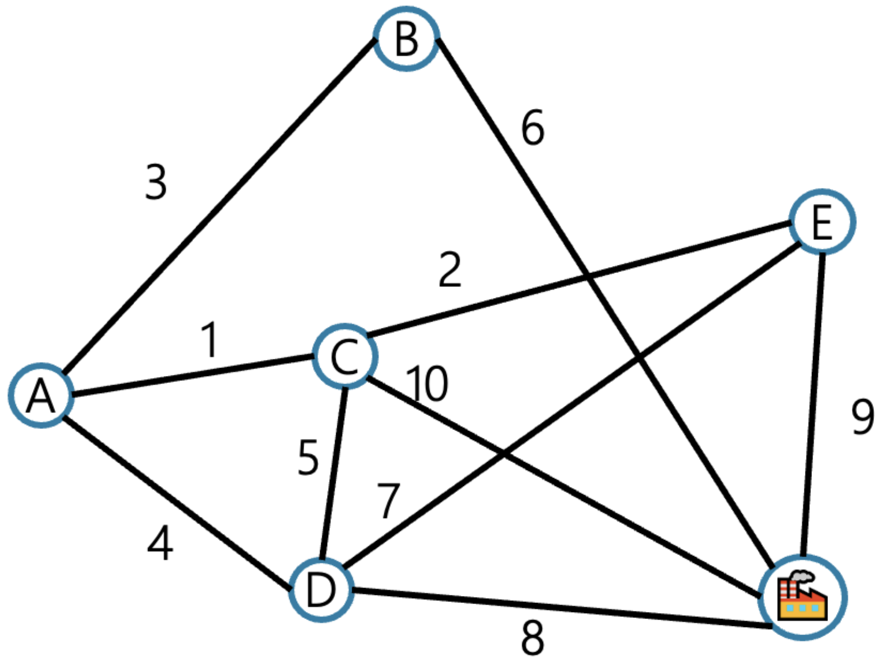
UNIT 3:
Greedy Method

Dr. Keshav Sinha

MST
Prim’s Algorithm
Kruskal’s Algorithm

Time travel – 1920s

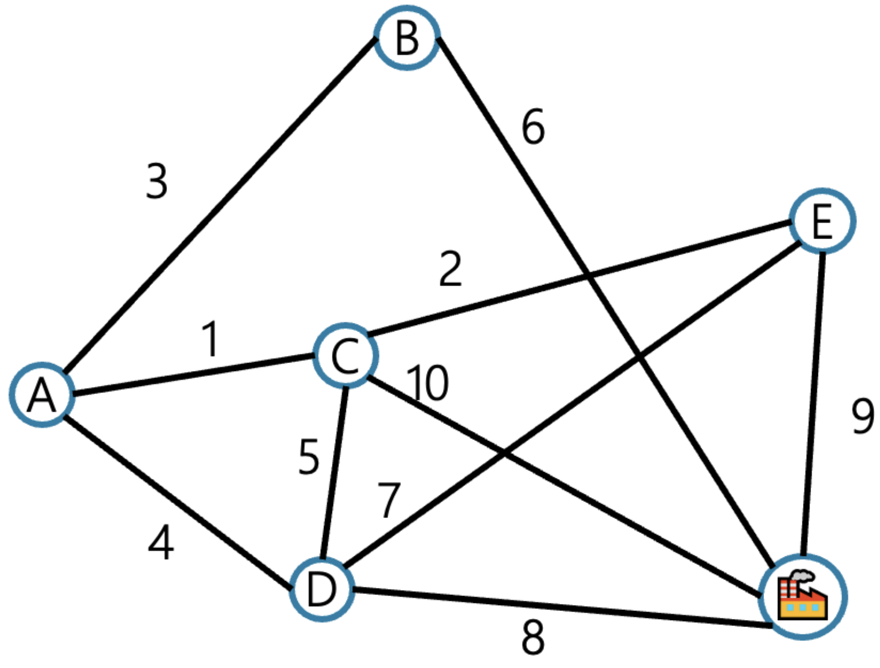
It's the 1920's. Your friend works at an electric company. They want to know where to build electrical wires to connect all cities to the powerplant.



They know how much it would cost to lay electric wires between any pair of locations, and they want the cheapest way to make sure there's electricity from the plant to every city.

Time travel – 1950s

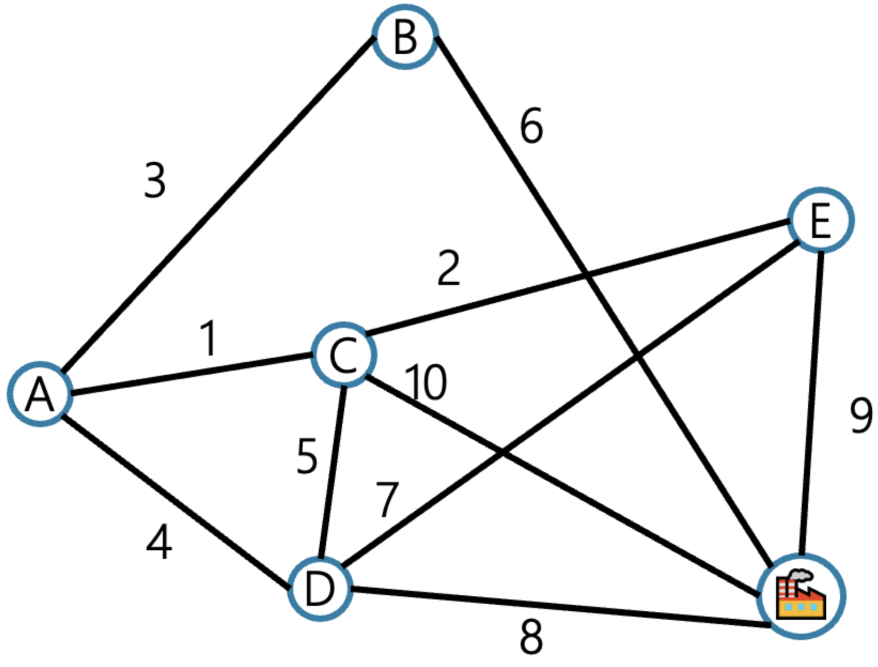
It's the 1950's. Your friend works at a phone company. They want to know where to build phone wires to connect phones to each other.



They know how much it would cost to lay phone wires between any pair of locations, and they want the cheapest way to make sure everyone can call everyone else.

Time travel – 2020s

It's 2022! Your friend works at an internet company. They want to know where to build internet cables to connect all cities to the Internet.



They know much it would cost to lay internet cables between any pair of locations, and they want the cheapest way to make sure everyone can reach the server.

What are we looking for?

A set of edges such that…

1. Every vertex touches at least one edge
2. Graph on these edges is connected
3. Sum of edge weights is minimized

Claim: The set of edges we pick never has a cycle

Aside: Tree!

1. Don’t need a root
2. Varying numbers of children
3. Connected and no cycles

Tree (when talking about undirected graphs)
An undirected, connected acyclic graph.

MST Problem

A set of edges such that…

1. Edges span the graph
2. Graph on these edges is connected
3. Sum of edge weights is minimized
4. Contains no cycle

We need a minimum spanning tree

Minimum Spanning Tree Problem

Given: an undirected, weighted graph G
Find: A minimum-weight set of edges such that you can get from any vertex of G to any other on only those edges.

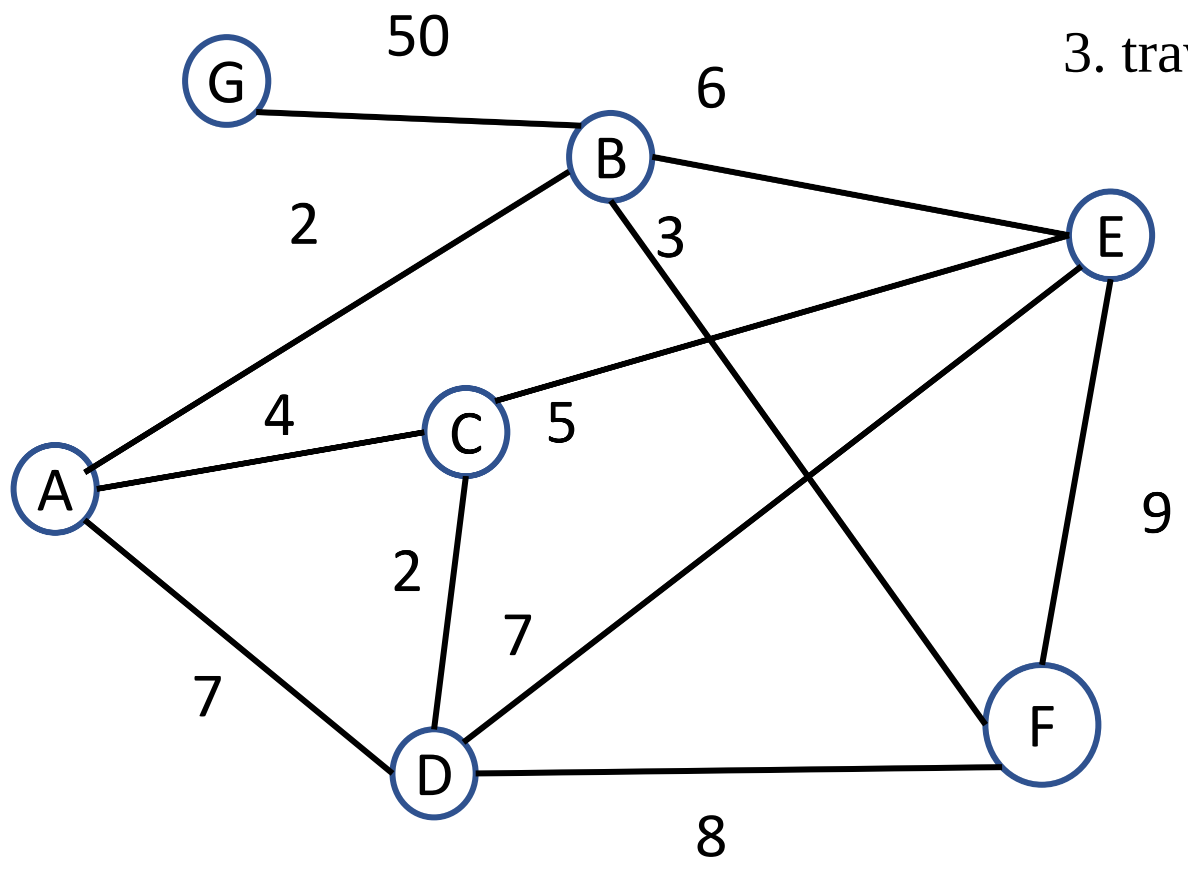
Prim’s Algorithm

- Choose arbitrary starting point
- Add a new edge to the result each step
- How to choose the new edge?
    - Will let you reach more vertices
    - Is as light as possible

Try it out

Prim's travelling strategy as a broke college student!

1. research neighboring cities (update table)
2. decide on the cheapest city to travel to
3. travel to that city (mark graph)!

