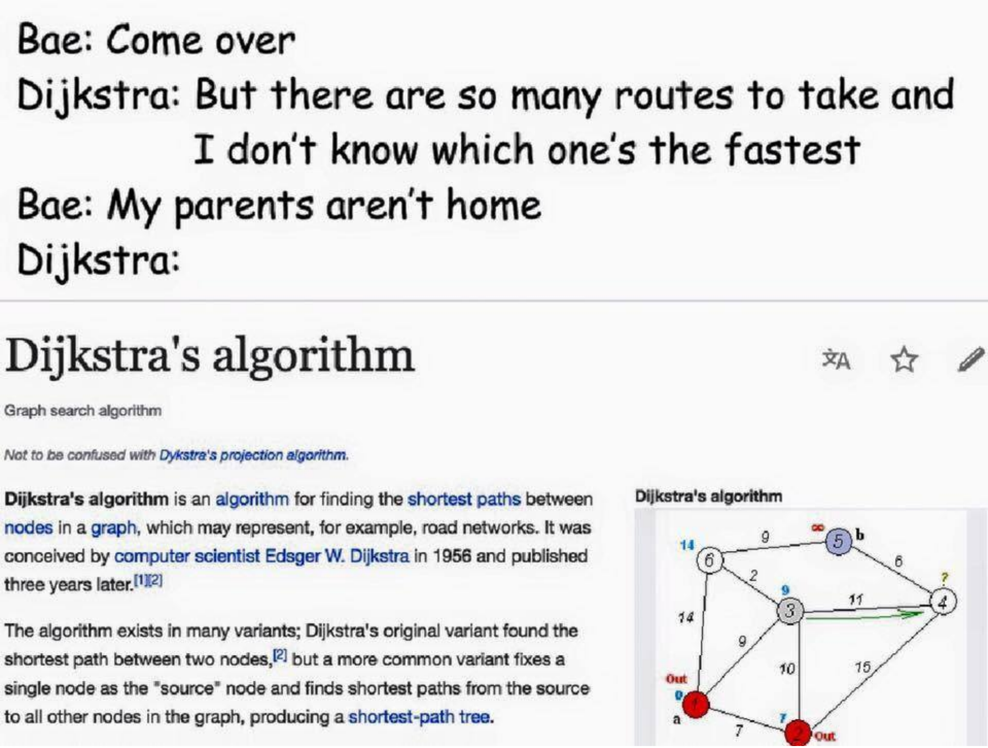
CSE 373 SP21 Section 6

Graphs



Bae: Come over

Dijkstra: But there are so many routes to take and I don't know which one's the fastest

Bae: My parents aren't home

Dijkstra:

Dijkstra's algorithm

Graph search algorithm

Not to be confused with Dykstra's projection algorithm.

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.[1][2]

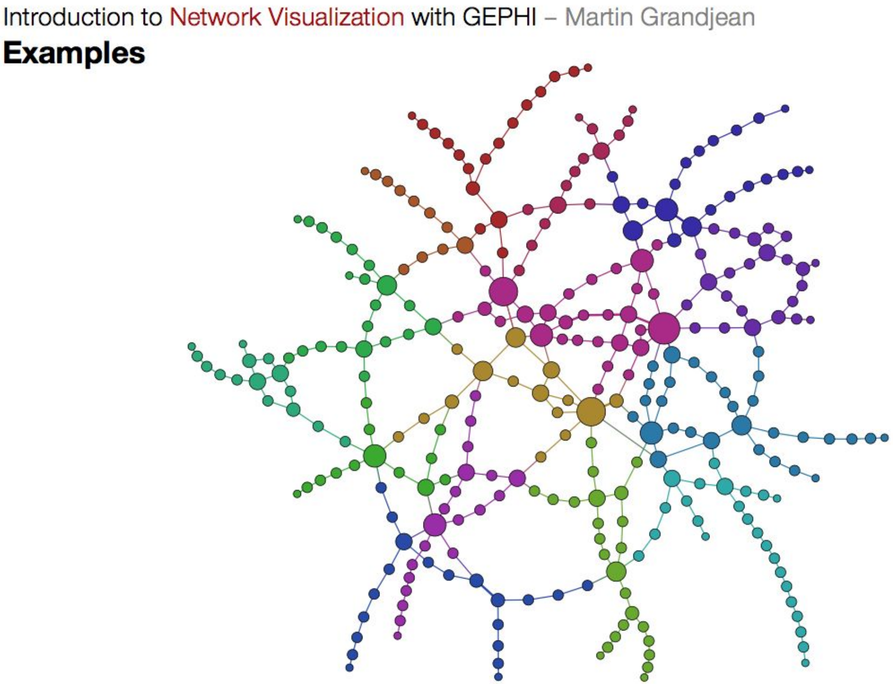
The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes,[2] but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

Dijkstra's algorithm

MicroTeach: Graph Intro

Graphs

Graph: A set of *nodes* (also called *vertices*) connected pairwise by *edges*.



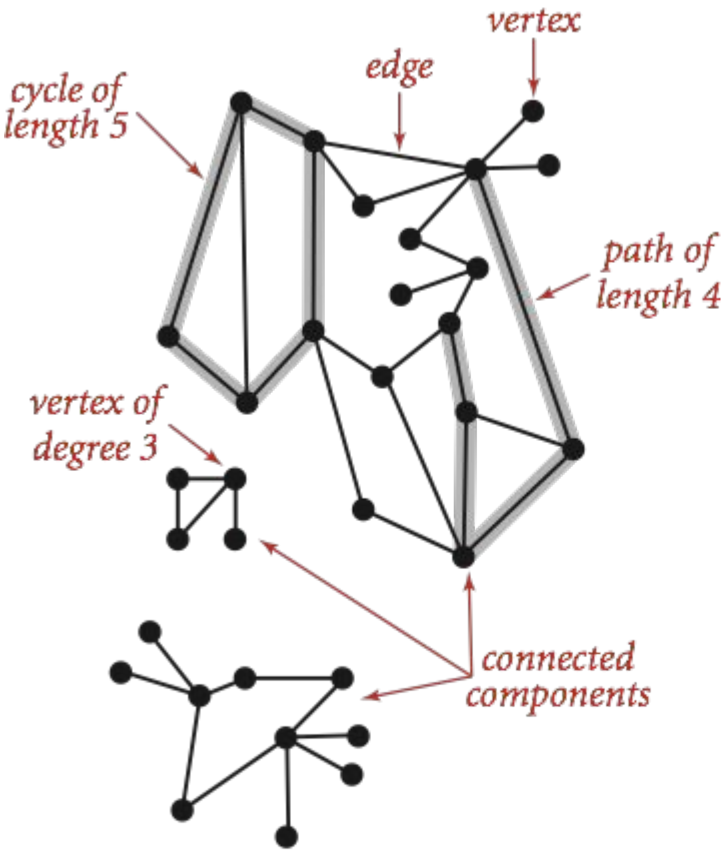
Introduction to Network Visualization with GEPHI – Martin Grandjean

Examples

Graph Terminology

- Graph:
  - Set of ***vertices***, a.k.a. ***nodes***.
  - Set of ***edges***: Pairs of vertices.
  - Vertices with an edge between are ***adjacent***.
  - The **degree** of a vertex is the number of edges directly connected to it.

- A ***path*** is a sequence of vertices connected by edges.
- A ***cycle*** is a path whose first and last vertices are the same.
  - A graph with a cycle is 'cyclic'.

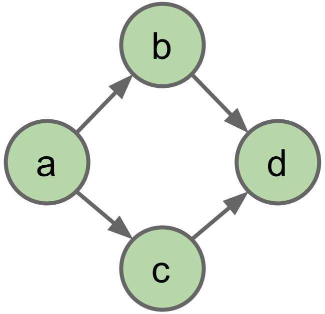


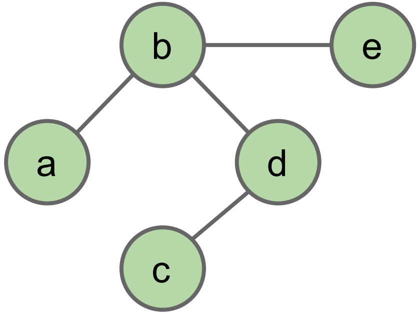
Some Graph Types

Directed

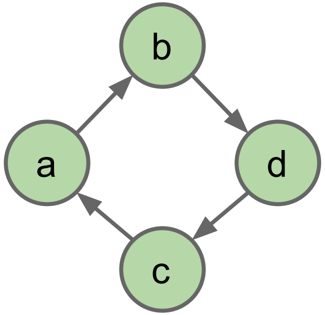
Undirected

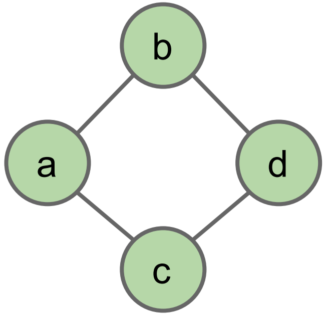
Acyclic:





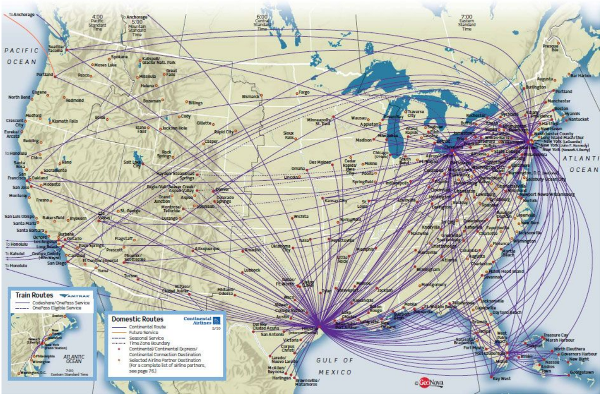
Cyclic:

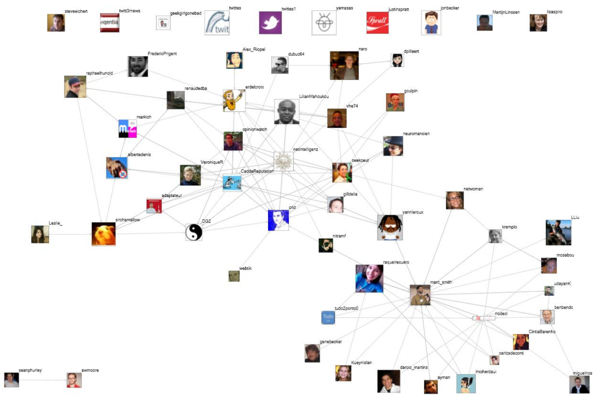




Graph Applications

- Physical Maps
  - Airline maps
  - Traffic
- Relationships
  - Social media graphs
  - Code bases
- Influence
  - Biology
- Related topics
  - Web Page Ranking
  - Wikipedia
- Many more...

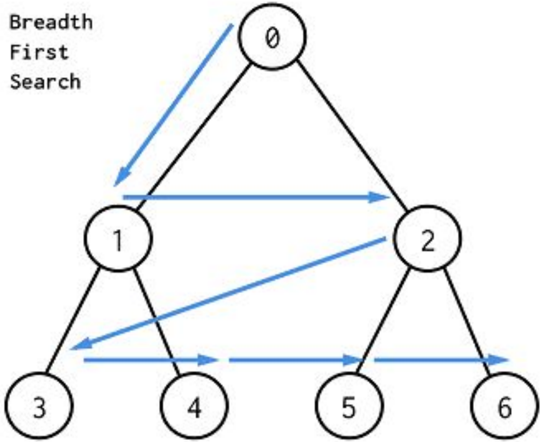




MicroTeach: DFS/BFS

BFS Pseudocode (simplified)

```
Queue q

add Vertex start to q
mark start as discovered

while q is not empty {
    Vertex from = q.remove()
    for each edge {from,d} {
        if d is not discovered {
            add d to q
            mark d as discovered
        }
    }
}
```

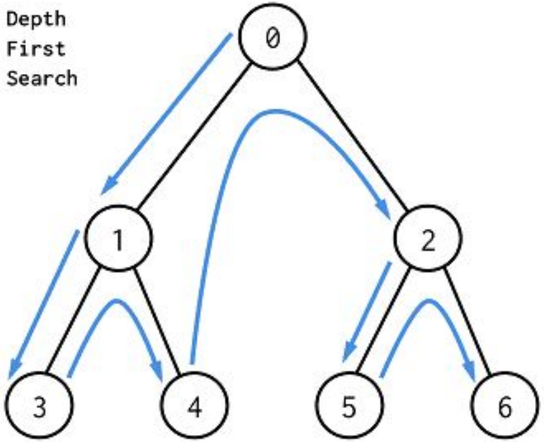


BFS Pseudocode

```
bfs(Graph graph, Vertex start) {
    // stores the remaining vertices to visit in the BFS
    Queue<Vertex> perimeter = new Queue<>();

    // stores the set of discovered vertices so we don't revisit them multiple times
    Set<Vertex> discovered = new Set<>();

    // kicking off our starting point by adding it to the perimeter
    perimeter.add(start);
    discovered.add(start);
    while (!perimeter.isEmpty()) {
        Vertex from = perimeter.remove();
        for (E edge : graph.outgoingEdgesFrom(from)) {
            Vertex to = edge.to();
            if (!discovered.contains(to)) {
                perimeter.add(to);
                discovered.add(to);
            }
        }
    }
}
```

DFS Pseudocode (simplified)

```
Stack s

add Vertex start to s

while s is not empty {
    Vertex from = s.remove()
    if from is not discovered {
        for each edge {from,d} {
            add d to s
        }
        mark from as discovered
    }
}
```



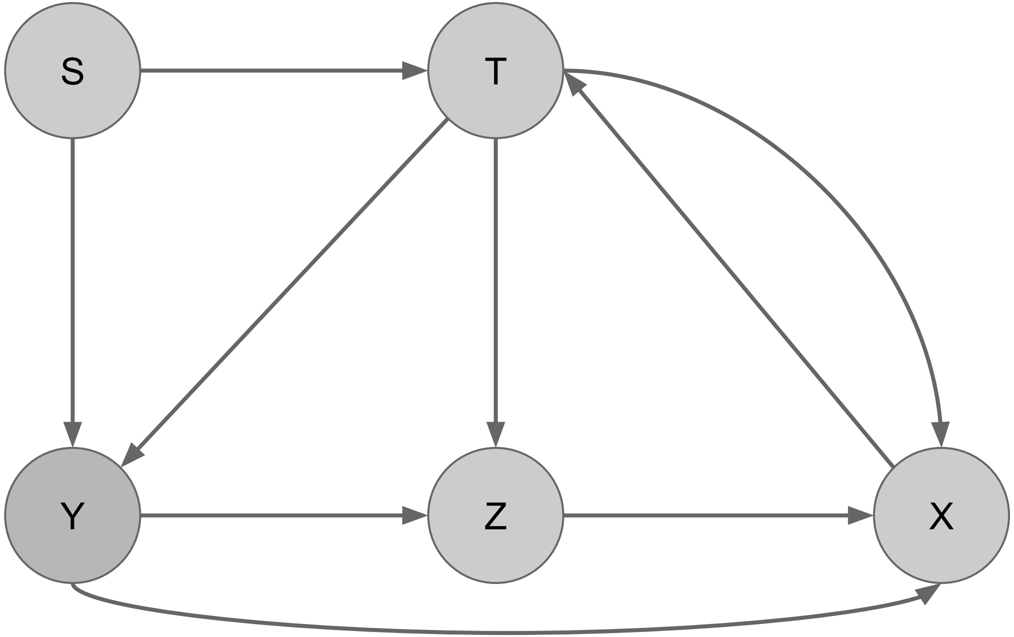
*** Fixes the "bug" Kasey mentioned in Lecture 17!**
Can you spot the change? :)

DFS Pseudocode

```
dfs(Graph graph, Vertex start) {
    // stores the remaining vertices to visit in the DFS
    Stack<Vertex> perimeter = new Stack<>();

    // stores the set of discovered vertices so we don't revisit them multiple times
    Set<Vertex> discovered = new Set<>();

    // kicking off our starting point by adding it to the perimeter
    perimeter.add(start);

    while (!perimeter.isEmpty()) {
        Vertex from = perimeter.remove();
        if (!discovered.contains(from)) {
            for (E edge : graph.outgoingEdgesFrom(from)) {
                Vertex to = edge.to();
                perimeter.add(to);
            }
            discovered.add(from);
        }
    }
}
```

*** Fixes the “bug” Kasey mentioned in Lecture 17! Can you spot the change? :)**

Problem 3:
Simulating BFS



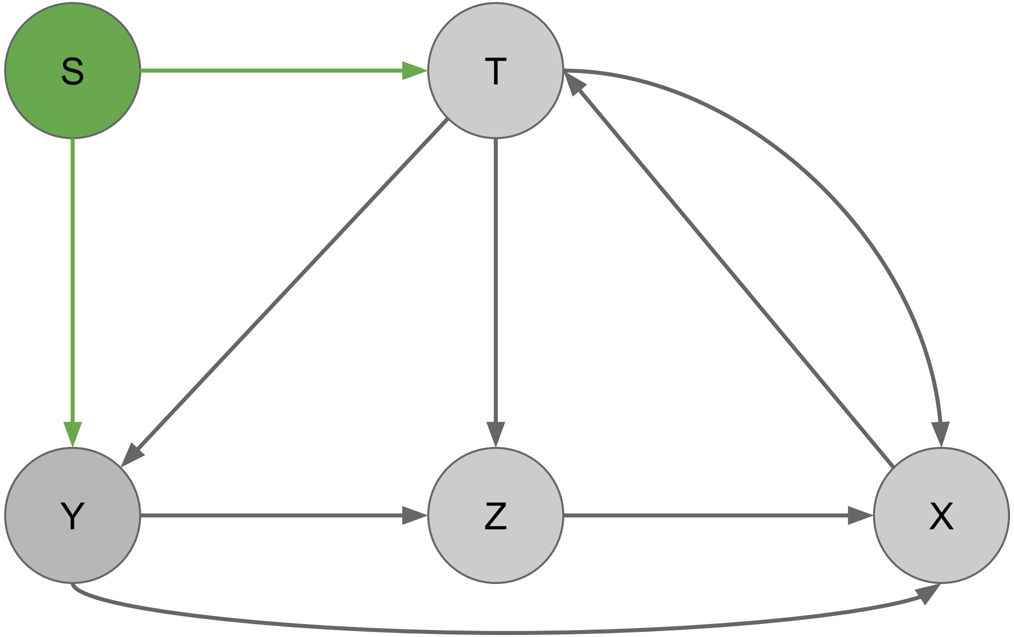
Vertex	Pred	Processed (?)
S	--	
T	--	
X	--	
Y	--	
Z	--	

Queue of Vertices to Explore:

S				

Begin with the start vertex in the queue!

