Roadmap

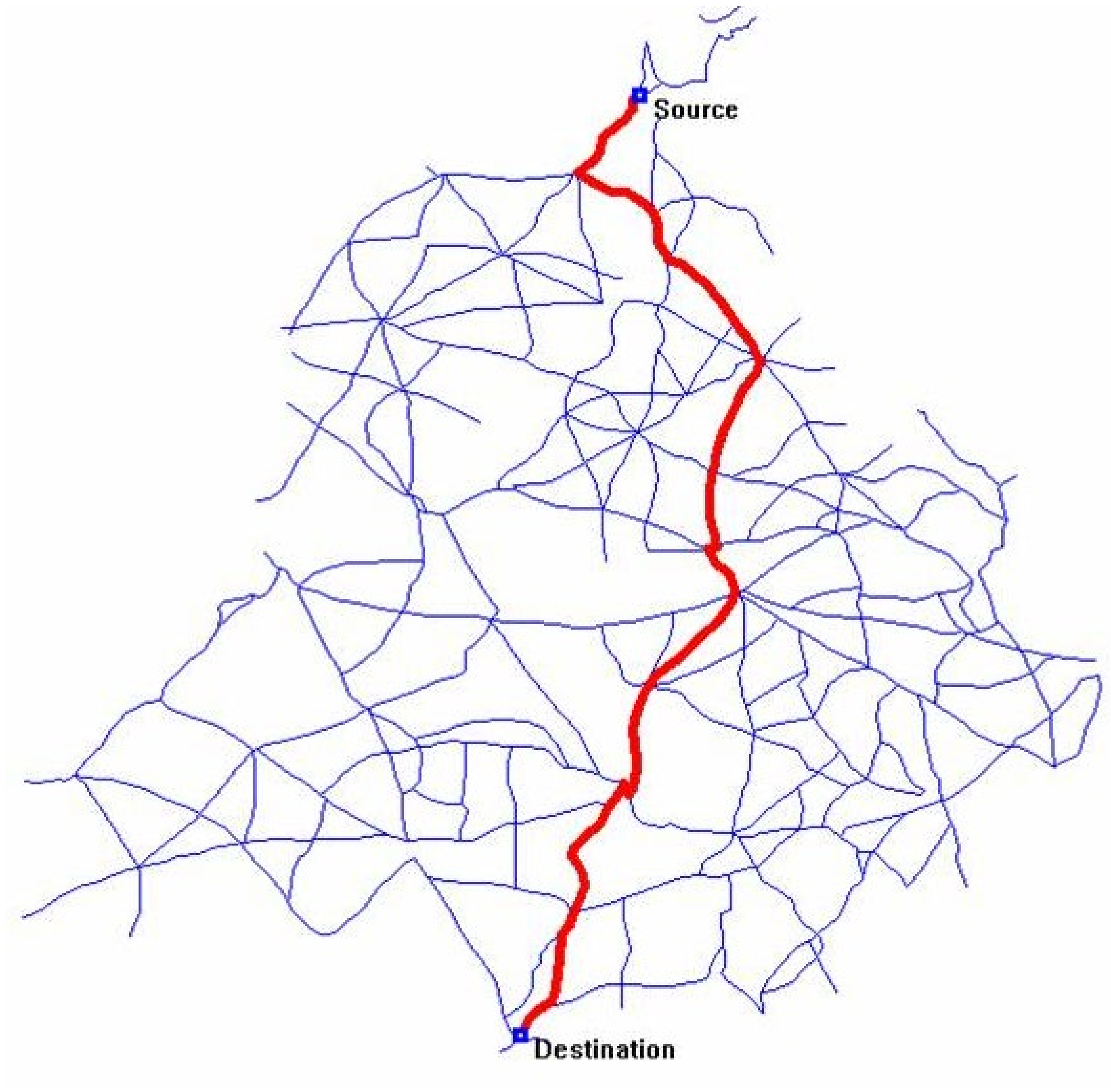
Part I: Shortest Paths

- Dijkstra

Part II: Applications of Shortest Paths

- DNA Alignment
- Constraint Systems

SHORTEST PATHS
(ON WEIGHTED GRAPHS)



Shortest Path Problem

Basic question: find the shortest path!

- Source-to-destination: one vertex to another
- Single source: one vertex to every other
- All pairs: between all pairs of vertices

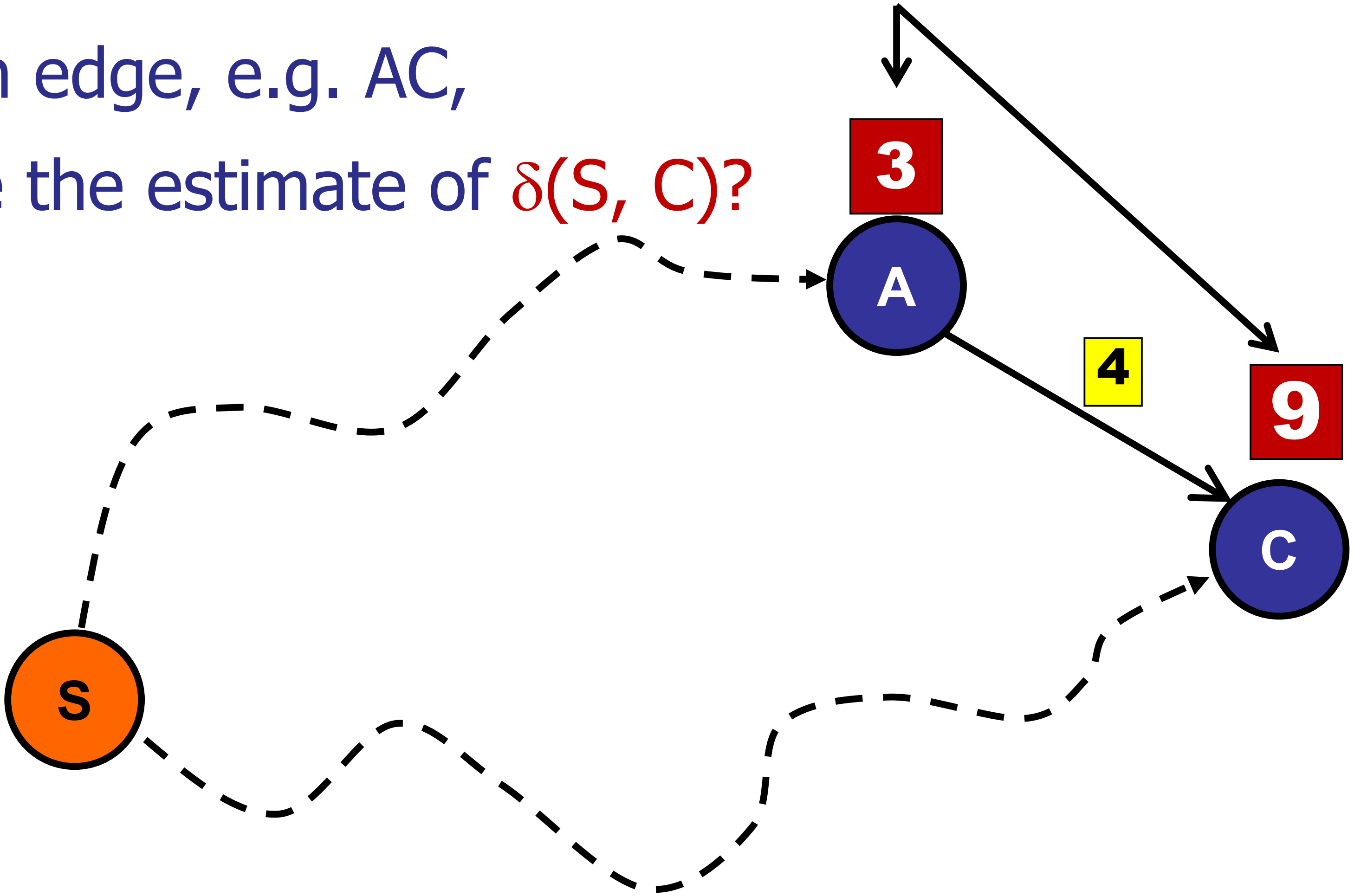
Variants:

- Edge weights: non-negative, arbitrary, Euclidean, …
- Cycles: cyclic, acyclic, no negative cycles

Shortest Paths

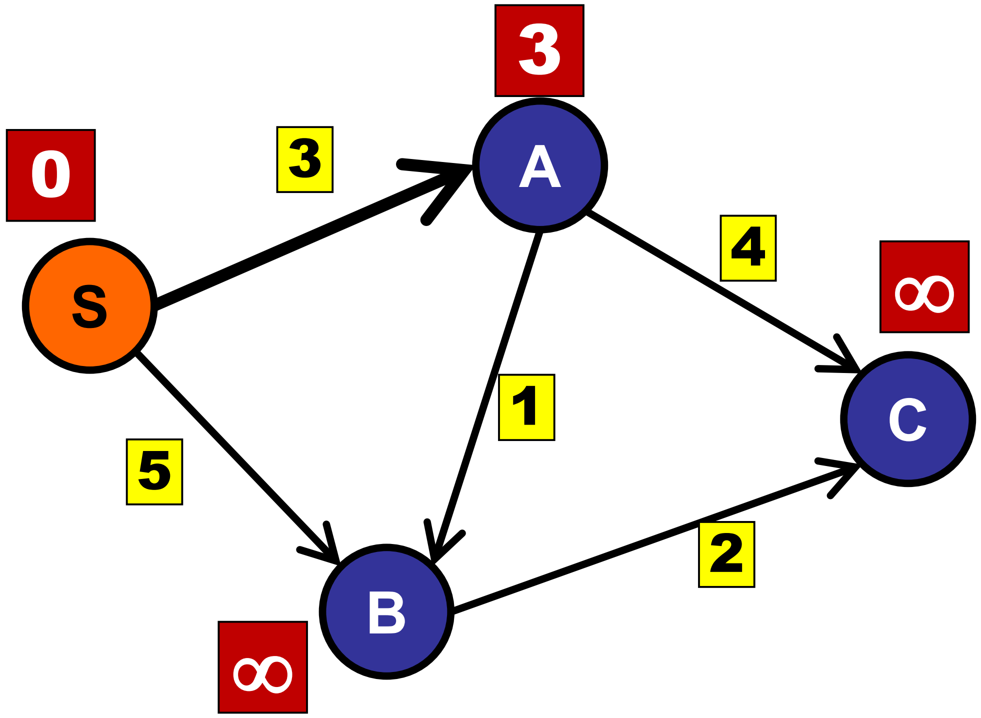
Maintain estimate for each distance:

- Lets say we have some estimate already
- For each edge, e.g. AC,
- Improve the estimate of δ(S, C)?



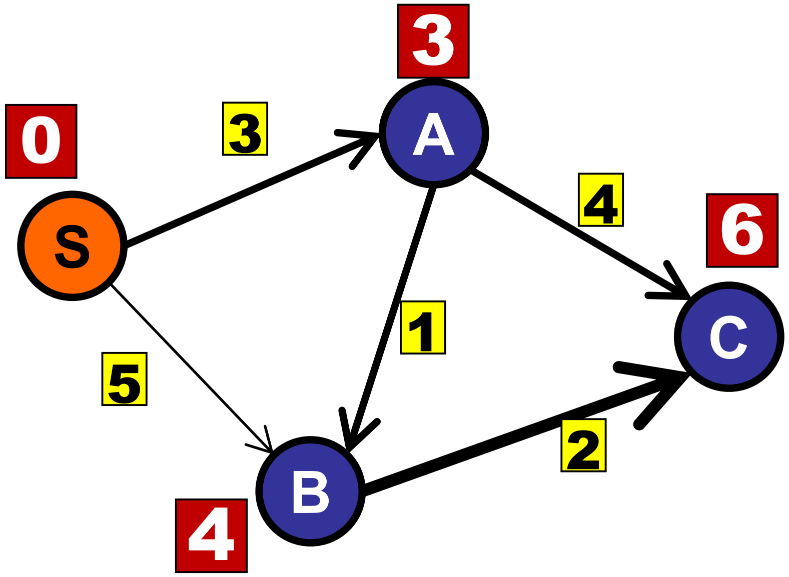
Shortest Paths

```
relax(int u, int v){
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
}
```



Bellman-Ford

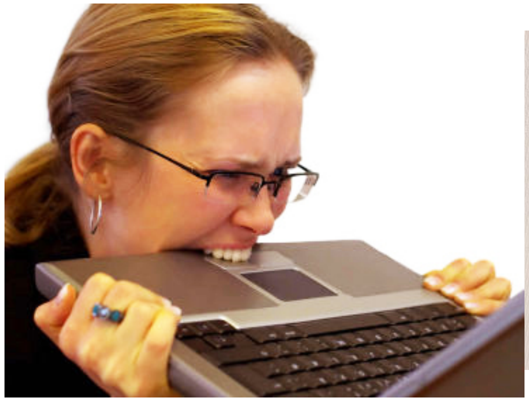
```
n = V.length;
for (i=0; i<n; i++)
    for (each edge e in the graph)
        relax(e)
```



Bellman-Ford Summary

Basic idea:

- Repeat |V| times: relax every edge
- Stop when “converges”.
- O(VE) time.



Special issues:

- If negative weight-cycle: impossible.
- Use Bellman-Ford to **detect** negative weight cycle.
- If all weights are the same, use BFS.

Today

Key idea:

Relax the edges in the "right" order.

Only relax each edge once:

- O(E) cost (for relaxation step).

Edsger W. Dijkstra

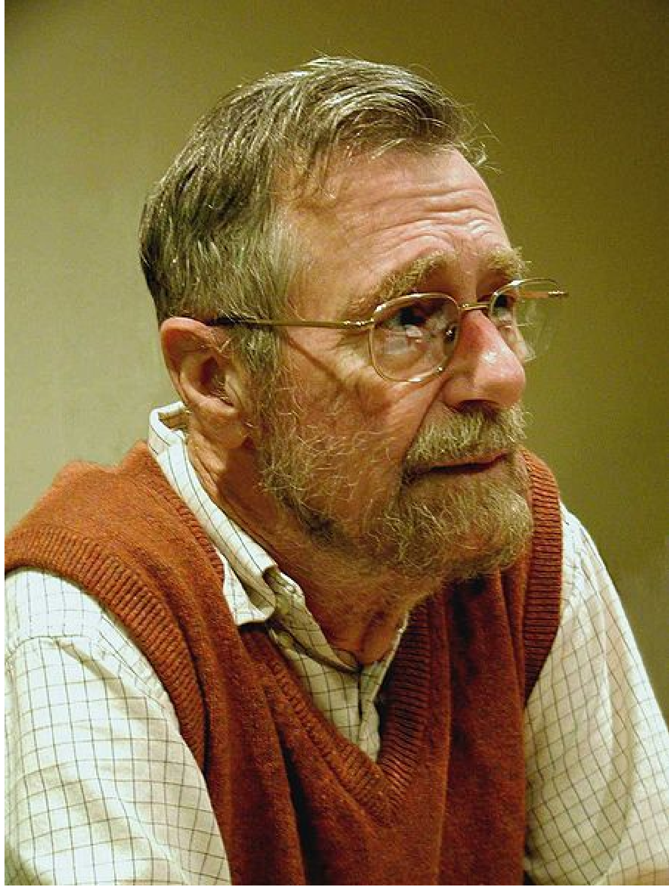
"Computer science is no more about computers than astronomy is about telescopes."

"The question of whether a computer can think is no more interesting than the question of whether a submarine can swim."

"There should be no such thing as boring mathematics."

"Elegance is not a dispensable luxury but a factor that decides between success and failure."

"Simplicity is prerequisite for reliability."


1930-2002

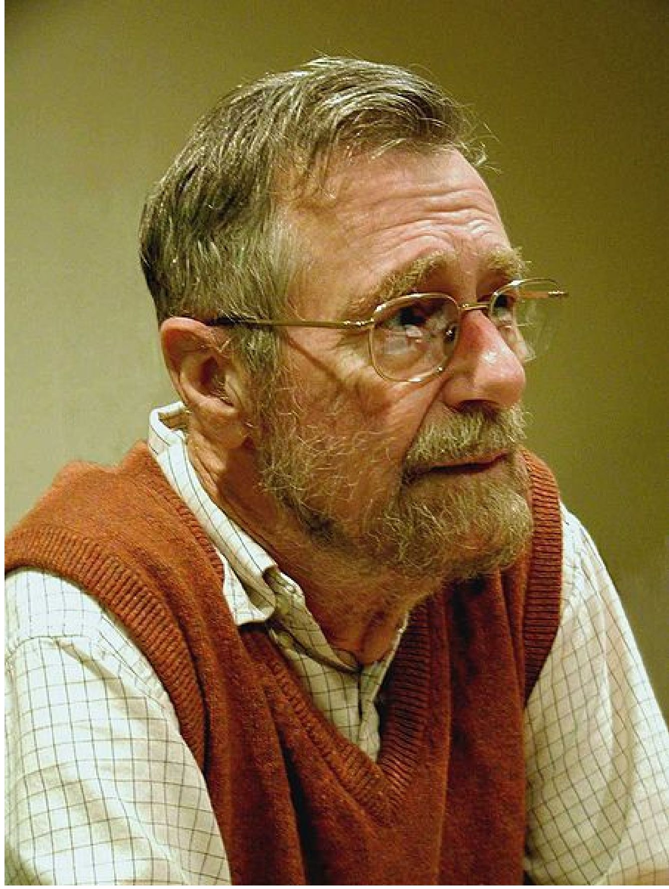
Edsger W. Dijkstra

"It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration."

"The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense."

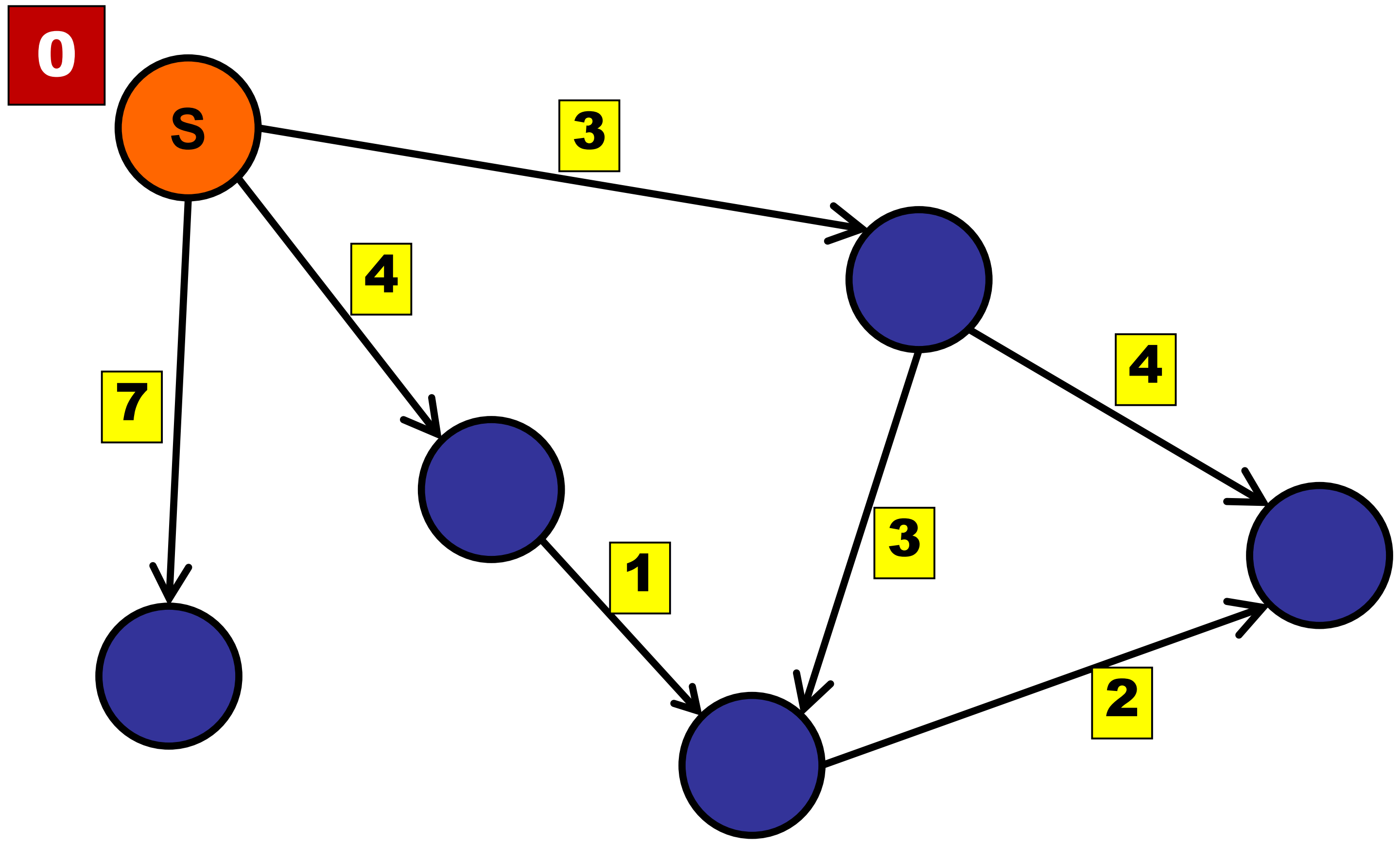
"APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums."

"Object-oriented programming is an exceptionally bad idea which could only have originated in California."



