm



	A	B	C	D	E	F	G	H	I	∑	P
A	🟥	15	25							0	-
B	15	🟥			10			5	25	15	A
C	25		🟥	10	20					25	A
D			10	🟥						35	C
E		10	20		🟥	10	5			25	B
F					10	🟥				35	E
G					5		🟥			30	E
H		5						🟥	15	20	B
I		25						15	🟥		

Now the only node that is left unlabeled is I. I can be reached via B (25+15 = 40) or H (15+20 = 35). The latter is shorter, so label I, Sum = 35 and predecessor = H.

Dijkstra’s algorithm



	A	B	C	D	E	F	G	H	I	∑	P
A	🟥	15	25							0	-
B	15	🟥			10			5	25	15	A
C	25		🟥	10	20					25	A
D			10	🟥						35	C
E		10	20		🟥	10	5			25	B
F					10	🟥				35	E
G					5		🟥			30	E
H		5						🟥	15	20	B
I		25						15	🟥	35	H

Ready! The uncolored elements of the matrix represent edges that will not be used.

PERT

- Graphs can be very informative when planning projects and evaluating their completion time
- One method that employs a graph for this task is called *PERT (Program Evaluation and Review Technique)*
- In PERT we divide our project to tasks and estimate their a) duration and b) list the preceding tasks that have to be completed in order to begin said task
- A PERT graph can be drawn in two ways:
  1) Nodes are numbers that represent the project phase, tasks are represented by edges whose weight is the duration (# of phases = # of tasks + 1)
  2) Nodes are tasks, nodes are weighted by their duration
- Both ways work and are used, but 2) is maybe a bit more intuitive, since in 1) the “phases” are not that easy to link

Example: Black Friday campaign

- A large home electronics chain is preparing for Black Friday, which is only a month away. Store relies in traditional paper advertisements, and they are in a hurry: they haven't done anything yet. The manager is worried whether they will be able to get the campaign ready to launch in time.
- Before they can launch it, they'd need to decide what to put on sale (both department managers, who focus on what they think would be in high demand, as well as buyers, who focus on what's available for cheap), price the items for the ad, prepare the artwork, prepare the descriptions and design the advertisement. And work doesn't stop there: they still have to compile the mailing list, print the labels for mailing, print the advertisement, affix the labels to advertisements and then deliver the ads. All this takes time.

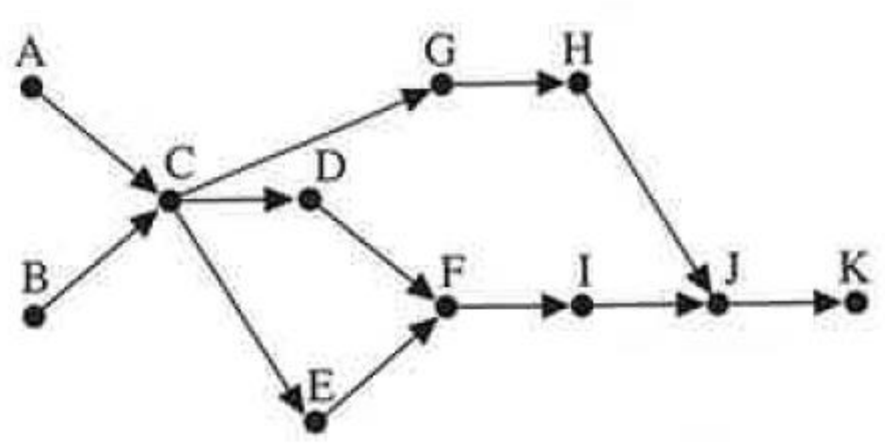
Example: Black Friday campaign

- Divide to tasks and collate in a table:

Task	Task description	Time (days)	Preceding?
A	Choose items (managers)	3	None
B	Choose items (buyers)	2	None
C	Price items	2	A,B
D	Prepare artwork	4	C
E	Prepare descriptions	3	C
F	Design advertisement	2	D,E
G	Compile mailing list	3	C
H	Print labels	1	G
I	Print advertisements	5	F
J	Affix labels	2	H,I
K	Deliver advertisements	10	J