Simulating BFS

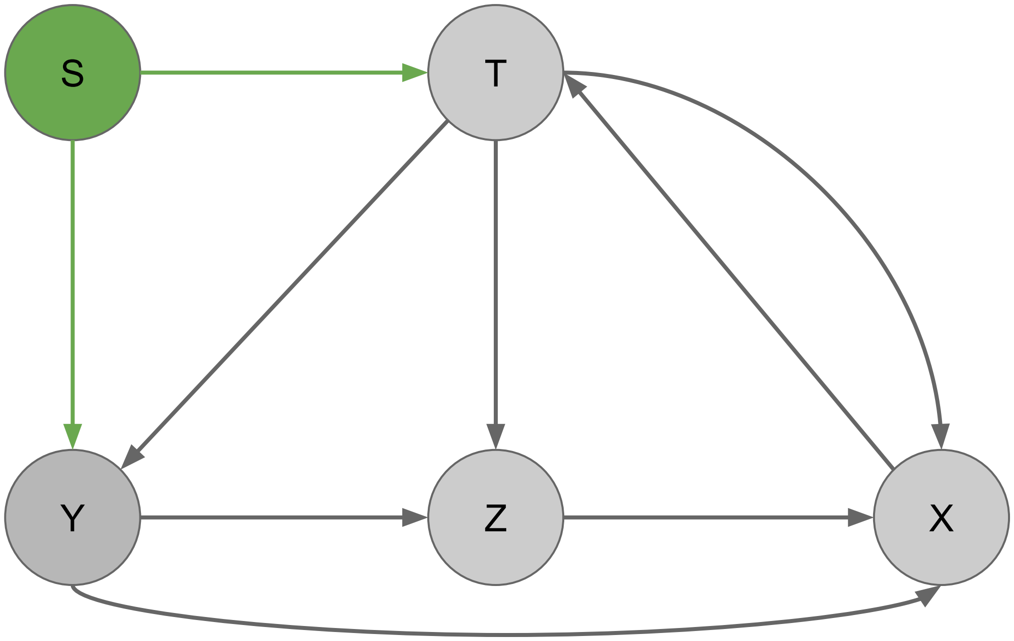


Queue of Vertices to Explore:

Pop S to explore!

Vertex	Pred	Processed (?)
S	--	✓
T	--	
X	--	
Y	--	
Z	`--	

Simulating BFS



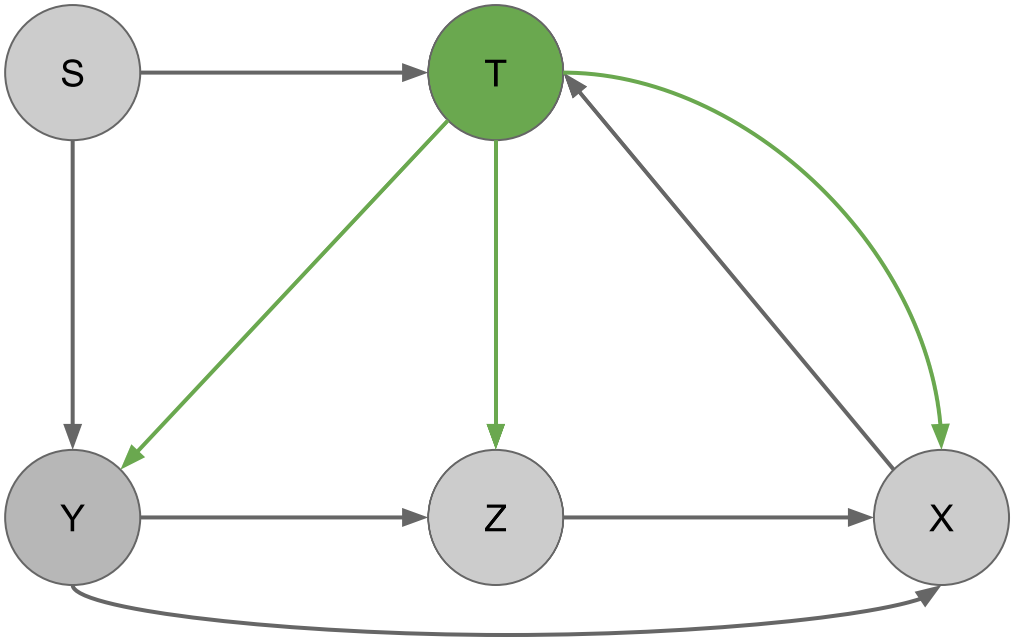
Vertex	Pred	Processed (?)
S	--	✓
T	S	
X	--	
Y	S	
Z	--	

Queue of Vertices to Explore:

T	Y			

Push neighbors of S onto queue to be explored

Simulating BFS



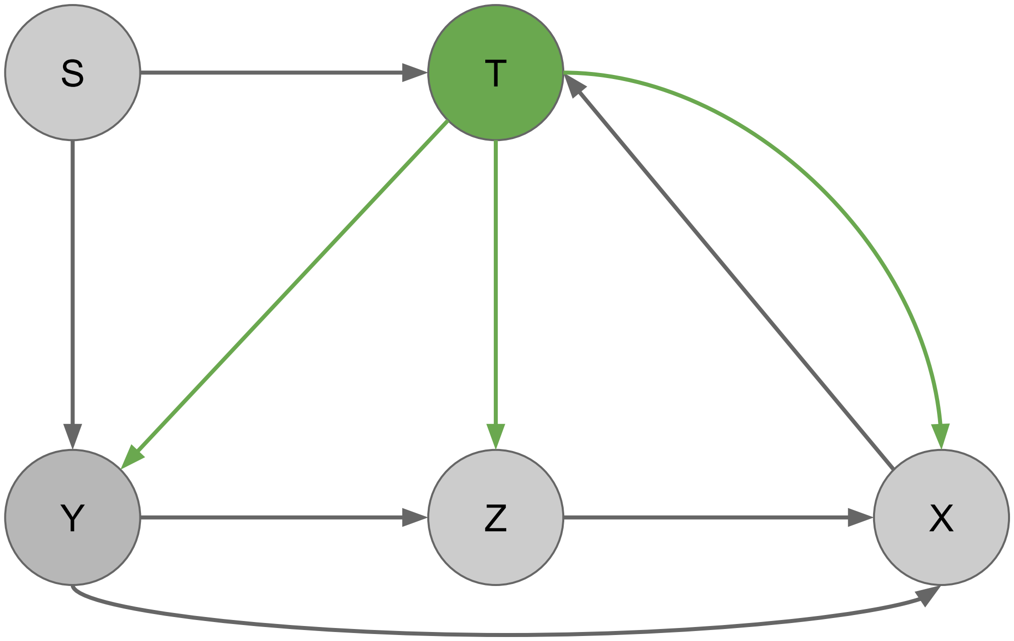
Vertex	Pred	Processed (?)
S	--	✓
T	S	✓
X	--	
Y	S	
Z	--	

Queue of Vertices to Explore:

Y				

Pop T to explore!

Simulating BFS



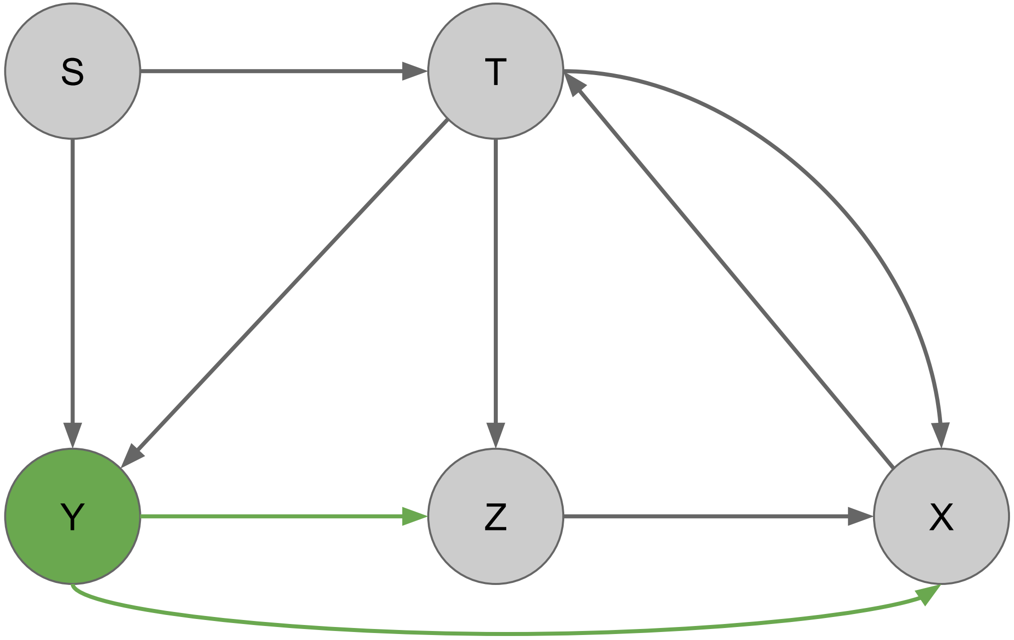
Queue of Vertices to Explore:

Y	X	Z		

Push neighbors of T onto queue to be explored

Vertex	Pred	Processed (?)
S	--	✓
T	S	✓
X	T	
Y	S	
Z	T	

Simulating BFS



Vertex	Pred	Processed (?)
S	--	✓
T	S	✓
X	T	
Y	S	✓
Z	T	

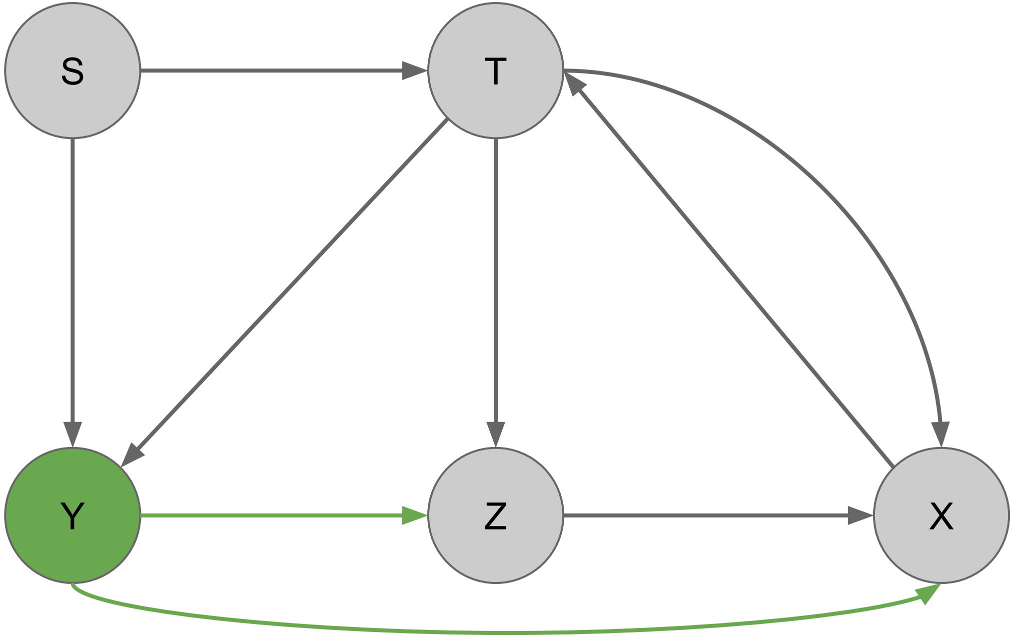
Queue of Vertices to Explore:

X	Z			

Pop Y to explore!

Simulating BFS

Simulating BFS

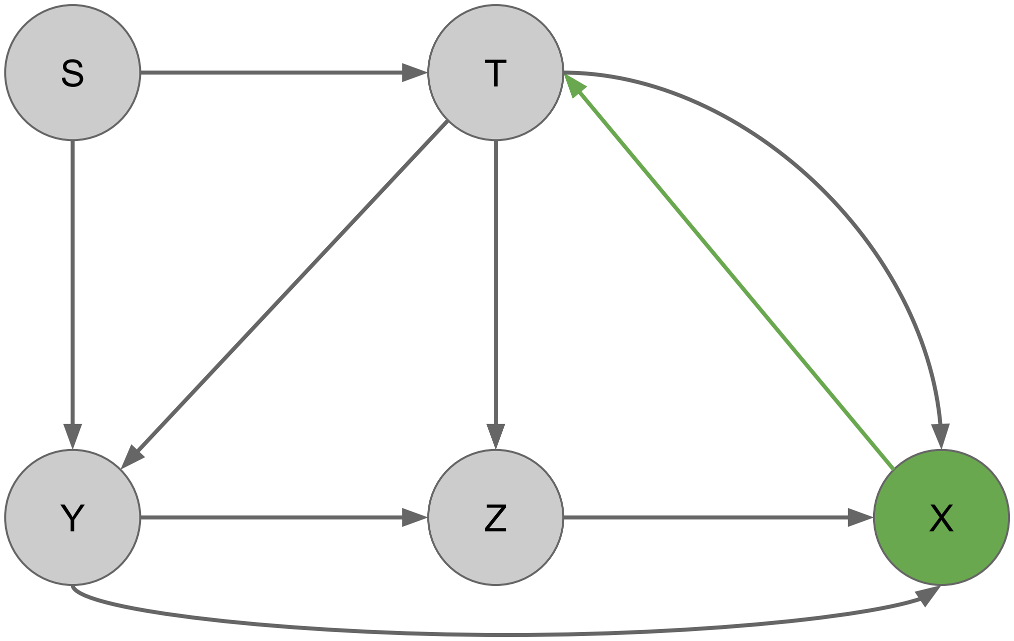


Queue of Vertices to Explore:

X	Z			

Vertex	Pred	Processed (?)
S	--	✓
T	S	✓
X	T	
Y	S	✓
Z	T	

(Nothing happens!)



Vertex	Pred	Processed (?)
S	--	✓
T	S	✓
X	T	✓
Y	S	✓
Z	T	

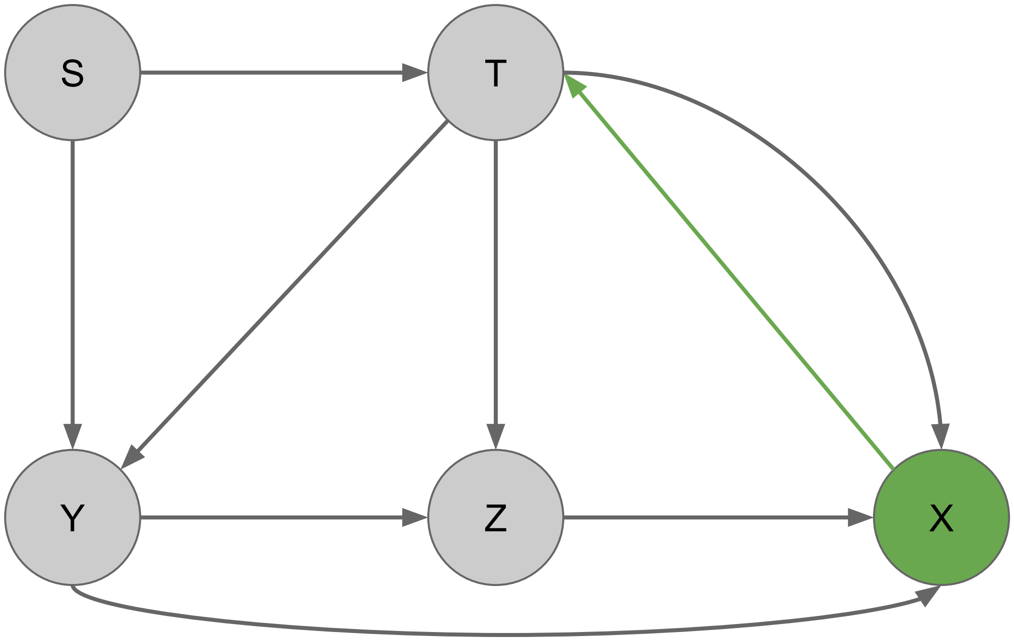
Queue of Vertices to Explore:

Z				

Pop X to explore!