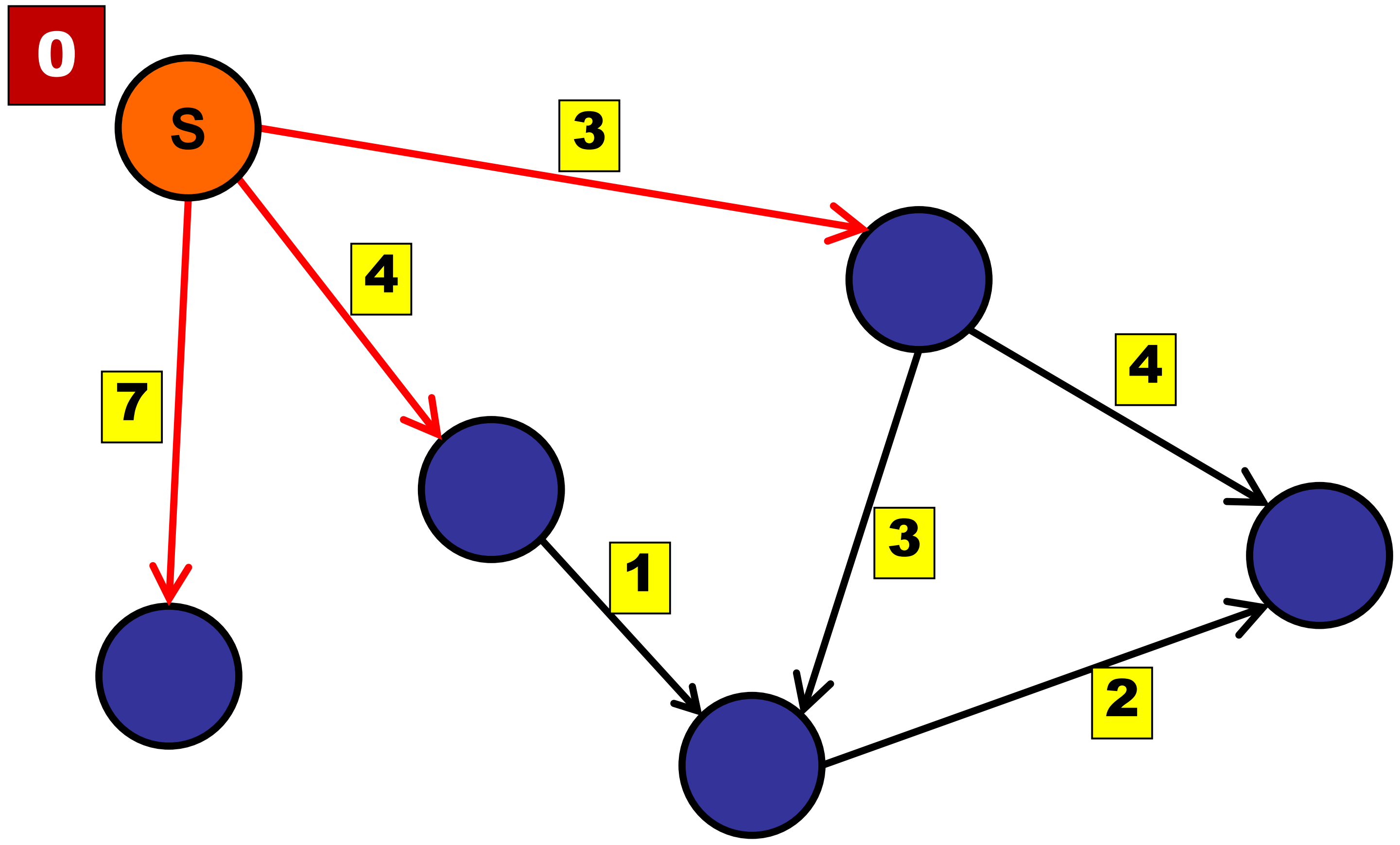
Dijkstra's Algorithm (First Try)

Relax shortest edge first



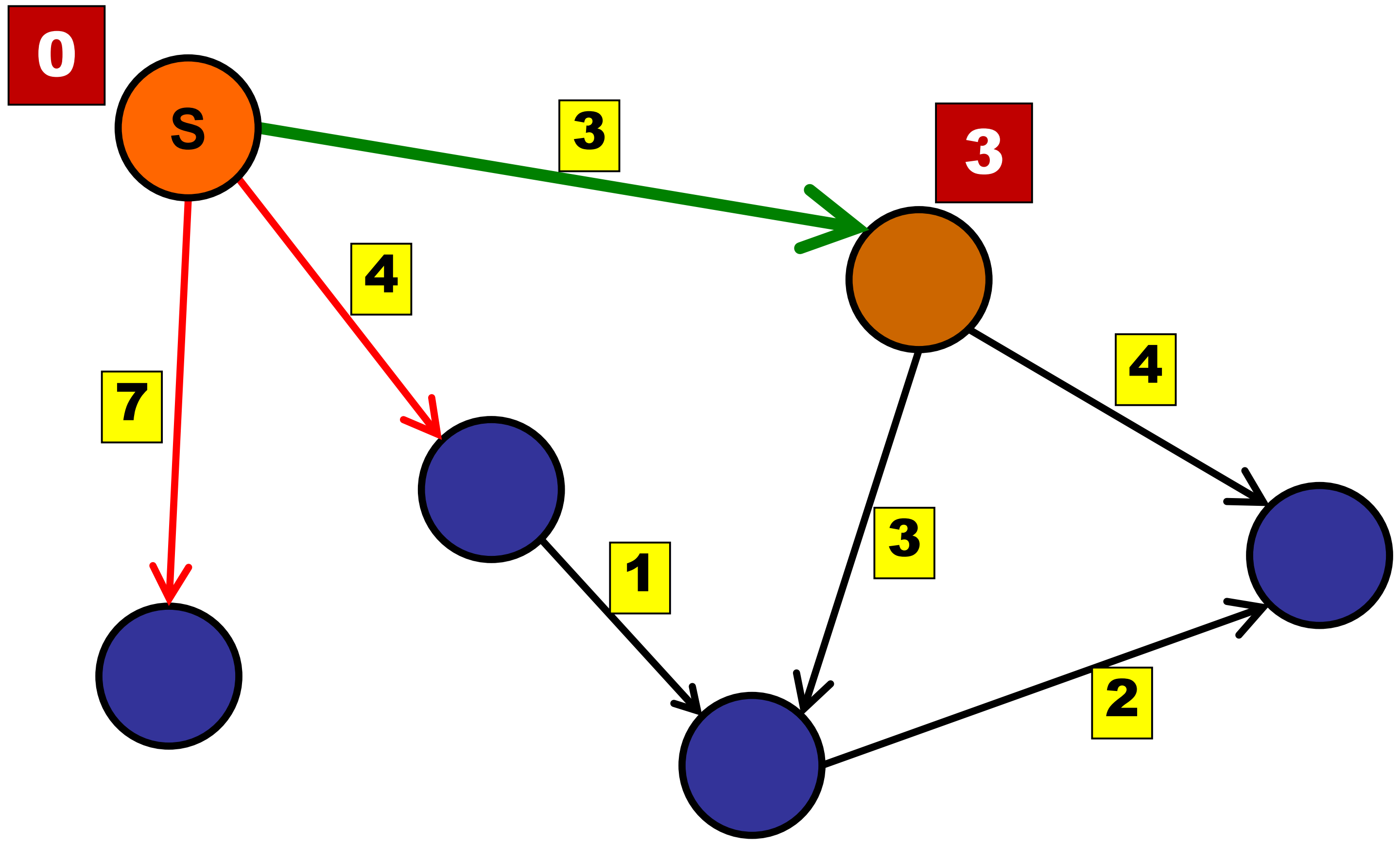
Dijkstra's Algorithm (First Try)

Relax shortest edge first



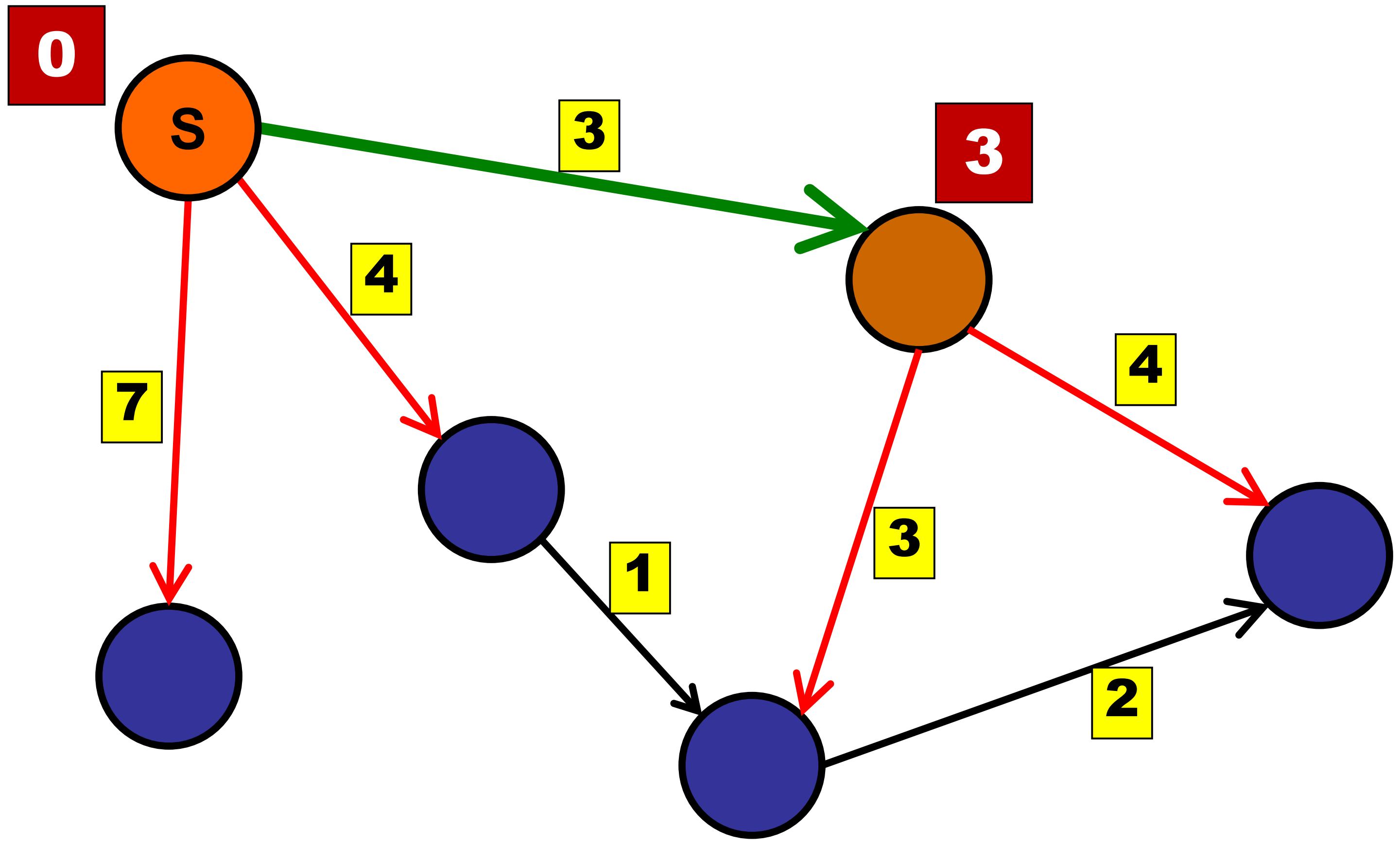
Dijkstra's Algorithm (First Try)

Relax shortest edge first



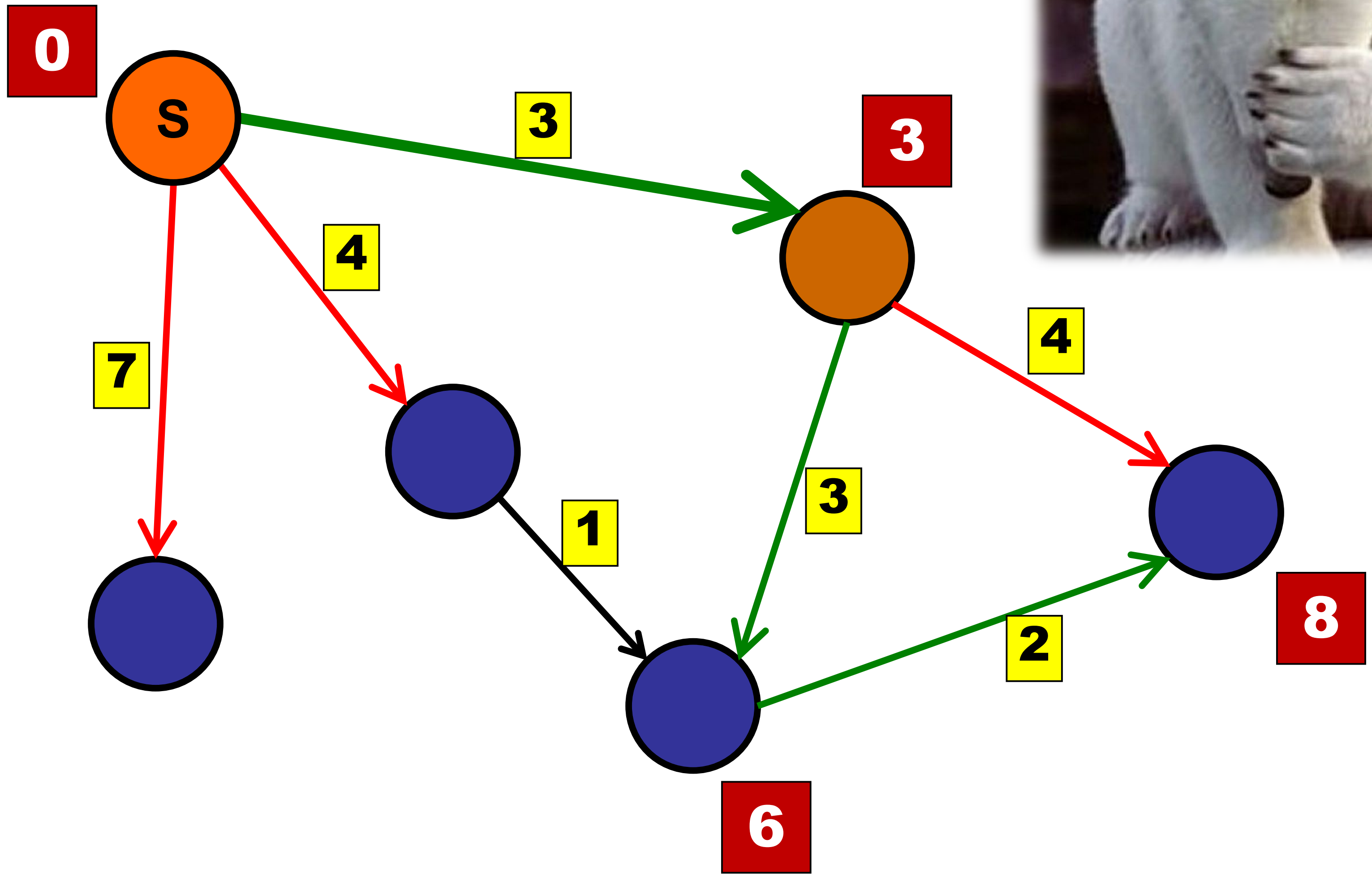
Dijkstra's Algorithm (First Try)

Relax shortest edge first



Dijkstra's Algorithm (Failed Try)

Oops….



Dijkstra's Algorithm

Basic idea:

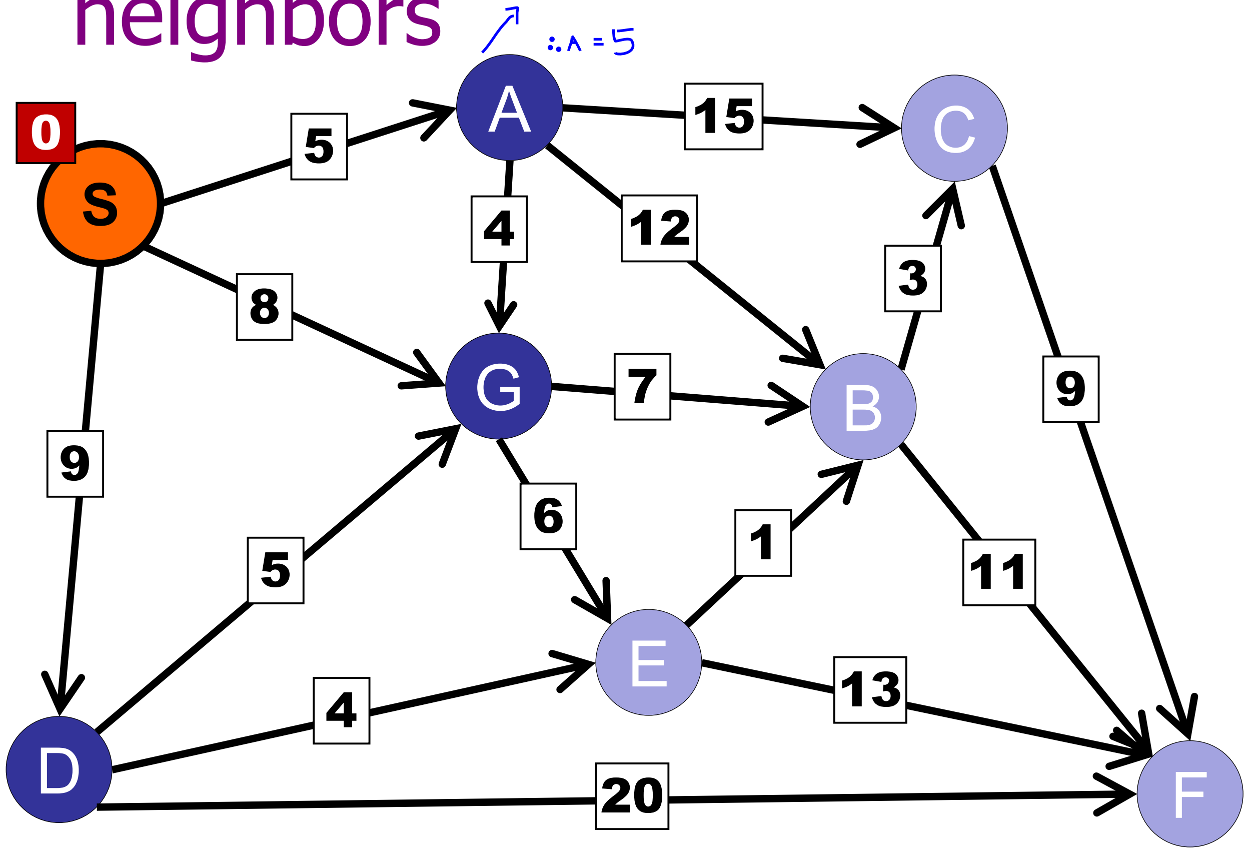
- Initialize:
  - Put all vertices into a priority queue
  - Set all priorities to estimated distances as infinity
  - Set the starting vertex estimated distance as 0
- Repeat until the priority is empty:
  - Extract the vertex v in the priority queue with the shortest estimated distance
  - Relax all the neighbors of v in the priority queue and update their estimated distance

Dijkstra’s Algorithm

Extract S and relax/update neighbors

$\infty > \delta(S, A) = 5$

$\therefore A = 5$

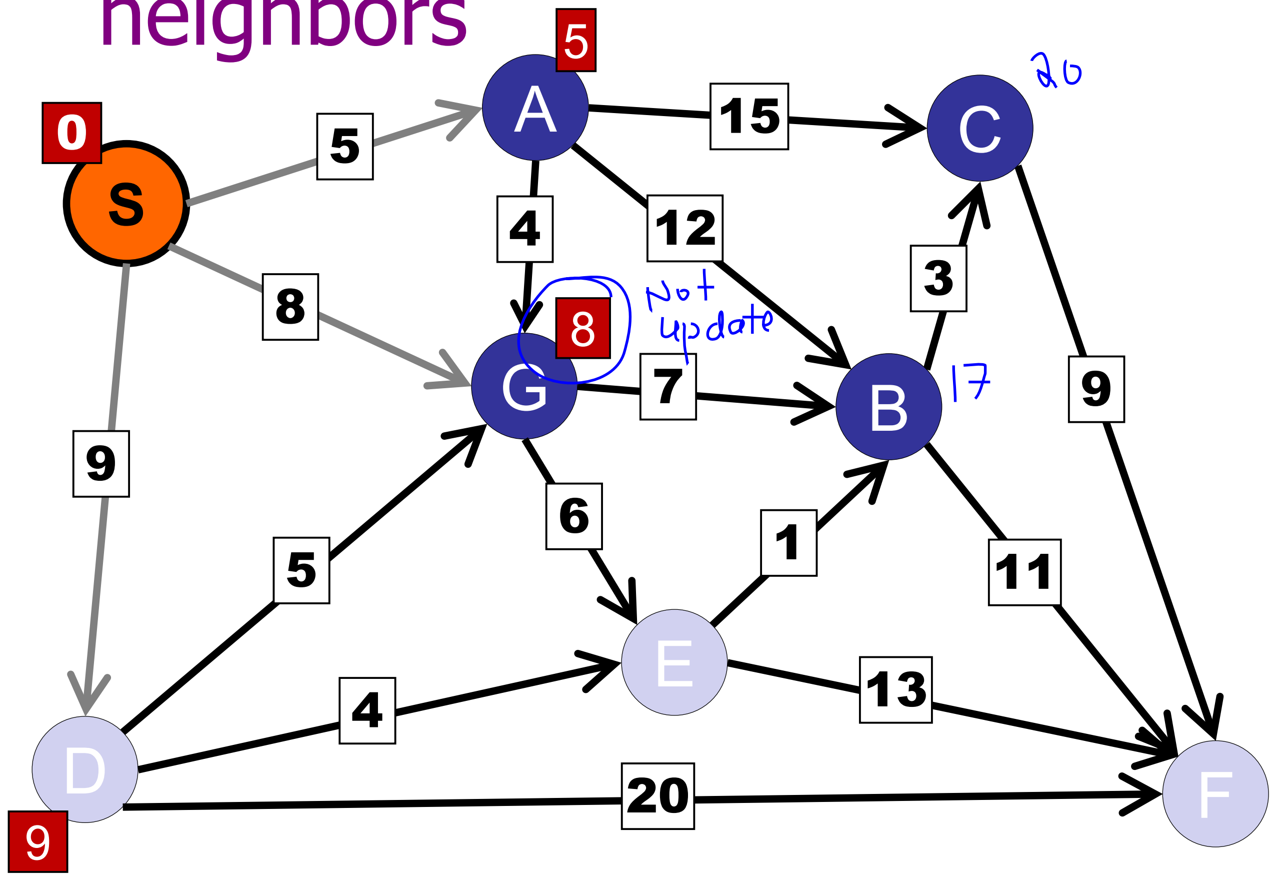


Vertex	Dist.
S	0
A	∞
G	∞
C	∞
B	∞

(Not showing vertices with distance = infinity)