Example: Black Friday campaign

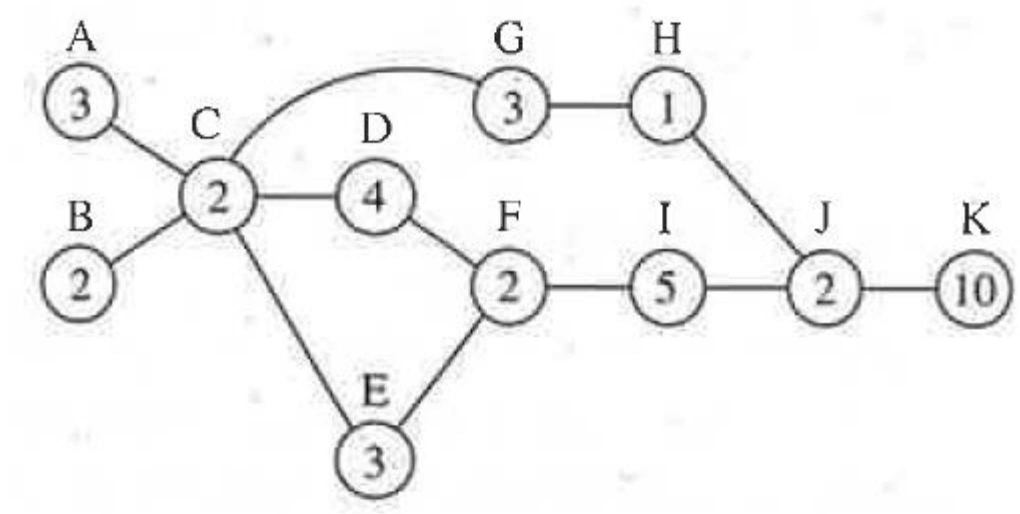
- Let’s sketch a diagram of the process:



- When drawn this way, all arrows go from left to right as the project progresses. Let’s omit the arrowheads for simplicity and add the durations (let’s use the method where we set durations to node weights for a change).

Example: Black Friday campaign

- New graph with weights:



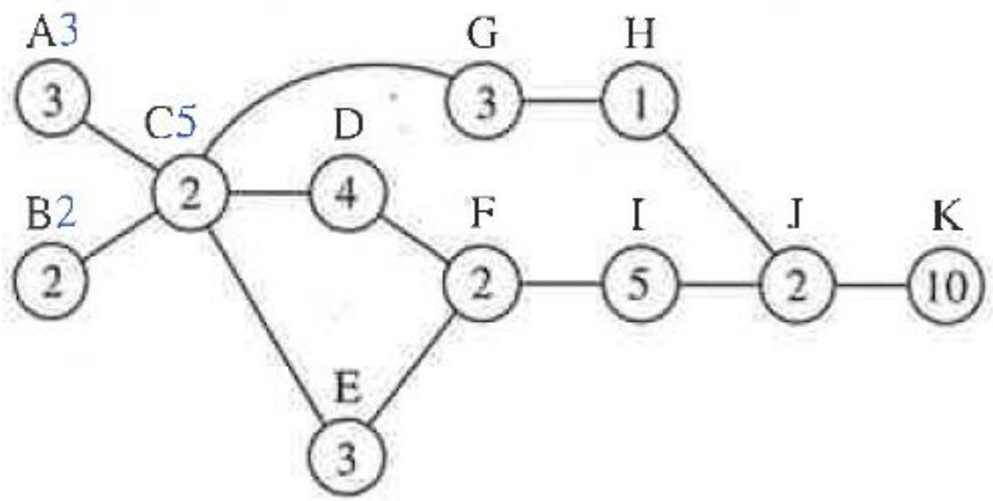
- This provides us a good illustration. But what is the shortest time in which we can complete the project?
- In order to find it out, we'll use a tool called *critical path method*

Critical path method

- In critical path method, we start from the initial node i and calculate the maximum length of the path all the way up until our terminal node j
  - Why maximum? Well, because all tasks must be completed
- The solution tells us two things:
  - How long will the project take in best-case-scenario
  - What is the critical path – so, which tasks are critical for the realization of best-case scenario and which are not
- Also, it tells us how much there is flexibility in certain non-critical tasks
  - The amount of flexibility is called *float*
- Can be done either using node weights or edge weights – technique remains the same

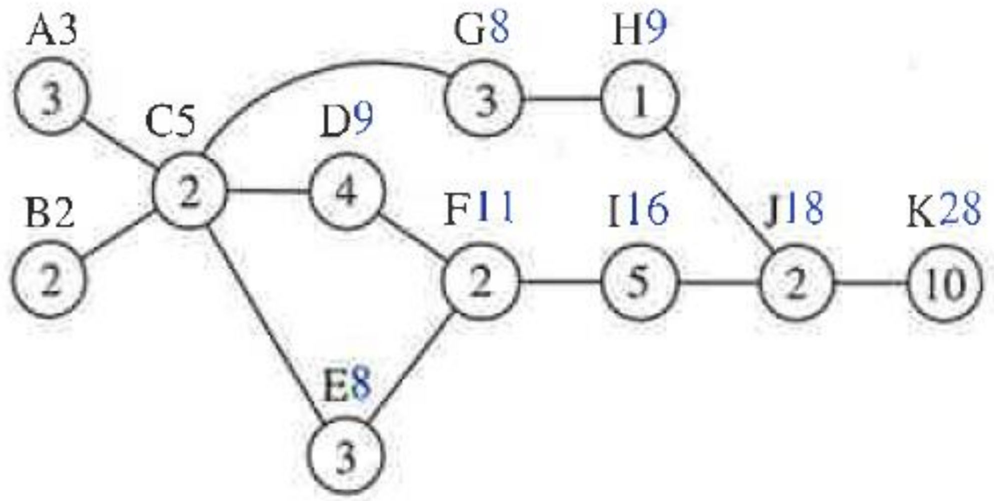
Example: Black Friday campaign

- Let’s get back to this and optimize. First two steps:
  - Length of task A = 3 days, length of task B = 2 days
  - Length of path to C = max(A,B) + C = 3 + 2 = 5