Simulating BFS

Simulating BFS

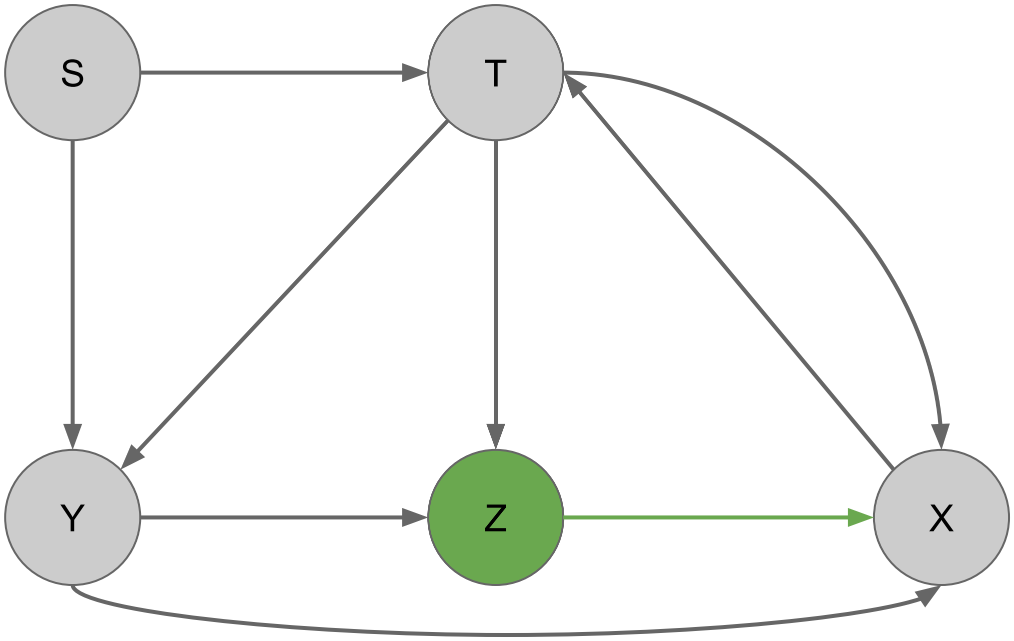


Queue of Vertices to Explore:

Z				

Vertex	Pred	Processed (?)
S	--	✓
T	S	✓
X	T	✓
Y	S	✓
Z	T	

(Nothing happens!)

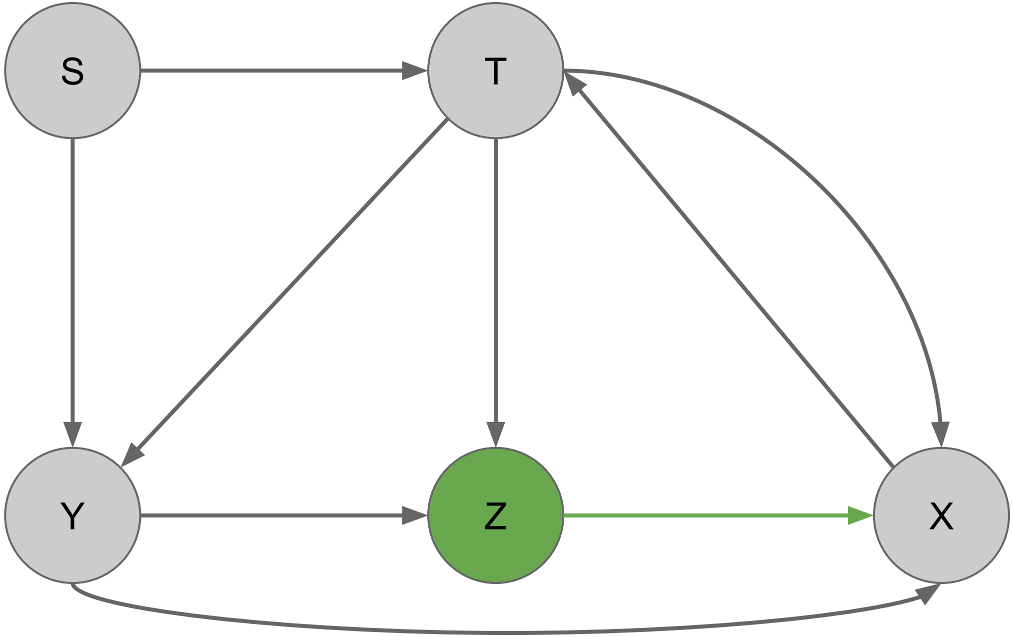


Vertex	Pred	Processed (?)
S	--	✓
T	S	✓
X	T	✓
Y	S	✓
Z	T	✓

Queue of Vertices to Explore:

Pop Z to explore!

Simulating BFS

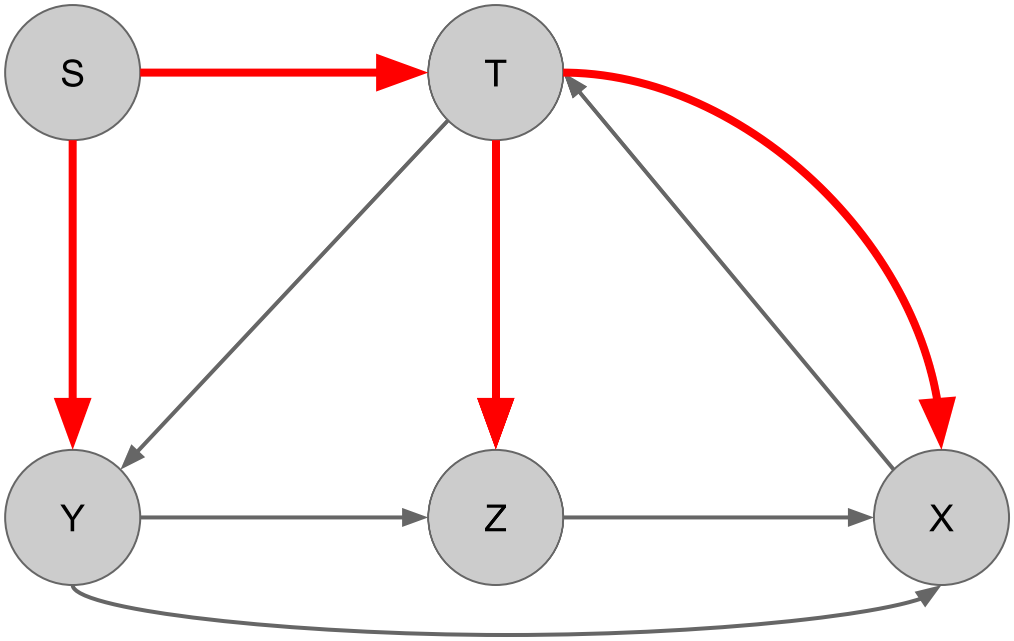


Queue of Vertices to Explore:

Vertex	Pred	Processed (?)
S	--	✓
T	S	✓
X	T	✓
Y	S	✓
Z	T	✓

(Nothing happens!)

Simulating BFS



Resulting SPT

Vertex	Pred	Processed (?)
S	--	✓
T	S	✓
X	T	✓
Y	S	✓
Z	T	✓

Simulating BFS

How do we interpret the final table?

To check if there exists a path from a given start node to given target…

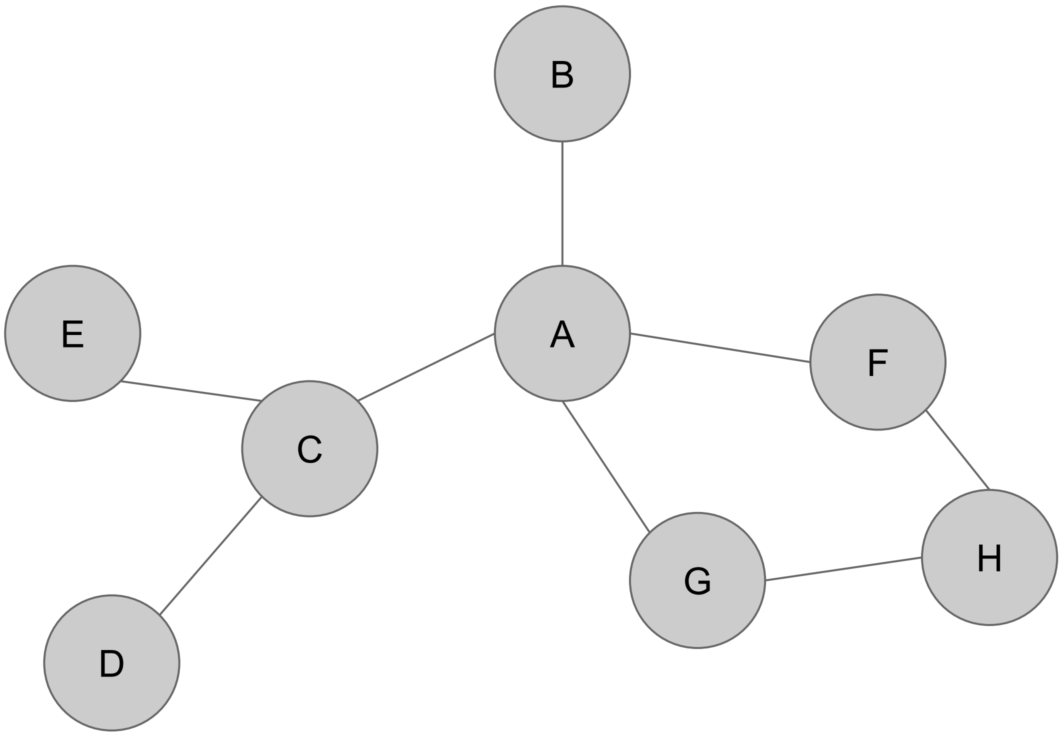
- Locate the target vertex in the table
- Backtrace through its predecessors
- If the start vertex is one of its predecessors, then there exists a path between them, otherwise there does not

To find the resulting shortest paths tree (SPT)…

- For each vertex, backtrace from its predecessors until you reach the source vertex
- This is the same as getting the shortest path from the source to each vertex
- By combining the shortest paths to each vertex in the graph, you will get the SPT for the graph

Vertex	Pred	Processed (?)
S	--	✓
T	S	✓
X	T	✓
Y	S	✓
Z	T	✓

Problem 2:
Graph Traversal



If we traverse this using breadth-first search, what are the two possible orderings of the nodes we visit?

What if we use depth-first search

Leetcode Problem: Find if Path Exists

(https://leetcode.com/problems/find-if-path-exists-in-graph/)

Leetcode!

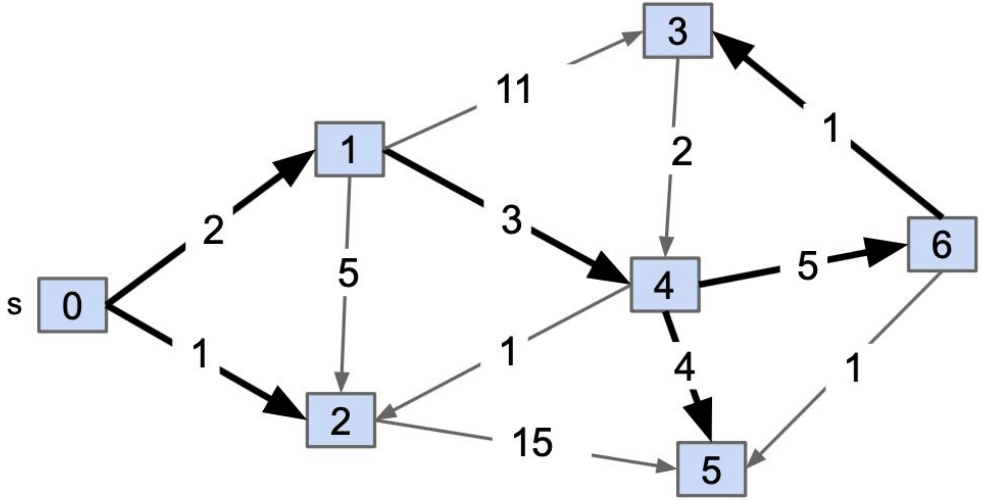
Before coding the solution…

1. Read the problem in its entirety
2. Ensure you understand any edge cases
3. Think about the different possible solutions
    a. You will most likely start thinking about the brute force solution first, and that's okay!
    b. Consider runtime (and memory) complexity
        i. Most likely in terms of Big-O
4. Write pseudocode
5. Code out solution
6. Test Solution
7. Optimize -> repeat steps (3-8)

MicroTeach: Dijkstra’s

Dijkstra’s Algorithm: single-pair-shortest-path

- Pathfinding on a *weighted* graph!
- Main idea: find shortest path/shortest distance from *start* node in graph to every other node.
- Uses a Priority Queue, where priorities of nodes are their distance from the start node
- We pull the closest node off the queue each iteration, and update the distances for its adjacent nodes. Then repeat.



Dijkstra’s Pseudocode

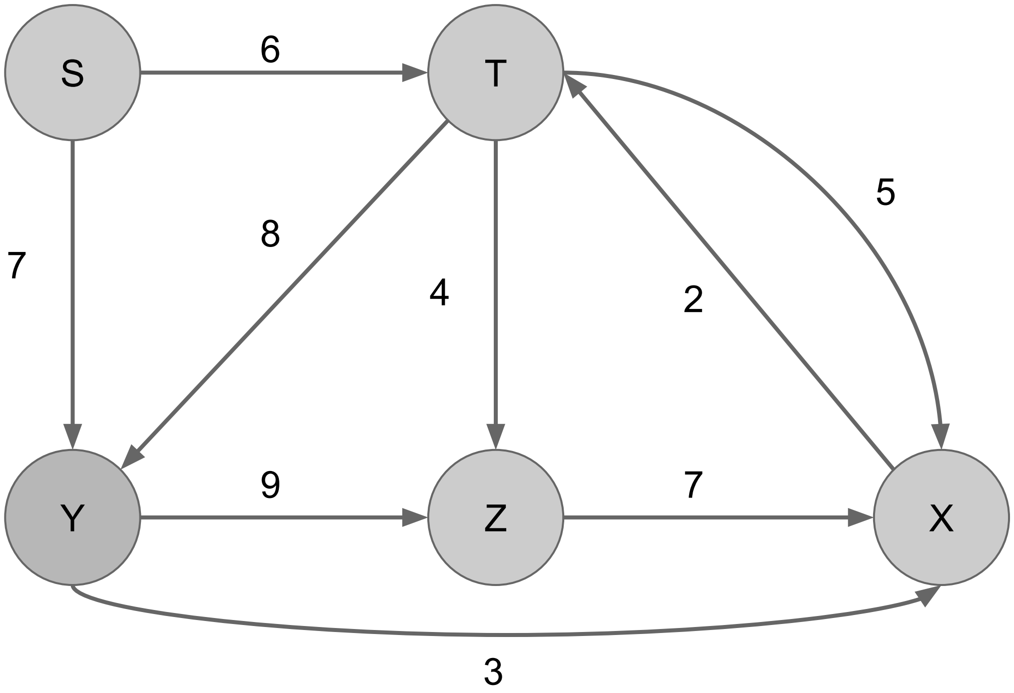
```
Dijkstra(Graph G, Vertex source)
    initialize distances to ∞
    mark all vertices unprocessed
    mark source as distance 0
    while(there are unprocessed vertices){
        let u be the closest unprocessed vertex
        for each(edge (u,v) leaving u){
            if(u.dist+weight(u,v) < v.dist){
                v.dist = u.dist+weight(u,v)
                v.predecessor = u
            }
        }
        mark u as processed
    }
```

Q: How to get the shortest path?

A: After running Dijkstra, start from the target node and **follow the backpointers!**

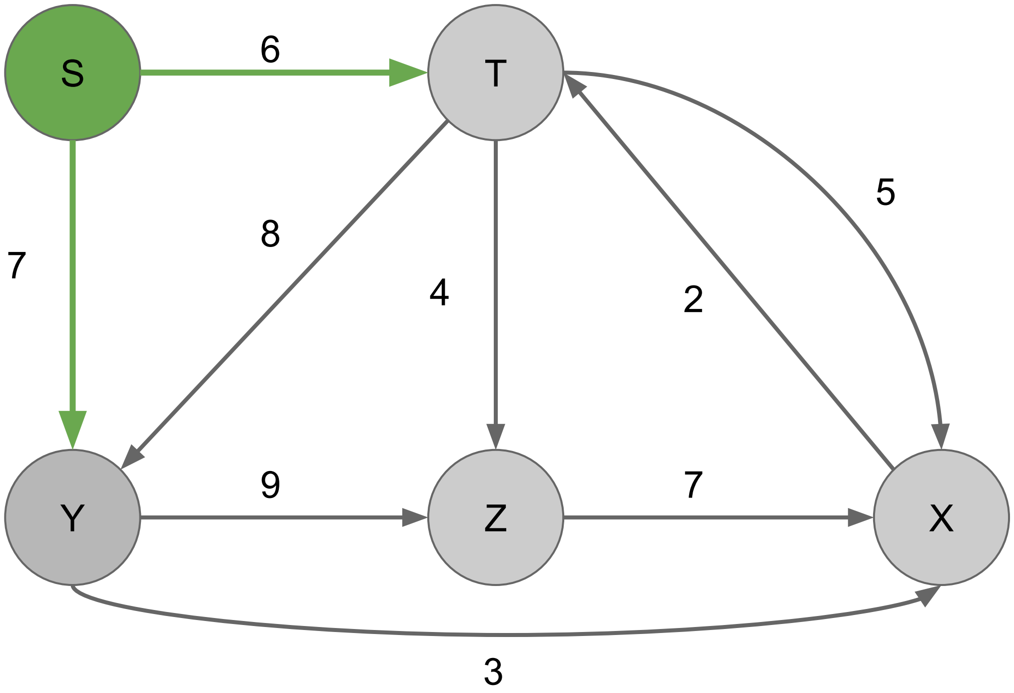
```
GetPath(Graph G, Vertex source, Vertex target)
    // We never reached the target :(
    if (target.dist == INFINITY)
        return null

    path = []
    curNode = target
    path.add_back(target)

    while(curNode != source)
        curNode = curNode.predecessor
        path.add_back(curNode)

    // If we want the path to go from source -> goal.
    return path.reversed()
```