Dijkstra's Algorithm

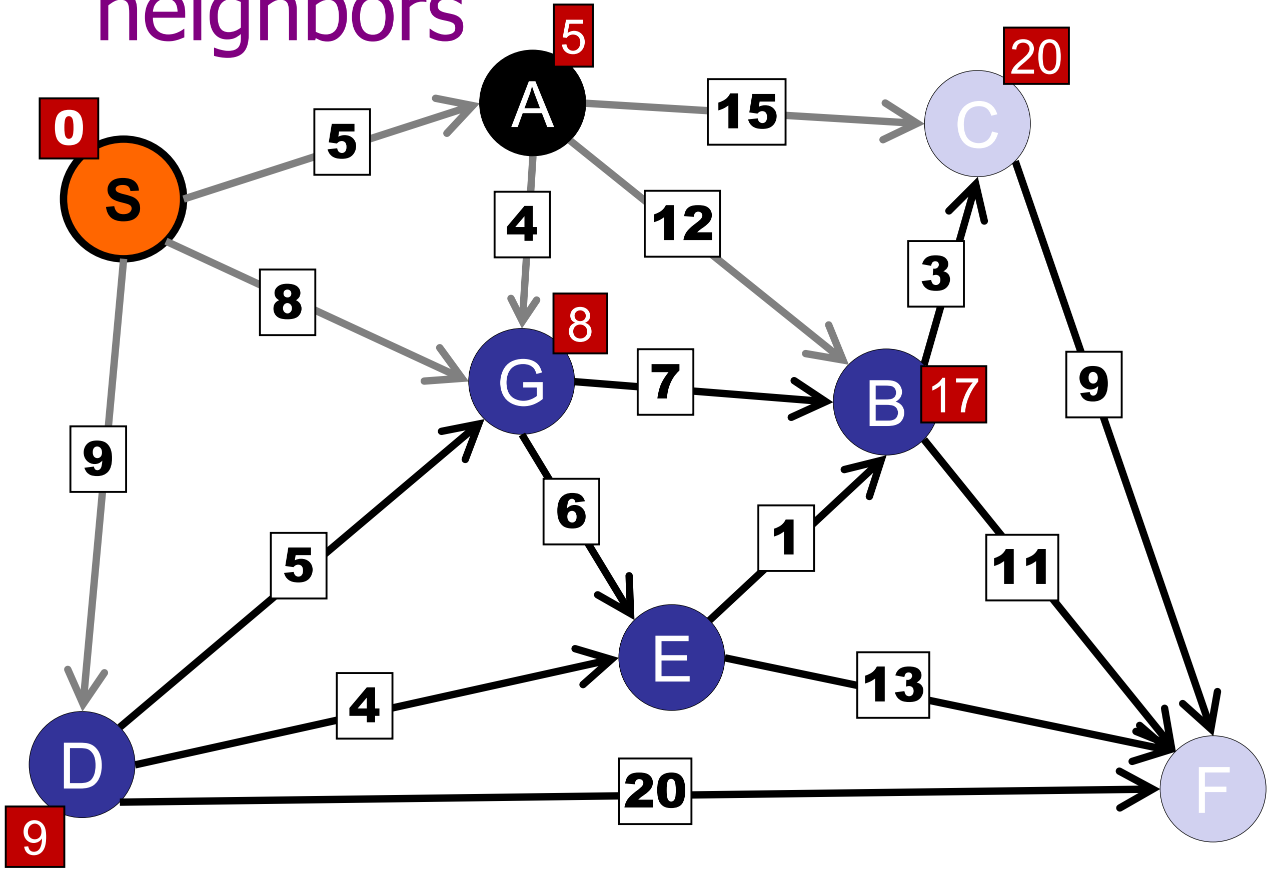
Extract A and relax/update neighbors



Vertex	Dist.
A	5
G	8
D	9

Dijkstra’s Algorithm

Extract G and relax/update neighbors

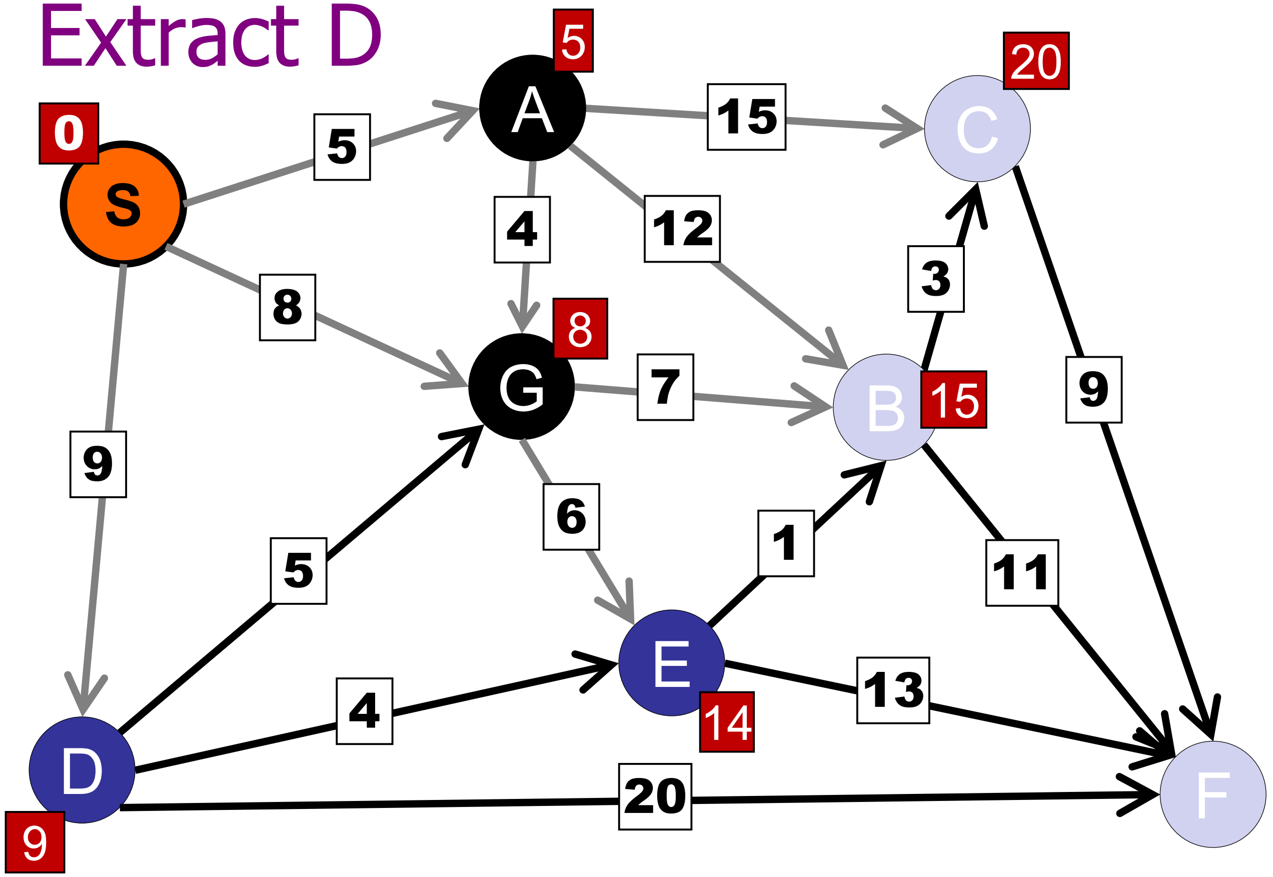


Vertex	Dist.
G	8
D	9
B	17
C	20

Dijkstra’s Algorithm

Dist. of B updated to 15

Extract D

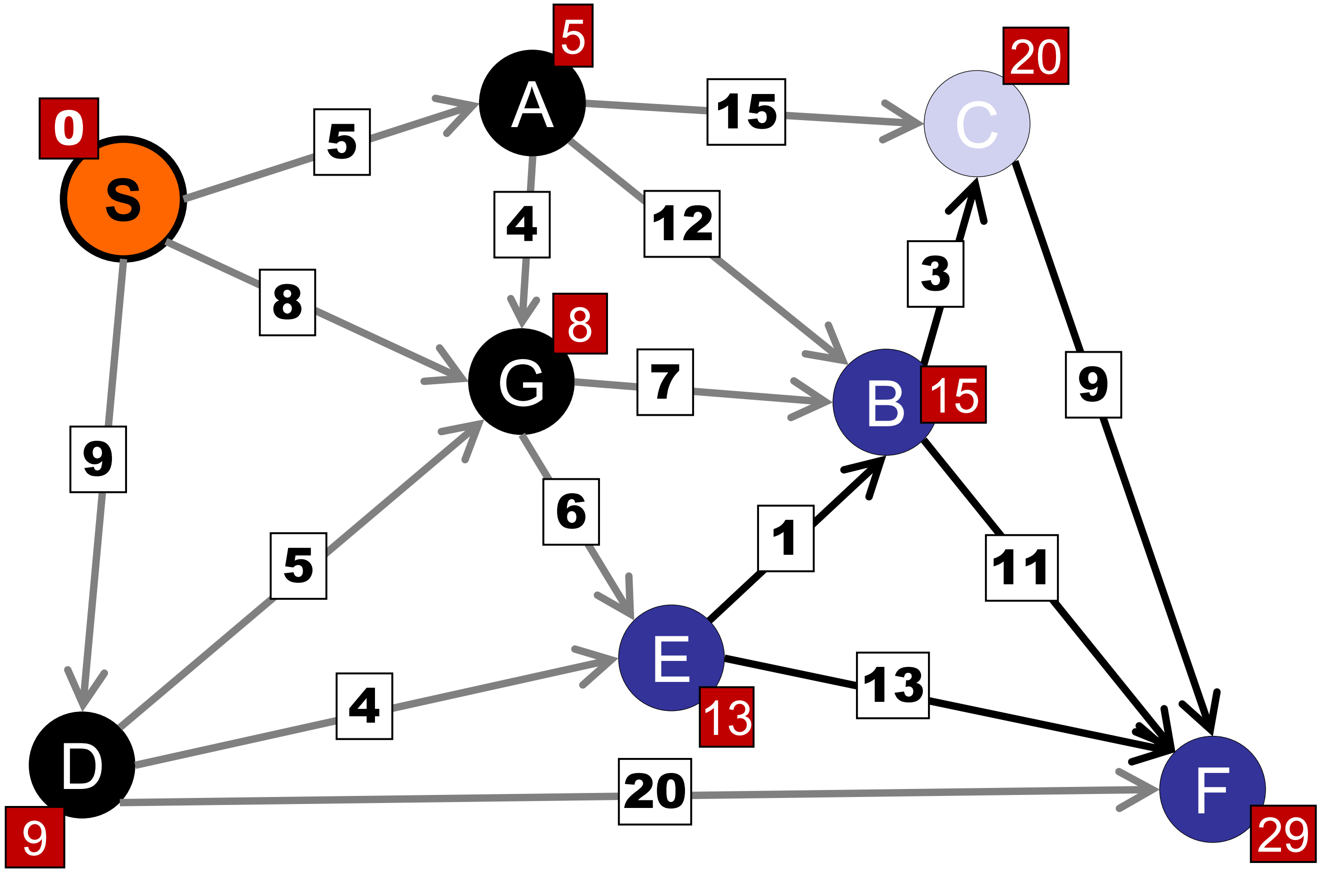


Vertex	Dist.
D	9
E	14
B	15
C	20

G is a neigbor of D. However, since G is already ”dequeued”, G won’t be added back to the PQ anymore

Dijkstra's Algorithm

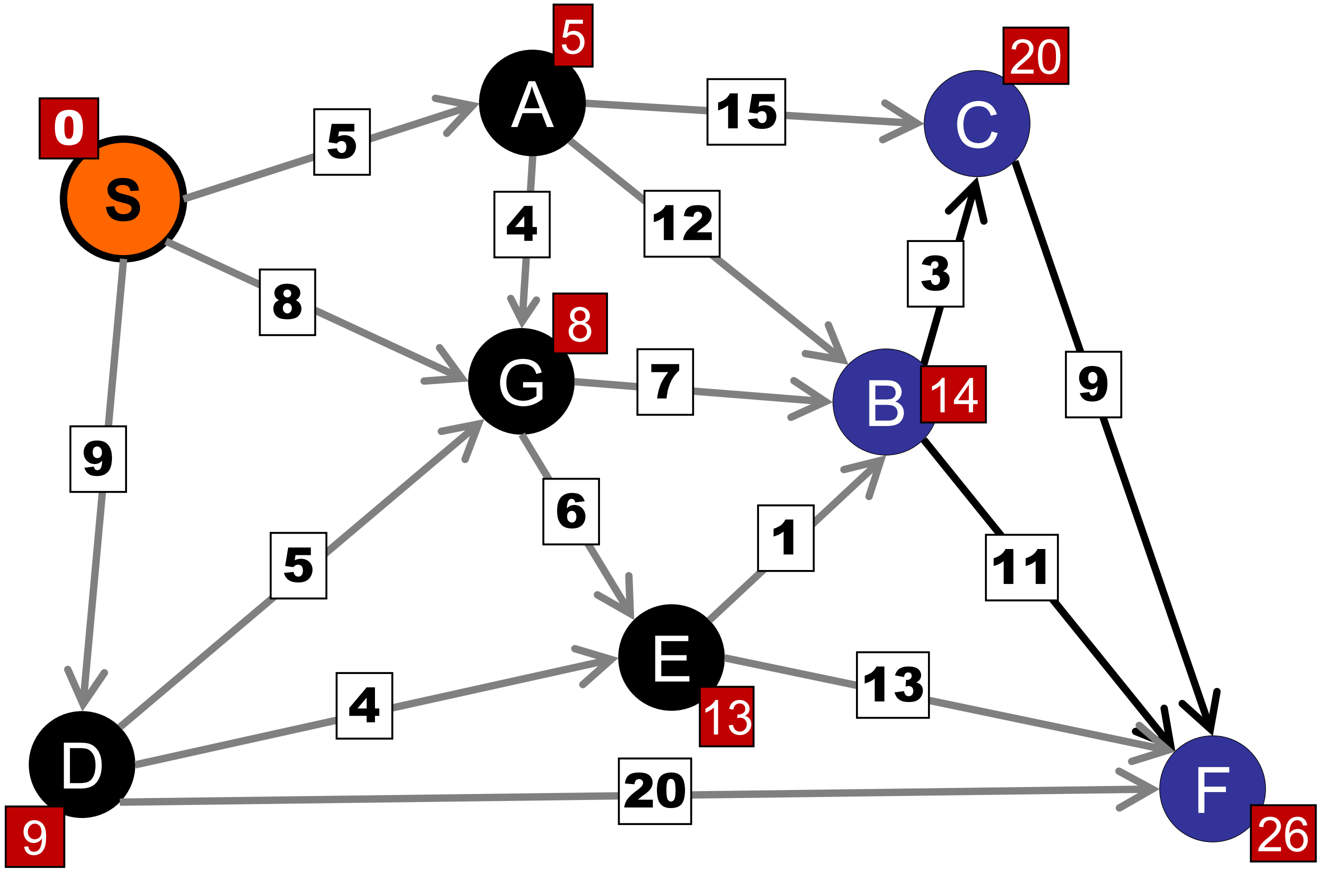
Extract E



Vertex	Dist.
E	**13**
B	15
C	20
F	**29**

Dijkstra's Algorithm

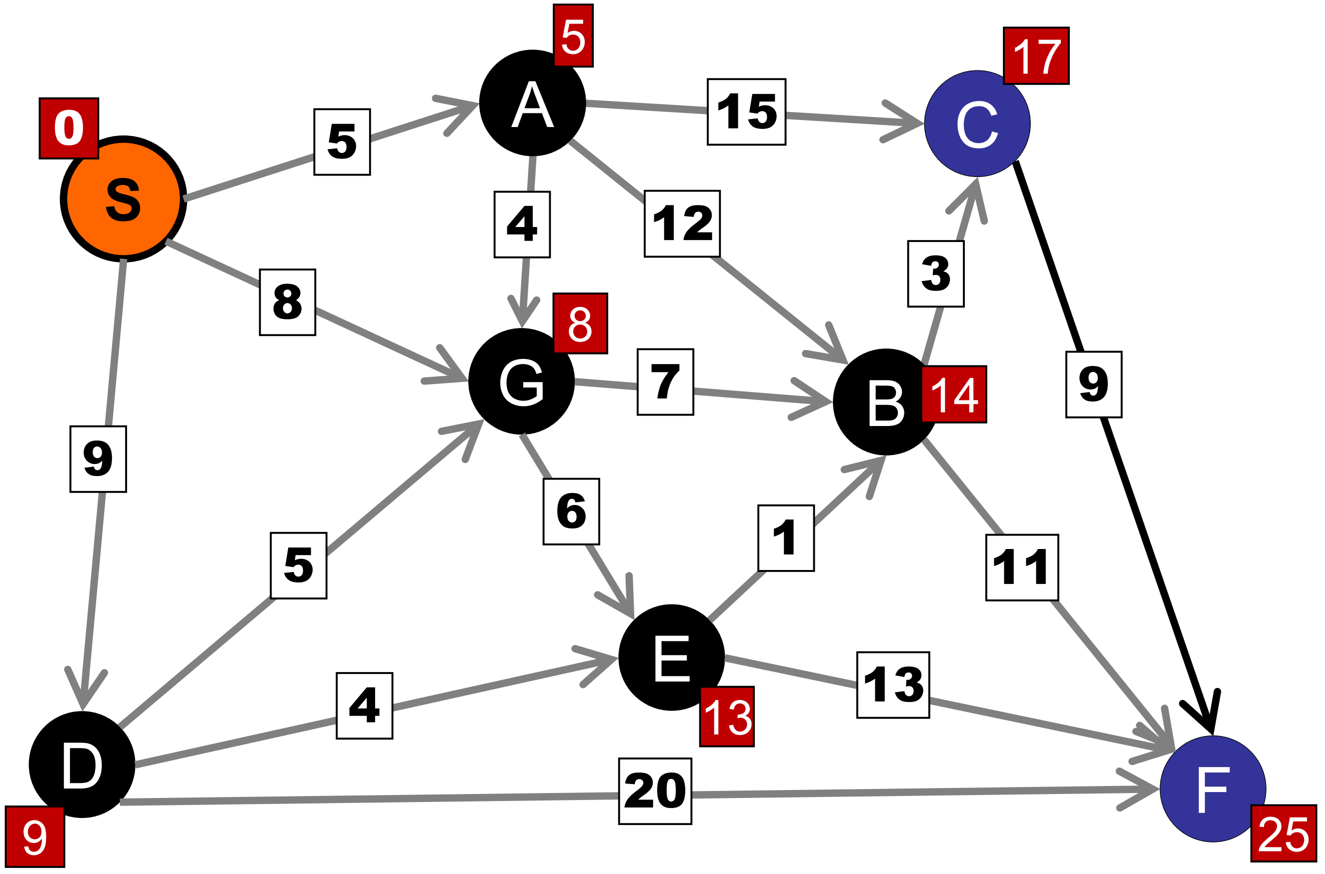
Extract B



Vertex	Dist.
B	**14**
C	20
F	**26**

Dijkstra's Algorithm

Extract C

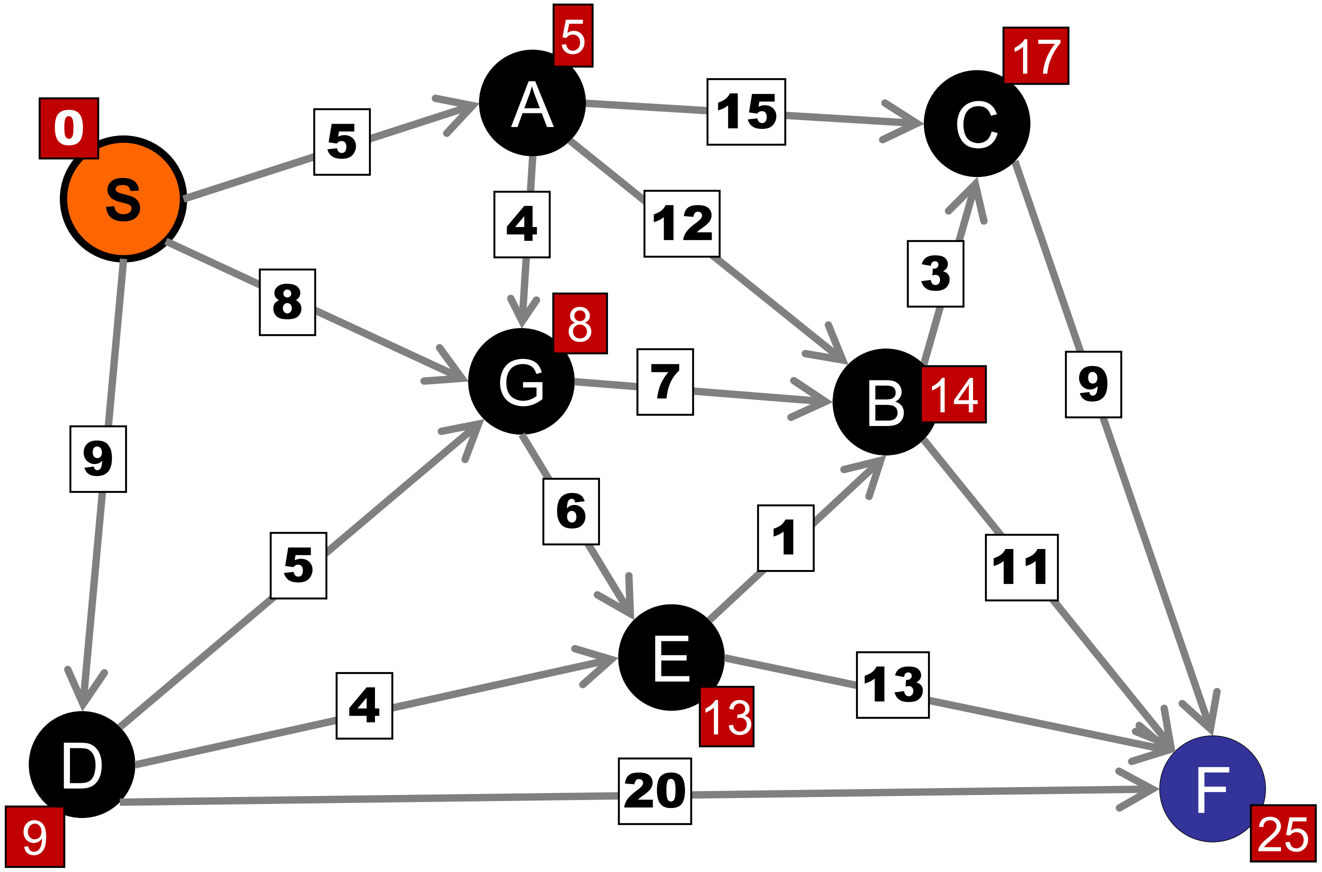


Vertex	Dist.
C	~~20~~ 17
F	25

Dijkstra's Algorithm

Extract F

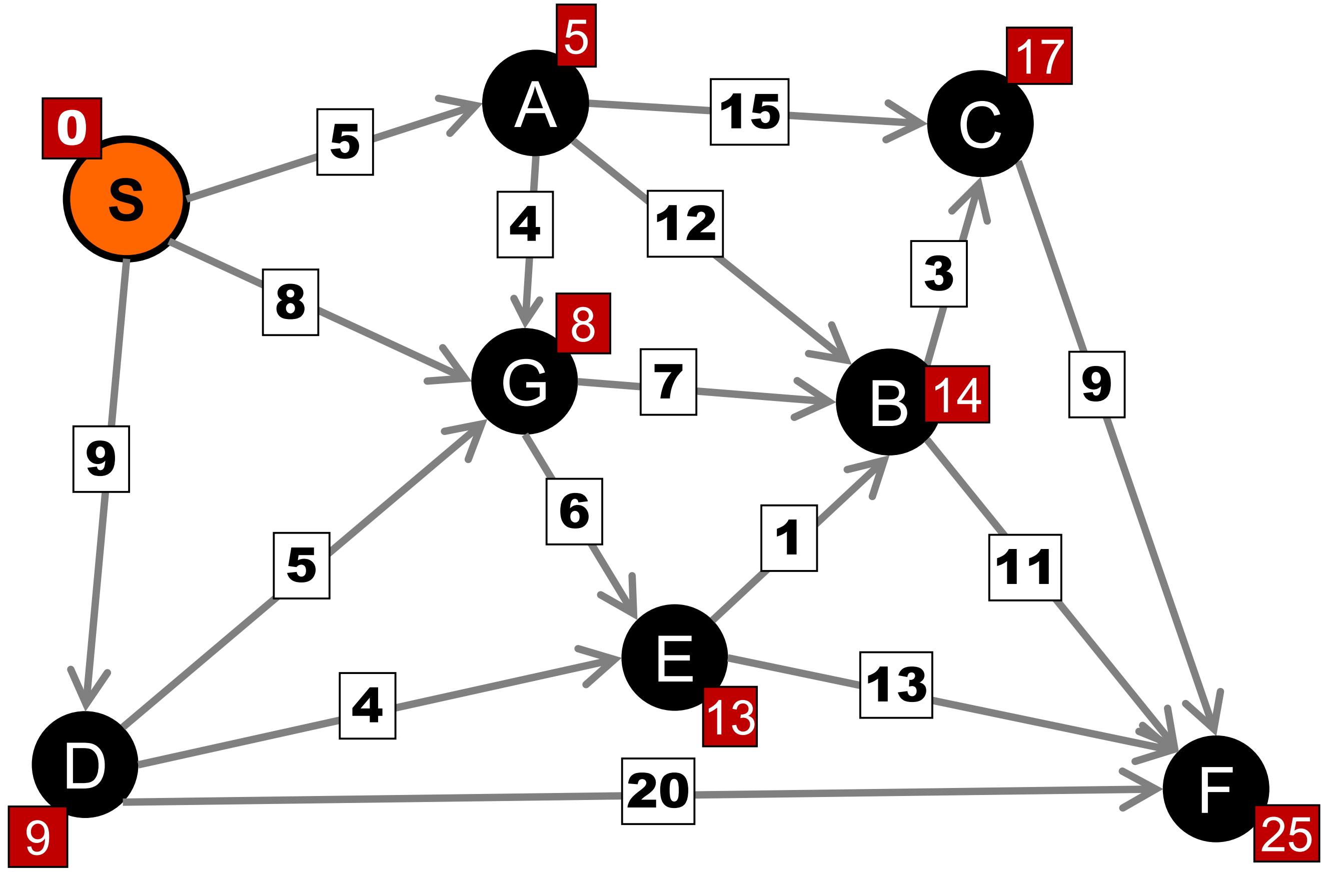
Vertex	Dist.
F	25



Dijkstra’s Algorithm

Done!

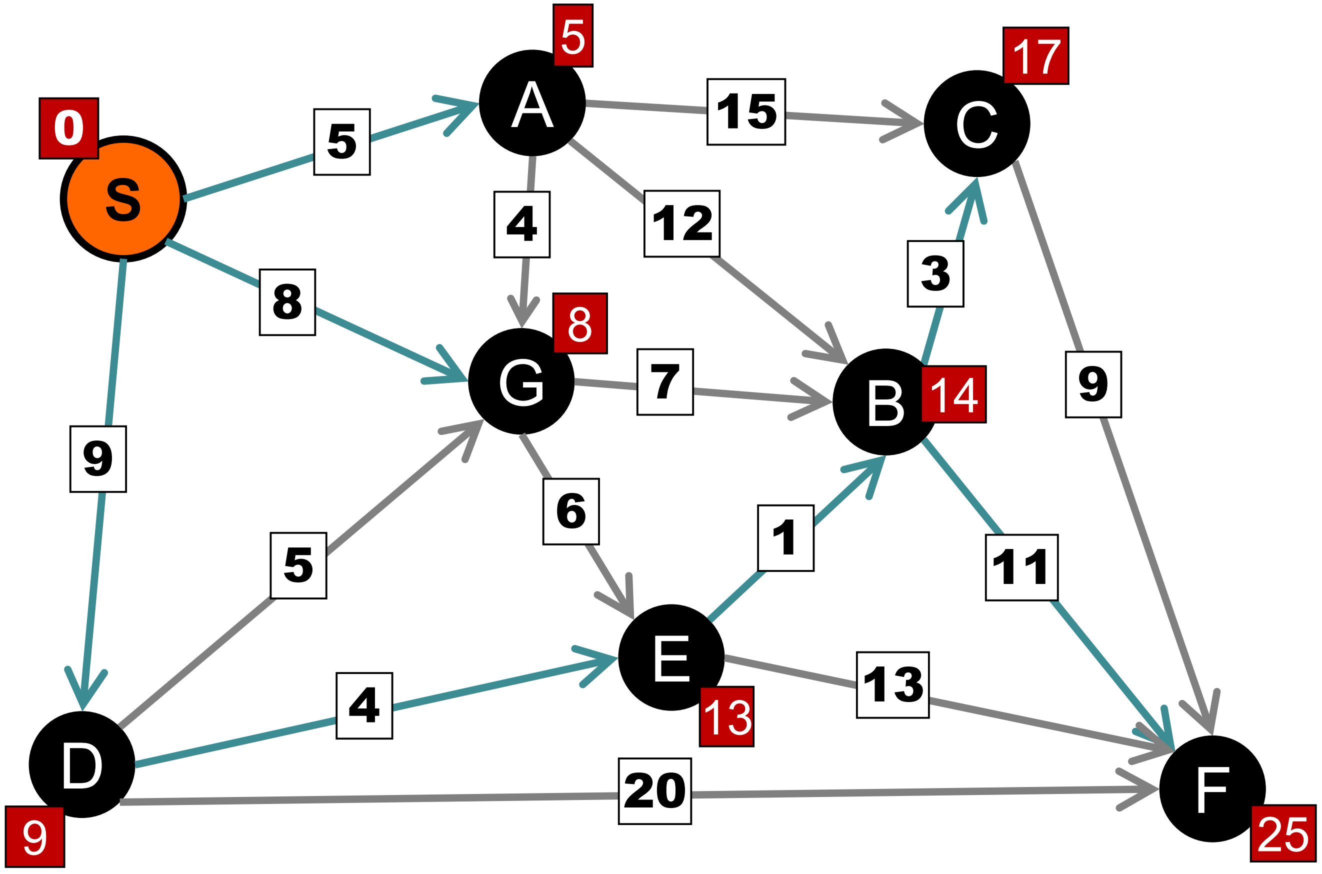
Vertex	Dist.