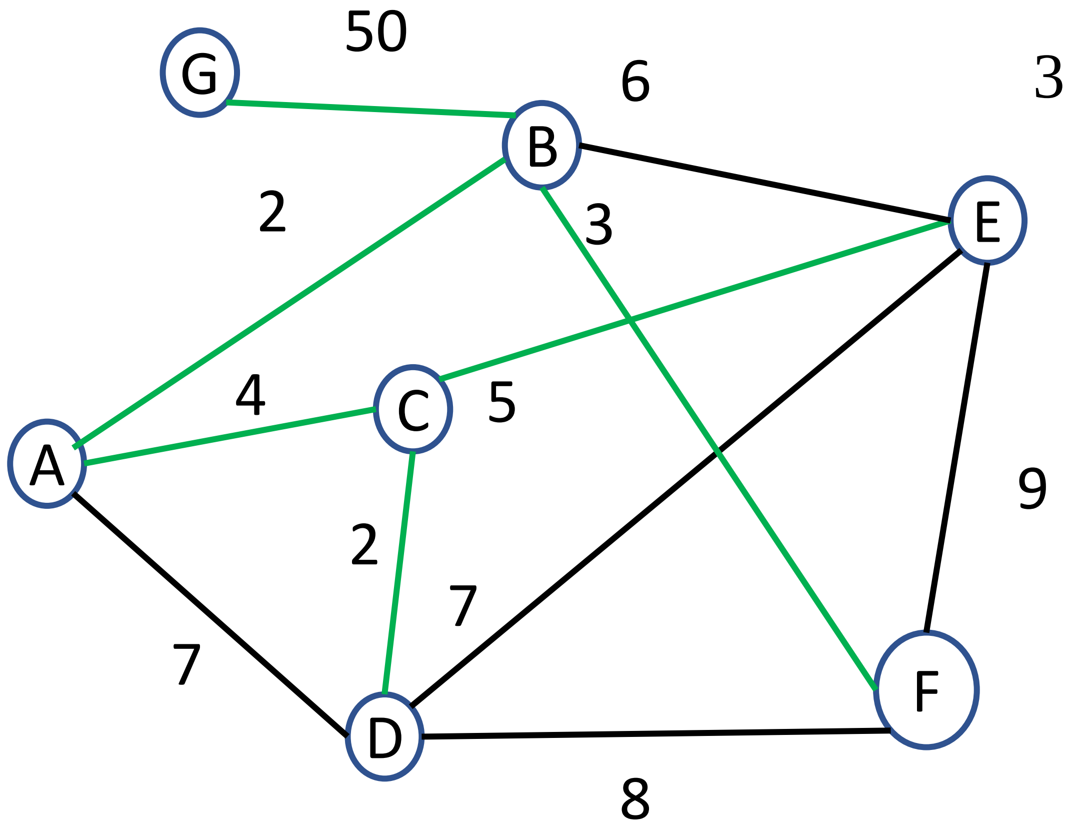


Vertex	Dist.	Best Edge	Processed
A			
B			
C			
D			
E			
F			
G			

Try it out

Prim's travelling strategy as a broke college student!

1. research neighboring cities (update table)
2. decide on the cheapest city to travel to
3. travel to that city (mark graph)!



Vertex	Dist.	Best Edge	Processed
A	--	--	Yes
B	2	(A,B)	Yes
C	4	(A,C)	Yes
D	~~7~~ 2	~~(A,D)~~(C,D)	Yes
E	~~6~~ 5	~~(B,E)~~(C,E)	Yes
F	3	(B,F)	Yes
G	50	(B,G)	Yes

Pseudocode

```
initialize distances to $\infty$
    mark source as distance 0
    mark all vertices unprocessed
    foreach(edge (source, v) )
        v.dist = w(source,v)
    while(there are unprocessed vertices){
        let u be the closest unprocessed vertex
        add u.bestEdge to spanning tree
        foreach(edge (u,v) leaving u){
            if(w(u,v) < v.dist){
                v.dist = w(u,v)
                v.bestEdge = (u,v)
        }
        }
  mark u as processed
}
```

Does Prim’s Algorithm Always Work?

- Prim’s Algorithm is a greedy algorithm
- Always select the best option available now
- Once it decides to include an edge in the MST, it never reconsiders its decision
- Life lesson? Be greedy!

A different Approach

- Prim’s algorithm - vertex by vertex

- What if you think edge by edge instead?

- Start from the lightest edge
    - add it if it connects new things to each other
    - don’t add it if it would create a cycle

- This is Kruskal’s Algorithm

Kruskal’s Algorithm

```
KruskalMST(Graph G)
  initialize each vertex to be a connected component
        sort the edges by weight
        foreach(edge (u, v) in sorted order){
                if(u and v are in different components){
                        add (u, v) to the MST
                        Update u and v to be in the same component
                }
        }
```

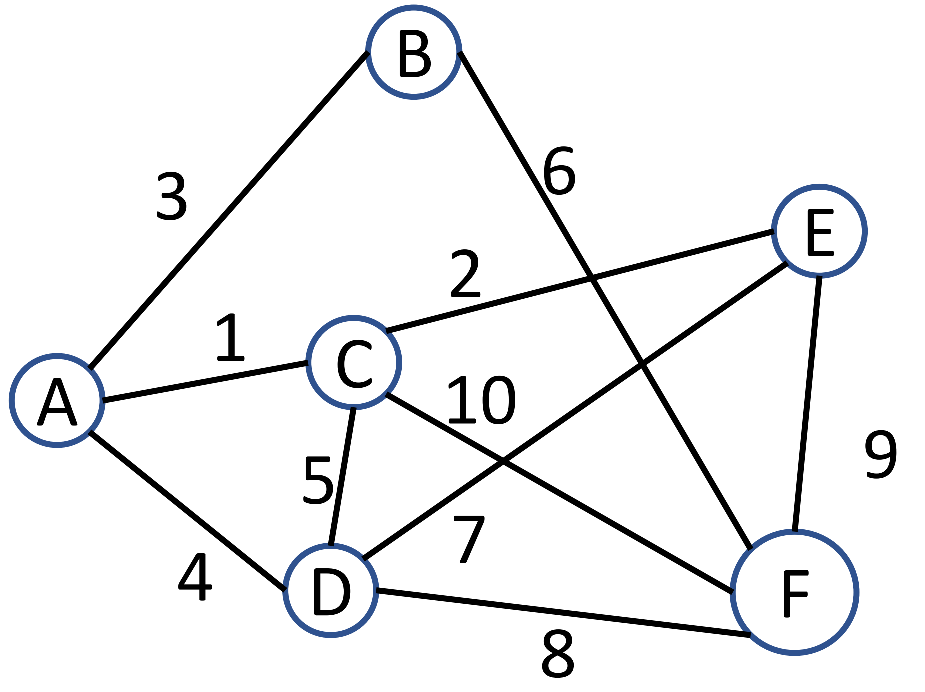
Try It Out

Two simple steps:

1. If edge connects different clouds

   then add edge, combine clouds

2. Otherwise, ignore



Edge	Include?	Reason
(A,C)		
(C,E)		
(A,B)		
(A,D)		
(C,D)		

Edge	Include?	Reason
(B,F)		
(D,E)		
(D,F)		
(E,F)		
(C,F)		

Kruskal’s Algorithm: Running Time

```
KruskalMST(Graph G)
  initialize each vertex to be a connected component
        sort the edges by weight
        foreach(edge (u, v) in sorted order){
                if(u and v are in different components){
                        add (u, v) to the MST
                        Update u and v to be in the same component
                }
        }
```

Dijkstra’s Algorithm

Shortest Path Applications

- Network Routing
- Driving Directions
- Cheap Flight Tickets
- Critical Paths In Project Management

Single Source Shortest Paths

- Done: BFS to find the minimum path length from **s** to **t** in *O*(|E|+|V|)

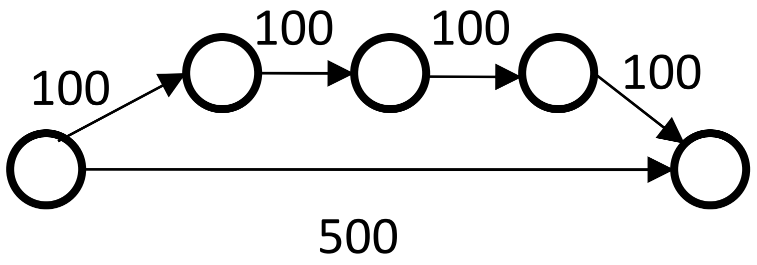
- We found the minimum path length from **s** to *every node*
  - Still *O*(|E|+(|V|)
  - No faster way for a "distinguished" destination in the worst-case

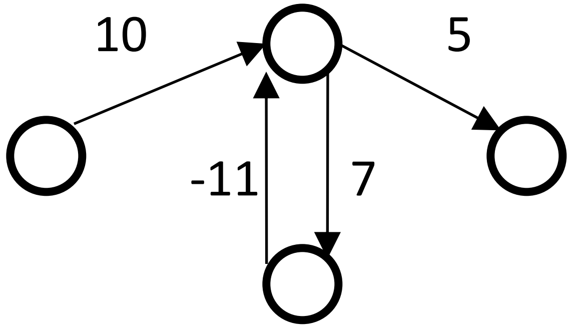
- Now: Weighted graphs

Given a weighted graph and node **s**,
find the minimum-cost path from **s** to every node

- As before, asymptotically no harder than for one destination
- Unlike before, BFS will not work

Not as Easy





Why BFS won't work: Shortest path may not have the fewest edges

- Annoying when this happens with costs of flights

We will assume there are no negative weights

- *Problem* is *ill-defined* if there are negative-cost *cycles*
- *Today's algorithm* is *wrong* if *edges* can be negative

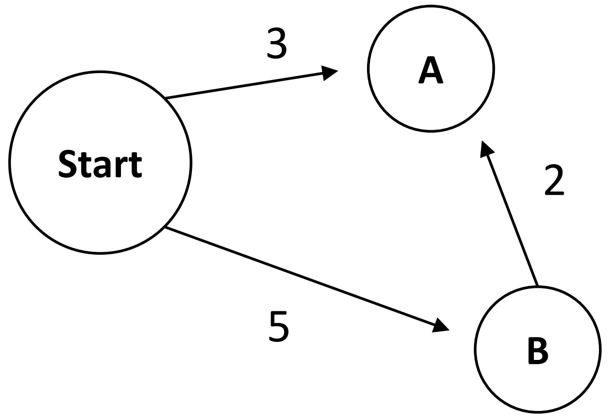
Dijkstra’s Algorithm

- Named after its inventor Edsger Dijkstra (1930-2002)
  - Truly one of the “founders” of computer science;
    1972 Turing Award; this is just one of his many contributions
  - Sample quotation: “computer science is no more about computers than astronomy is about telescopes”

- The idea: reminiscent of BFS, but adapted to handle weights
  - Grow the set of nodes whose shortest distance has been computed
  - Nodes not in the set will have a “best distance so far”
  - A priority queue will turn out to be useful for efficiency

Dijkstra’s Intuition

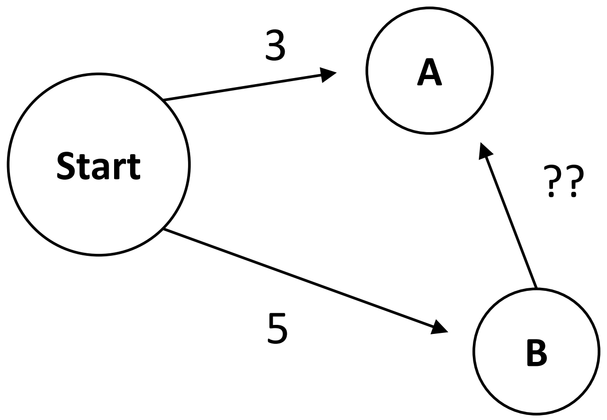
At each step, process the next closest vertex to our start.



Without trying other paths to A, why do we know that the shortest path to A must be a total of cost 3?