Problem 4A: Dijkstra



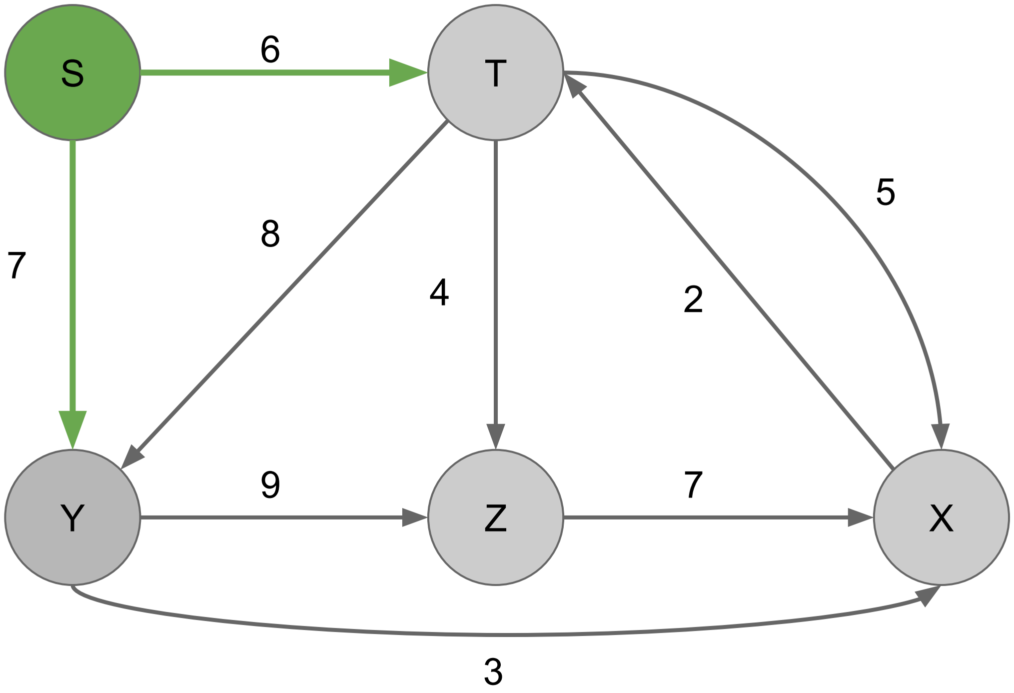
Vertex	Distance	Pred	Processed (?)
S	0	--	
T	inf	--	
X	inf	--	
Y	inf	--	
Z	inf	--	

Problem 4A: Dijkstra



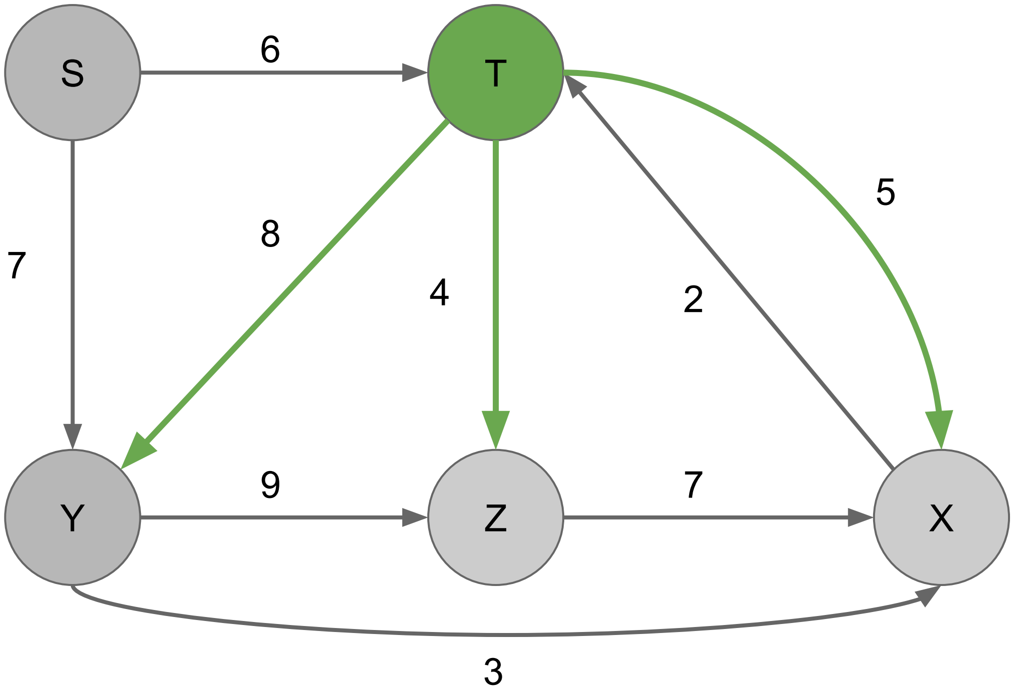
Vertex	Distance	Pred	Processed (?)
S	0	--	✓
T	inf	--	
X	inf	--	
Y	inf	--	
Z	inf	--	

Problem 4A: Dijkstra



Vertex	Distance	Pred	Processed (?)
S	0	--	✓
T	~~inf~~ **6**	**S**	
X	inf	--	
Y	~~inf~~ **7**	**S**	
Z	inf	--	

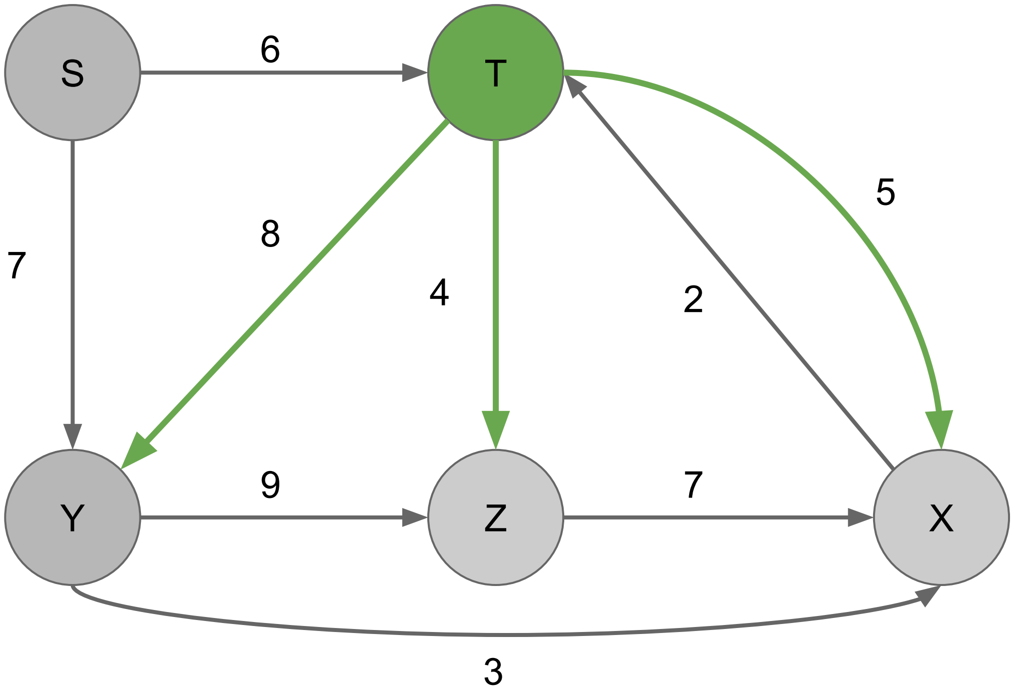
Problem 4A: Dijkstra



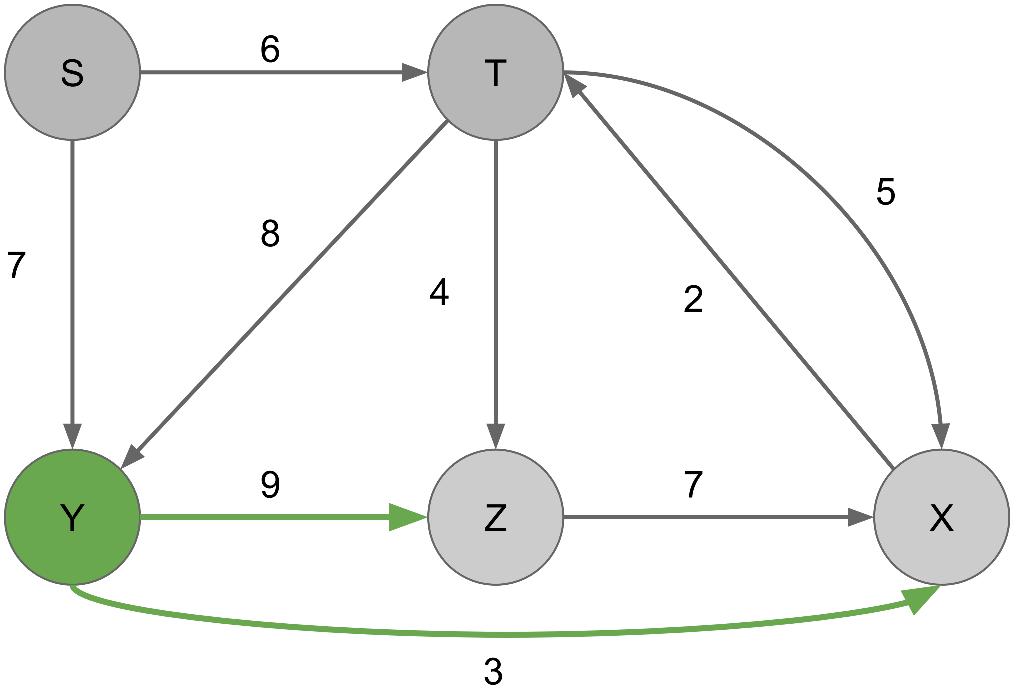
Vertex	Distance	Pred	Processed (?)
S	0	--	✓
T	~~inf~~ 6	S	✓
X	inf	--	
Y	~~inf~~ 7	S	
Z	inf	--	

Problem 4A: Dijkstra

Problem 4A: Dijkstra

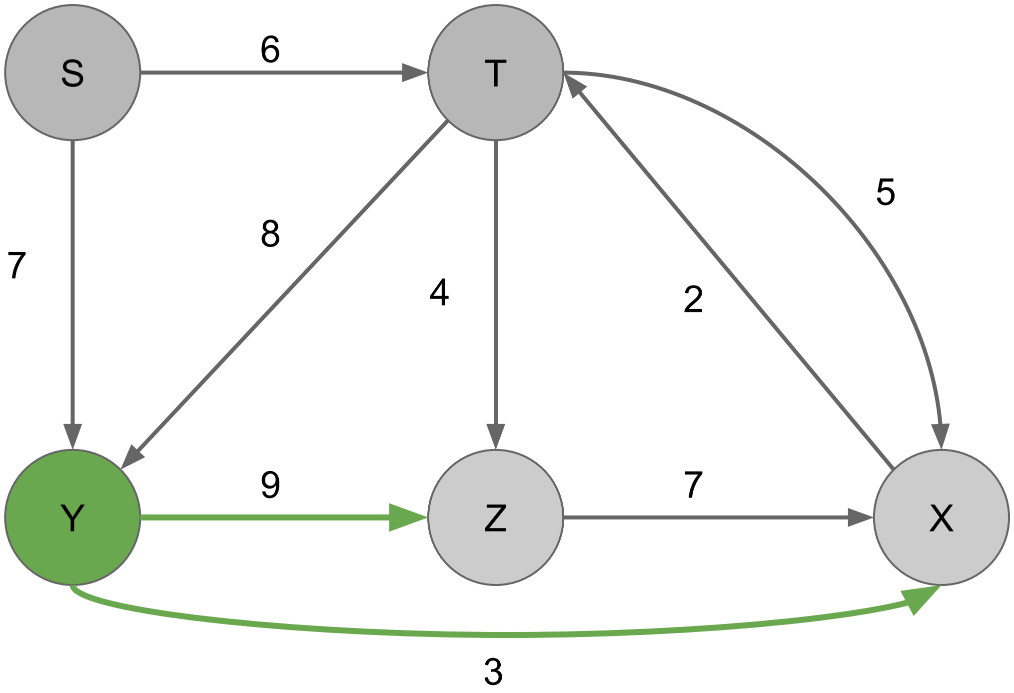


Vertex	Distance	Pred	Processed (?)
S	0	--	✓
T	~~inf~~ 6	S	✓
X	~~inf~~ **11**	**T**	
Y	~~inf~~ 7	S	
Z	~~inf~~ **10**	**T**	



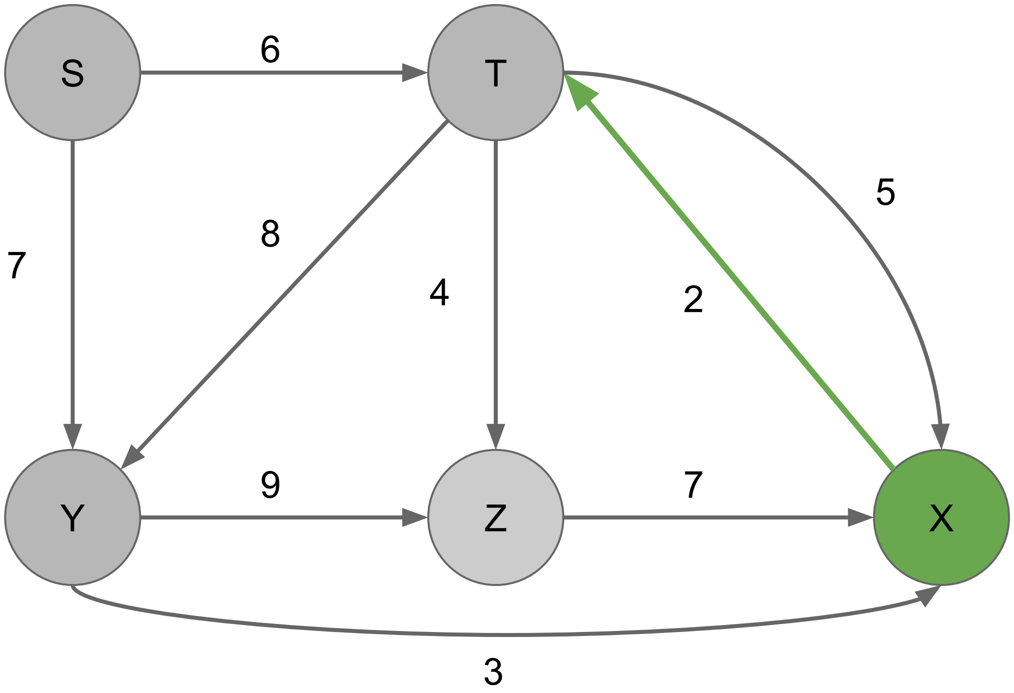
Vertex	Distance	Pred	Processed (?)
S	0	--	✓
T	~~inf~~ 6	S	✓
X	~~inf~~ 11	T	
Y	~~inf~~ 7	S	✓
Z	~~inf~~ 10	T	

Problem 4A: Dijkstra



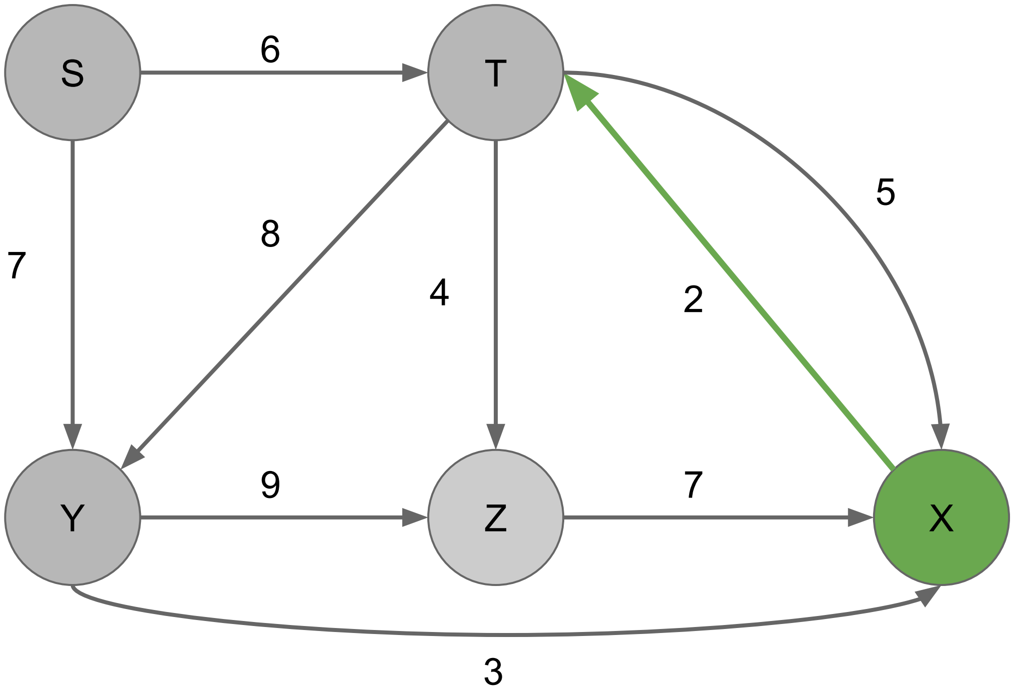
Vertex	Distance	Pred	Processed (?)
S	0	--	✓
T	~~inf~~ 6	S	✓
X	~~inf~~ ~~11~~ **10**	~~T~~ **Y**	
Y	~~inf~~ 7	S	✓
Z	~~inf~~ 10	T	

Problem 4A: Dijkstra



Vertex	Distance	Pred	Processed (?)
S	0	--	✓
T	~~inf~~ 6	S	✓
X	~~inf 11~~ 10	~~T~~ Y	✓
Y	~~inf~~ 7	S	✓
Z	~~inf~~ 10	T	