Dijkstra’s Algorithm

Vertex	Dist.

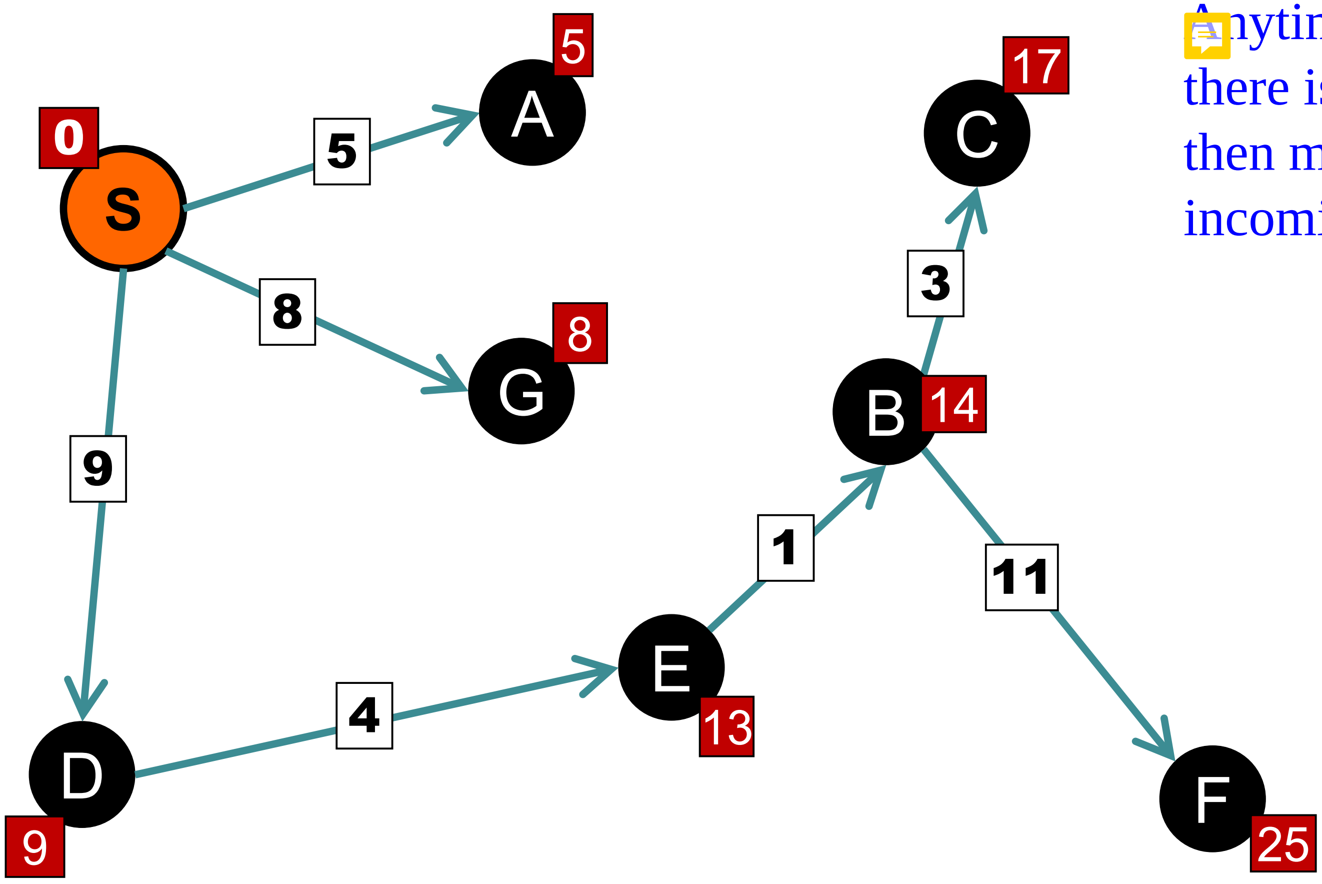
Done



Dijkstra’s Algorithm

Shortest Path Tree

Vertex	Dist.



Anytime there is relaxation, if there is an improve in distance, then make it the parent of incoming vertex

Abstract Data Type

Priority Queue

`void`	`insert(Key k, Priority p)`	*insert k with priority p*
`Data`	`extractMin()`	*remove key with minimum priority*
`void`	`decreaseKey(Key k, Priority p)`	*reduce the priority of key k to priority p*
`boolean`	`contains(Key k)`	*does the priority queue contain key k?*
`boolean`	`isEmpty()`	*is the priority queue empty?*

Notes:

Assume data items are unique.

Dijkstra's Algorithm

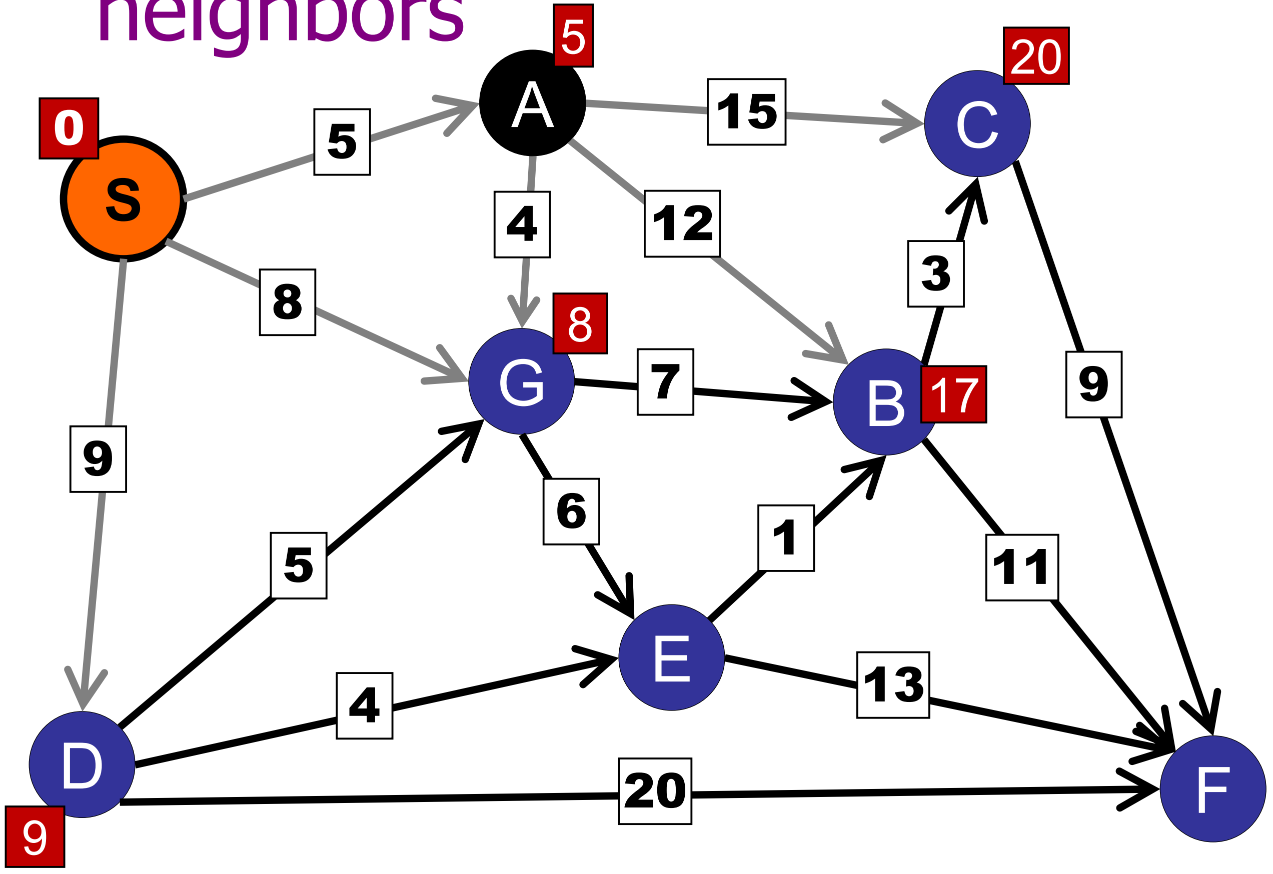
```
relax(Edge e) {
     int v = e.from();
     int w = e.to();
     double weight = e.weight();
     if (distTo[w] > distTo[v] + weight) {
          distTo[w] = distTo[v] + weight;
          parent[w] = v;
          pq.decreaseKey(w, distTo[w]);
     }
}
```

Dijkstra’s Algorithm

```
relax(Edge e) {
    int v = e.from();
    int w = e.to();
    double weight = e.weight();
    if (distTo[w] > distTo[v] + weight) {
        distTo[w] = distTo[v] + weight;
        parent[w] = v;
        pq.decreaseKey(w, distTo[w]);
    }
}
```

Dijkstra's Algorithm

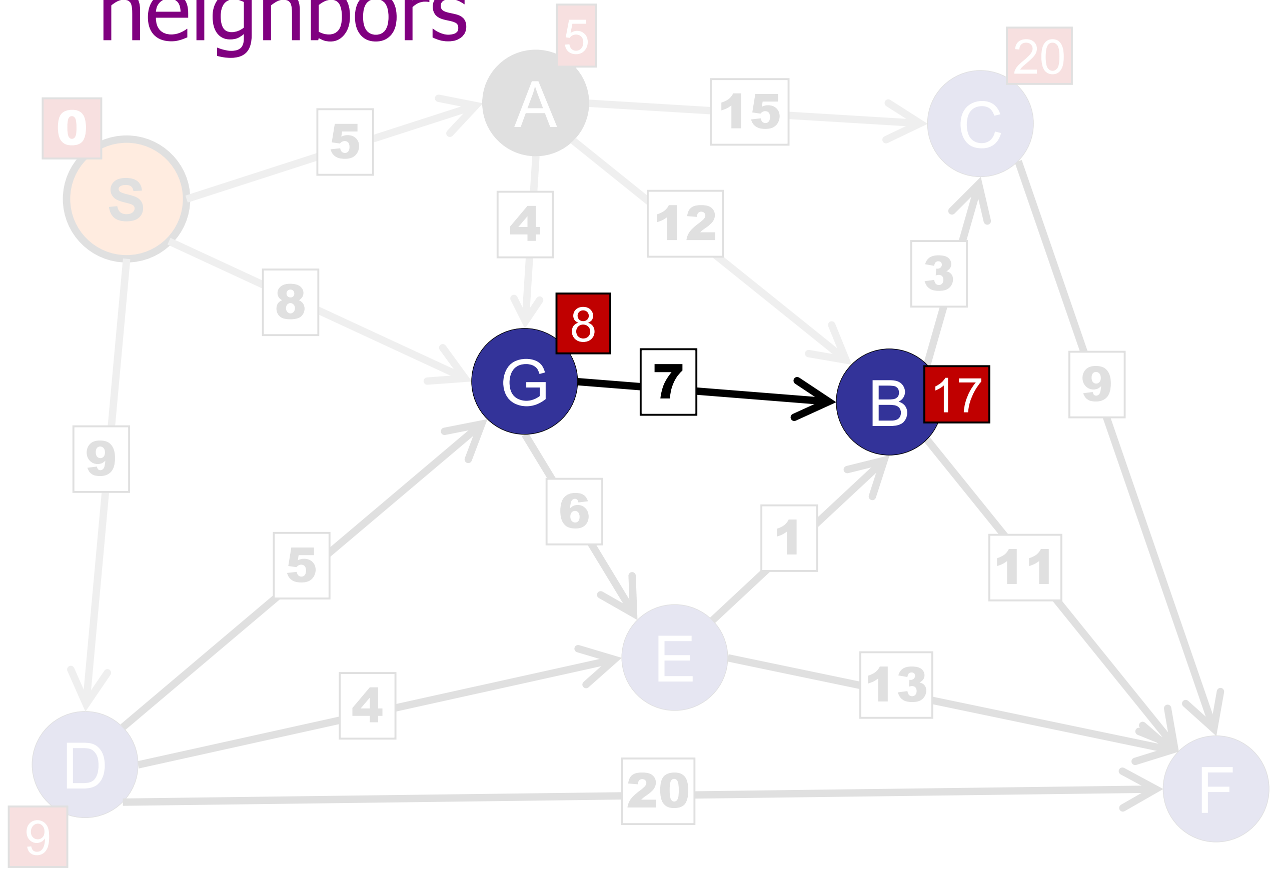
Remove G and relax/update neighbors



Vertex	Dist.
G	8
D	9
B	17
C	20

Dijkstra's Algorithm

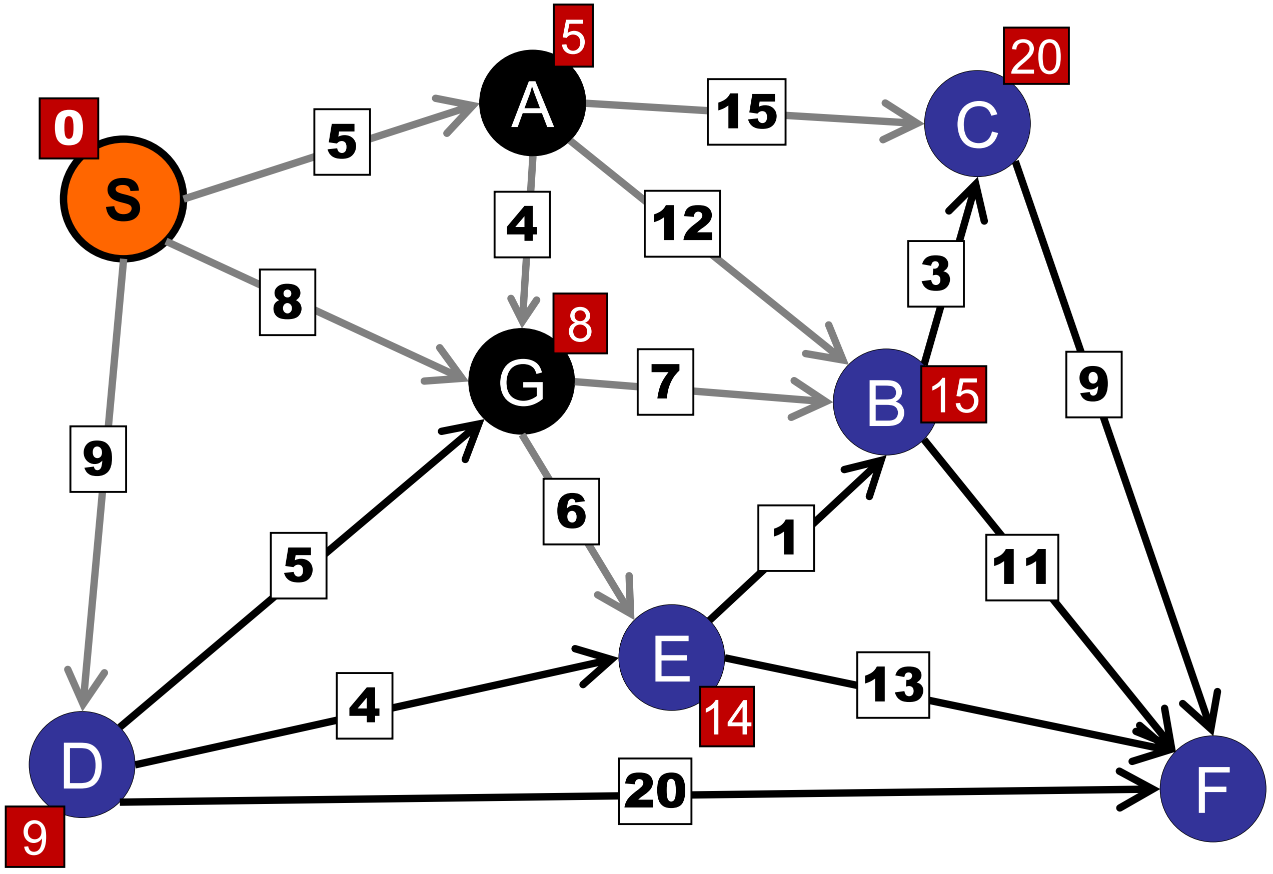
Remove G and relax/update neighbors



Vertex	Dist.
G	8
D	9
B	17
C	20

Dijkstra's Algorithm

Remove G and relax.



Vertex	Dist.
D	9
E	14
B	15
C	20

Dijkstra’s Algorithm

```
relax(Edge e) {
    int v = e.from();
    int w = e.to();
    double weight = e.weight();
    if (distTo[w] > distTo[v] + weight) {
        distTo[w] = distTo[v] + weight;
        parent[w] = v;
        pq.decreaseKey(w, distTo[w]);
    }
}
```

Dijkstra's Algorithm

Since every edge is only relaxed 1 time (since only will travel the edge once since only extract each vertex 1 time

```
relax(Edge e) {
    int v = e.from();
    int w = e.to();
    double weight = e.weight();
    if (distTo[w] > distTo[v] + weight) {
        distTo[w] = distTo[v] + weight;
        parent[w] = v;
        pq.decreaseKey(w, distTo[w]);
    }
}
```

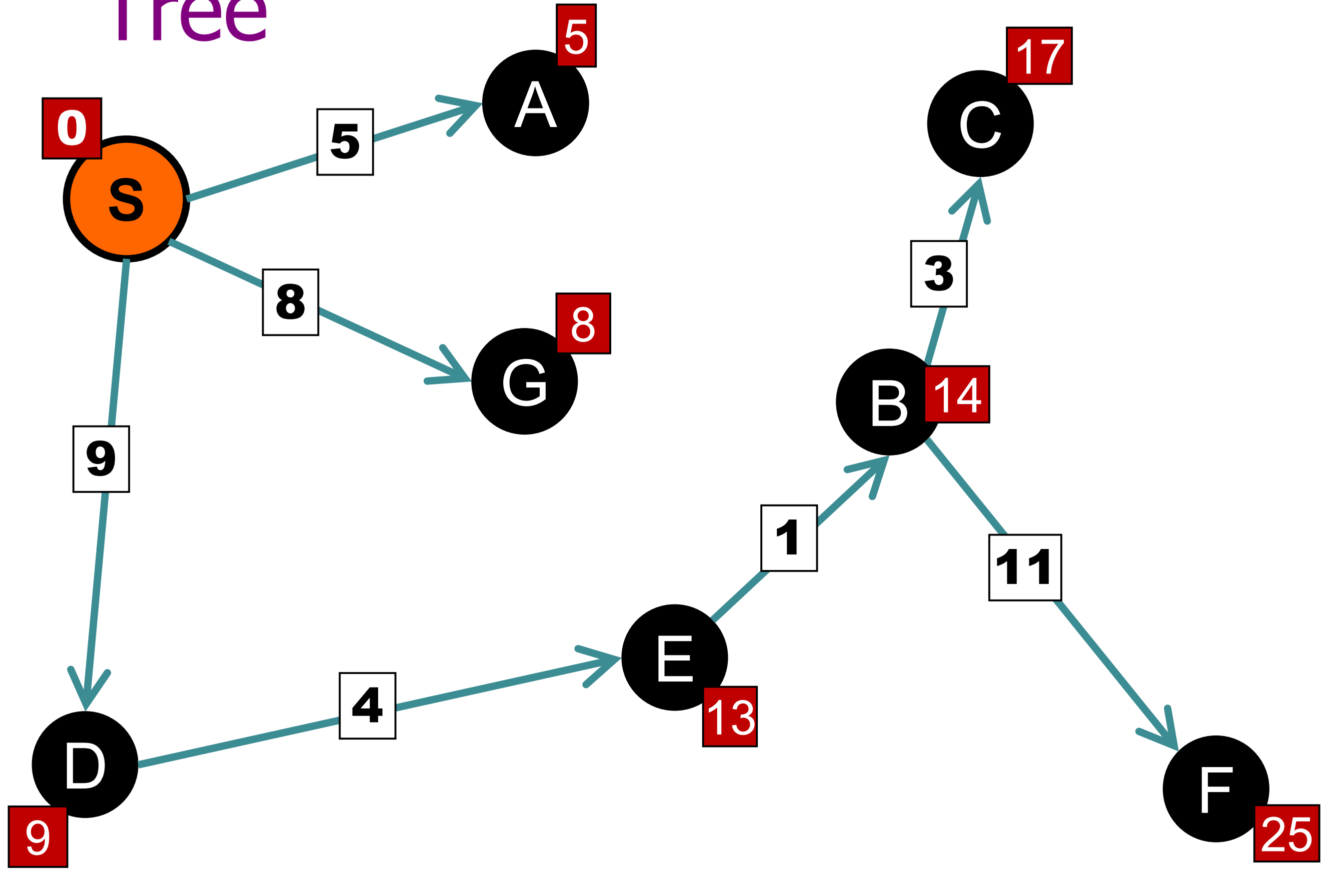
Dijkstra's Algorithm

Analysis:

- deleteMin: |V| times each
  - Each node is added to the priority queue **once**.
- relax / decreaseKey: |E| times
  - Each edge is relaxed once.
- Priority queue operations: O(log V)
- Total: O((V+E)log V) = O(E log V)

Dijkstra’s Algorithm

Following the parents: Yields the Shortest Path Tree



Dijkstra’s Algorithm

Why does it work?

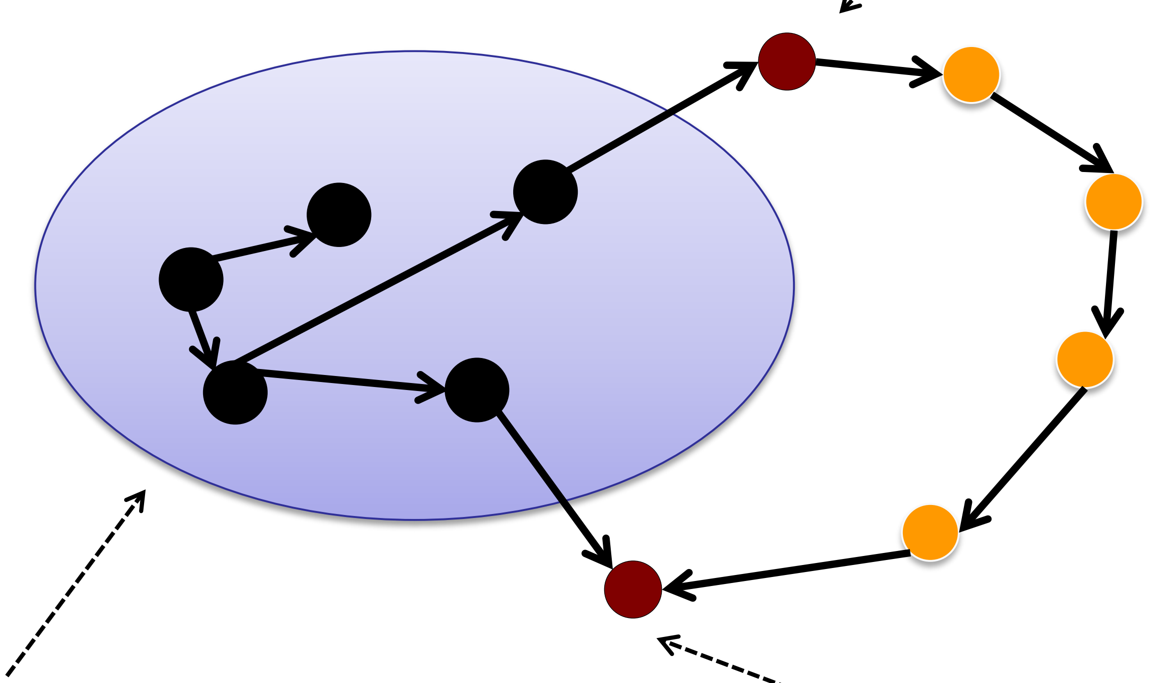
Dijkstra’s Algorithm

Proof by induction:

- Every “finished” (dequeued) vertex has a correct estimate.
  - Namely, shortest path is found for that vertex
- Initially: only “finished” vertex is start.

Dijkstra's Algorithm

Every edge crossing the boundary has been relaxed.



fringe vertices: neighbor of a finished vertex.

other vertices: no known estimate

finished vertices: distance is accurate.

fringe vertices: top in priority queue neighbor of a finished vertex.