Dijkstra’s Intuition

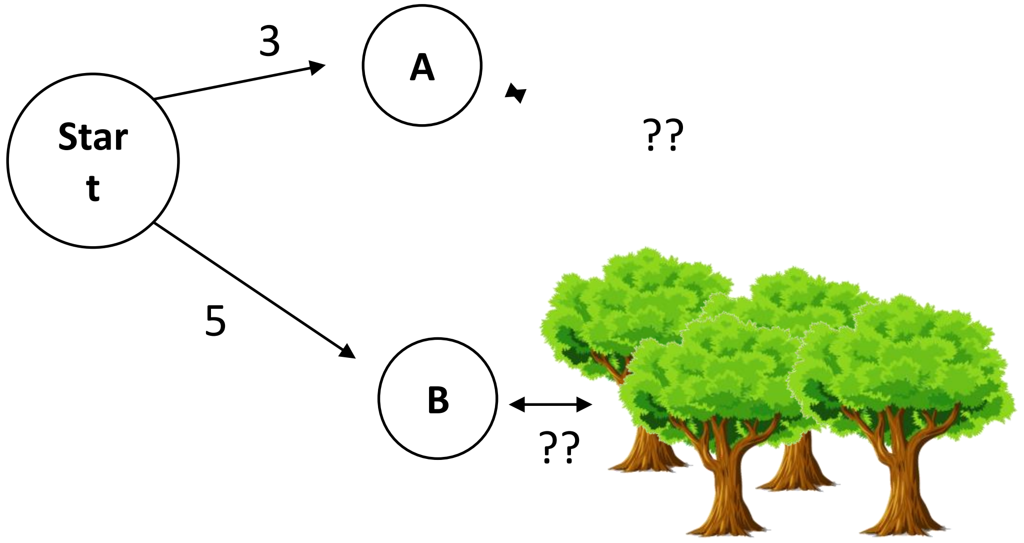
At each step, process the next closest vertex to our start.



Without trying other paths to A, why do we *still* know that the shortest path to A must be a total of cost 3?

Dijkstra's Intuition

At each step, process the next closest vertex to our start.

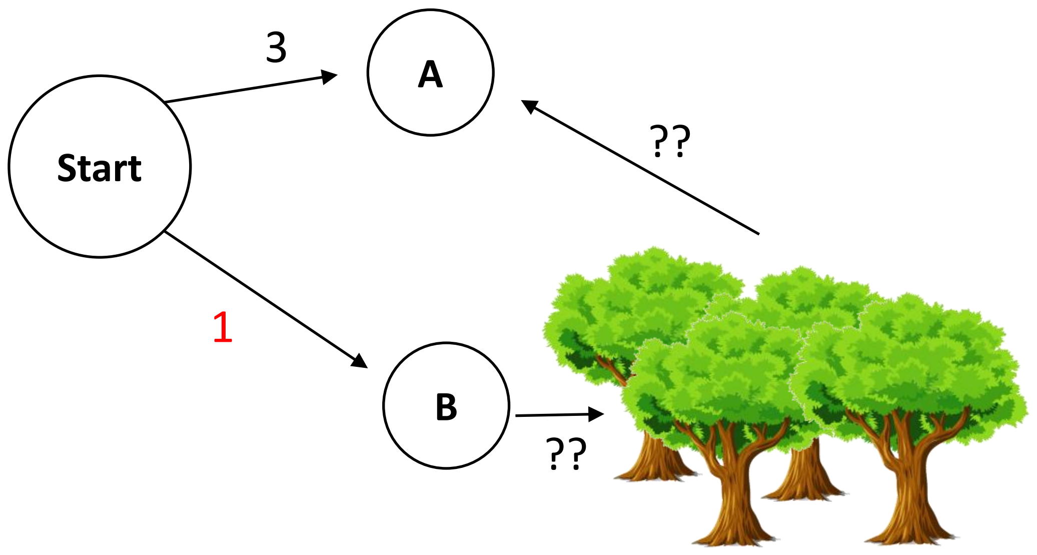


Without trying other paths to A, why do we *still, still* know that the shortest path to A must be a total of cost 3?

Can we make the claim that the shortest path to B must be 5?

Dijkstra’s Intuition

At each step, process the next closest vertex to our start.

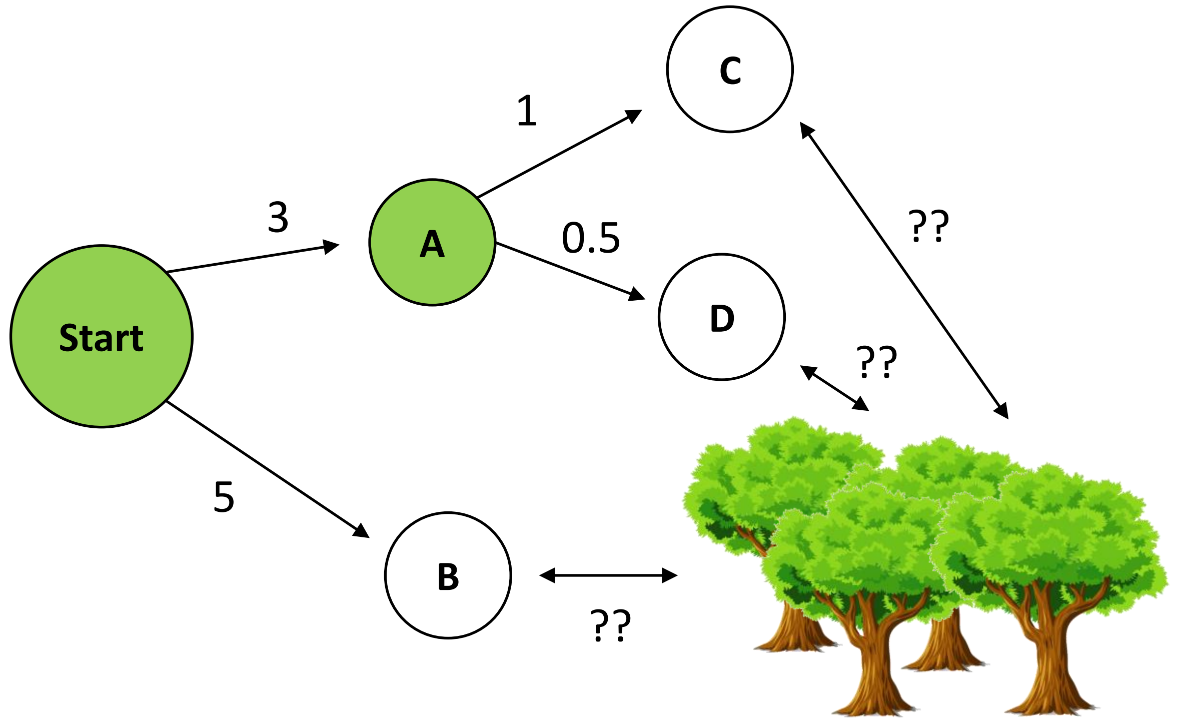


What if the weights were different?
Do we still know that the shortest path to A costs 3?

Dijkstra’s Intuition

At each step, process the next closest vertex to our start, which we know must be the shortest possible distance to that node.

Rinse and repeat.

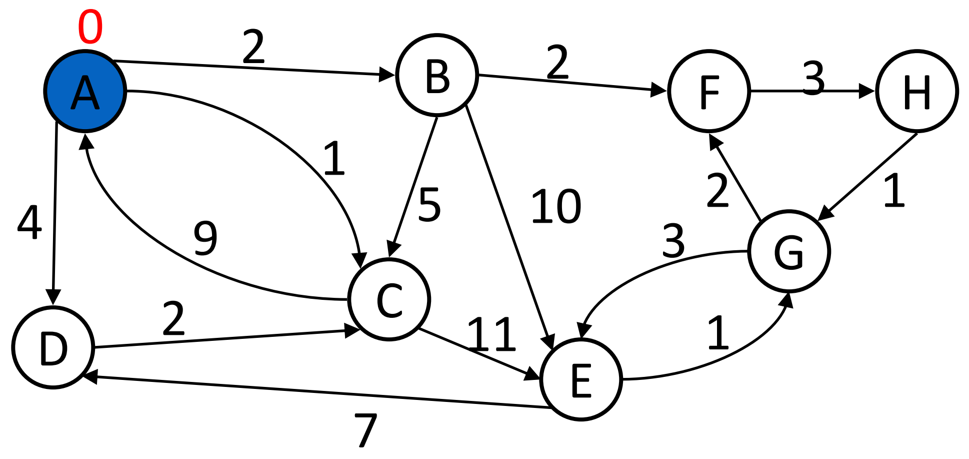


The Algorithm

1. For each node **v**, set **v.cost = ∞** and **v.known = false**
2. Set **source.cost = 0**
3. While there are unknown nodes in the graph
   a) Select the unknown node **v** with lowest cost
   b) Mark **v** as known
   c) For each edge **(v,u)** with weight **w**, if **u** is unknown,

```
c1 = v.cost + w // cost of best path through v to u
c2 = u.cost   // cost of best path to u previously known
 if(c1 < c2){ // if the path through v is better
     u.cost = c1
     u.path = v // for computing actual paths
 }
```

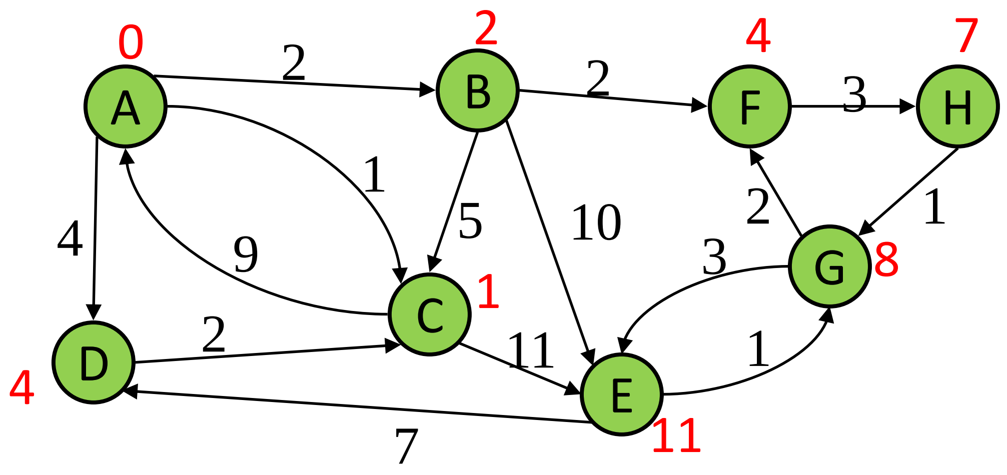
Example #1



Order Added to Known Set:

vertex	known?	cost	path
A			
B			
C			
D			
E			
F			
G			
H			

Example #1



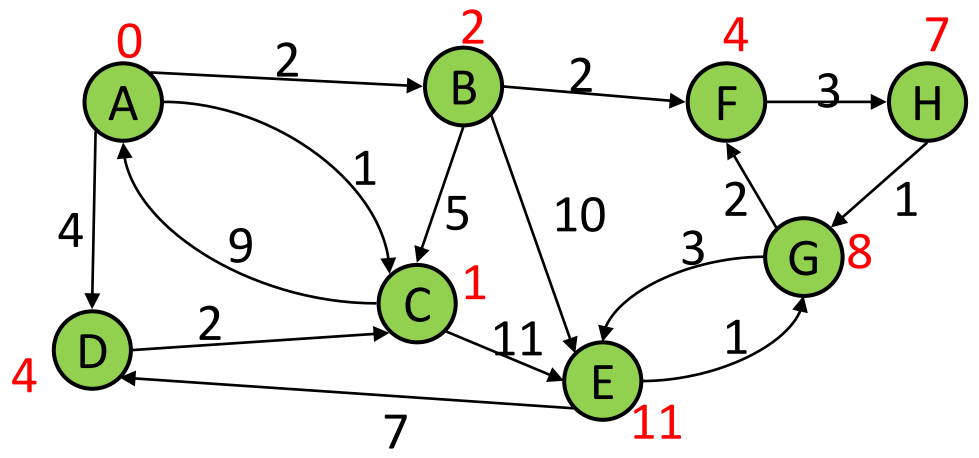
Order Added to Known Set:

A, C, B, D, F, H, G, E

vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Interpreting the Results

- Now that we're done, how do we get the path from, say, A to E?