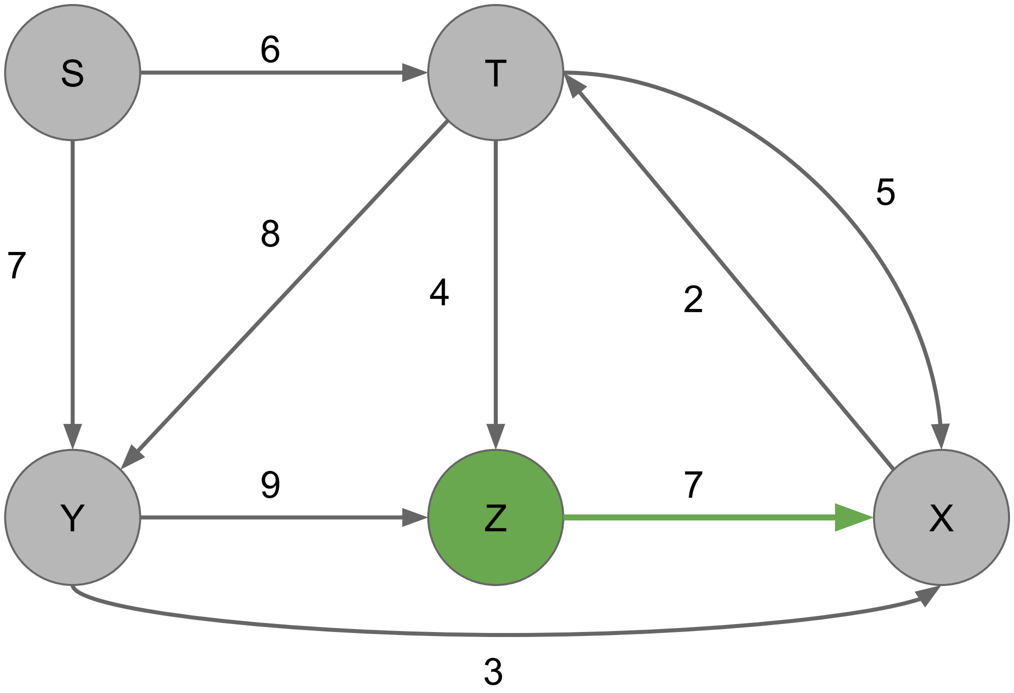
Problem 4A: Dijkstra



Vertex	Distance	Pred	Processed (?)
S	0	--	✓
T	~~inf~~ 6	S	✓
X	~~inf 11~~ 10	~~T~~ Y	✓
Y	~~inf~~ 7	S	✓
Z	~~inf~~ 10	T	

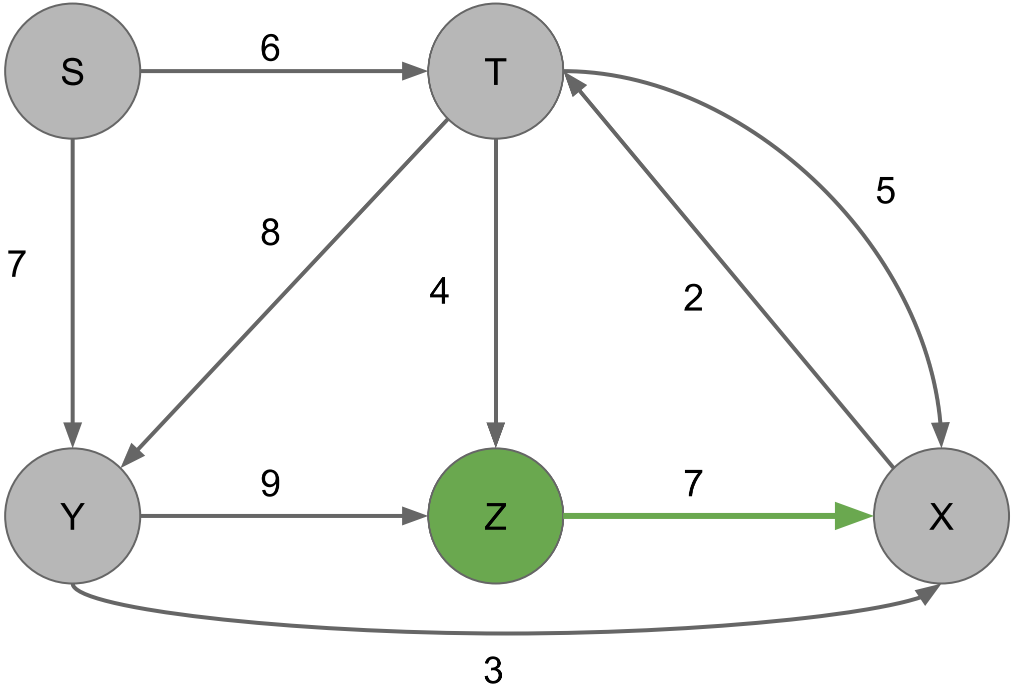
(Nothing happens!)

Problem 4A: Dijkstra



Vertex	Distance	Pred	Processed (?)
S	0	--	✓
T	~~inf~~ 6	S	✓
X	~~inf 11~~ 10	~~T~~ Y	✓
Y	~~inf~~ 7	S	✓
Z	~~inf~~ 10	T	✓

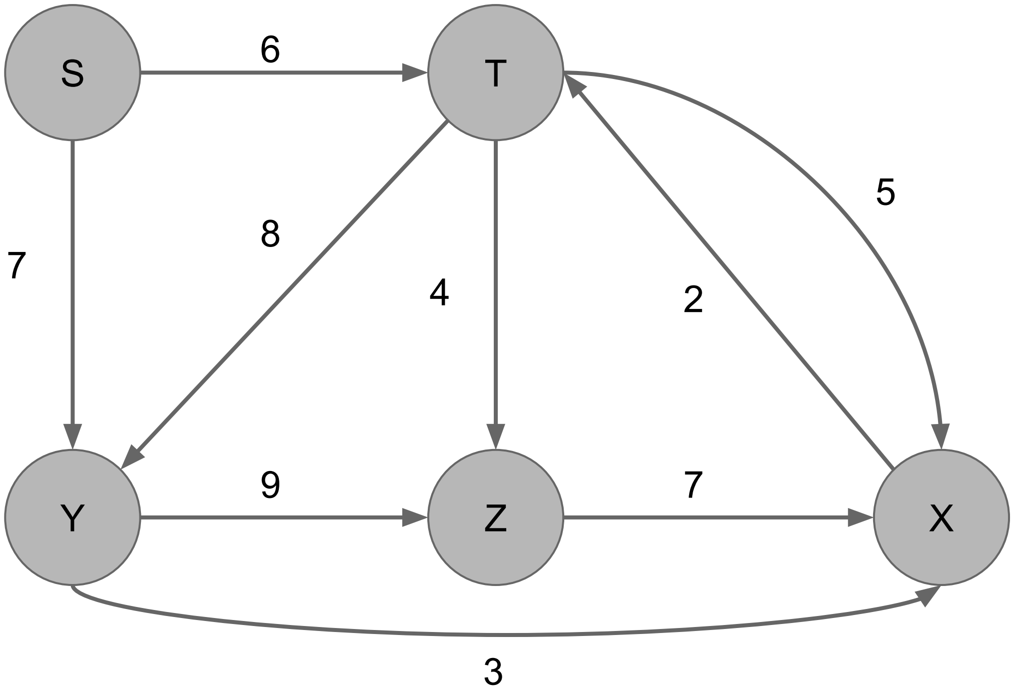
Problem 4A: Dijkstra



Vertex	Distance	Pred	Processed (?)
S	0	--	✓
T	~~inf~~ 6	S	✓
X	~~inf 11~~ 10	~~T~~ Y	✓
Y	~~inf~~ 7	S	✓
Z	~~inf~~ 10	T	✓

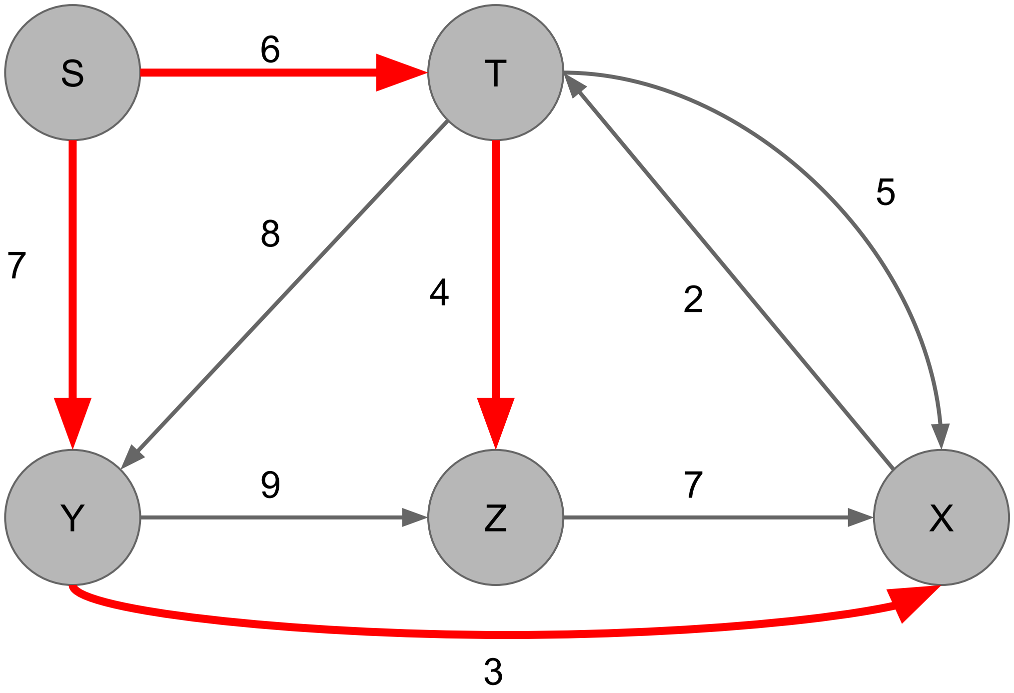
(Nothing happens!)

Problem 4A: Dijkstra



Vertex	Distance	Pred	Processed (?)
S	0	--	✓
T	~~inf~~ 6	S	✓
X	~~inf 11~~ 10	~~T~~ Y	✓
Y	~~inf~~ 7	S	✓
Z	~~inf~~ 10	T	✓

Problem 4A: Dijkstra



Resulting SPT

Vertex	Distance	Pred	Processed (?)
S	0	--	✓
T	~~inf~~ 6	S	✓
X	~~inf 11~~ 10	~~T~~ Y	✓
Y	~~inf~~ 7	S	✓
Z	~~inf~~ 10	T	✓

Problem 4A: Dijkstra

Why It Works - Understanding Dijkstra Invariants

Invariants

predecessor[**v**]: best known predecessor of **v**.

distTo[**v**]: best known distance of **s** to **v**.

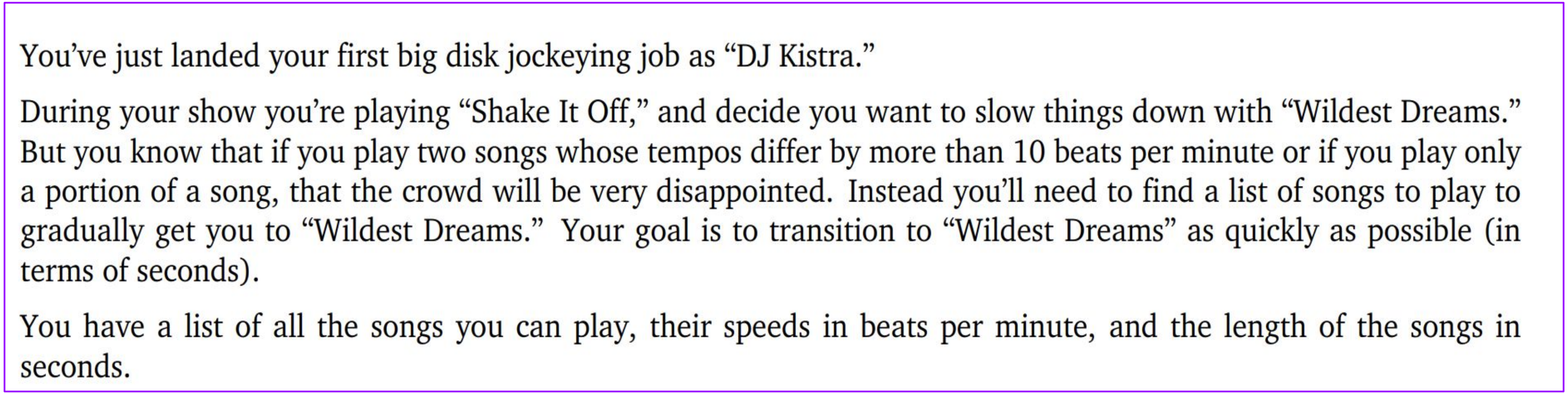
PQ maintains vertices based on distTo.

Important properties

Always visits vertices in order of total distance from source.

Problem 6: DJ Kistra

Problem 6



You’ve just landed your first big disk jockeying job as “DJ Kistra.”

During your show you’re playing “Shake It Off,” and decide you want to slow things down with “Wildest Dreams.” But you know that if you play two songs whose tempos differ by more than 10 beats per minute or if you play only a portion of a song, that the crowd will be very disappointed. Instead you’ll need to find a list of songs to play to gradually get you to “Wildest Dreams.” Your goal is to transition to “Wildest Dreams” as quickly as possible (in terms of seconds).

You have a list of all the songs you can play, their speeds in beats per minute, and the length of the songs in seconds.

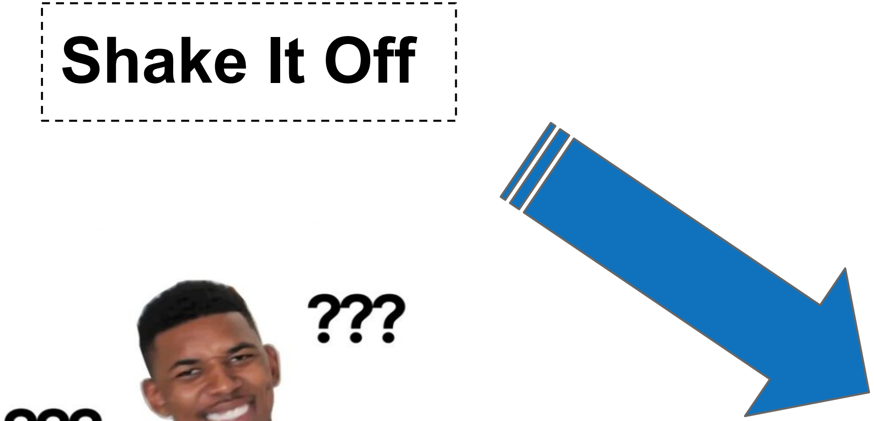
Problem 6A

(a) Describe a graph you could construct to help you solve the problem. At the very least you'll want to mention what the vertices and edges are, and whether the edges are weighted or unweighted and directed or undirected.

Problem 6A

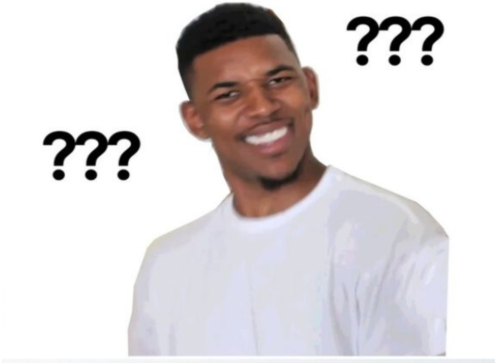
Q: How do we get from Shake It Off to Wildest Dreams while obeying the rule:

Two consecutive songs' tempos must differ by *no more* than 10 beats per minute (BPM)



Shake It Off

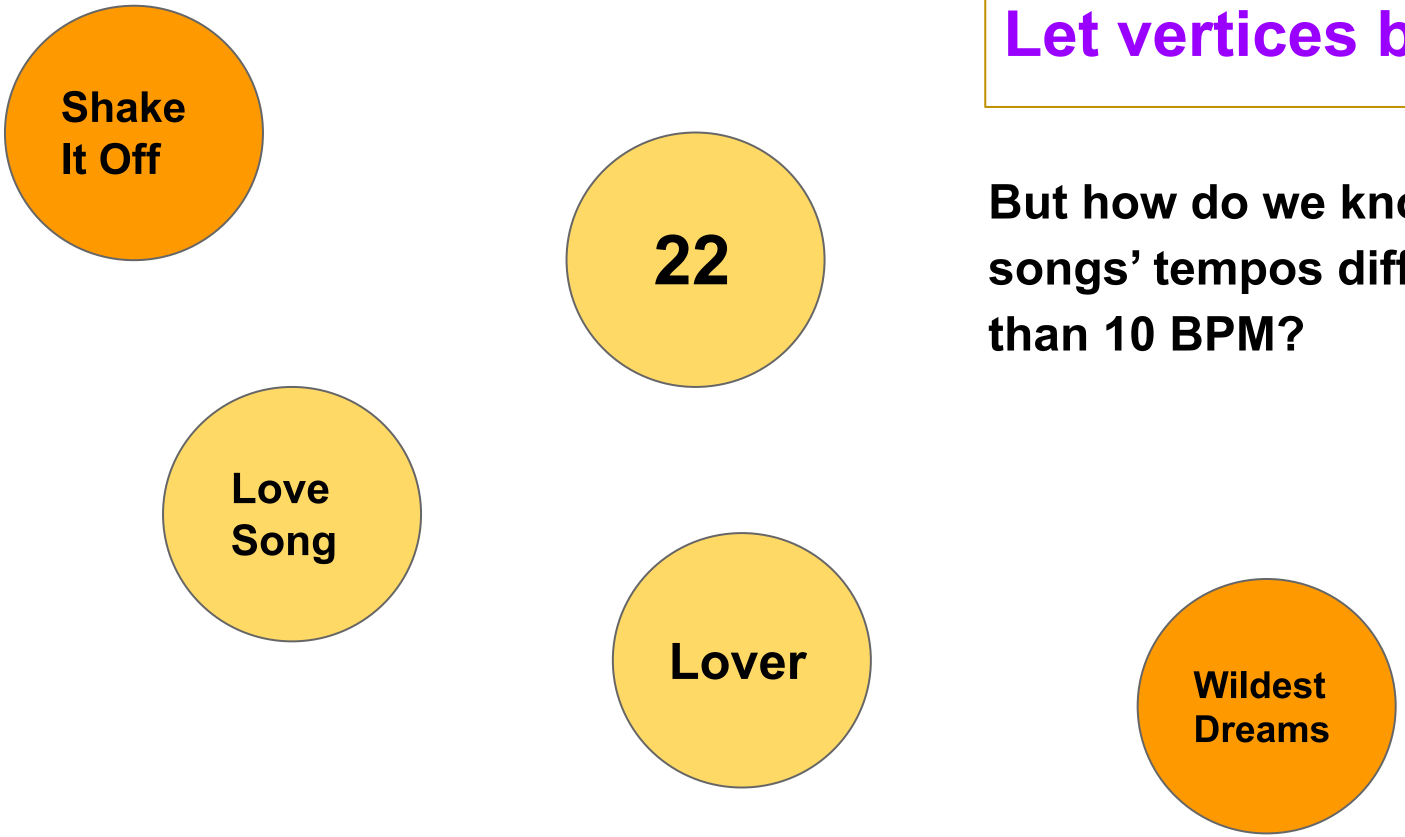
Wildest Dreams



Problem 6A

Let vertices be songs!