Example: Black Friday campaign

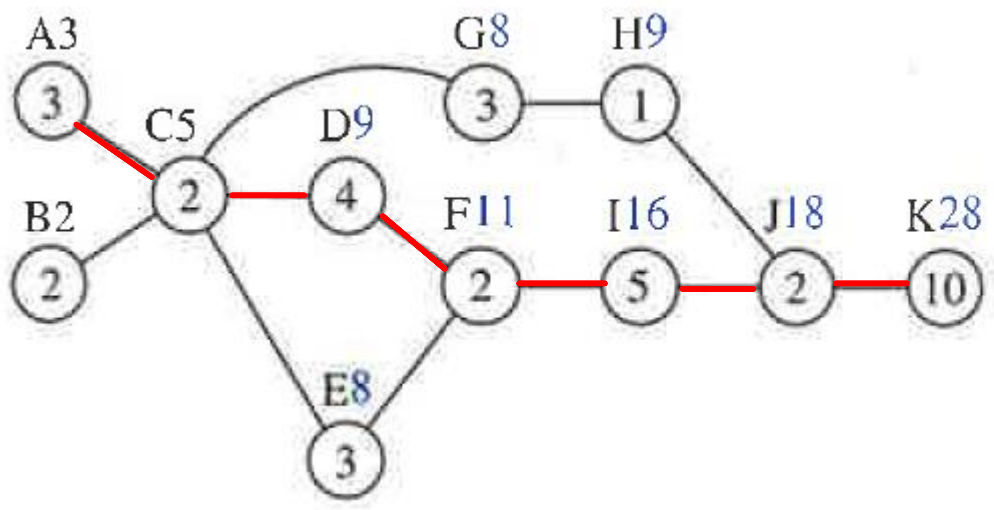
- Continuing the process forward through the whole graph ends up in this case:



- 28 days, and we had a month of time!
- Notice: Along the upper path we can get to J in 9 days, but the stages F and I are not ready yet – they require at least 16 days to complete! Therefore, completion of J takes 16+2 = 18 days.

Example: Black Friday campaign

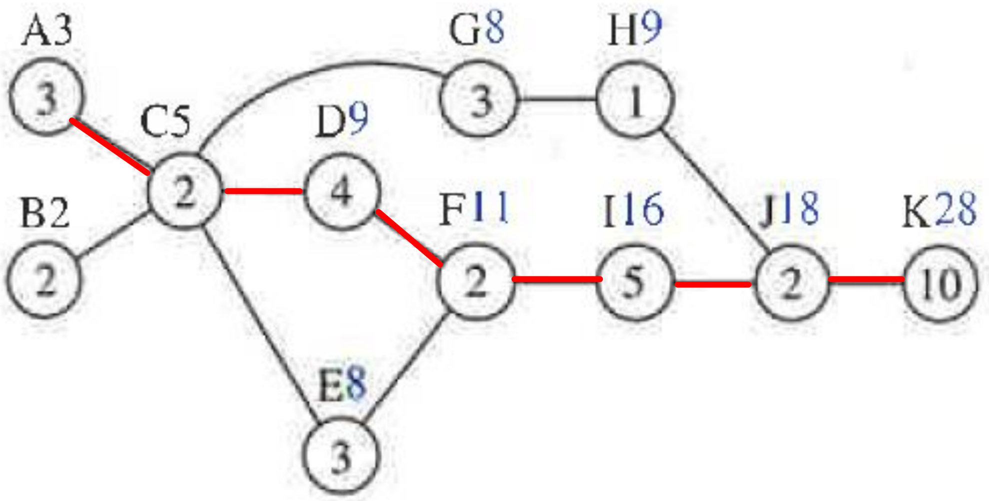
- Critical path is then the following:



  - All tasks on the critical path (A, C, D, F, I, J, K) are critical: any delays in these will cause the whole project to be delayed

Example: Black Friday campaign

- Float times of non-critical tasks:
  - B = 3 – 2 = 1 day
  - E = 9 – 8 = 1 day
  - G and H = 16 – 9 = 7 days
- NOTE: Delays in these don't change the completion time of the project, if the delay is shorter than float time



Thank you!