Dijkstra’s Algorithm

Proof by induction:

- Every “finished” vertex has correct estimate.
- Initially: only “finished” vertex is start.

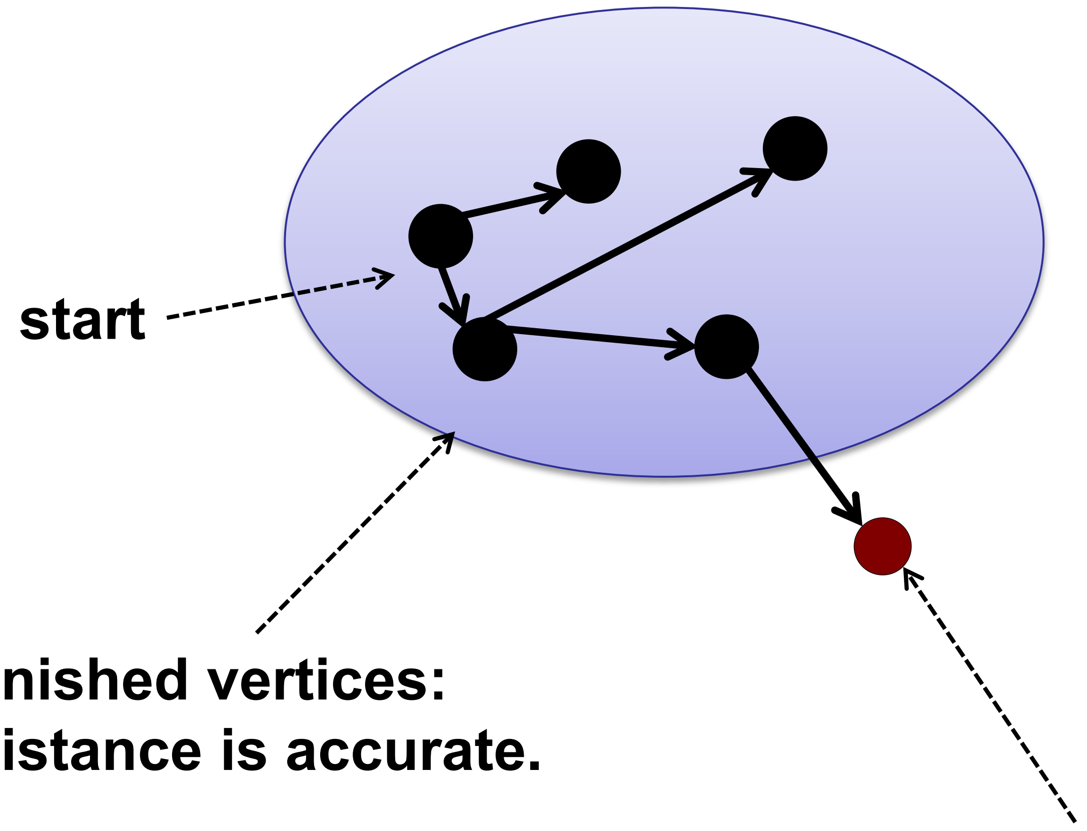
Dijkstra’s Algorithm

Proof by induction:

- Every “finished” vertex has correct estimate.
- Initially: only “finished” vertex is start.

- Inductive step:
  - Remove vertex from priority queue.
  - Relax its edges.
  - Add it to finished.
  - **Claim: it has a correct estimate.**

Dijkstra's Algorithm



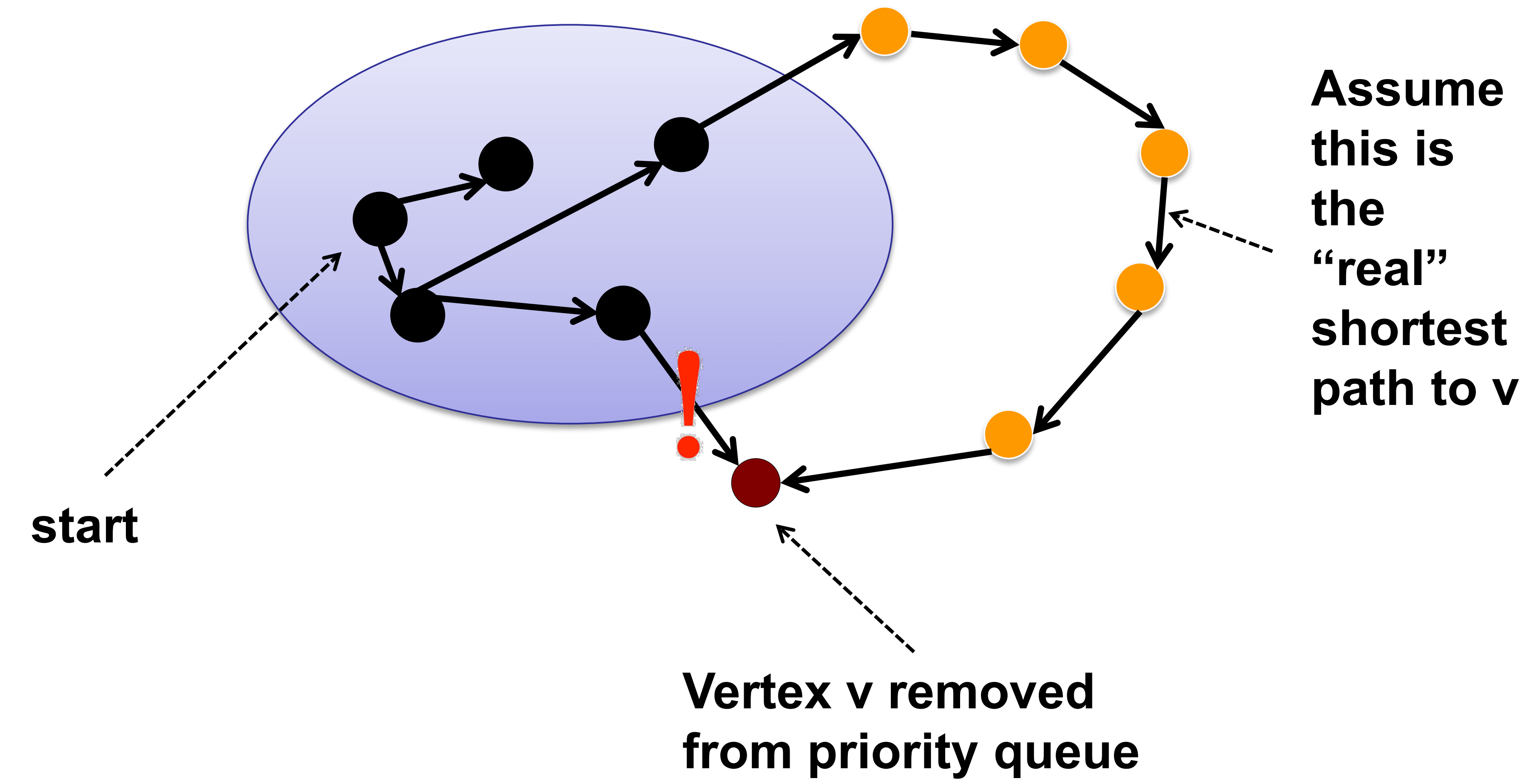
Let this distance of v be d_v

Vertex v going to be removed from priority queue next. Thus, with minimum distance amount the unfinished

Dijkstra’s Algorithm

trying to prove by contradiction

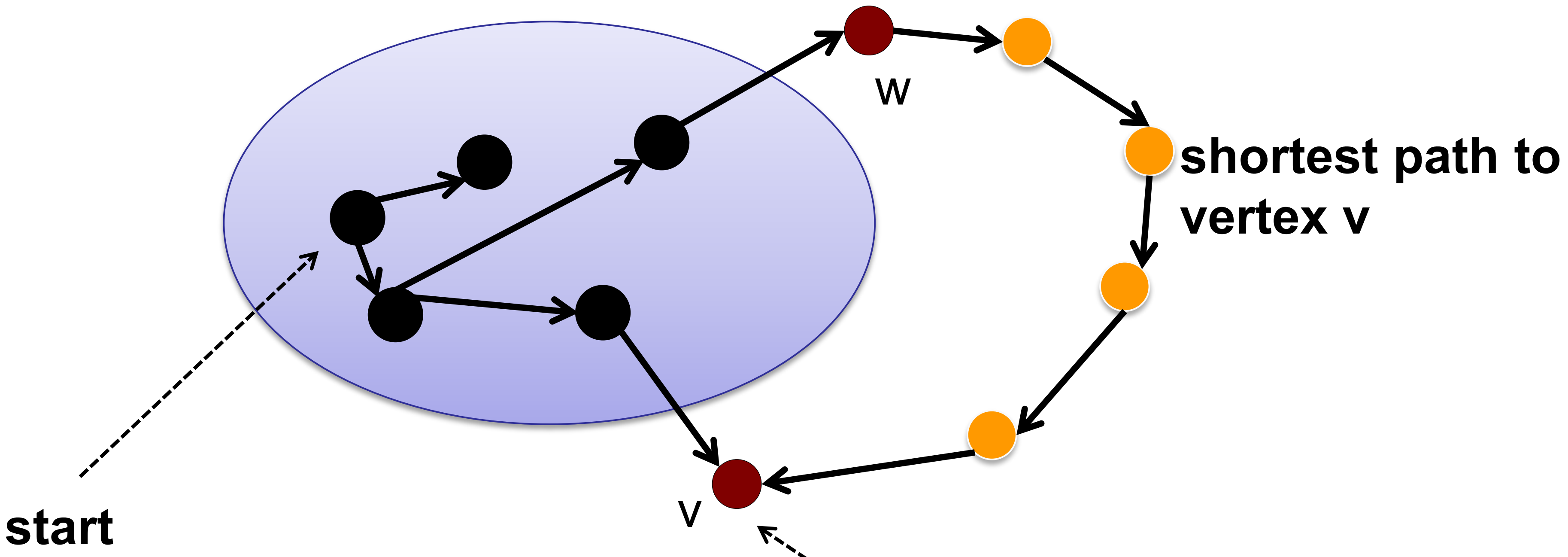
Assume NOT. The current estimate is not the shortest path. And the new path has dist. < d_v


Assume this is the “real” shortest path to v
start
Vertex v removed from priority queue
!

Dijkstra's Algorithm

There must be a vertex w in the current PQ on this "real" path.

And let this "real" path has distance $r_v < d_v$

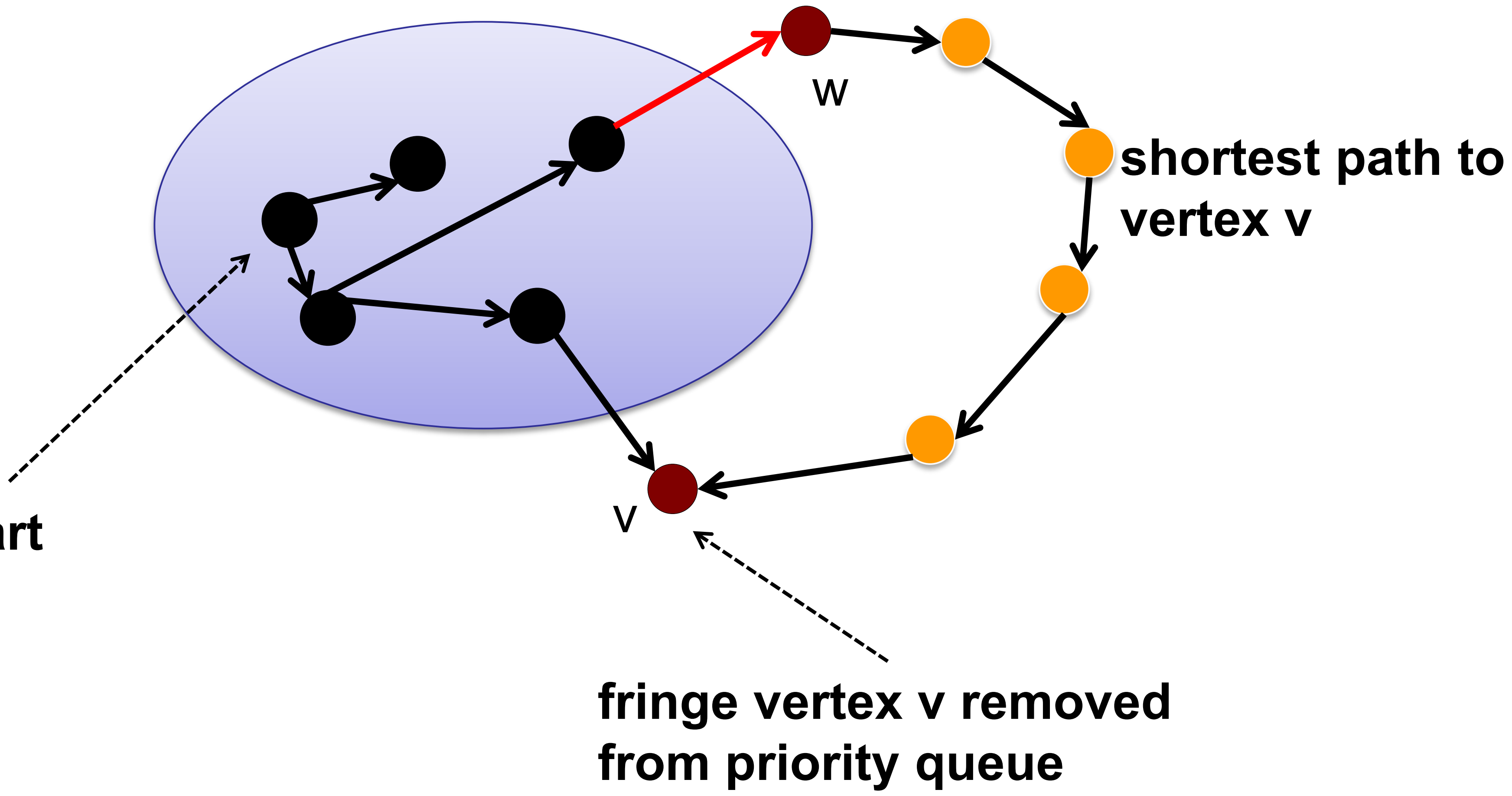


fringe vertex v removed from priority queue

Dijkstra's Algorithm

If P is shortest path to v, then prefix of P is shortest path to w.

Then distTo[w] is accurate. And **$\text{distTo}[w] < r_v < d_v = \text{distTo}[v]$**

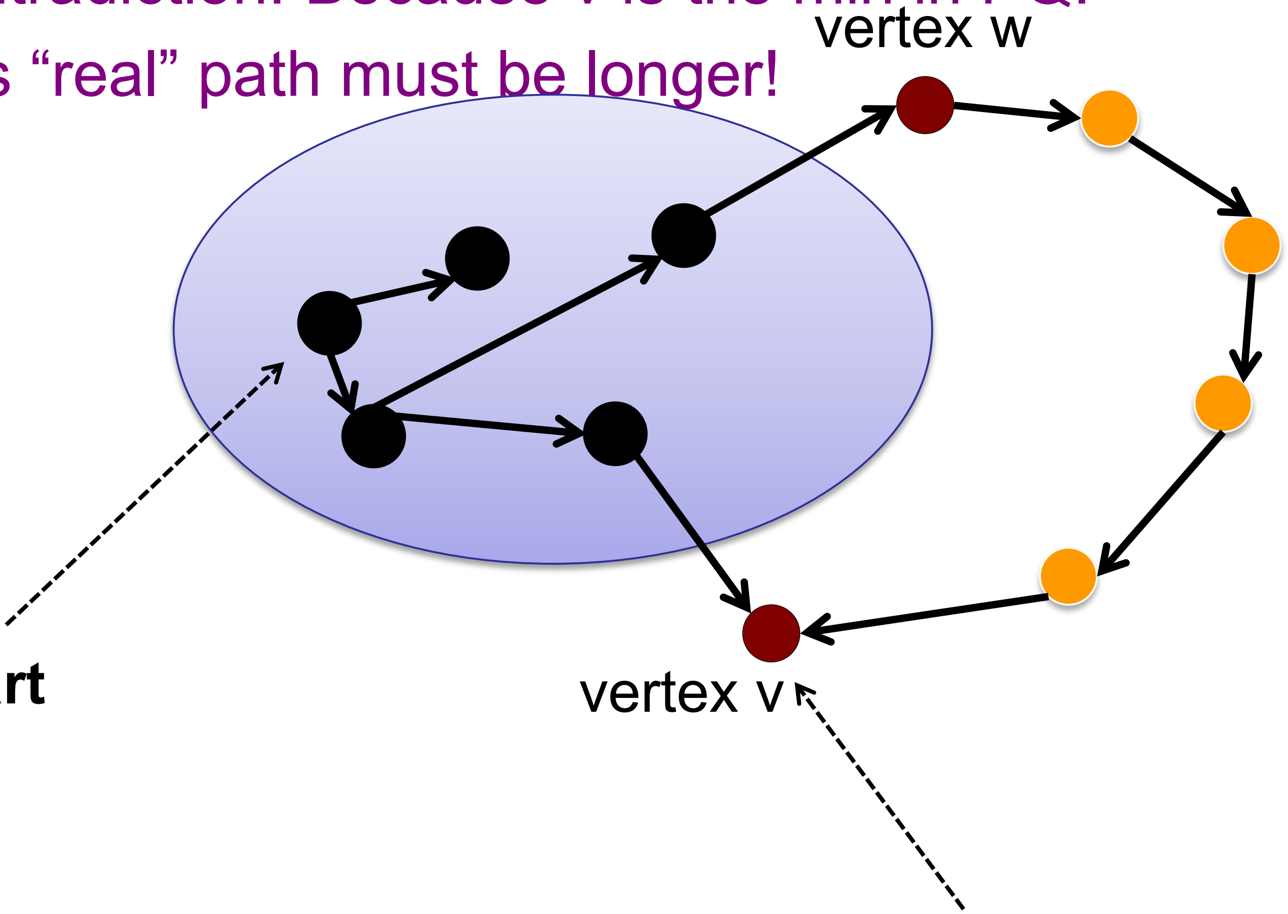

w
shortest path to vertex v
start
v
fringe vertex v removed from priority queue

Dijkstra's Algorithm

But `distTo[w] >= distTo[v]` according to PQ!

Contradiction! Because v is the min in PQ!

This "real" path must be longer!



Vertex v going to be removed from priority queue next.
Thus, with minimum distance amount the unfinished

Dijkstra’s Algorithm

Proof by induction:

- Every “finished” vertex has correct estimate.
- Initially: only “finished” vertex is start.

- Inductive step:
  - Remove vertex from priority queue.
  - Relax its edges.
  - Add it to finished.
  - **Claim: it has a correct estimate.**

Dijkstra’s Algorithm

```
relax(Edge e) {
    int v = e.from();
    int w = e.to();
    double weight = e.weight();
    if (distTo[w] > distTo[v] + weight) {
        distTo[w] = distTo[v] + weight;
        parent[w] = v;
        pq.decreaseKey(w, distTo[w]);
    }
}
```

Dijkstra's Algorithm

Analysis:

- insert / deleteMin: |V| times each
  - Each node is added to the priority queue **once**.
- decreaseKey: |E| times
  - Each edge is relaxed once.
- Priority queue operations: O(log V)
- Total: O((V+E)log V) = O(E log V)

Dijkstra's Algorithm

Source-to-Destination:

- What if you stop the first time you dequeue the destination?

- Recall:
  - a vertex is "finished" when it is dequeued
  - if the destination is finished, then stop

Dijkstra Summary

Basic idea:

- Maintain distance estimates.
- Repeat:
  - Find unfinished vertex with smallest estimate.
  - Relax all outgoing edges.
  - Mark vertex finished.

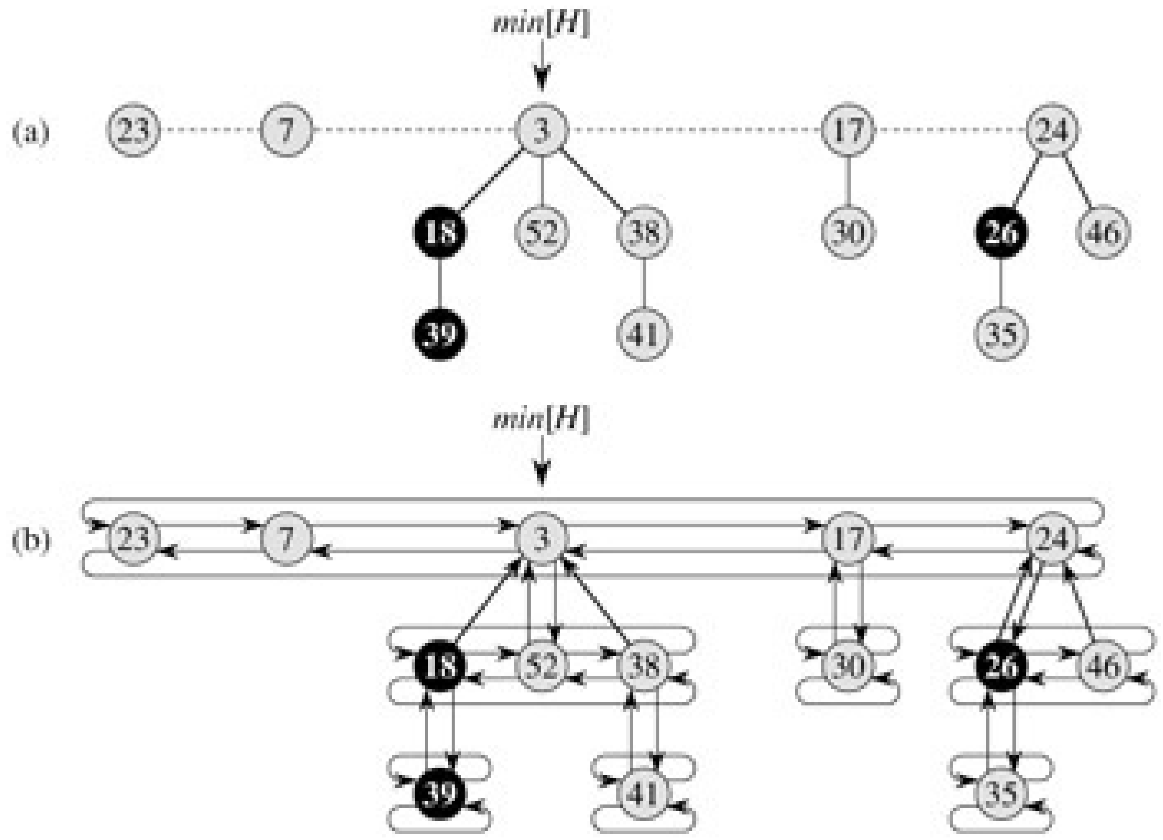
- O(E log V) time

Dijkstra's Performance

PQ Implementation	insert	deleteMin	decreaseKey	Total
Array	1	V	1	**$O(V^2)$**
AVL Tree	log V	log V	log V	**O(E log V)**
d-way Heap	$d\log_d V$	$d\log_d V$	$\log_d V$	**$O(E\log_{E/V} V)$**
Fibonacci Heap	1	log V	1	**O(E + V log V)**

Fibonacci Heap

- Not in this course

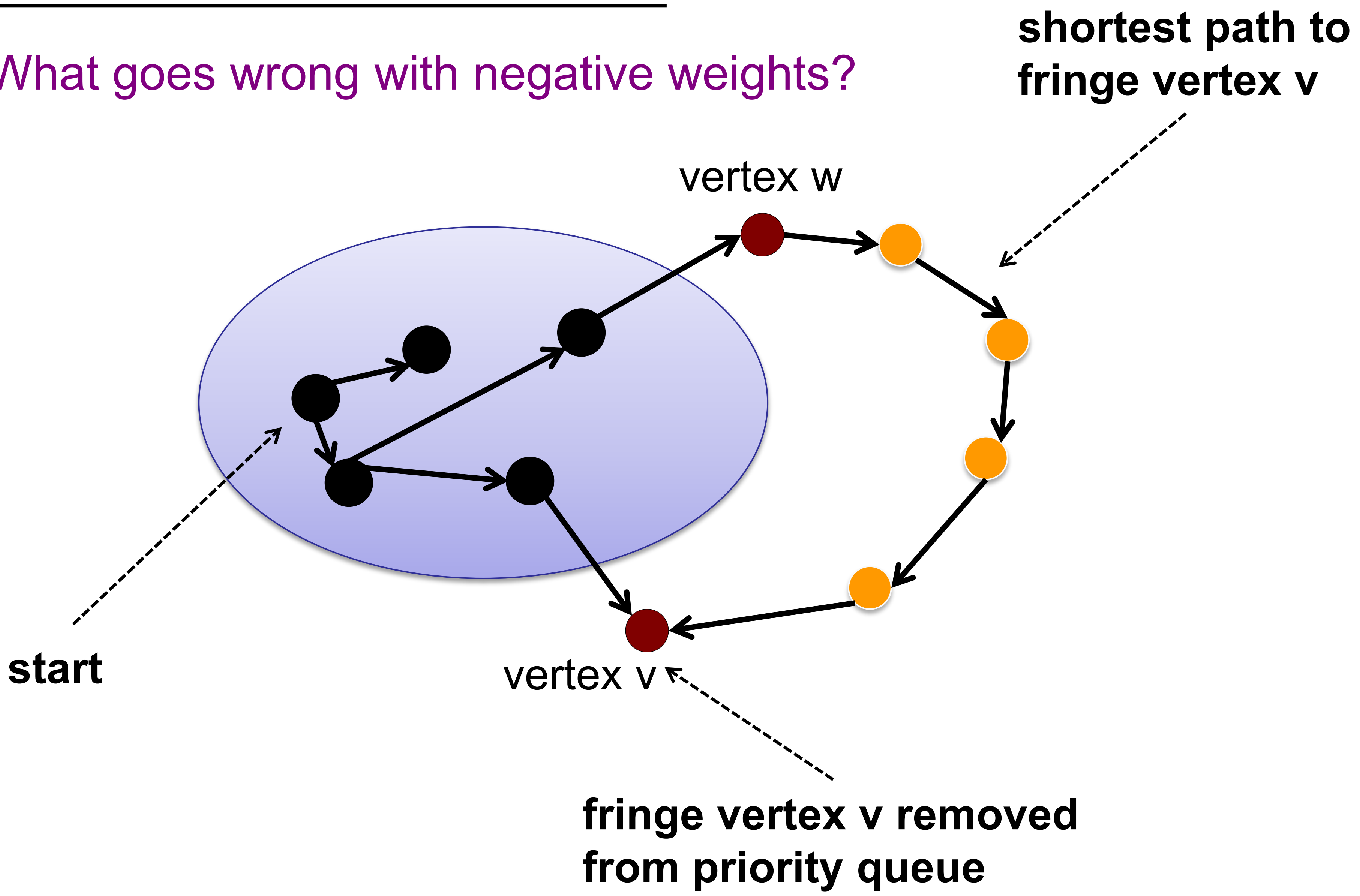


Dijkstra Summary

Edges with negative weights?