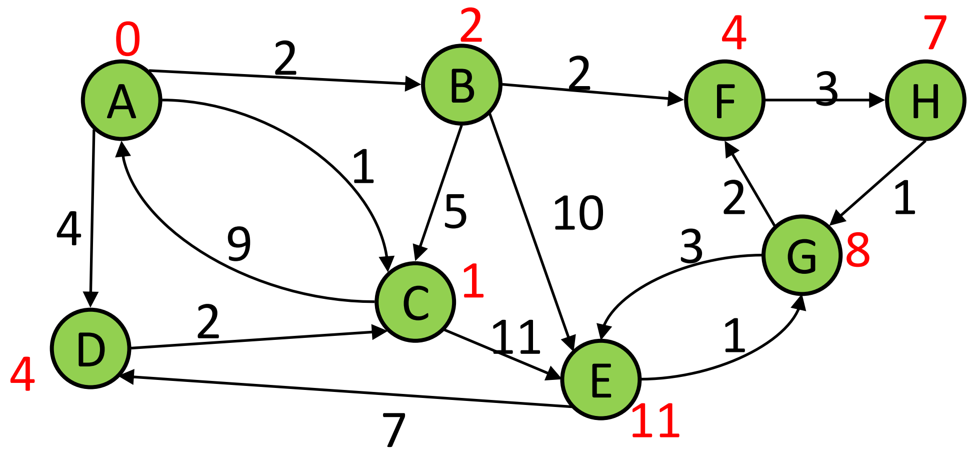
<u>Order Added to Known Set:</u>

A, C, B, D, F, H, G, E

vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Stopping Short

- How would this have worked differently if we were only interested in:
  - The path from A to G?
  - The path from A to D?

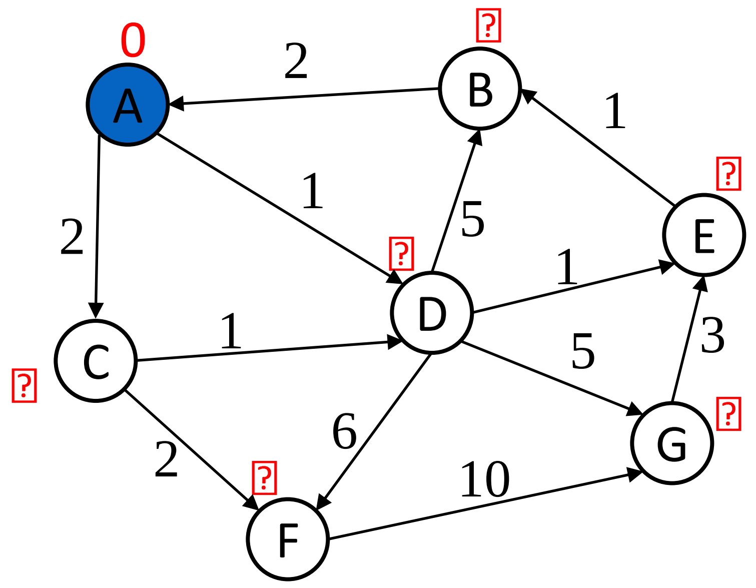


Order Added to Known Set:

A, C, B, D, F, H, G, E

vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

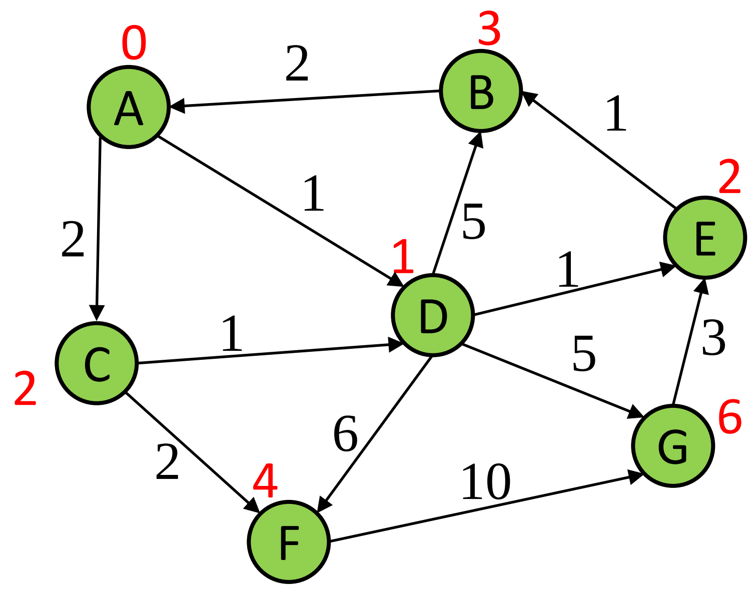
Example #2



Order Added to Known Set:

vertex	known?	cost	path
A		0	
B			
C			
D			
E			
F			
G			

Example #2



Order Added to Known Set:

A, D, C, E, B, F, G

vertex	known?	cost	path
A	Y	0	
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F	Y	4	C
G	Y	6	D

Features

- When a vertex is marked known,
  the cost of the shortest path to that node is known
    - The path is also known by following back-pointers

- While a vertex is still not known,
  another shorter path to it might still be found
    - The current cost we have is an upper-bound though!

Note: The "Order Added to Known Set" is not important

- A detail about how the algorithm works (client doesn't care)
- Not used by the algorithm (implementation doesn't care)
- It is sorted by path-cost, resolving ties in some way

A Greedy Algorithm

- Dijkstra’s algorithm
    - For single-source shortest paths in a weighted graph (directed or undirected) with no negative-weight edges

- An example of a *greedy algorithm*:
    - At each step, irrevocably does what seems best at that step
        - A locally optimal step, not necessarily globally optimal
    - Once a vertex is known, it is not revisited
        - Turns out to be globally optimal

When greed fails us

Making change – use fewest # of coins possible for 15¢

25, 10, 5, 1

25, 12, 10, 5, 1

Where are we?

- What should we do after learning an algorithm?
  - Prove it is correct
    - Not obvious!
    - We will sketch the key ideas
  - Analyze its efficiency
    - Will do better by using a data structure we learned earlier!

Correctness: Intuition

Rough intuition:

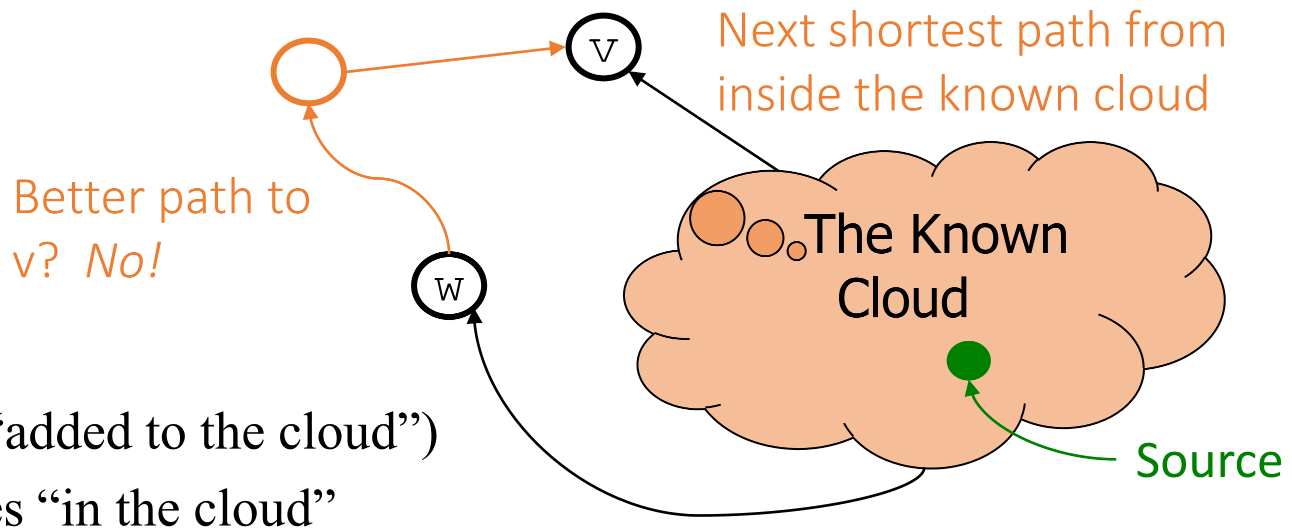
All the “known” vertices have the correct shortest path

- True initially: shortest path to start node has cost 0
- If it stays true every time, we mark a node “known”, then by induction this holds and eventually everything is “known”

Key fact we need: When we mark a vertex “known” we won’t discover a shorter path later!

- This holds only because Dijkstra’s algorithm picks the node with the next shortest path-so-far
- The proof is by contradiction…

Correctness: The Cloud (Rough Idea)



Suppose v is the next node to be marked known ("added to the cloud")

- The best-known path to v must have only nodes "in the cloud"
  - Since we've selected it, and we only know about paths through the cloud to a node right outside the cloud
- Assume (for contradiction) the actual shortest path to v is different
  - It won't use only cloud nodes, (or we would know about it), so it must use non-cloud nodes
  - Let w be the *first* non-cloud node on this path.
  - The part of the path up to w is already known and must be shorter than the best-known path to v. So, v would not have been picked.
    Contradiction!

Efficiency, First Approach

Use pseudocode to determine asymptotic run-time

- Notice each edge is processed only once

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  while(not all nodes are known) {
    b = find unknown node with smallest cost
    b.known = true
    for each edge (b,a) in G
     if(!a.known)
       if(b.cost + weight((b,a)) < a.cost){
         a.cost = b.cost + weight((b,a))
         a.path = b
       }
}
```

Improving Asymptotic Running Time

- So far: $O(|V|^2 + |E|)$

- We had a similar “problem” with topological sort being $O(|V|^2 + |E|)$ due to each iteration looking for the node to process next
  - We solved it with a queue of zero-degree nodes
  - But here we need the lowest-cost node and costs can change as we process edges

- Solution?

Efficiency, Second Approach

Use pseudocode to determine asymptotic run-time

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  build-heap with all nodes
  while(heap is not empty) {
    b = deleteMin()
    b.known = true
    for each edge (b,a) in G
     if(!a.known)
      if(b.cost + weight((b,a)) < a.cost){
        decreaseKey(a,"new cost – old cost")
        a.path = b
      }
}
```

Dense Vs. Sparse Again

First approach: $O(|V|^2 + |E|)$

Second approach: $O(|V|\log|V| + |E|\log|V|)$

So which is better?

Dense Vs. Sparse Again

First approach: $O(|V|^2 + |E|)$ or: $O(|V|^2)$

Second approach: $O(|V|\log|V| + |E|\log|V|)$

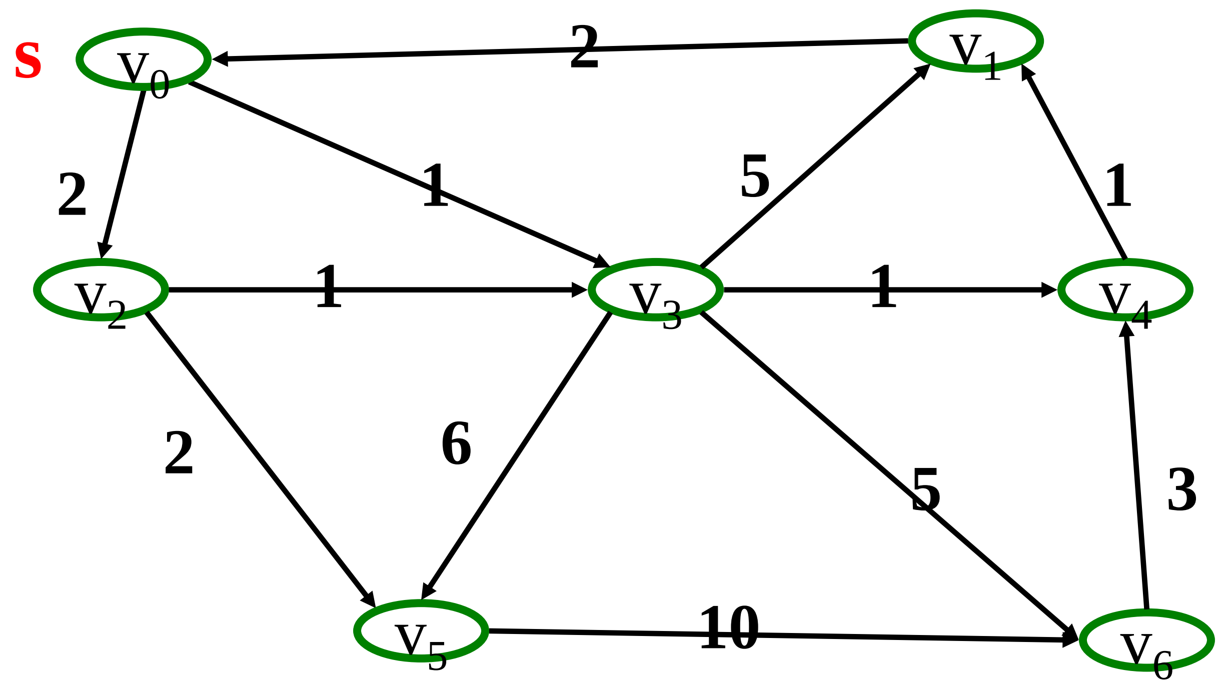
So which is better?