But how do we know if two songs’ tempos differ by more than 10 BPM?



Shake It Off

22

Love Song

Lover

Wildest Dreams

Problem 6A

Include BPM in vertices!

Storing multiple pieces of information in Vertex or Edge Objects is often useful.

Shake It Off
150 BPM

22
150 BPM

Love Song
140 BPM

Lover
130 BPM

Wildest Dreams
120 BPM

Problem 6A

Shake It Off
150 BPM

22
150 BPM

Love Song
140 BPM

Lover
130 BPM

Wildest Dreams
120 BPM

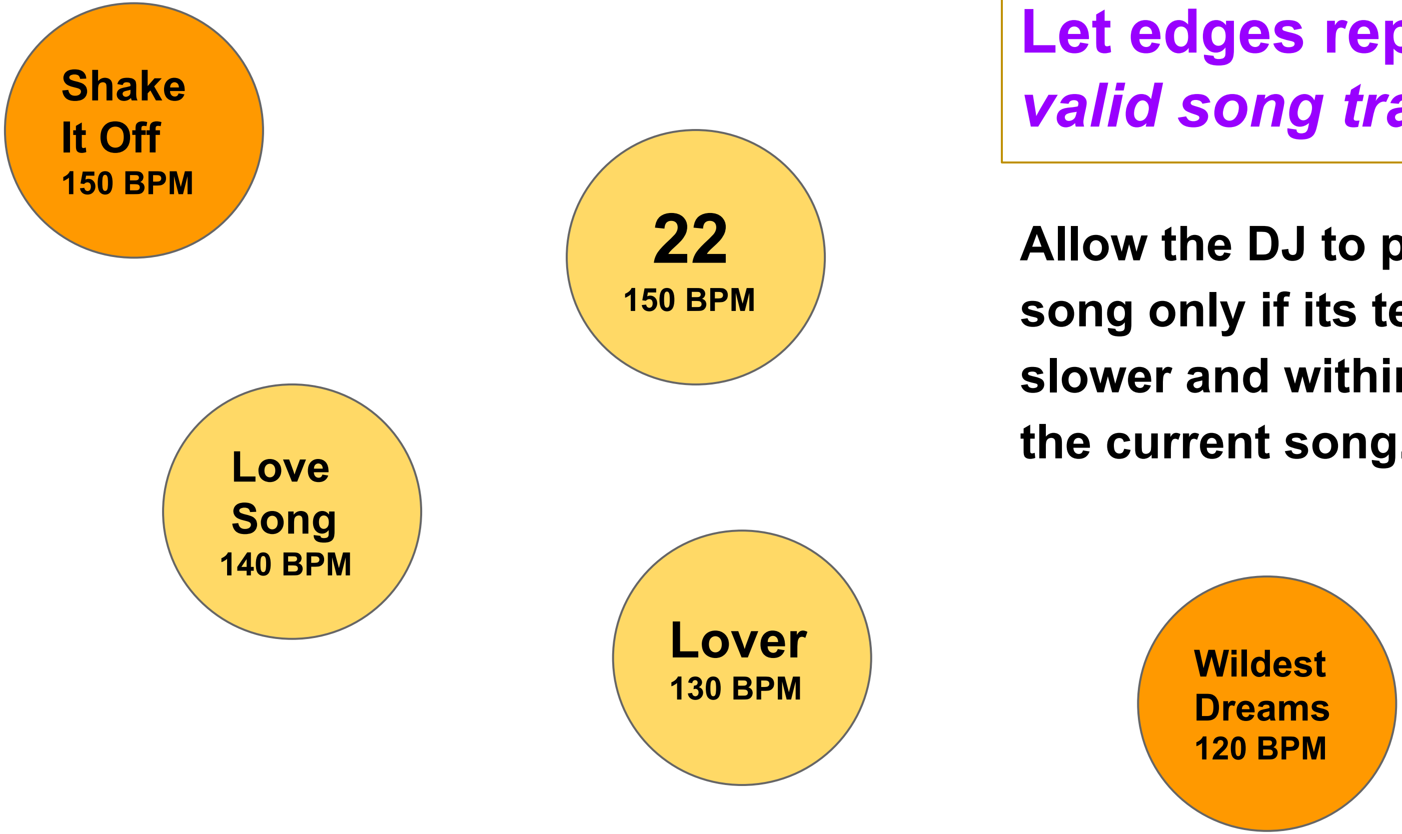
Q: What are our edges?

We know we want to slow down the tempo and create a path between Shake It Off and Wildest Dreams.

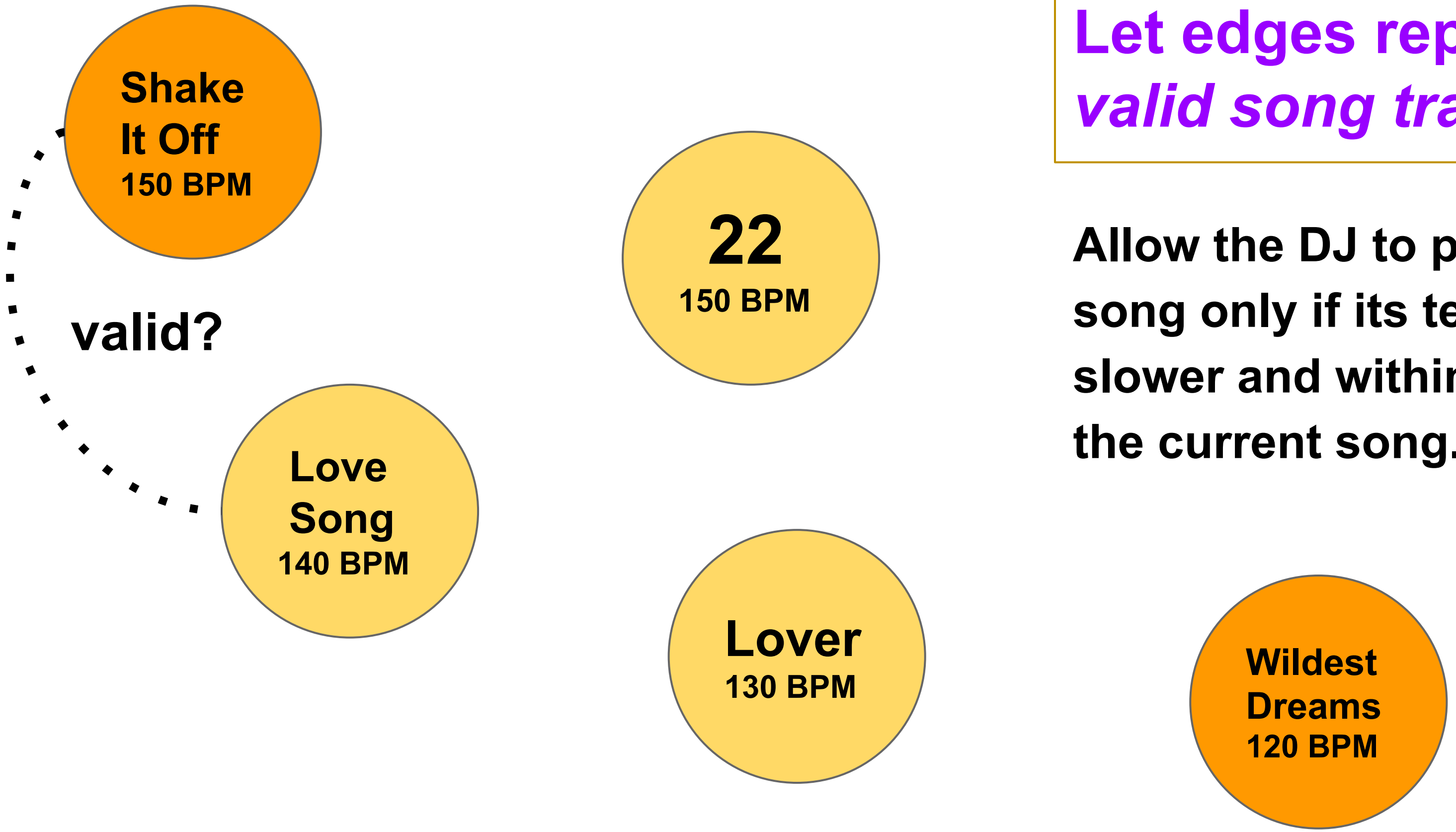
Problem 6A

Let edges represent *valid song transitions*!

Allow the DJ to play the next song only if its tempo is slower and within 10 BPM of the current song.



Problem 6A

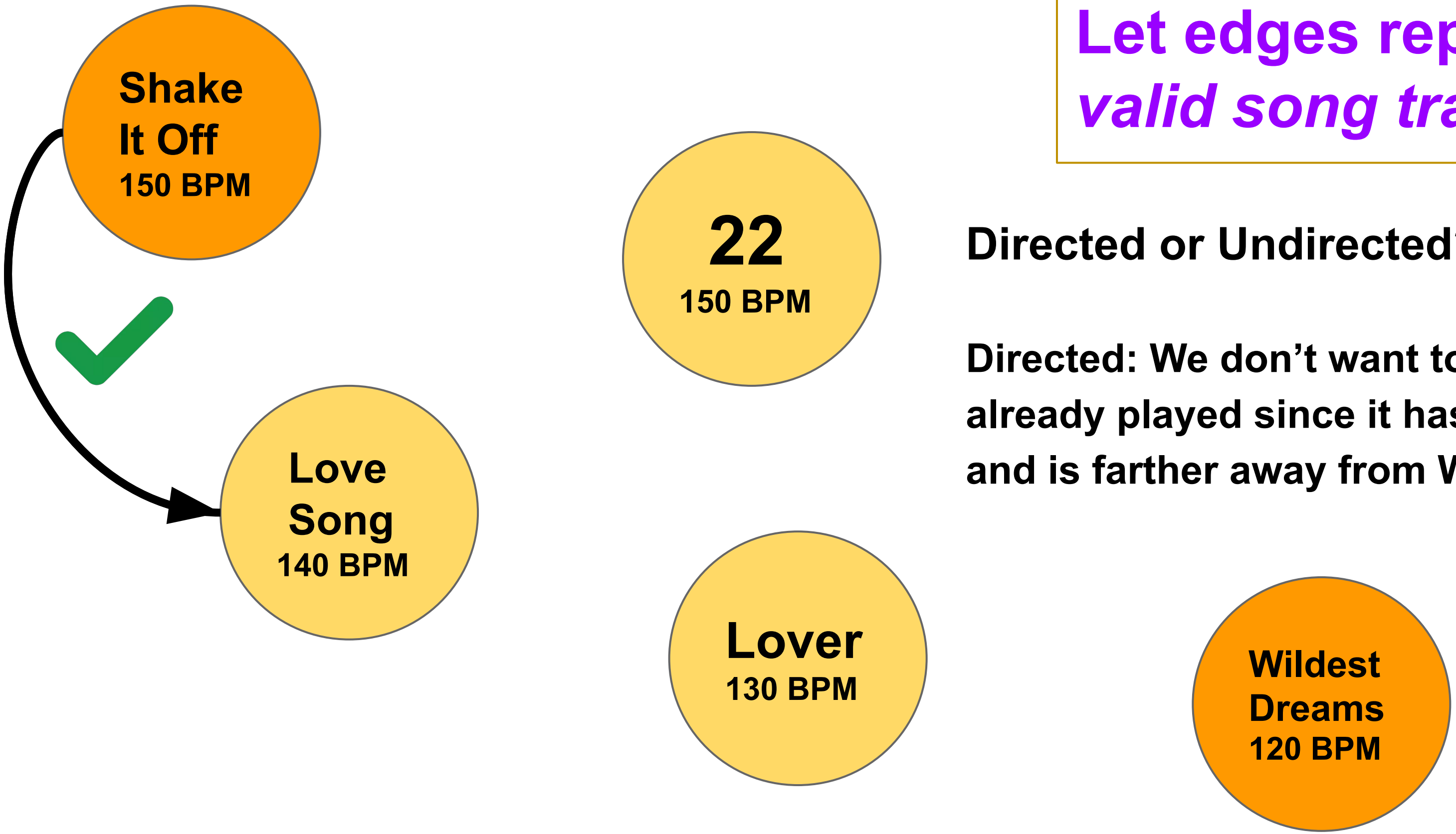


Let edges represent *valid song transitions*!

Allow the DJ to play the next song only if its tempo is slower and within 10 BPM of the current song.

Problem 6A

Let edges represent *valid song transitions*!



Shake It Off
150 BPM

22
150 BPM

Love Song
140 BPM

Lover
130 BPM

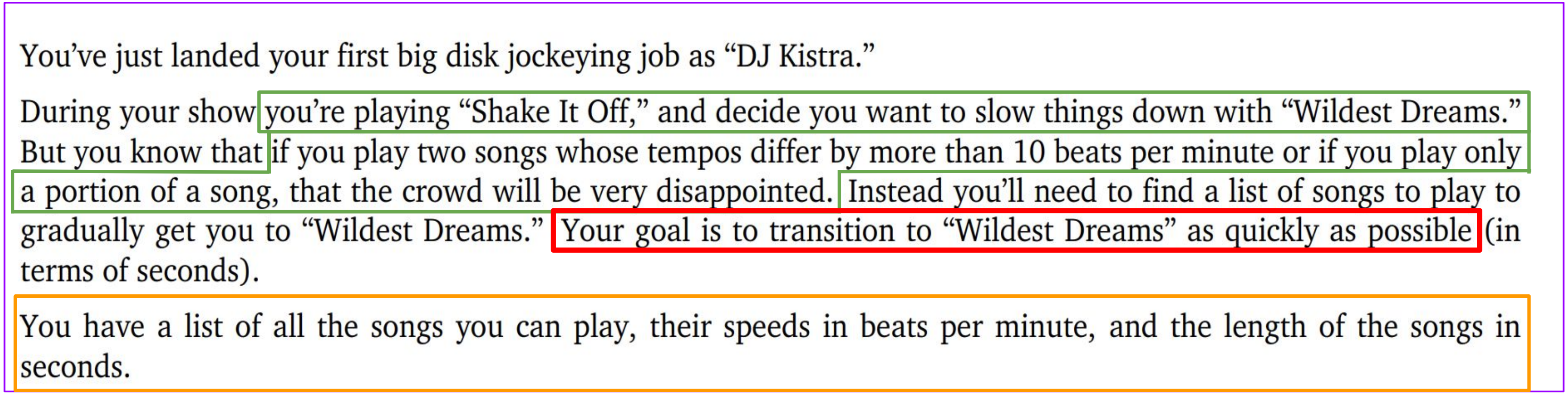
Wildest Dreams
120 BPM

Directed or Undirected?

Directed: We don’t want to play a song we already played since it has a faster tempo and is farther away from Wildest Dreams.

Problem 6A

Can we accomplish the task with the graph model we've built? Let's check.



You've just landed your first big disk jockeying job as "DJ Kistra."