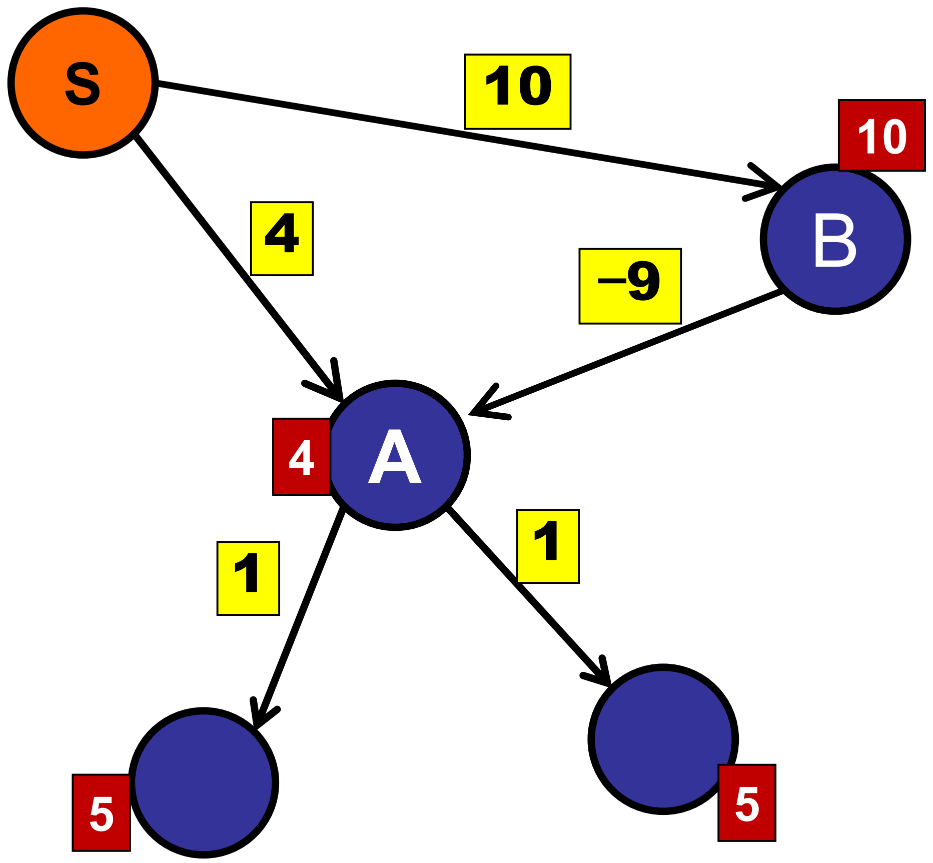
Dijkstra’s Algorithm

What goes wrong with negative weights?


shortest path to fringe vertex v
vertex w
start
vertex v
fringe vertex v removed from priority queue

Dijkstra's Algorithm

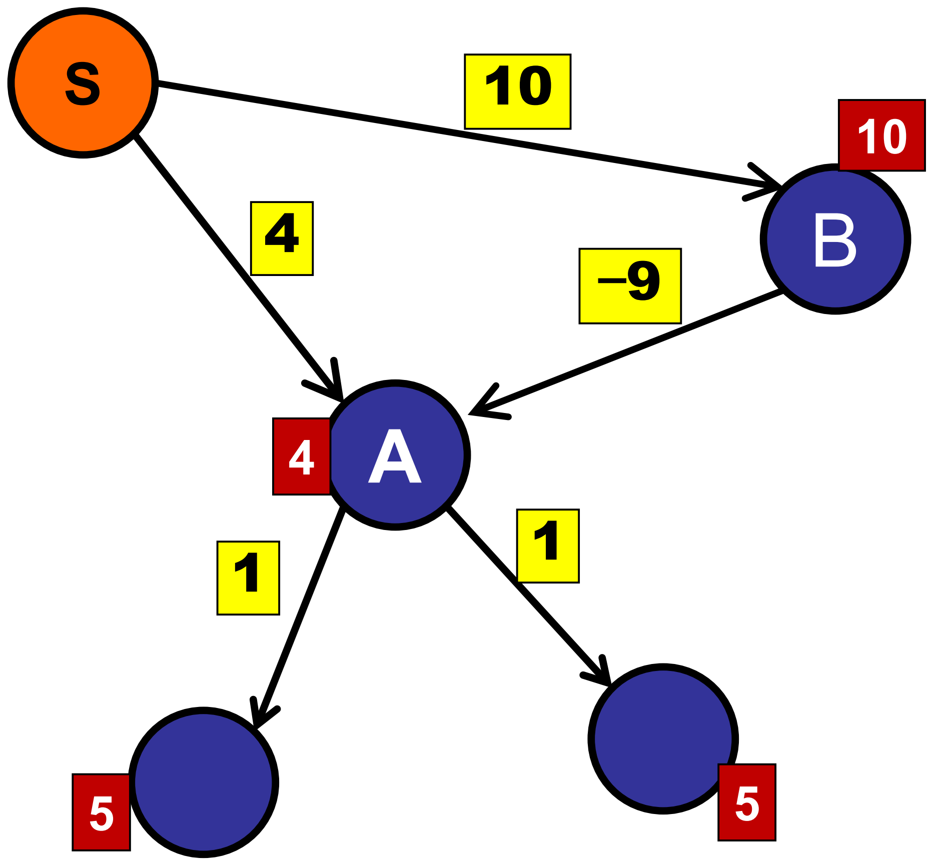
Edges with negative weights?



Step 1: Remove A.
Relax A.
Mark A done.

Dijkstra’s Algorithm

Edges with negative weights?



Step 1: Remove A.
Relax A.
Mark A done.

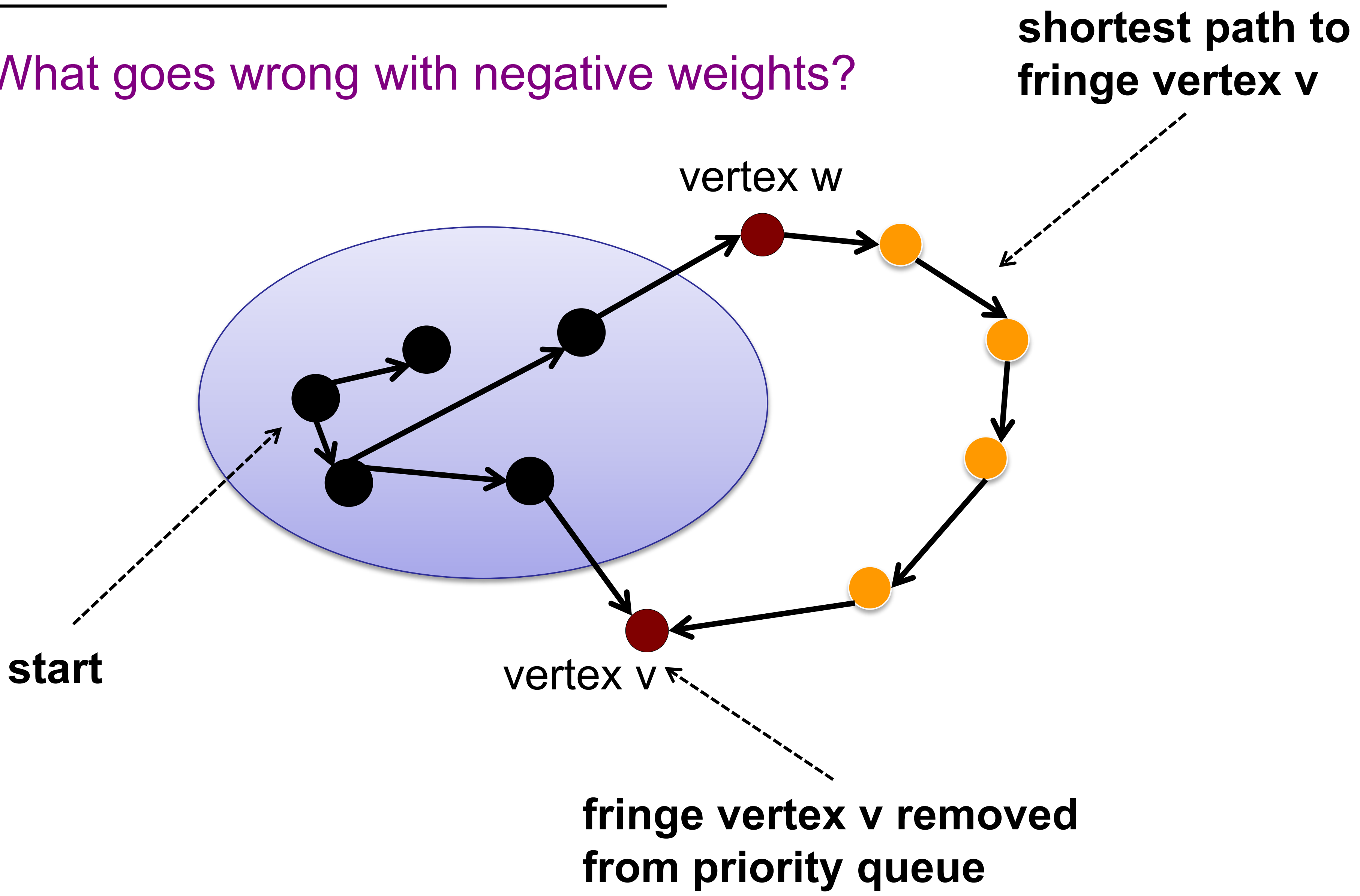
…

Step 4: Remove B.
Relax B.
Mark B done.

Oops: We need to update A.

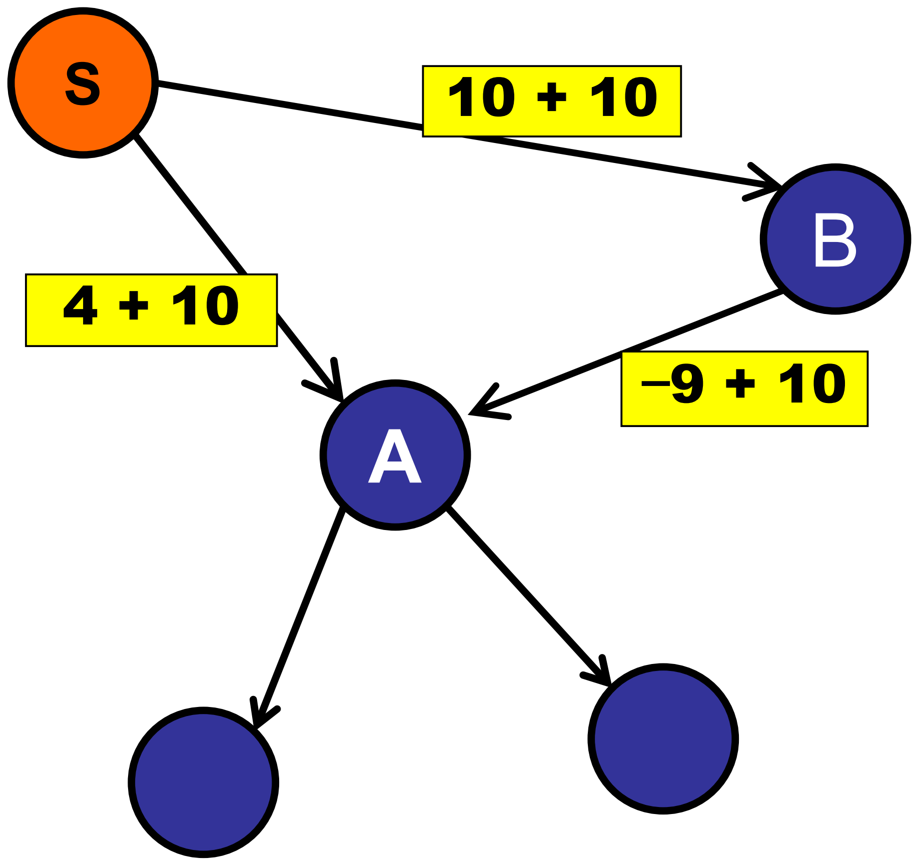
Dijkstra’s Algorithm

What goes wrong with negative weights?


shortest path to fringe vertex v
vertex w
start
vertex v
fringe vertex v removed from priority queue

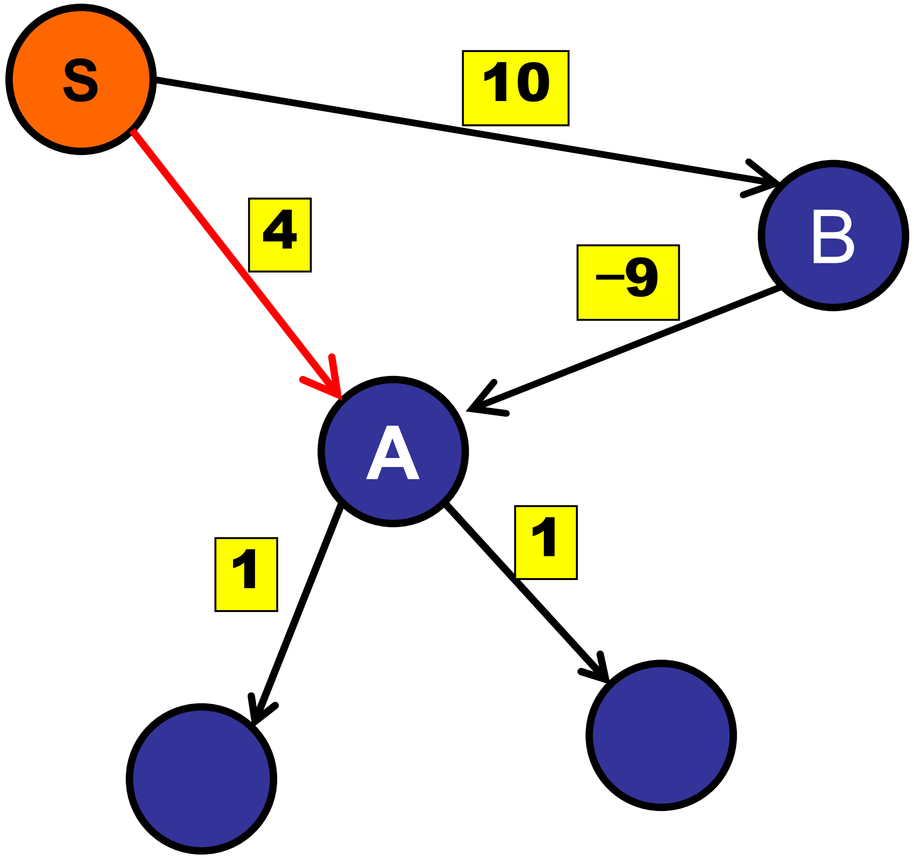
Dijkstra’s Algorithm

Can we reweight? e.g.: weight += 10



Dijkstra’s Algorithm

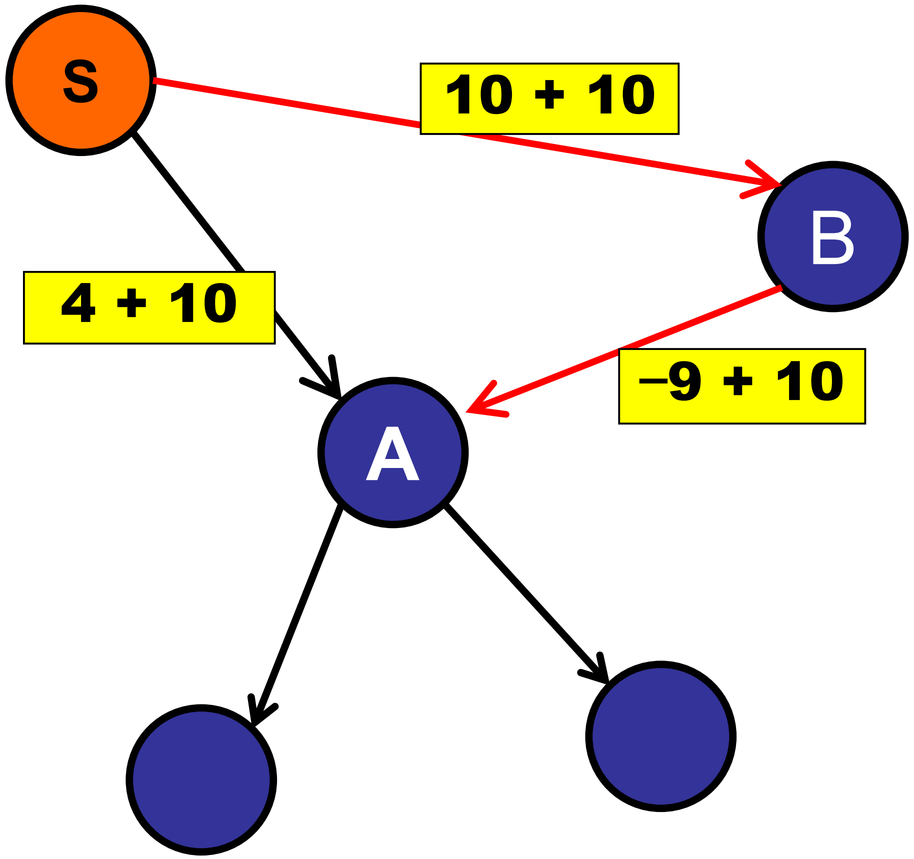
Can we reweight?



Path S-B-A:	1
Path S-A:	4

Dijkstra's Algorithm

Can we reweight?



Path S-B-A:	21
Path S-A:	14

Dijkstra Summary

Basic idea:

- Maintain distance estimates.
- Repeat:
    - Find unfinished vertex with smallest estimate.
    - Relax all outgoing edges.
    - Mark vertex finished.
- O(E log V) time (with AVL tree Priority Queue).
- No negative weight edges!

Dijkstra Comparison

Same algorithm:

- Maintain a set of explored vertices.
- Add vertices to the explored set by following edges that go from a vertex in the explored set to a vertex outside the explored set.

  – BFS: Take edge from vertex that was discovered **least** recently.

  – DFS: Take edge from vertex that was discovered **most** recently.

  – Dijkstra's:Take edge from vertex that is **closest** to source.

Dijkstra Comparison

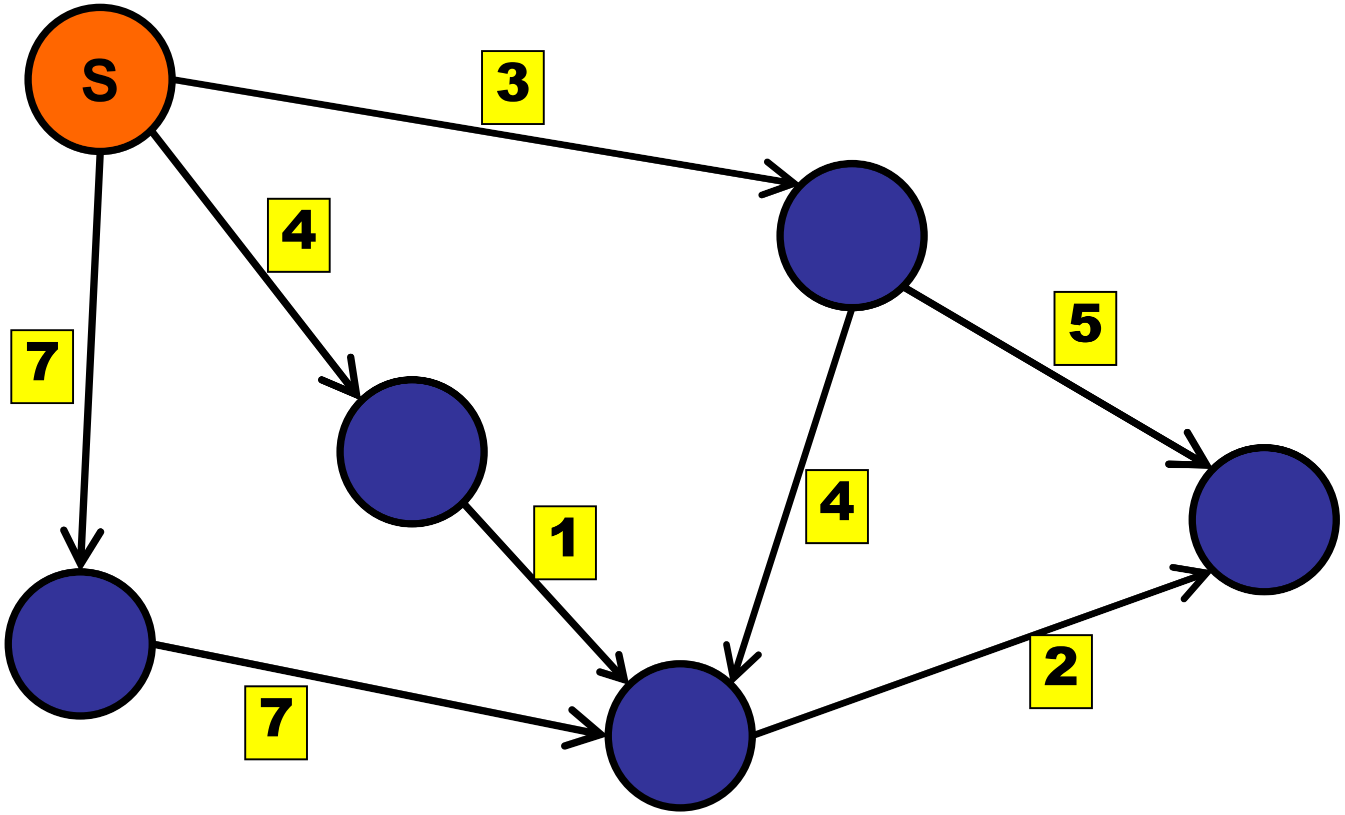
Same algorithm:

- Maintain a set of explored vertices.
- Add vertices to the explored set by following edges that go from a vertex in the explored set to a vertex outside the explored set.

  - BFS: Use queue.
  - DFS: Use stack.
  - Dijkstra's: Use priority queue.

Longest Paths

Any ideas?