- Sparse: $O(|V|\log|V| + |E|\log|V|)$ (if $|E| > |V|$, then $O(|E|\log|V|)$)
- Dense: $O(|V|^2 + |E|)$, or: $O(|V|^2)$

But, remember these are worst-case and asymptotic

- Priority queue might have slightly worse constant factors
- On the other hand, for “normal graphs”, we might call **decreaseKey** rarely (or not percolate far), making $|E|\log|V|$ more like $|E|$

Find the shortest path to each vertex from v_0



V	Known	Dist from s	Path
v0			
v1			
v2			
v3			
v4			
v5			
v6			

Order declared Known:

The Bellman-Ford
Shortest Path Algorithm

Class Overview

- ➢ The shortest path problem
- ➢ Differences
- ➢ The Bellman-Ford algorithm
- ➢ Time complexity

Shortest Path Problem

- Weighted path length (cost): The sum of the weights of all links on the path.

- The single-source shortest path problem: Given a weighted graph G and a source vertex s, find the shortest (minimum cost) path from s to every other vertex in G.

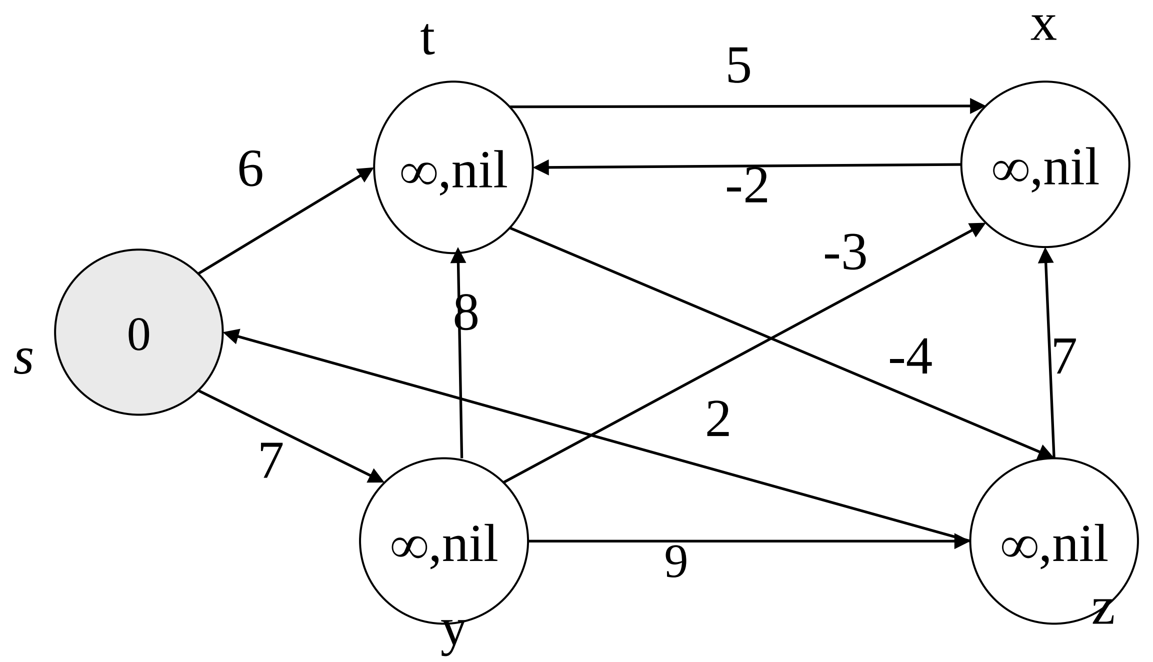
Differences

- Negative link weight: The Bellman-Ford algorithm works; Dijkstra’s algorithm doesn’t.

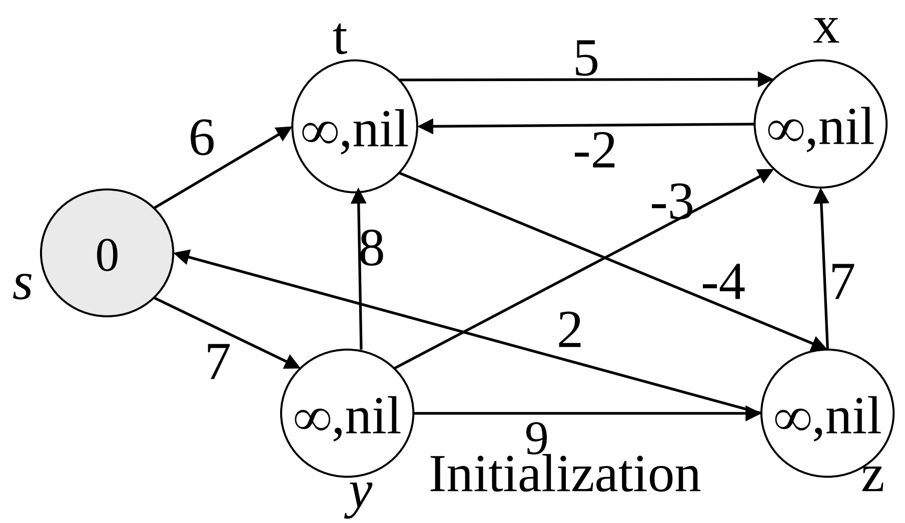
- Distributed implementation: The Bellman-Ford algorithm can be easily implemented in a distributed way. Dijkstra’s algorithm cannot.

- Time complexity: The Bellman-Ford algorithm is higher than Dijkstra’s algorithm.

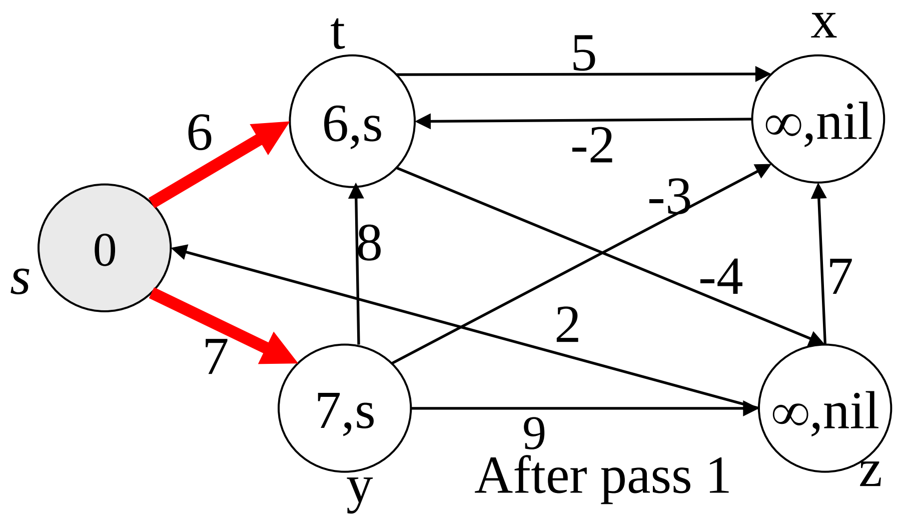
The Bellman-Ford Algorithm



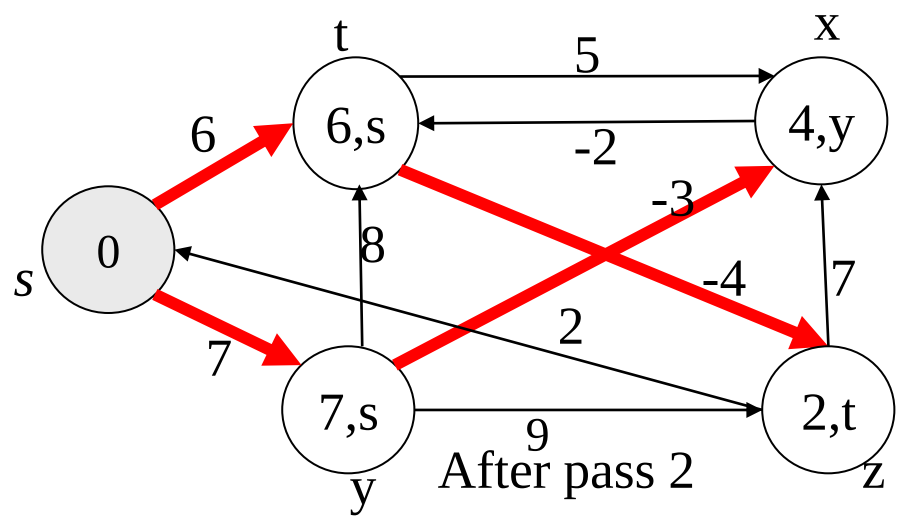
The Bellman-Ford Algorithm



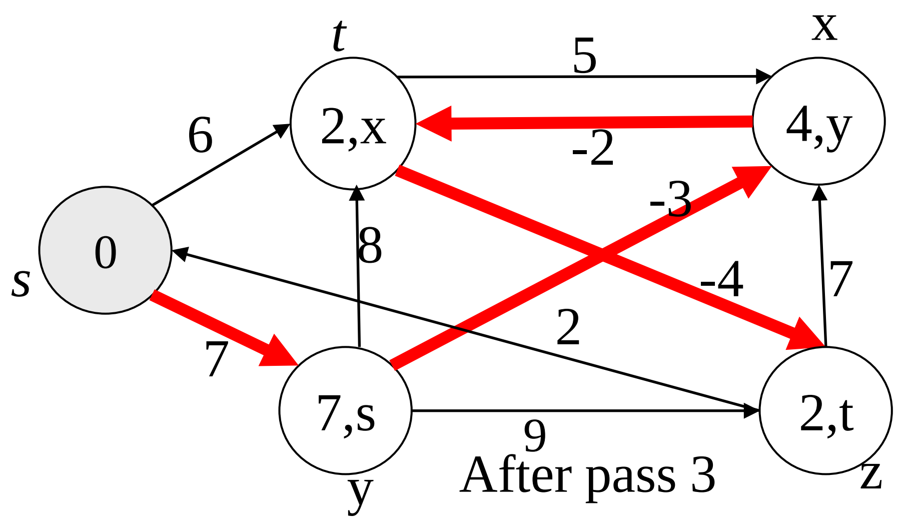
Initialization



After pass 1



After pass 2

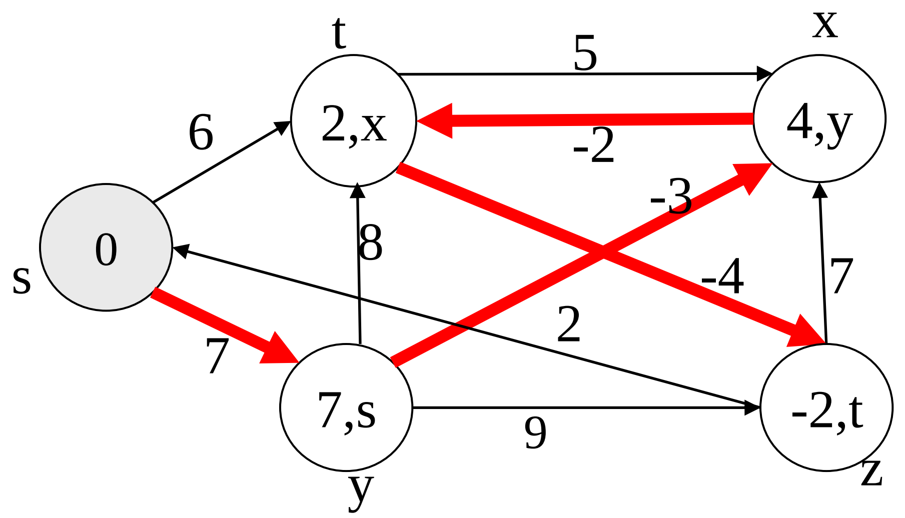


After pass 3

The order of edges examined in each pass:

(t, x), (t, z), (x, t), (y, x), (y, t), (y, z), (z, x), (z, s), (s, t), (s, y)

The Bellman-Ford Algorithm



After pass 4

The order of edges examined in each pass:

(t, x), (t, z), (x, t), (y, x), (y, t), (y, z), (z, x), (z, s), (s, t), (s, y)

The Bellman-Ford Algorithm

```
Bellman-Ford(G, w, s)
1.  Initialize-Single-Source(G, s)
2.  for i := 1 to |V| - 1 do
3.      for each edge (u, v) ∈ E do
4.          Relax(u, v, w)
5.  for each vertex v ∈ u.adj do
6.      if d[v] > d[u] + w(u, v)
7.          then return False     // there is a negative cycle
8.  return True
```

```
Relax(u, v, w)
if d[v] > d[u] + w(u, v)
    then  d[v] := d[u] + w(u, v)
          parent[v] := u
```

Time Complexity

```
Bellman-Ford(G, w, s)
1.  Initialize-Single-Source(G, s)                $\longrightarrow O(|V|)$
2.  for i := 1 to $|V| - 1$ do
3.      for each edge $(u, v) \in E$ do
                                                  $\longrightarrow O(|V||E|)$
4.          Relax(u, v, w)
5.  for each vertex $v \in$ u.adj do               $\longrightarrow O(|E|)$
6.      if $d[v] > d[u] + w(u, v)$
7.          then return False      // there is a negative cycle
8.  return True
```

Time complexity: $O(|V||E|)$

Huffman Codes

Encoding messages

- Encode a message composed of a string of characters
- Codes used by computer systems
  - ASCII
    - uses 8 bits per character
    - can encode 256 characters
  - Unicode
    - 16 bits per character
    - can encode 65536 characters
    - includes all characters encoded by ASCII
- ASCII and Unicode are *fixed-length codes*
  - all characters represented by same number of bits

Problems

- Suppose that we want to encode a message constructed from the symbols **A**, **B**, **C**, **D**, and **E** using a fixed-length code
  - How many bits are required to encode each symbol?
    - at least 3 bits are required
    - 2 bits are not enough (can only encode four symbols)
  - How many bits are required to encode the message **DEAACAAAAABA**?
    - there are twelve symbols, each requires 3 bits
    - 12*3 = 36 bits are required

Drawbacks of fixed-length codes

- **Wasted space**
  - Unicode uses twice as much space as ASCII
    - inefficient for plain-text messages containing only ASCII characters
- Same number of bits used to represent all characters
  - ‘a’ and ‘e’ occur more frequently than ‘q’ and ‘z’
- **Potential solution**
  - Use variable-length codes
  - variable number of bits to represent characters when frequency of occurrence is known
  - short codes for characters that occur frequently

Advantages of variable-length codes

- The advantage of variable-length codes over fixed-length is short codes can be given to characters that occur frequently
  - on average, the length of the encoded message is less than fixed-length encoding
- **Potential problem:** how do we know where one-character ends and another begins?
  - not a problem if number of bits is fixed!

A = 00
B = 01
C = 10
D = 11

001011011100111111111

A C D B A D D D D D

Prefix property

- A code has the **prefix property** if no character code is the prefix (start of the code) for another character
- Example:

Symbol	Code
P	000
Q	11
R	01
S	001
T	10

01001101100010

R S T Q P T

- 000 is not a prefix of 11, 01, 001, or 10
- 11 is not a prefix of 000, 01, 001, or 10 …

Code without prefix property

- The following code does **not** have prefix property

Symbol	Code
P	0
Q	1
R	01
S	10
T	11

- The pattern **1110** can be decoded as **QQQP**, **QTP**, **QQS**, or **TS**

Problem

- Design a variable-length prefix-free code such that the message **DEAACAAAAABA** can be encoded using 22 bits
- Possible solution:
  - **A** occurs eight times while **B**, **C**, **D**, and **E** each occur once
  - represent **A** with a one-bit code, say 0
    - remaining codes cannot start with 0
  - represent **B** with the two-bit code 10
    - remaining codes cannot start with 0 or 10
  - represent **C** with 110
  - represent **D** with 1110
  - represent **E** with 11110

Encoded Message

DEAACAAAAABA

Symbol	Code
A	0
B	10
C	110
D	1110
E	11110

1110111100011000000100

22 bits

Another possible code

DEAACAAAAABA

Symbol	Code
A	0
B	100
C	101
D	1101
E	1111

1101111100101000001000

22 bits

Better code

DEAACAAAAABA

Symbol	Code
A	0
B	100
C	101
D	110
E	111

11011100101000001000

20 bits

What code to use?

- Question: Is there a variable-length code that makes the most efficient use of space?

Answer: Yes!

Huffman coding tree

- Binary tree
  - each leaf contains symbol (character)
  - label edge from node to left child with 0
  - label edge from node to right child with 1
- Code for any symbol obtained by following path from root to the leaf containing symbol
- Code has prefix property
  - leaf node cannot appear on path to another leaf
  - *note*: fixed-length codes are represented by a complete Huffman tree and clearly have the prefix property

Building a Huffman tree

- Find frequencies of each symbol occurring in message
- Begin with a forest of single node trees
    - each contain symbol and its frequency
- Do recursively
    - select two trees with smallest frequency at the root
    - produce a new binary tree with the selected trees as children and store the sum of their frequencies in the root
- Recursion ends when there is one tree
    - this is the Huffman coding tree