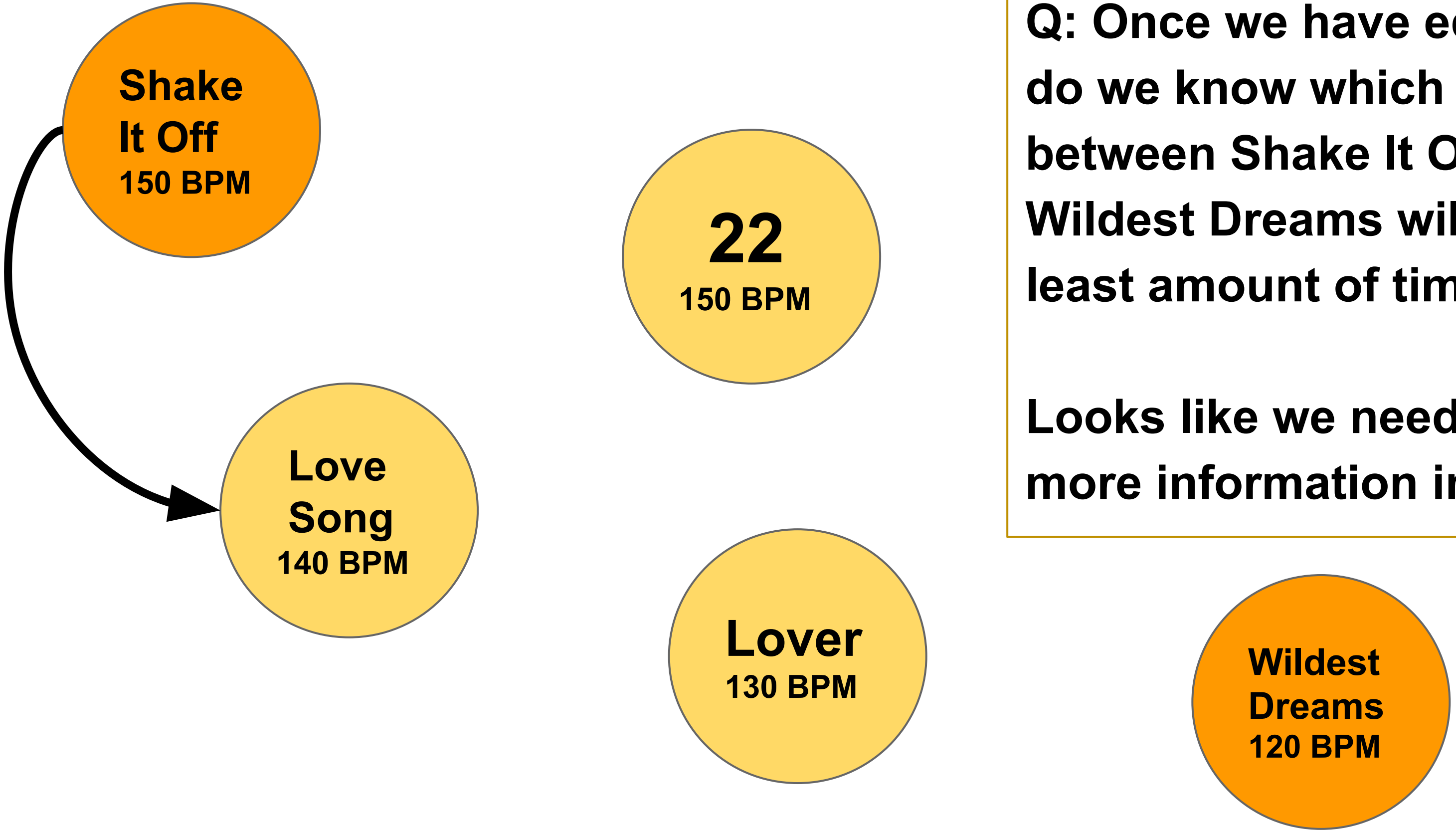
During your show you're playing "Shake It Off," and decide you want to slow things down with "Wildest Dreams." But you know that if you play two songs whose tempos differ by more than 10 beats per minute or if you play only a portion of a song, that the crowd will be very disappointed. Instead you'll need to find a list of songs to play to gradually get you to "Wildest Dreams." Your goal is to transition to "Wildest Dreams" as quickly as possible (in terms of seconds).

You have a list of all the songs you can play, their speeds in beats per minute, and the length of the songs in seconds.

We don't have a way to prioritize between different possible song transitions!

There's more information we haven't used. Does the length of the songs help us?

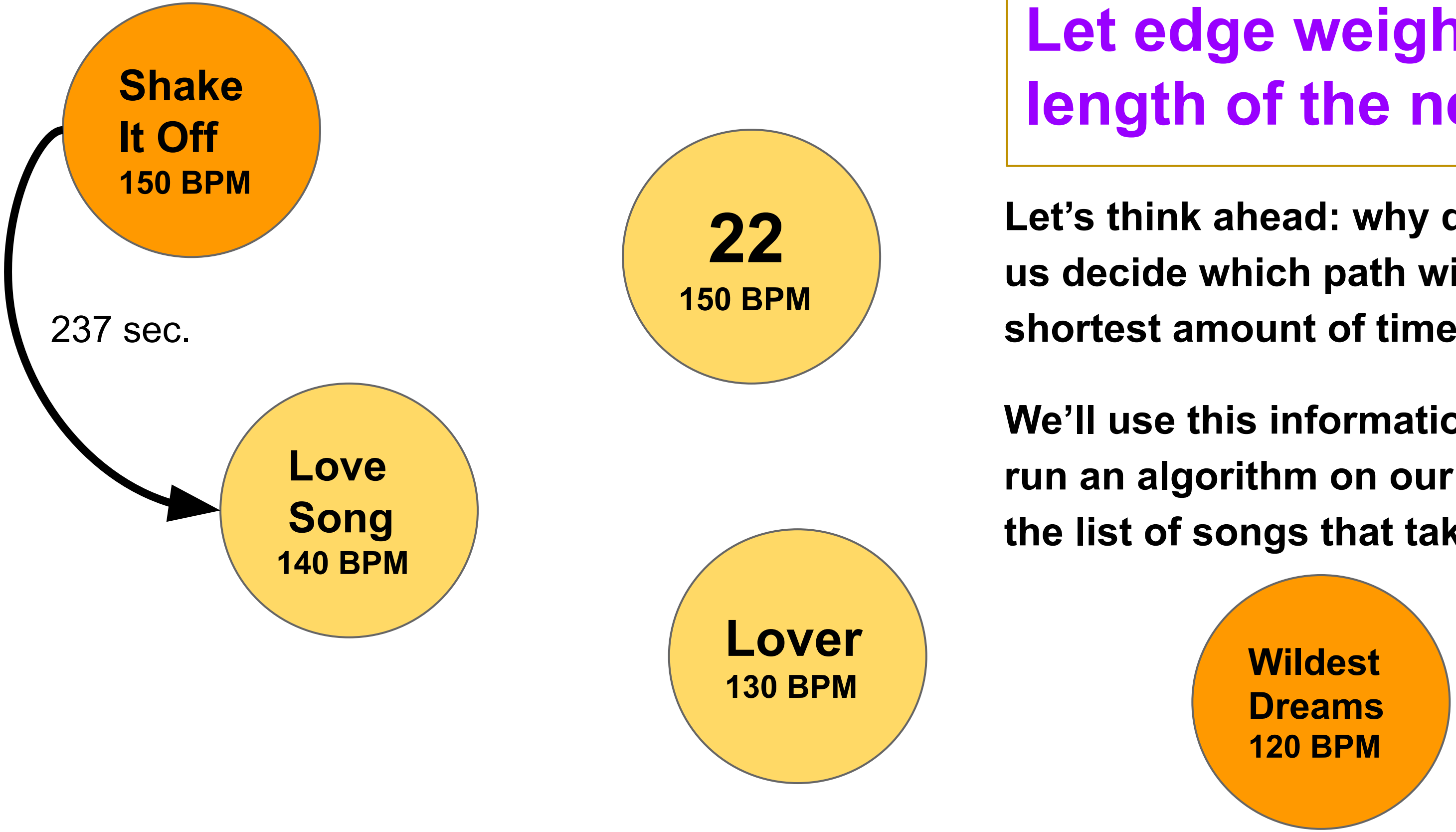
Problem 6A



Q: Once we have edges, how do we know which path between Shake It Off and Wildest Dreams will take the least amount of time?

Looks like we need to encode more information in our graph.

Problem 6A



Let edge weights be the length of the next song!

Let's think ahead: why does this help us decide which path will take the shortest amount of time?

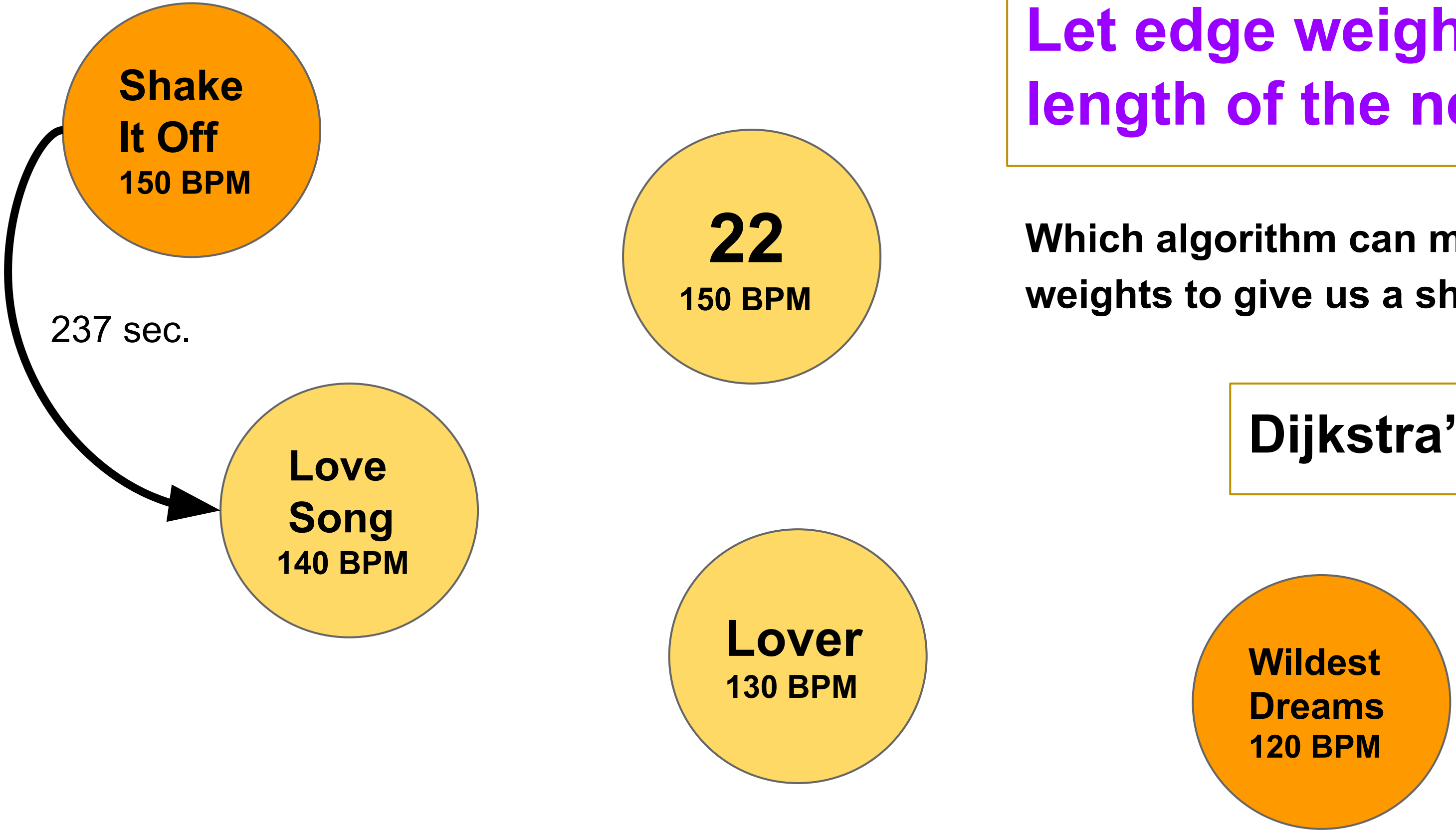
We'll use this information later when we run an algorithm on our graph to find the list of songs that take the least time.

Problem 6A

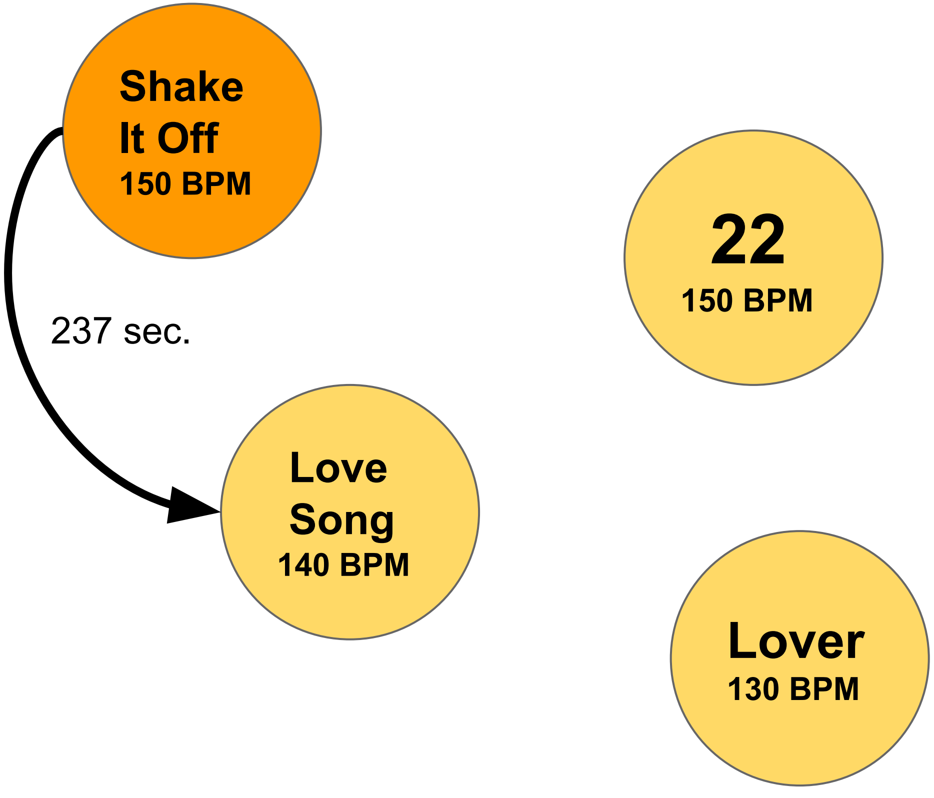
Let edge weights be the length of the next song!

Which algorithm can make use of edge weights to give us a shortest path?

Dijkstra's!



Problem 6A



Vertices: song and BPM

Edges: valid song transitions

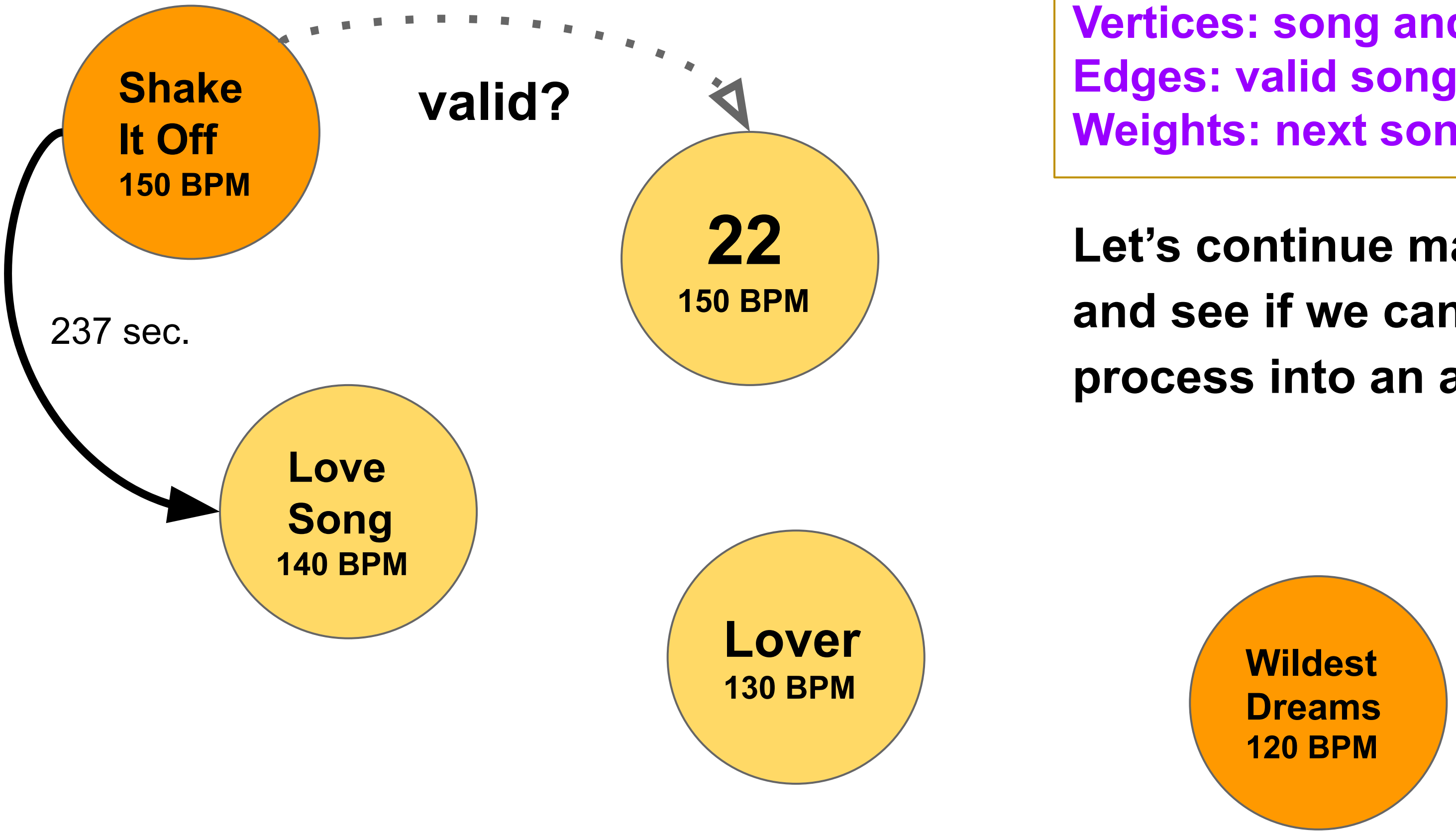
Weights: next song length

Now our graph model has everything it needs to find the shortest path between Shake It Off and Wildest Dreams!