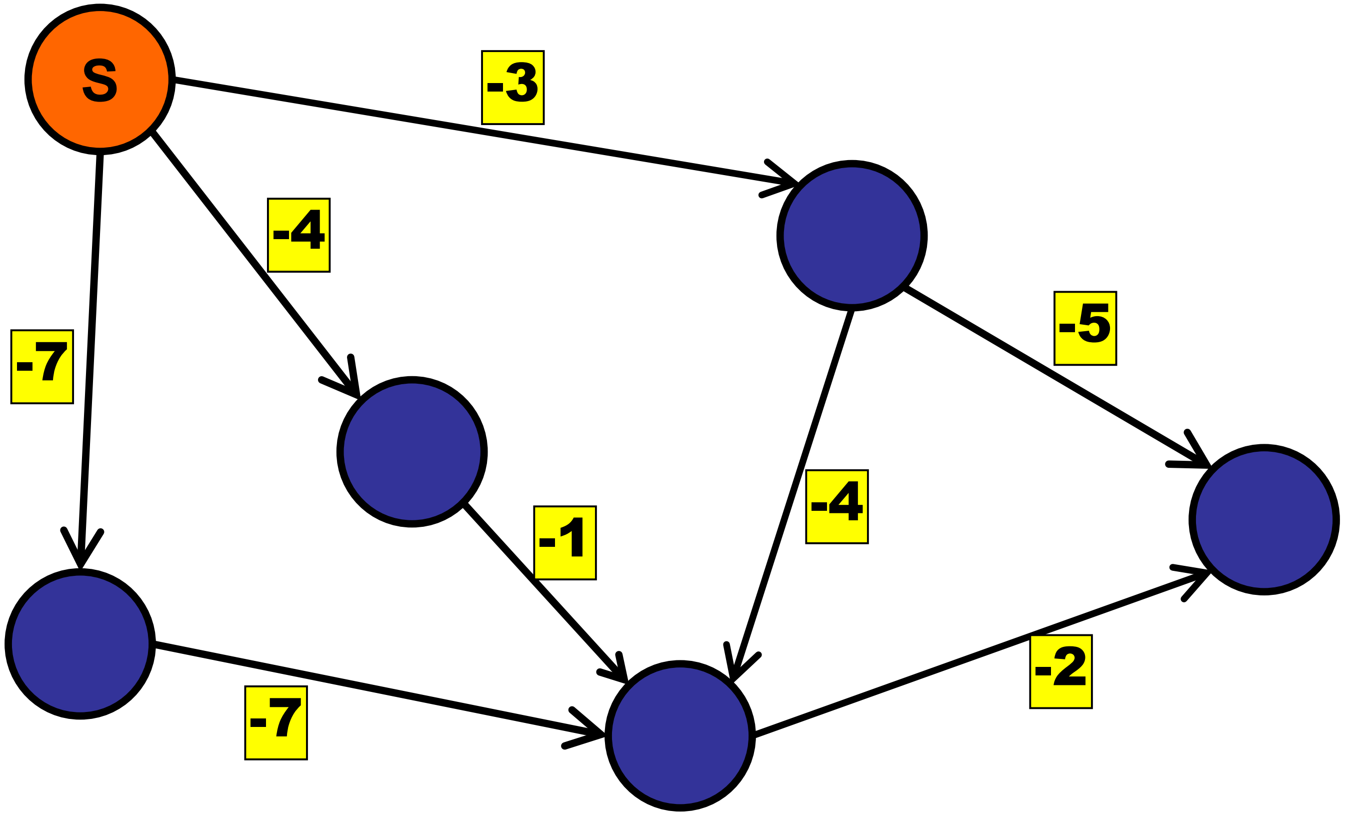


Longest Paths

Using BF not Dijkstras

Negate the edges?

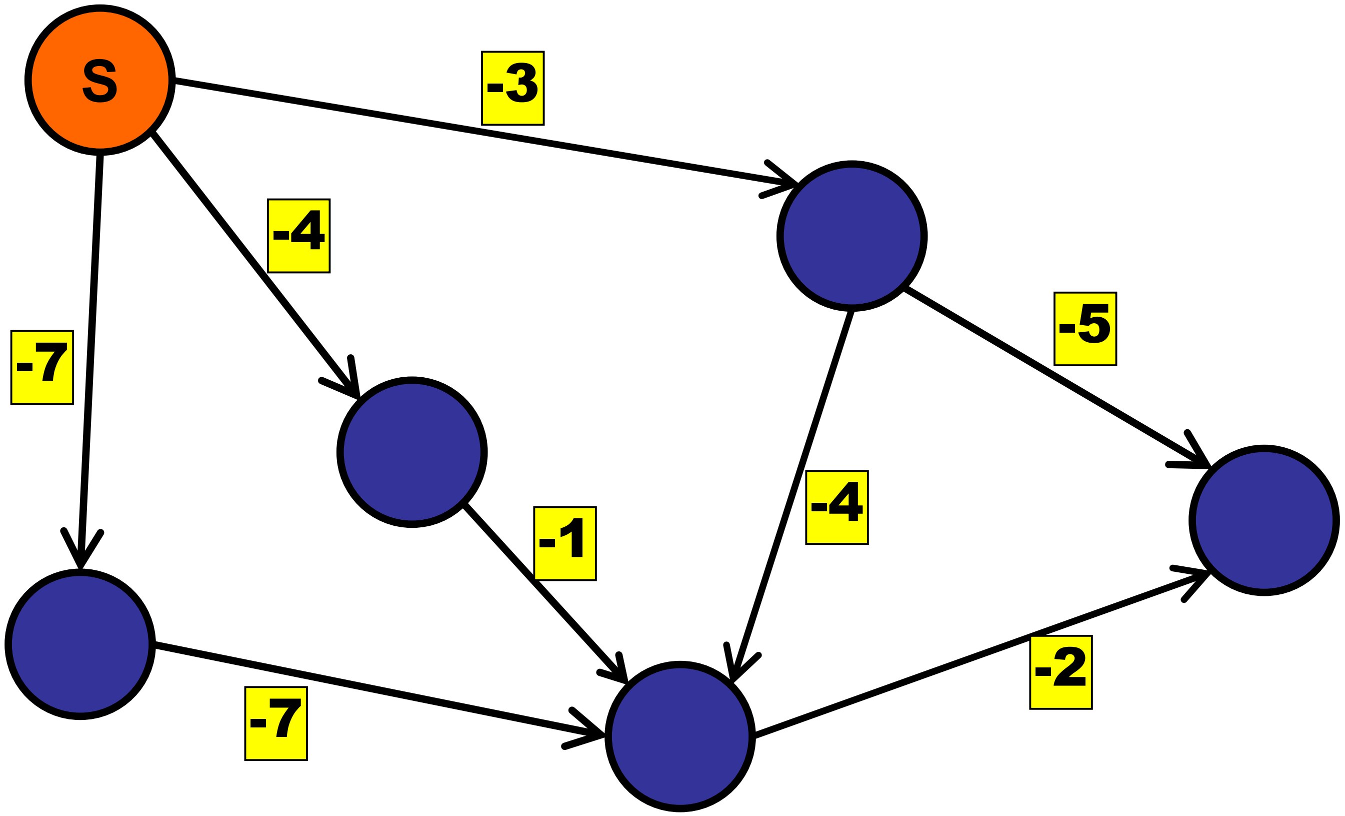
Only if your graph does not have a cycle!



Longest Paths

Acyclic Graph:

shortest path in negated=longest path in regular



Longest Path

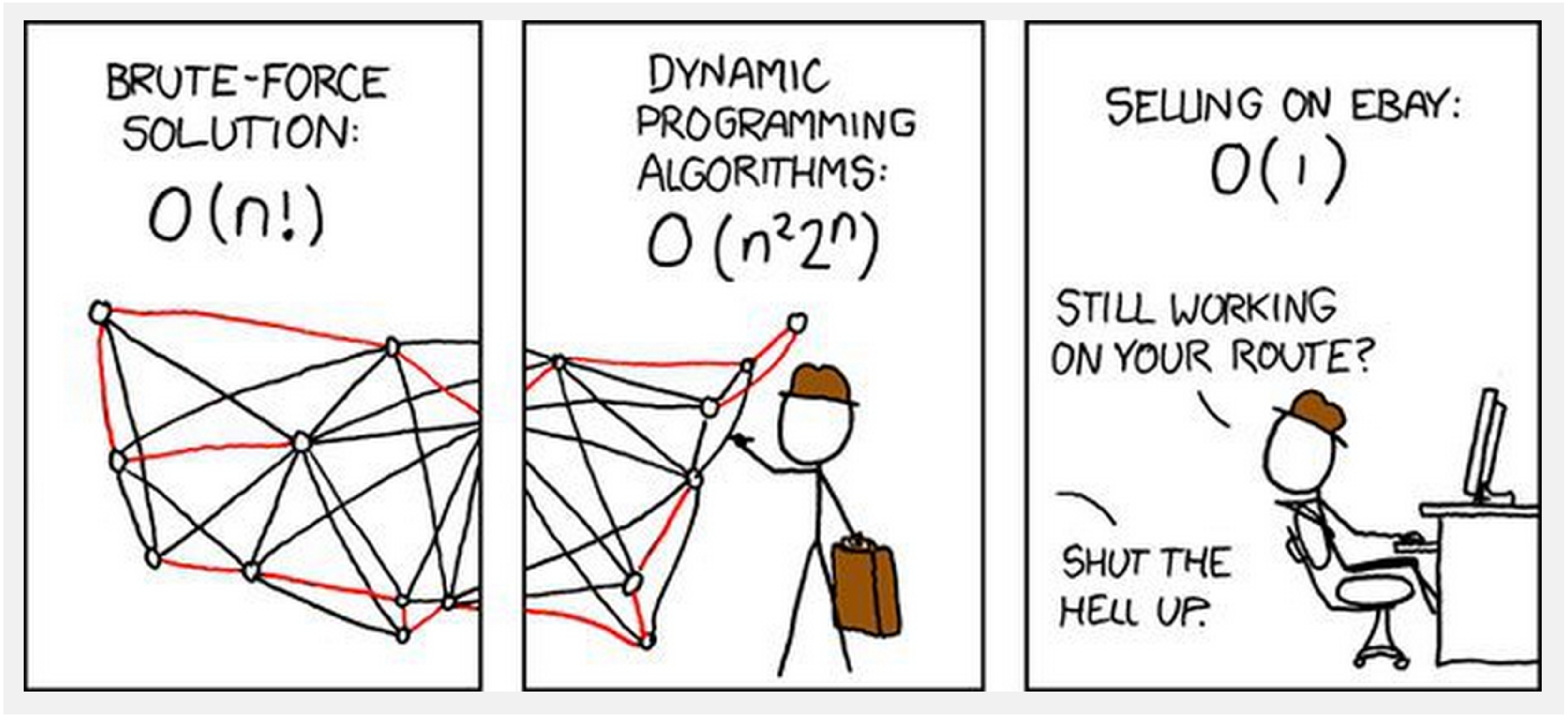
Directed Acyclic Graph:

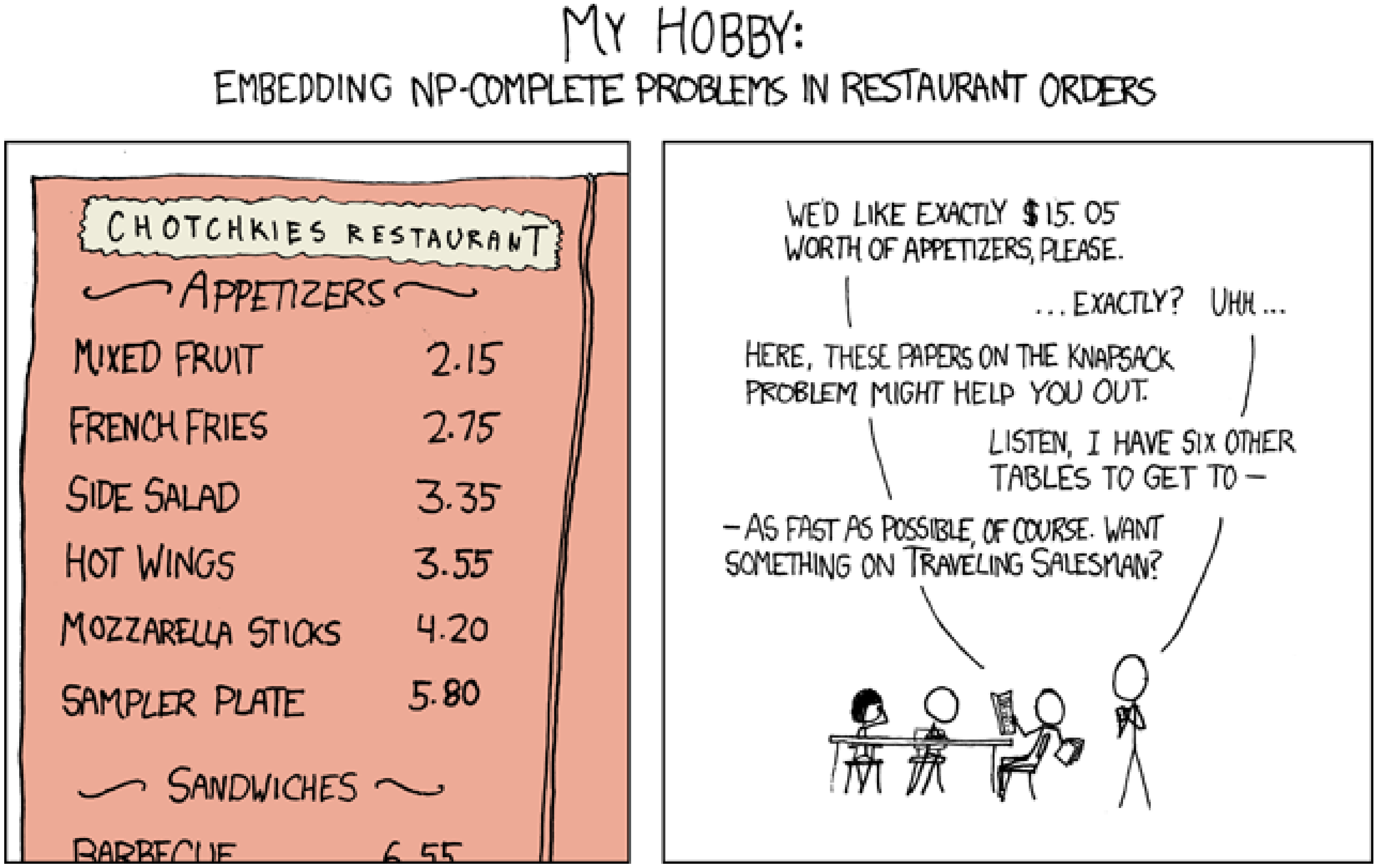
- Solvable efficiently using topological sort

General (cyclic) Graphs:

- NP-Hard
- Reduction from Hamiltonian Path:
  - If you could find the longest simple path, then you could decide if there is a path that visits every vertex.
  - Any polynomial time algorithm for longest path thus implies a polynomial time algorithm for HAMPATH.

Also called the Travelling Salemans Problem




MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS
CHOTCHKIES RESTAURANT
APPETIZERS
MIXED FRUIT 2.15
FRENCH FRIES 2.75
SIDE SALAD 3.35
HOT WINGS 3.55
MOZZARELLA STICKS 4.20
SAMPLER PLATE 5.80
SANDWICHES
WE'D LIKE EXACTLY $15.05 WORTH OF APPETIZERS, PLEASE.
...EXACTLY? UHH...
HERE, THESE PAPERS ON THE KNAPSACK PROBLEM MIGHT HELP YOU OUT.
LISTEN, I HAVE SIX OTHER TABLES TO GET TO —
— AS FAST AS POSSIBLE, OF COURSE. WANT SOMETHING ON TRAVELING SALESMAN?

Roadmap

Part I: Shortest Paths

Part II: Applications of Shortest Paths

- DNA Alignment
- Constraint Systems

Example: DNA Alignment

Input: two DNA strings:

- AGGAACCGTA
- AGAATCCGAA

How similar are they?

- Metric: edit distance

  How many operations to transform one DNA string into another?

Example: DNA Alignment

Input: two DNA strings:

- AGGAACCGTA $\longleftarrow$ delete G, delete T
- AGAATCCGA $\longleftarrow$ add T

Three operations:

- Delete a character
- Add a character
- Transform a character

Example: DNA Alignment

Input: two DNA strings:

- AGGAACCGTA $\leftarrow$ delete G, delete T
- AGAATCCGA $\leftarrow$ add T

Three operations:

– Delete a character	cost = d
– Add a character	cost = a
– Transform a character	cost = t

OR: minimum *cost* to transform A to B?

Example: DNA Alignment

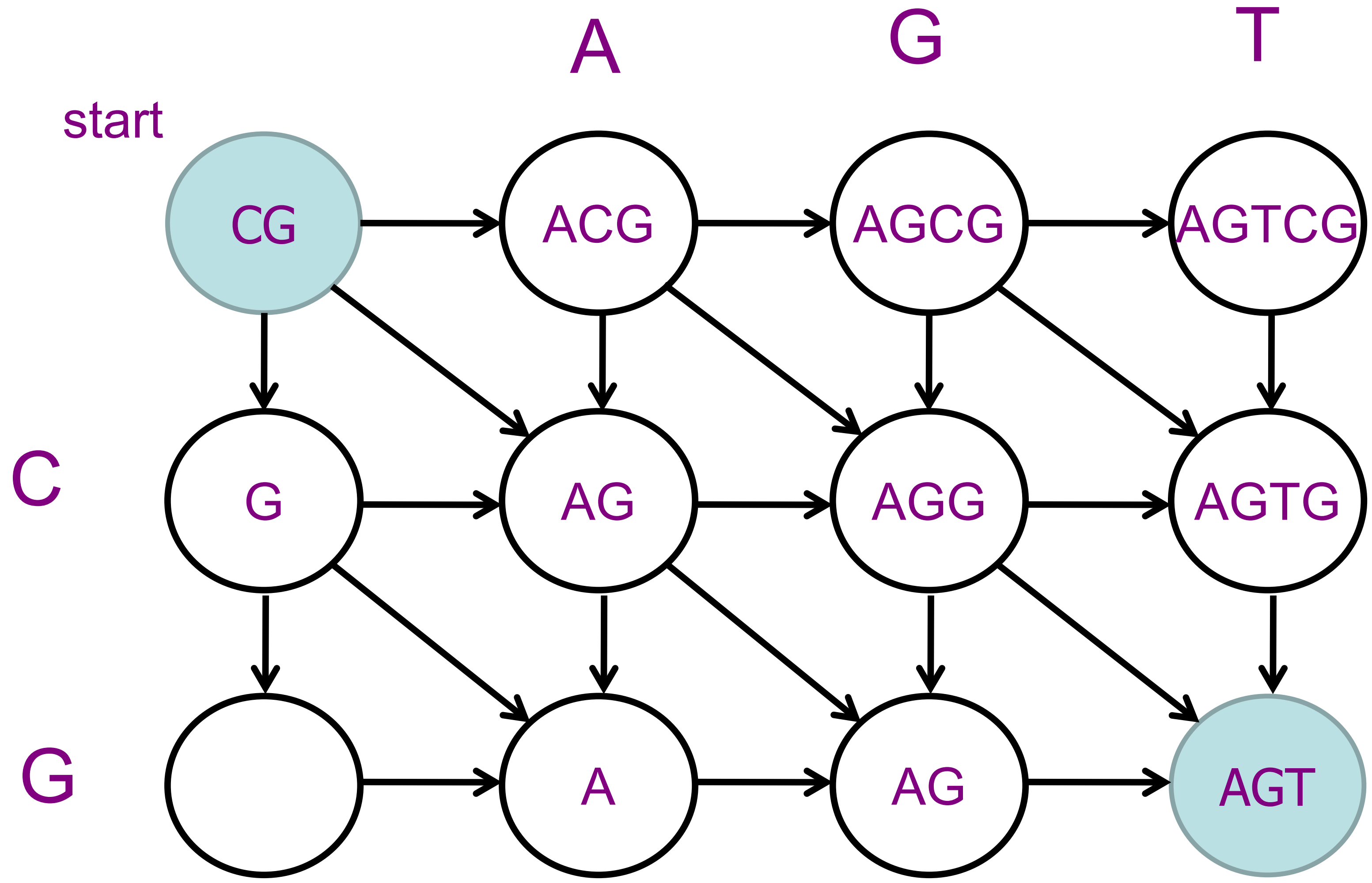
Model question as a directed graph:

- For each character i, character j:
  - Create a node in the graph N(i,j)
  - N(i,j) represents adapting position i of the old string to match position j of the new string.

- For node N(i,j), three outgoing edges:
  - insert character j+1 from new string after position i
  - delete character i+1 from old string
  - transform character i+1 to character j+1

Example: DNA Alignment

Transform: CG to AGT



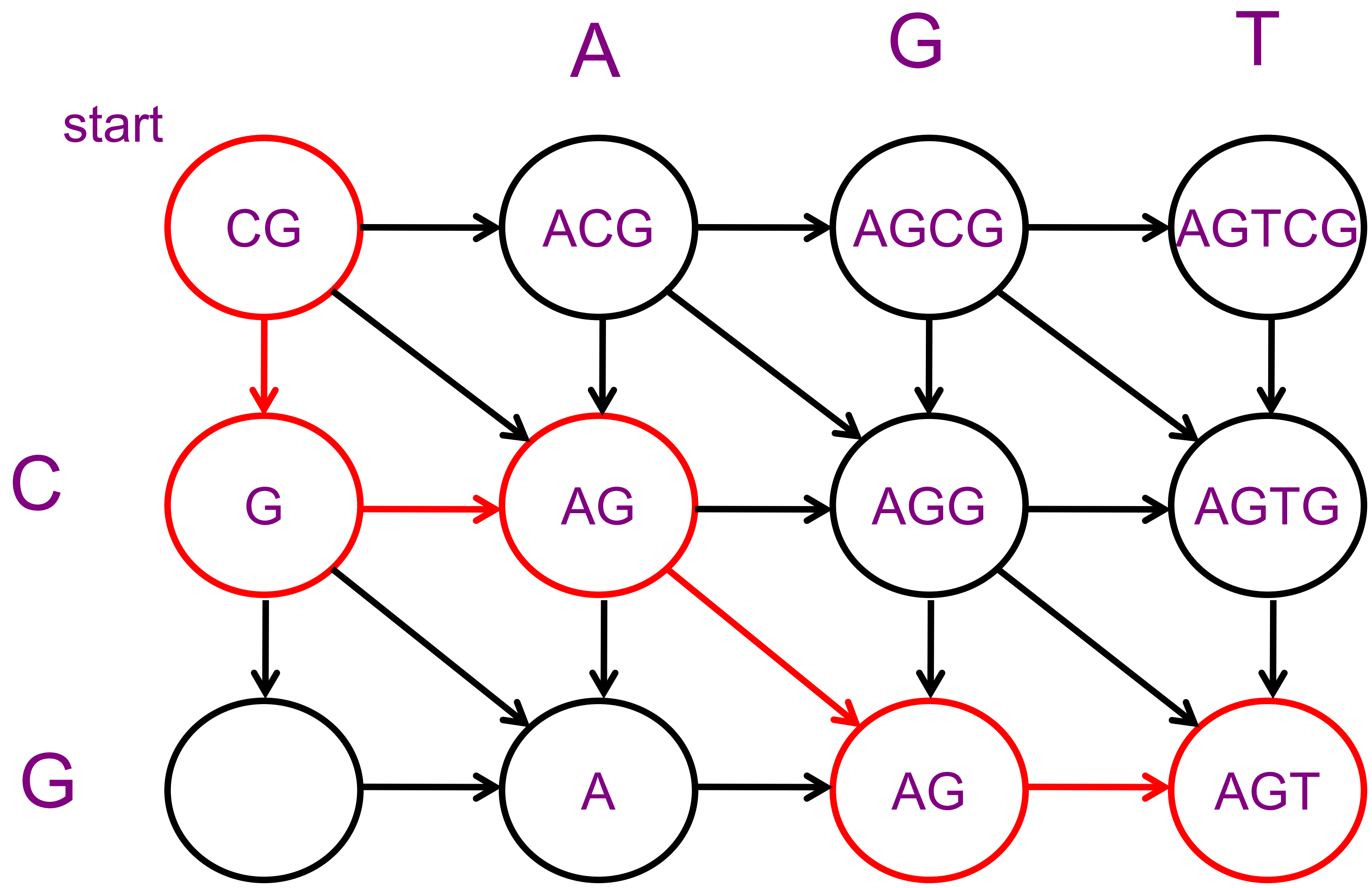
Vertical: delete character

Horizontal: add character

Diagonal: transform character

CG to AGT

Delete C, Add A, Leave G, Add T:



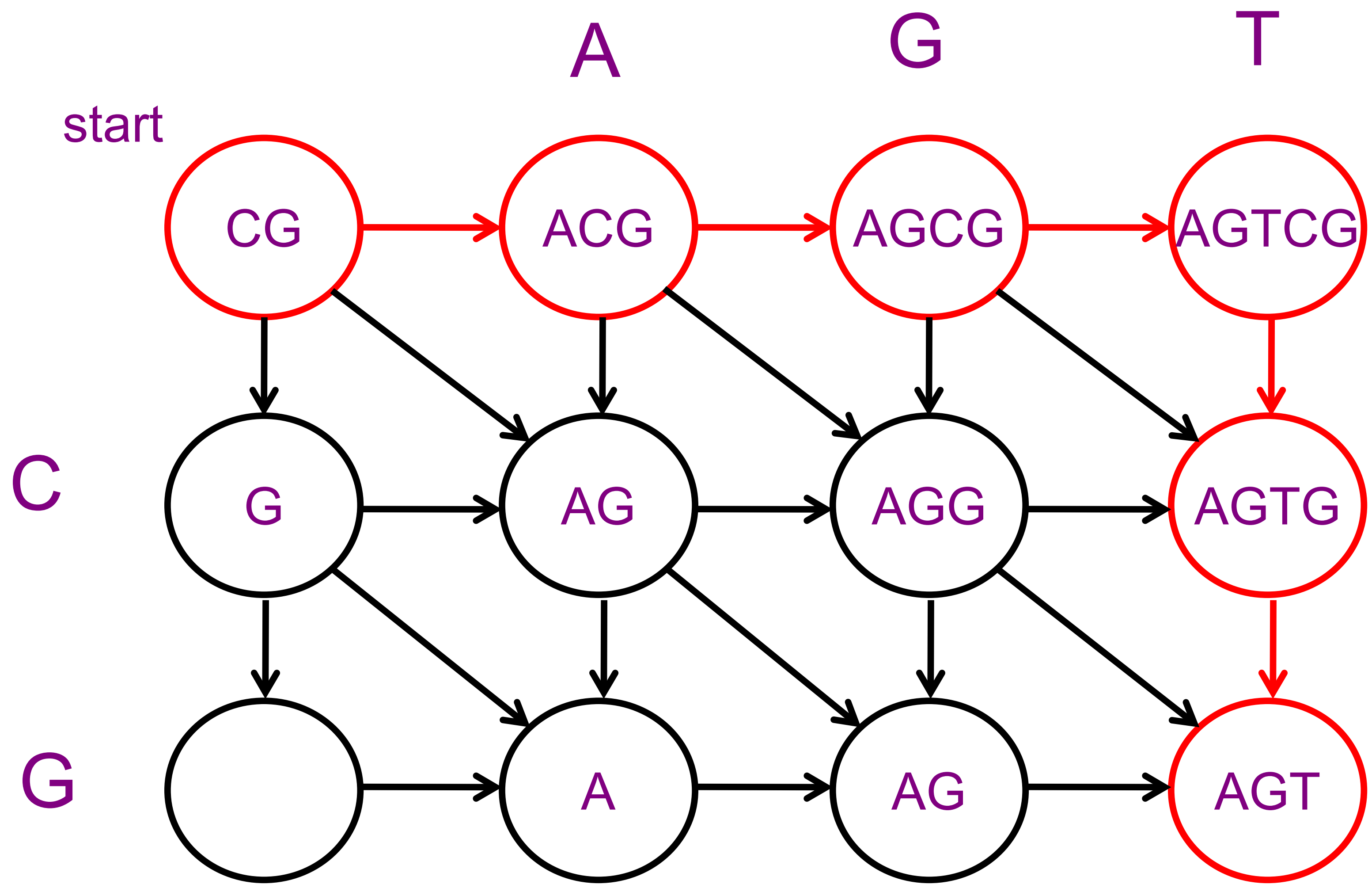
Vertical: delete character

Horizontal: add character

Diagonal: transform character

CG to AGT

Add A, Add G, Add T, Delete C, Delete G:



Vertical: delete character

Horizontal: add character

Diagonal: transform character

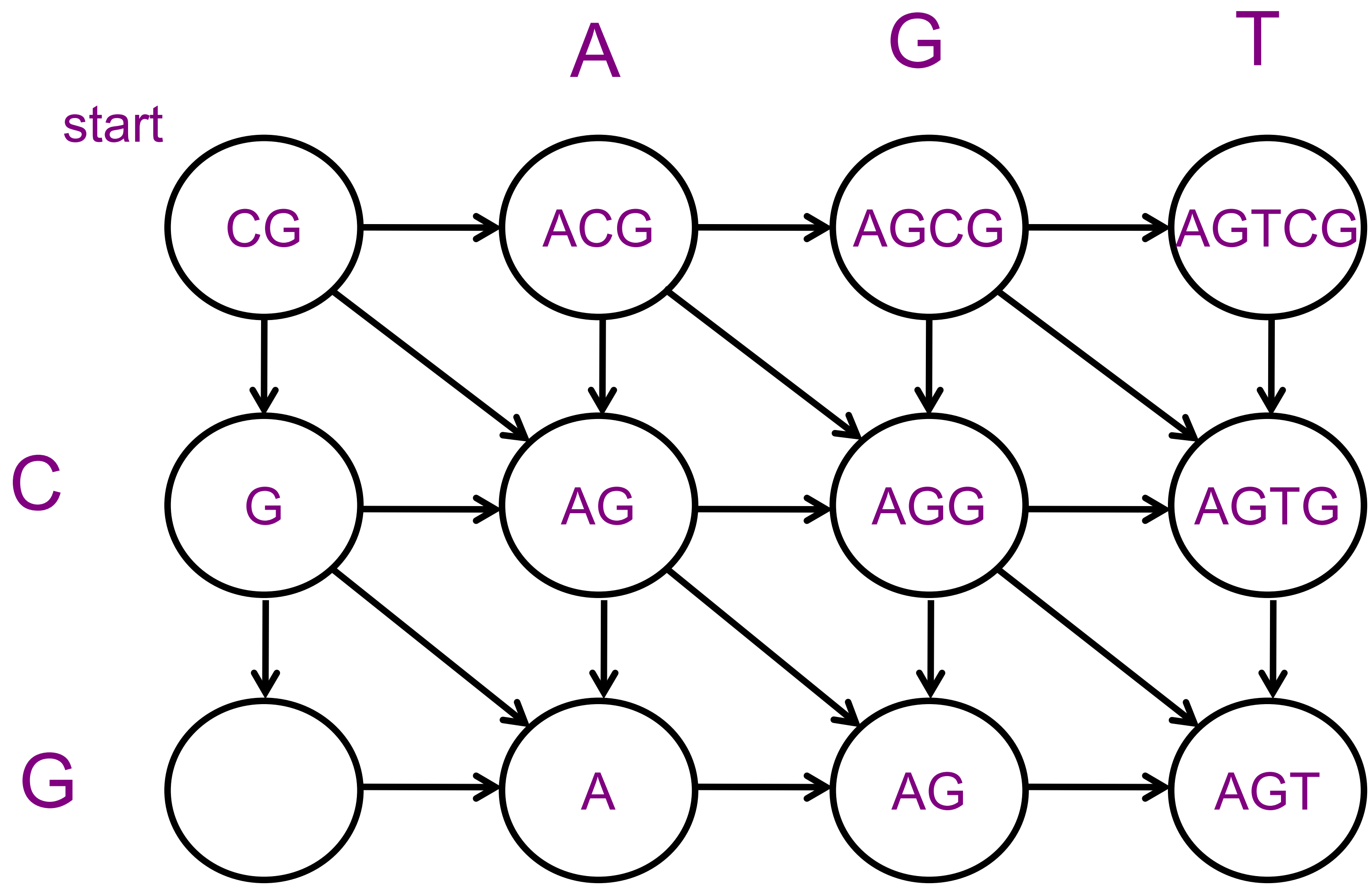
Example: DNA Alignment

Model question as a directed graph:

- For node N(i,j):
    - The first i letters of the old string have been replaced with the first j letters of the new string.
    - The shortest path to N(i,j) is the shortest set of changes to change the first i letters of the old string to the first j letters of the new string.

Example: DNA Alignment

Transform: CG to AGT



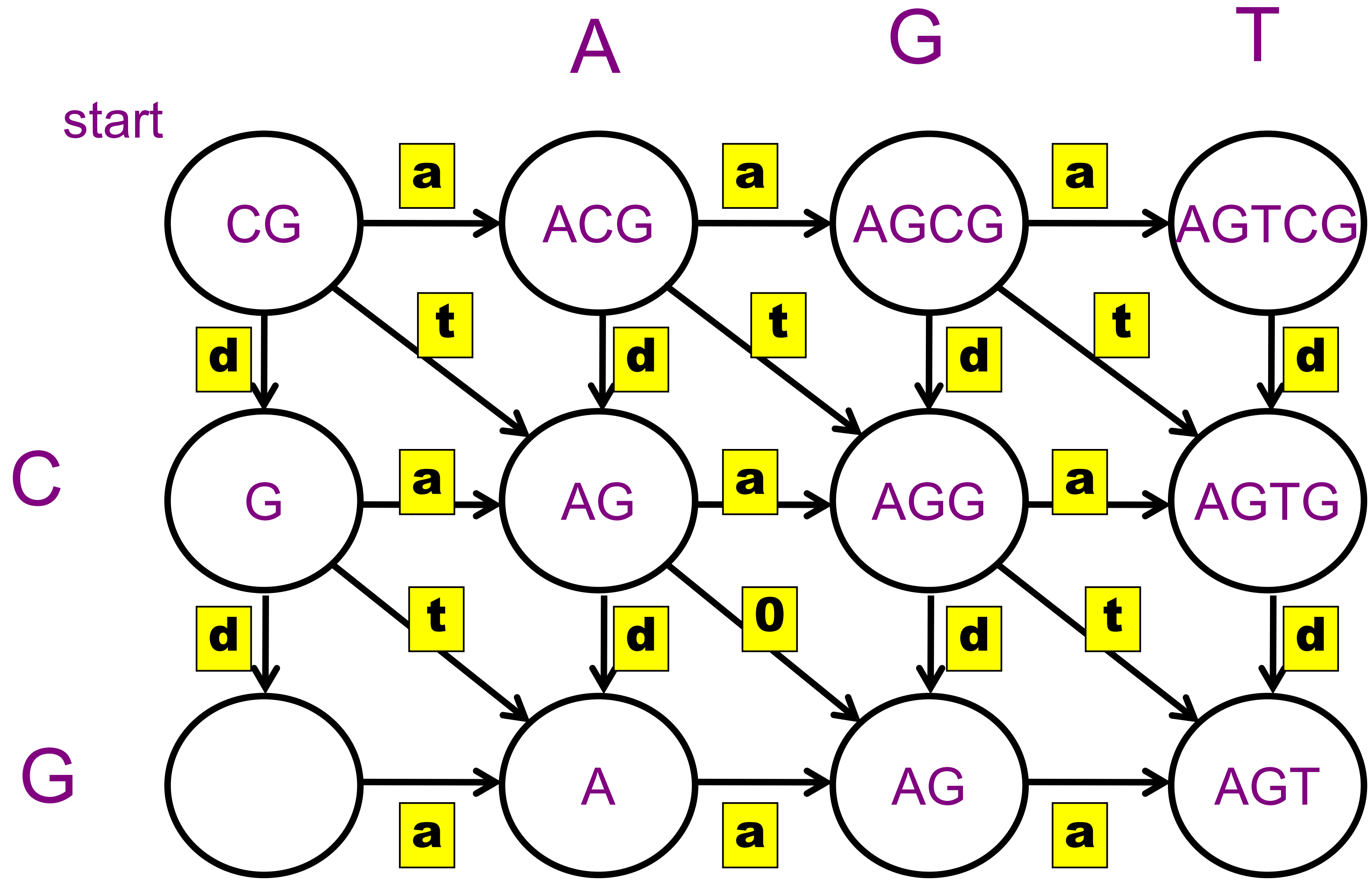
Vertical: delete character

Horizontal: add character

Diagonal: transform character

CG to AGT

Edge costs:



Vertical: delete character

Horizontal: add character

Diagonal: transform character

Example: DNA Alignment

Transform: CG to AGT

$$\begin{tikzcd}
 & & A & G & T \\
\text{start} & CG \arrow[r] \arrow[d] \arrow[dr] & ACG \arrow[r] \arrow[d] \arrow[dr] & AGCG \arrow[r] \arrow[d] \arrow[dr] & AGTCG \arrow[d] \\
C & G \arrow[r] \arrow[d] \arrow[dr] & AG \arrow[r] \arrow[d] \arrow[dr] & AGG \arrow[r] \arrow[d] \arrow[dr] & AGTG \arrow[d] \\
G & \bigcirc \arrow[r] & A \arrow[r] & AG \arrow[r] & AGT
\end{tikzcd}$$

Vertical: delete character

Horizontal: add character

Diagonal: transform character

N^2 algo

Roadmap

Part I: Shortest Paths

Part II: Applications of Shortest Paths

- DNA Alignment
- Constraint Systems

Example: Scheduling

Input:

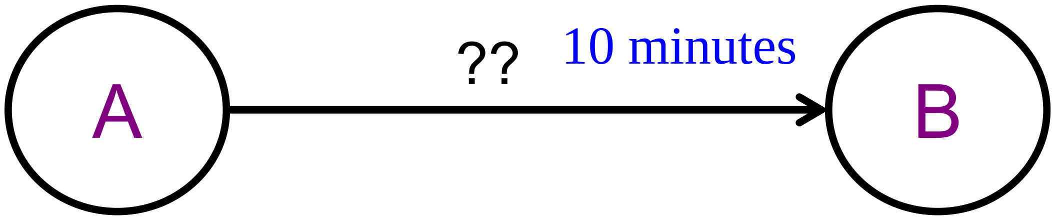
- Set of tasks: A, B, C, D, E, F
- Constraints:
  - A must be done at least 10 minutes before C
  - D must be done at most 20 minutes after E
  - B must be done after F

Output:

- Feasible?
- Schedule?

Example: Scheduling

B must be executed ~~**at least**~~ 10 minutes **after** A



Shortest path to B:
1) Went thru A
2) Didnt not go thru A

If the shortest path to B didnt go thru A, then Db /> Da + 10

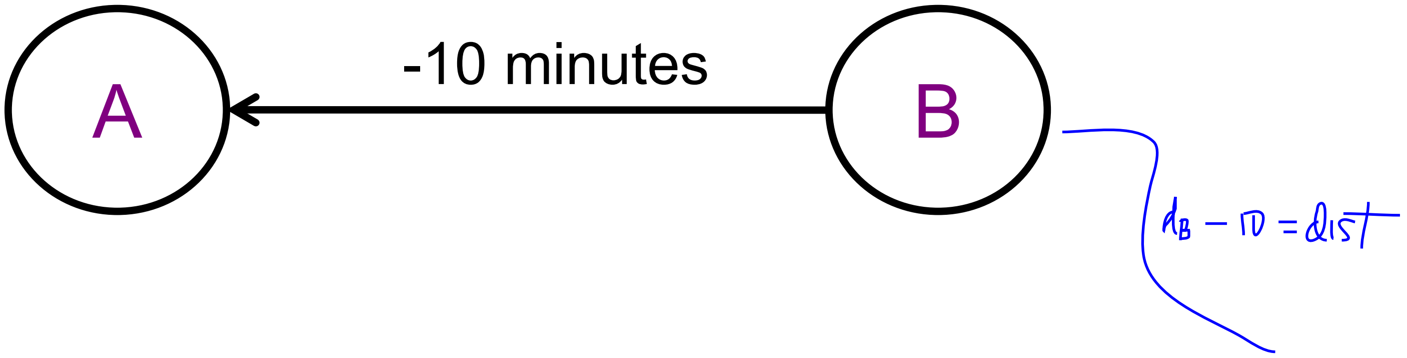
Shortest path = schedule time

Db <= Da + 10

triangle inequality: shortest path to B is at most 10 longer than shortest path to A

Example: Scheduling

B must be executed **at least** 10 minutes **after** A



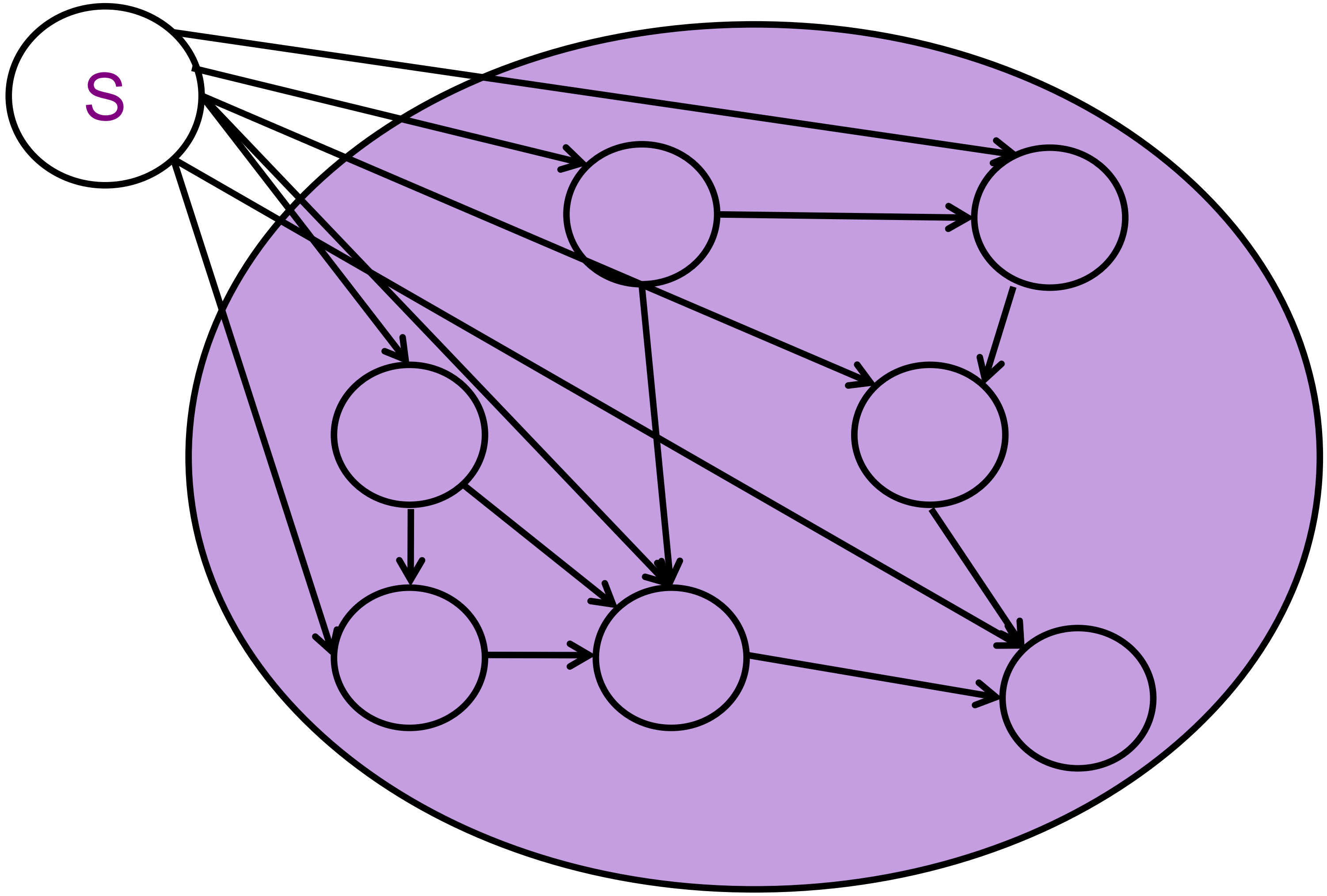
Shortest path = schedule time

$d_A \leq d_B - 10$

triangle inequality: shortest path to B is at least 10 longer than shortest path to A

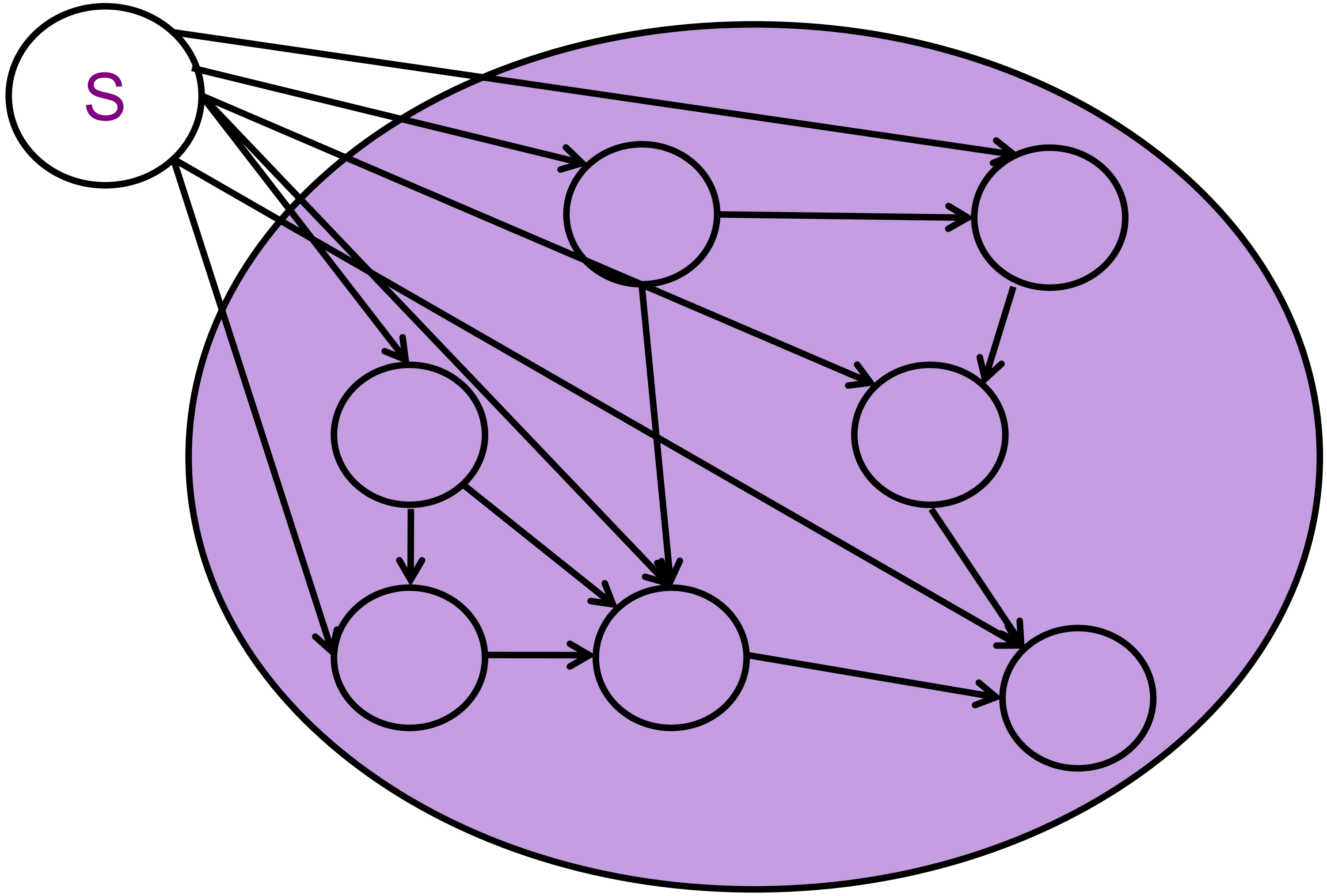
Example: Scheduling

Add source S connected by 0 weight edges to all.



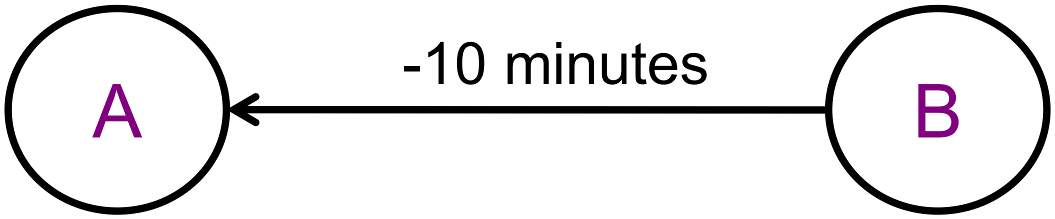
Example: Scheduling

Solve shortest paths.



Example: Scheduling

B must be executed **at least** 10 minutes **after** A



Negative edges: use Bellman-Ford!

no. vertex (jobs)

no. edge (constraints)

Running time: O(nm)

Example: Scheduling

Input:

- Set of tasks: A, B, C, D, E, F
- Constraints:
  - A must be done at least 10 minutes before C
  - D must be done at most 20 minutes after E
  - B must be done after F

Output:

- Shortest path guarantees constraints are met.
- Shortest path finishes all tasks in minimum time.

Roadmap

Part I: Shortest Paths

Part II: Applications of Shortest Paths

- DNA Alignment
- Constraint Systems