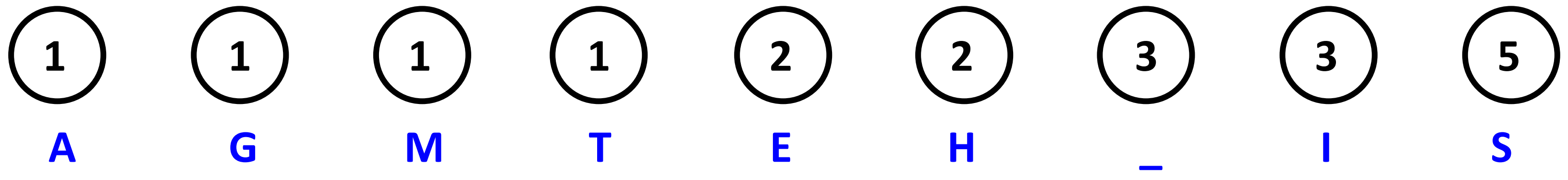
Example

- Build the Huffman coding tree for the message

This is his message

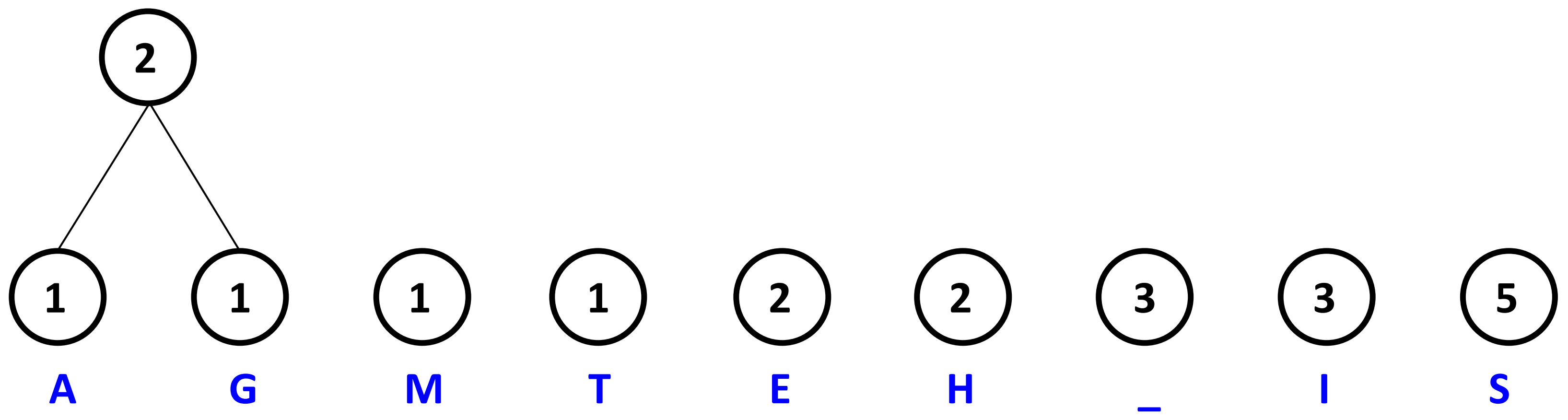
- Character frequencies

A	G	M	T	E	H	_	I	S
1	1	1	1	2	2	3	3	5

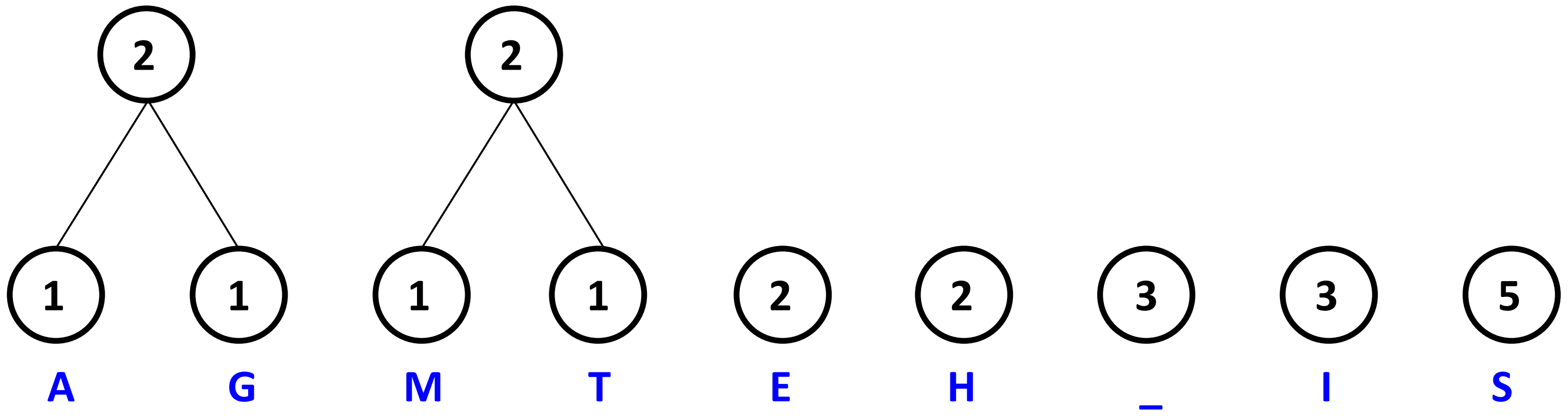


- Begin with forest of single trees

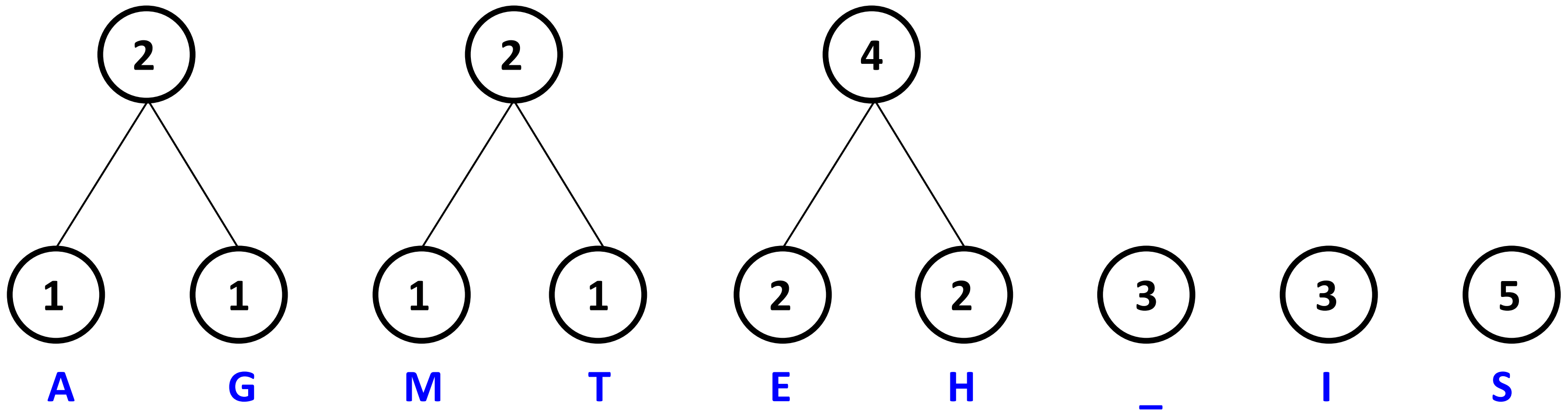
Step 1



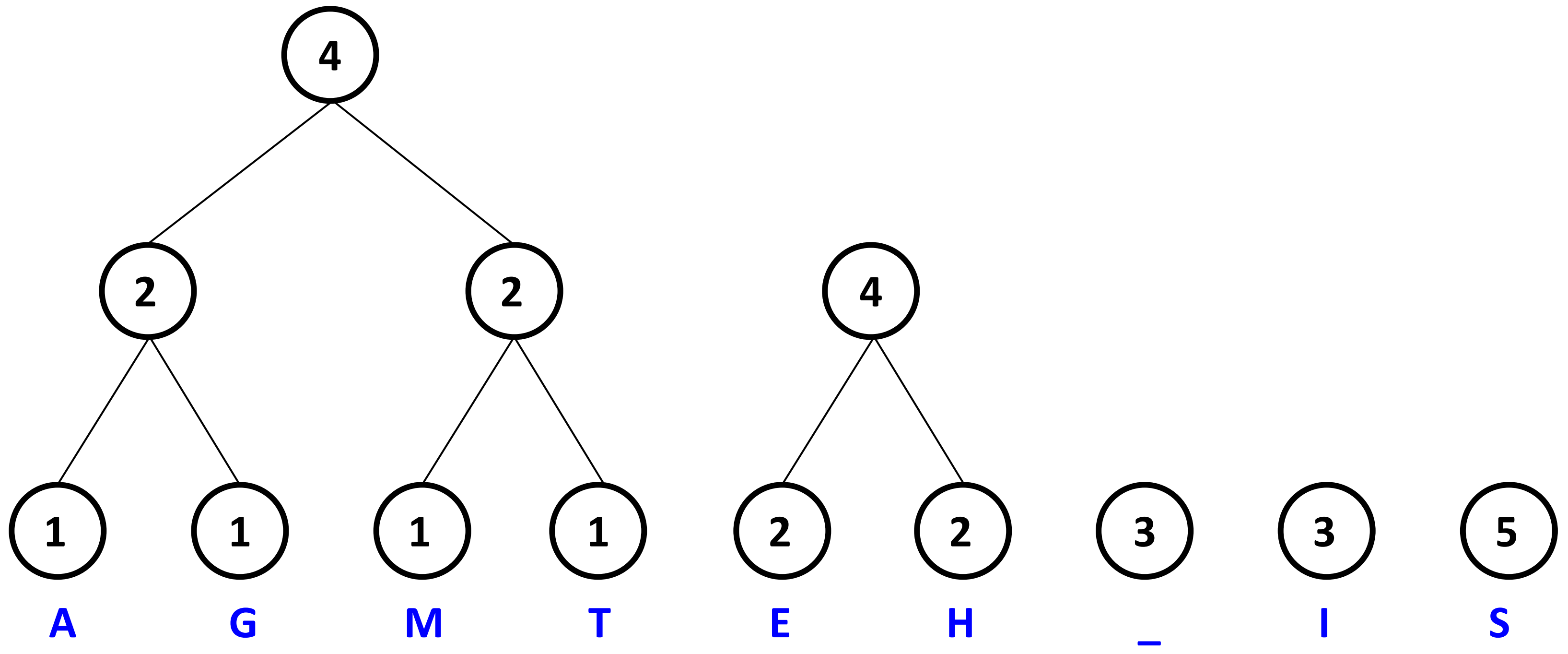
Step 2



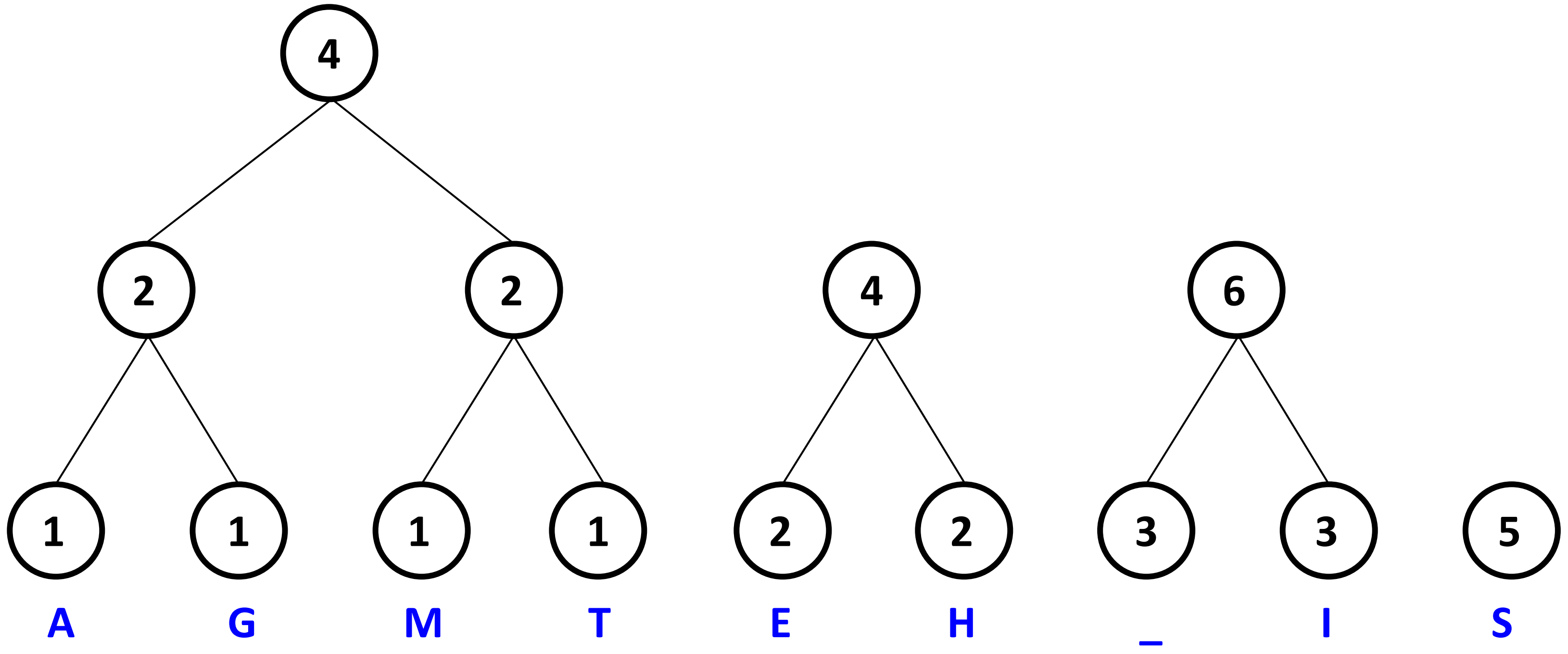
Step 3



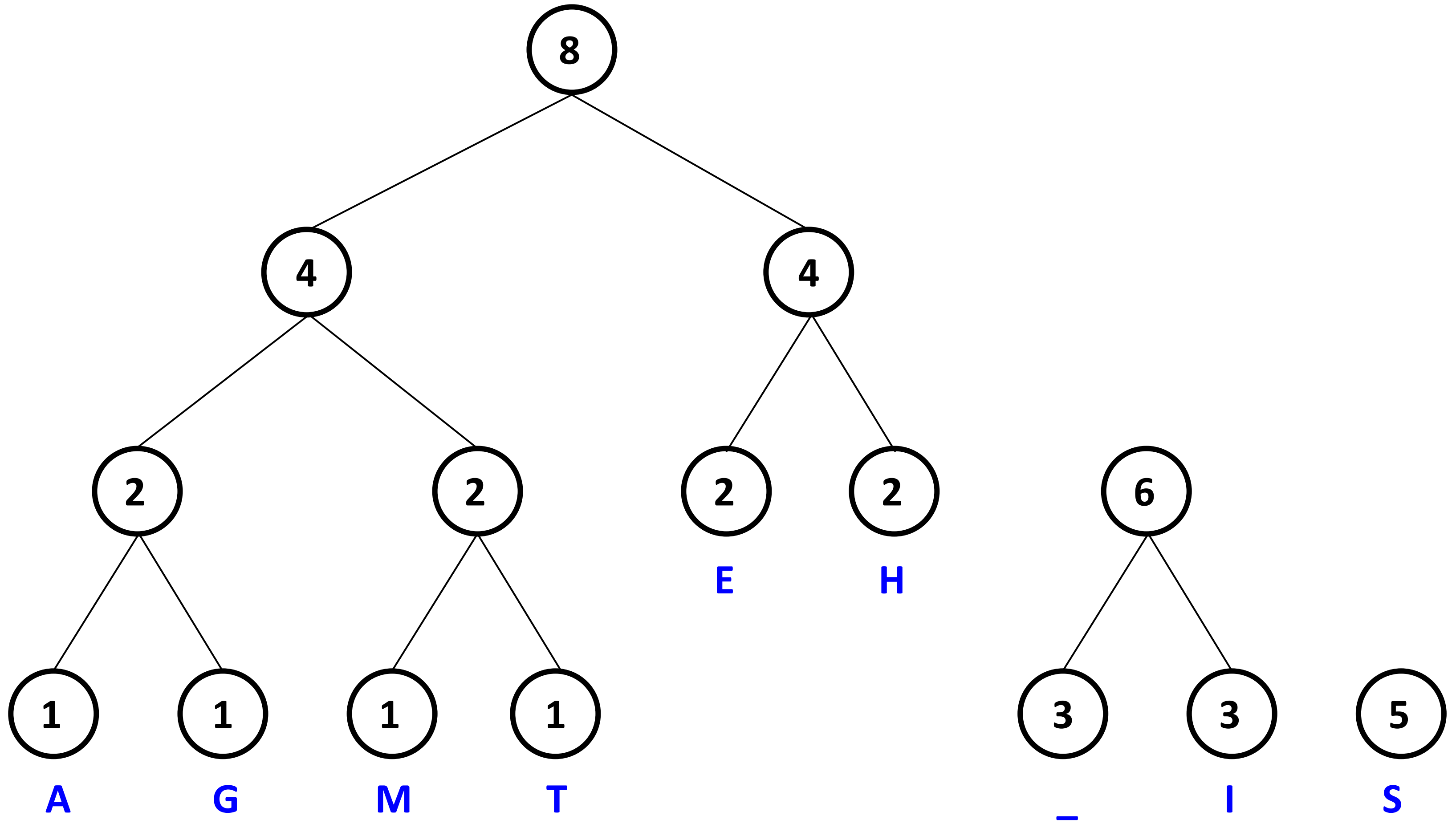
Step 4



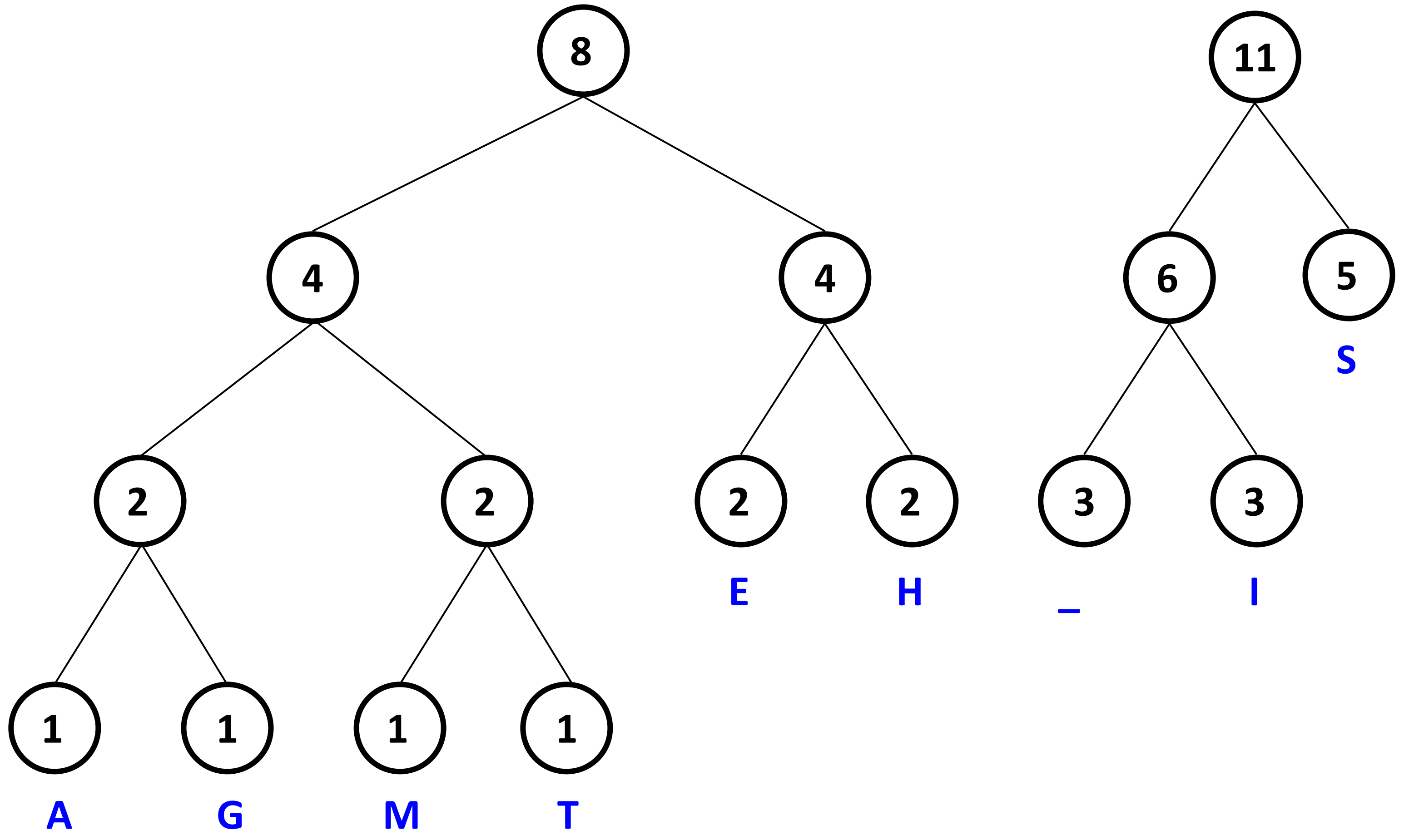
Step 5



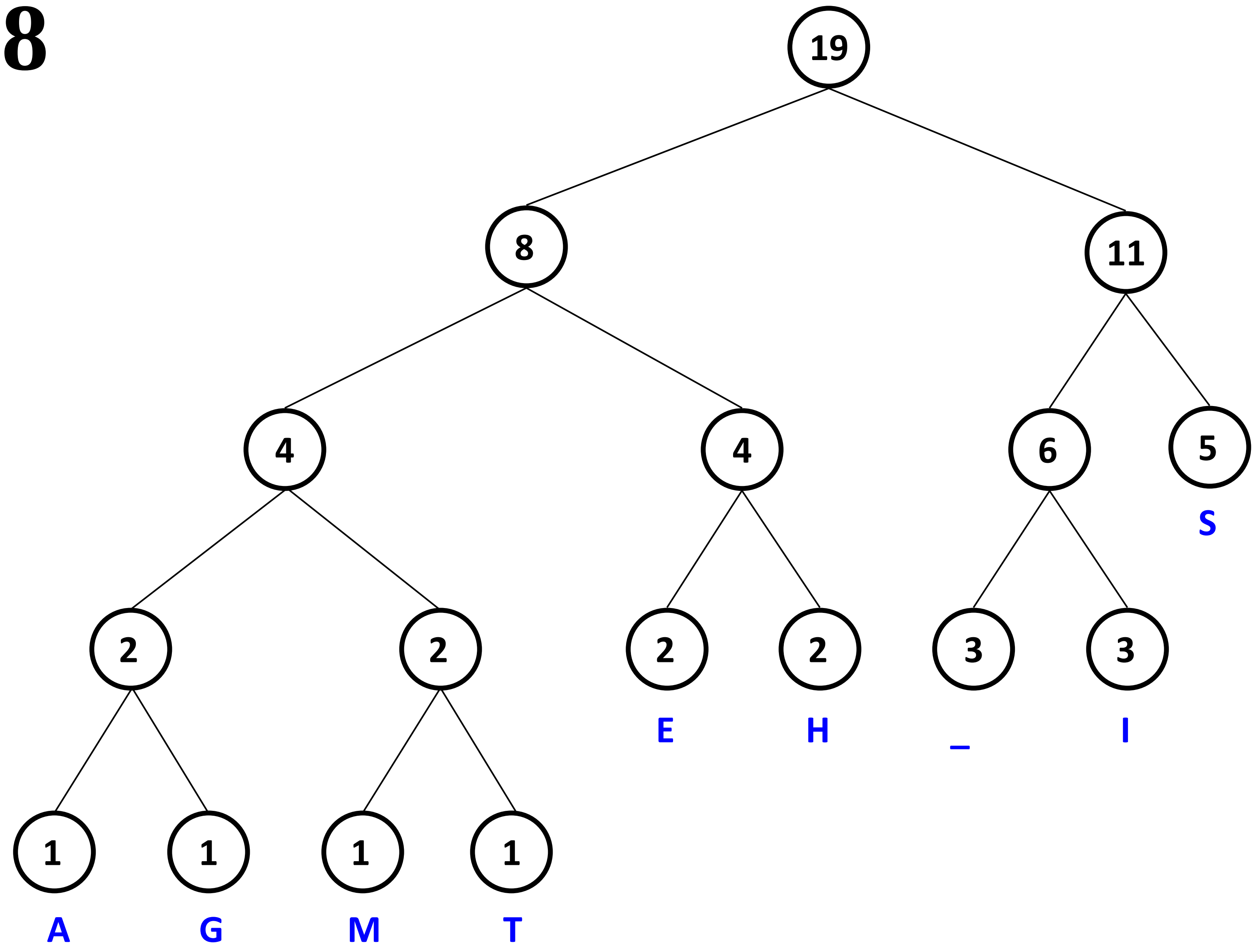
Step 6



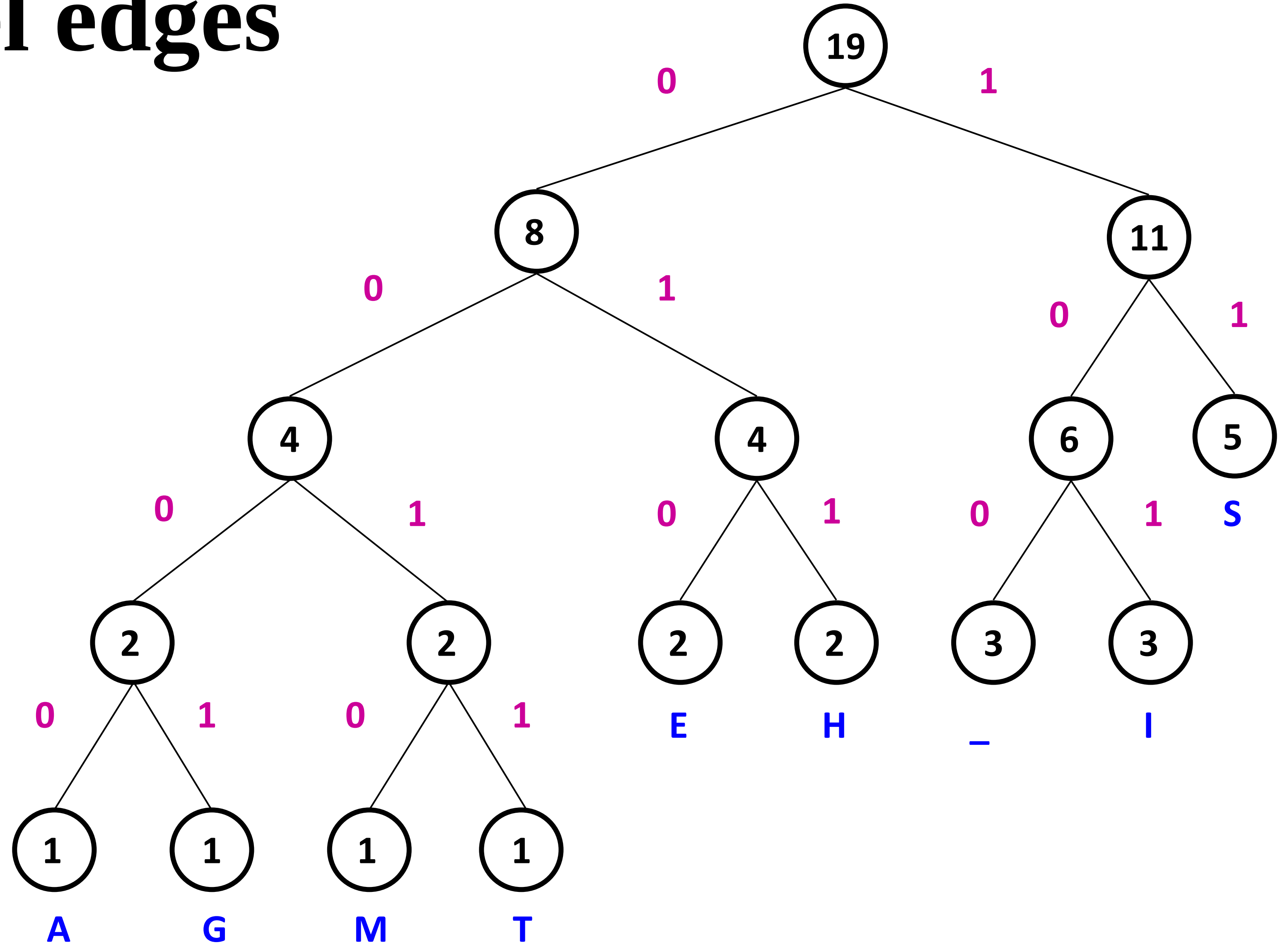
Step 7



Step 8



Label edges



Huffman code & encoded message

This is his message

S	11
E	010
H	011
_	100
I	101
A	0000
G	0001
M	0010
T	0011

0011011101111001011110001110111100001001011111000000001010