Problem 6B

(b) Describe an algorithm to construct your graph from the previous part. You may assume your songs are stored in whatever data structure makes this part easiest. Assume you have access to a method `makeEdge(v1, v2, w)` which creates an edge from `v1` to `v2` of weight `w`.

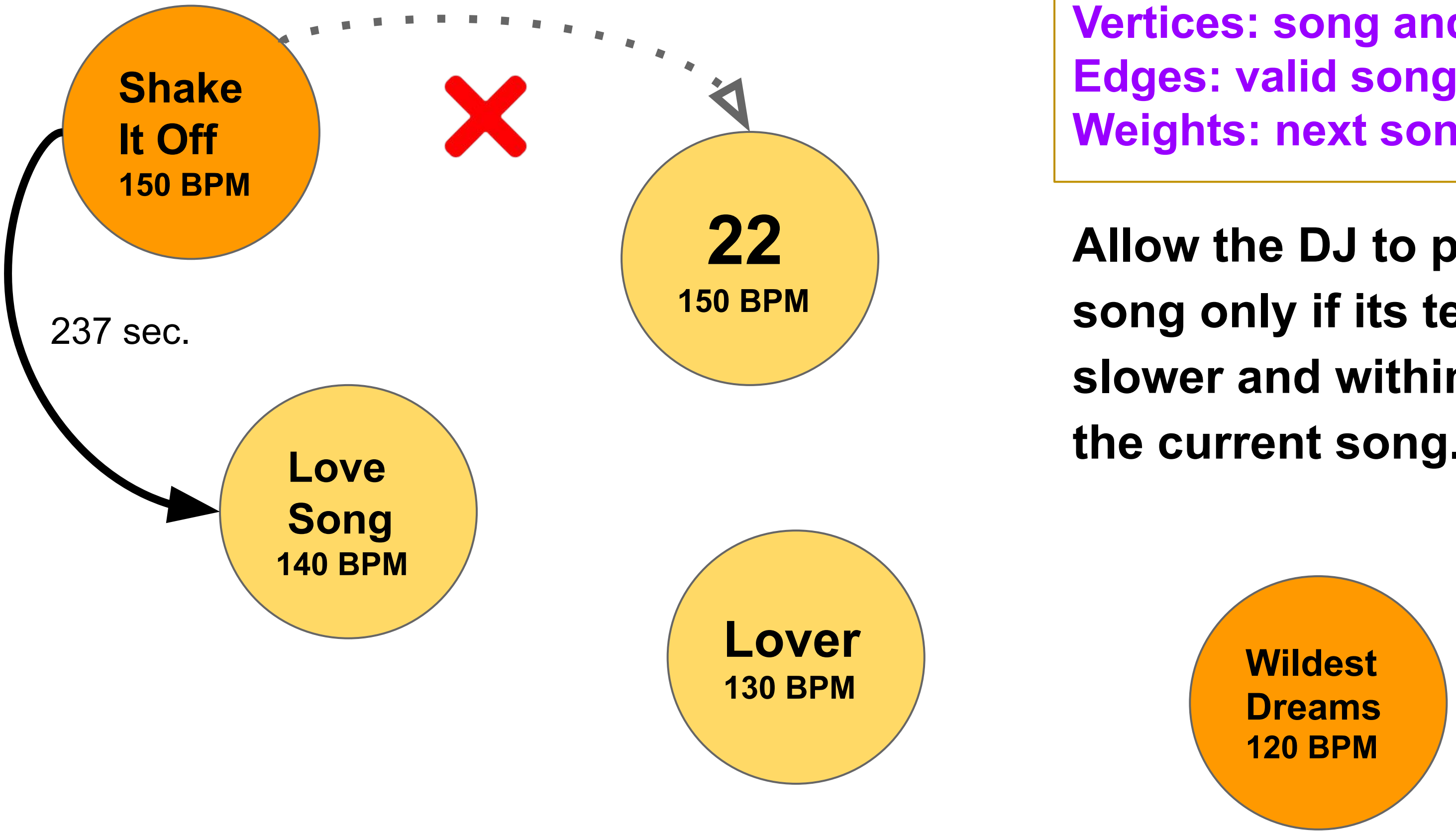
Problem 6B



Vertices: song and BPM
Edges: valid song transitions
Weights: next song length

Let's continue making edges and see if we can turn the process into an algorithm.

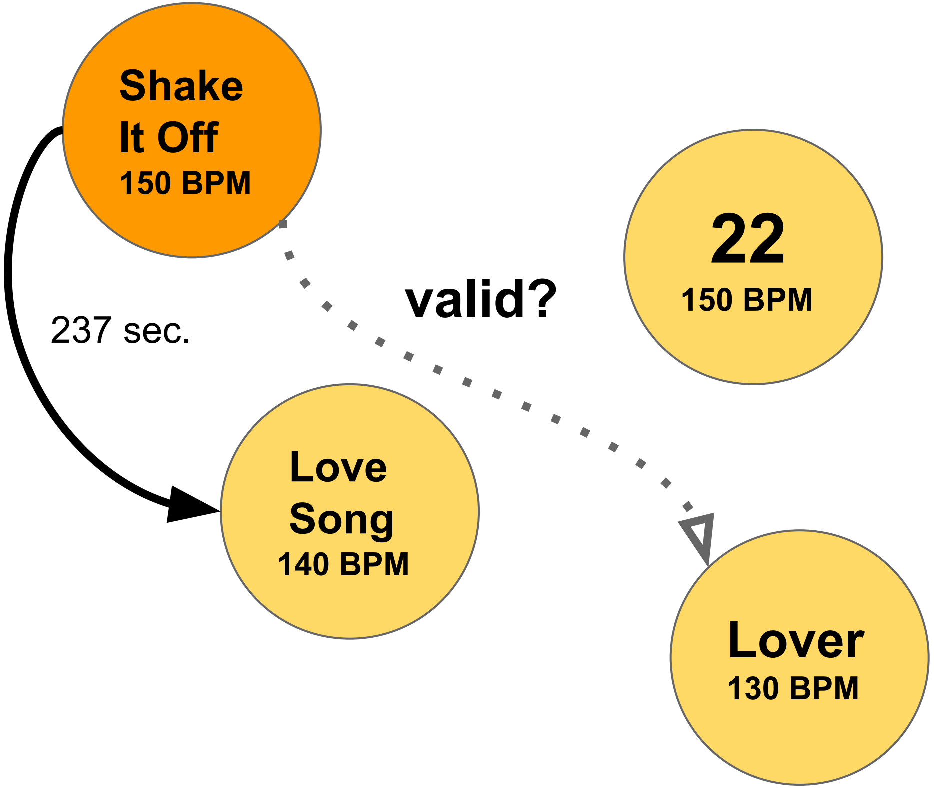
Problem 6B



Vertices: song and BPM
Edges: valid song transitions
Weights: next song length

Allow the DJ to play the next song only if its tempo is slower and within 10 BPM of the current song.

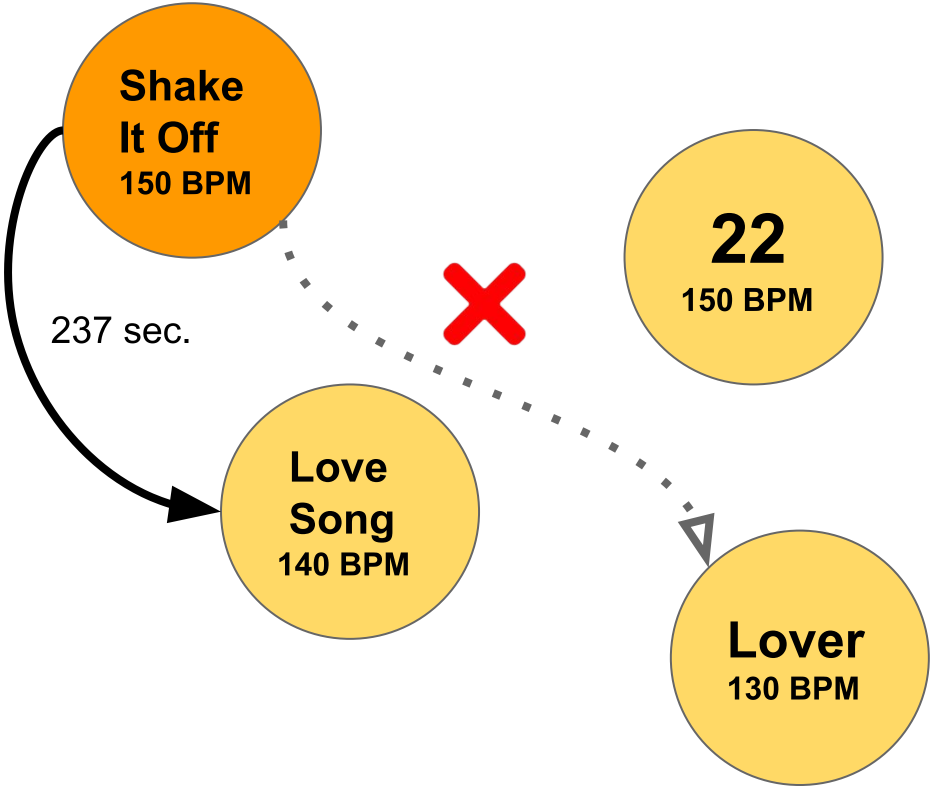
Problem 6B



Vertices: song and BPM
Edges: valid song transitions
Weights: next song length

Allow the DJ to play the next song only if its tempo is slower and within 10 BPM of the current song.

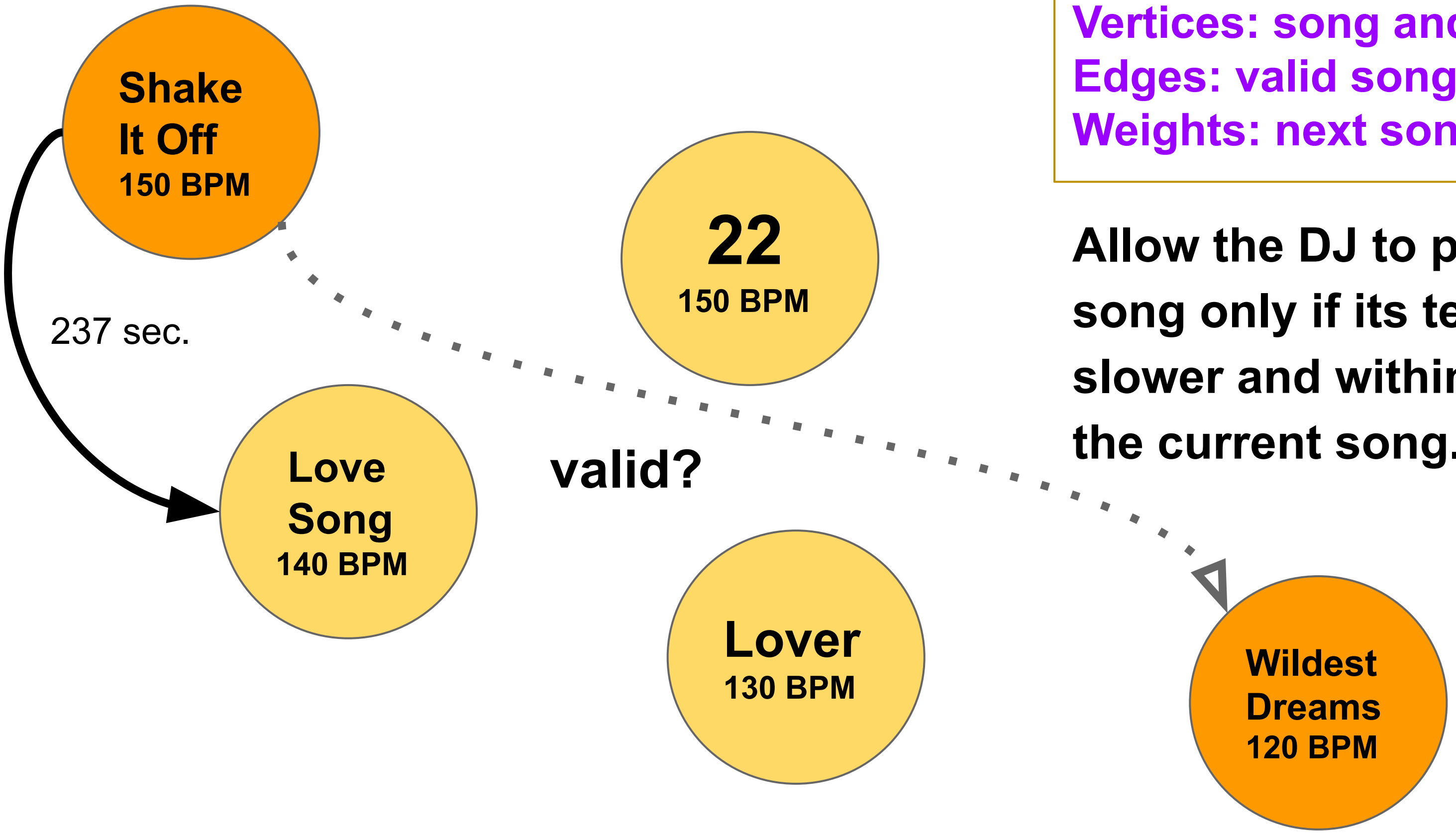
Problem 6B



Vertices: song and BPM
Edges: valid song transitions
Weights: next song length

Allow the DJ to play the next song only if its tempo is slower and within 10 BPM of the current song.

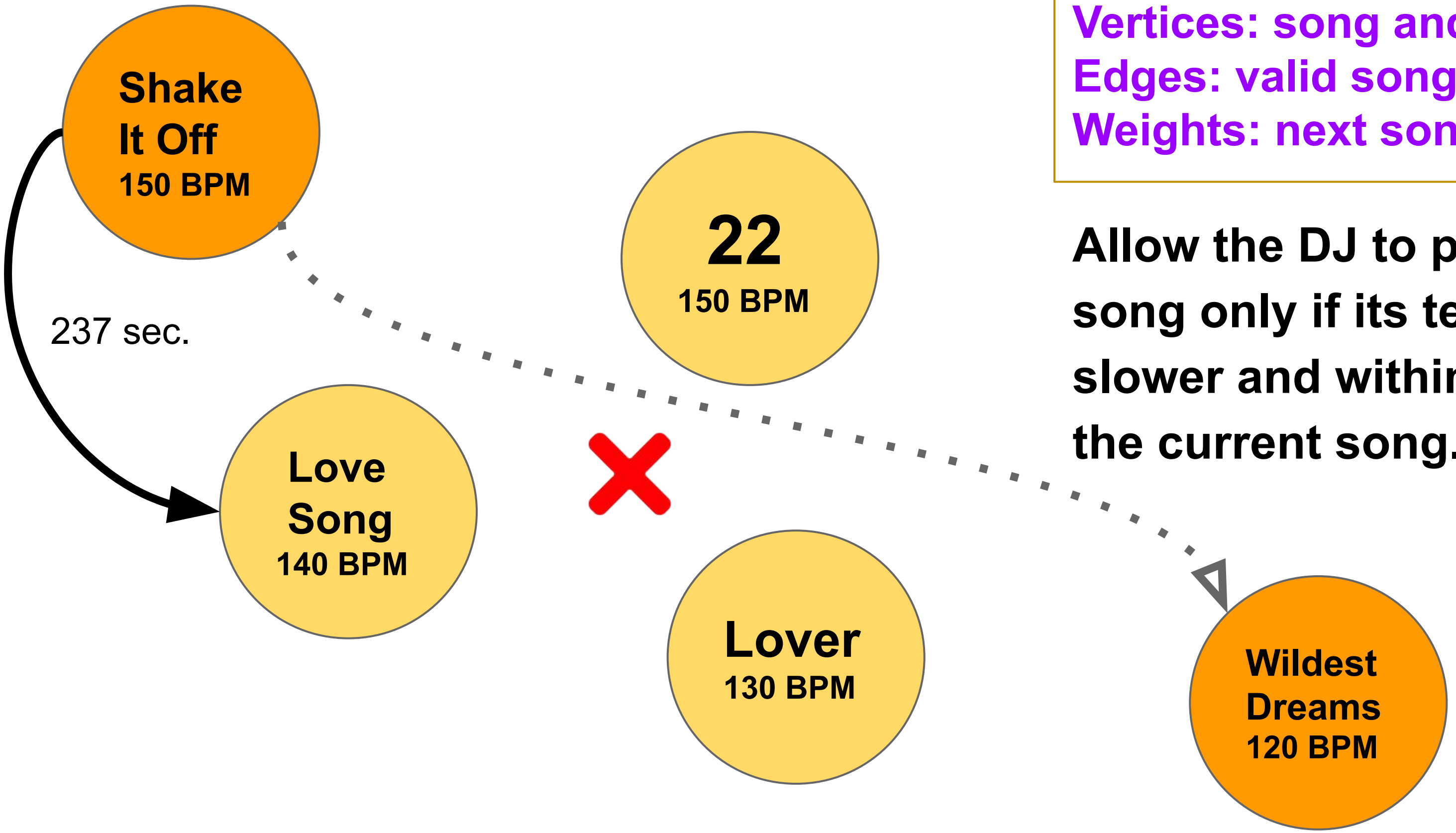
Problem 6B



Vertices: song and BPM
Edges: valid song transitions
Weights: next song length

Allow the DJ to play the next song only if its tempo is slower and within 10 BPM of the current song.

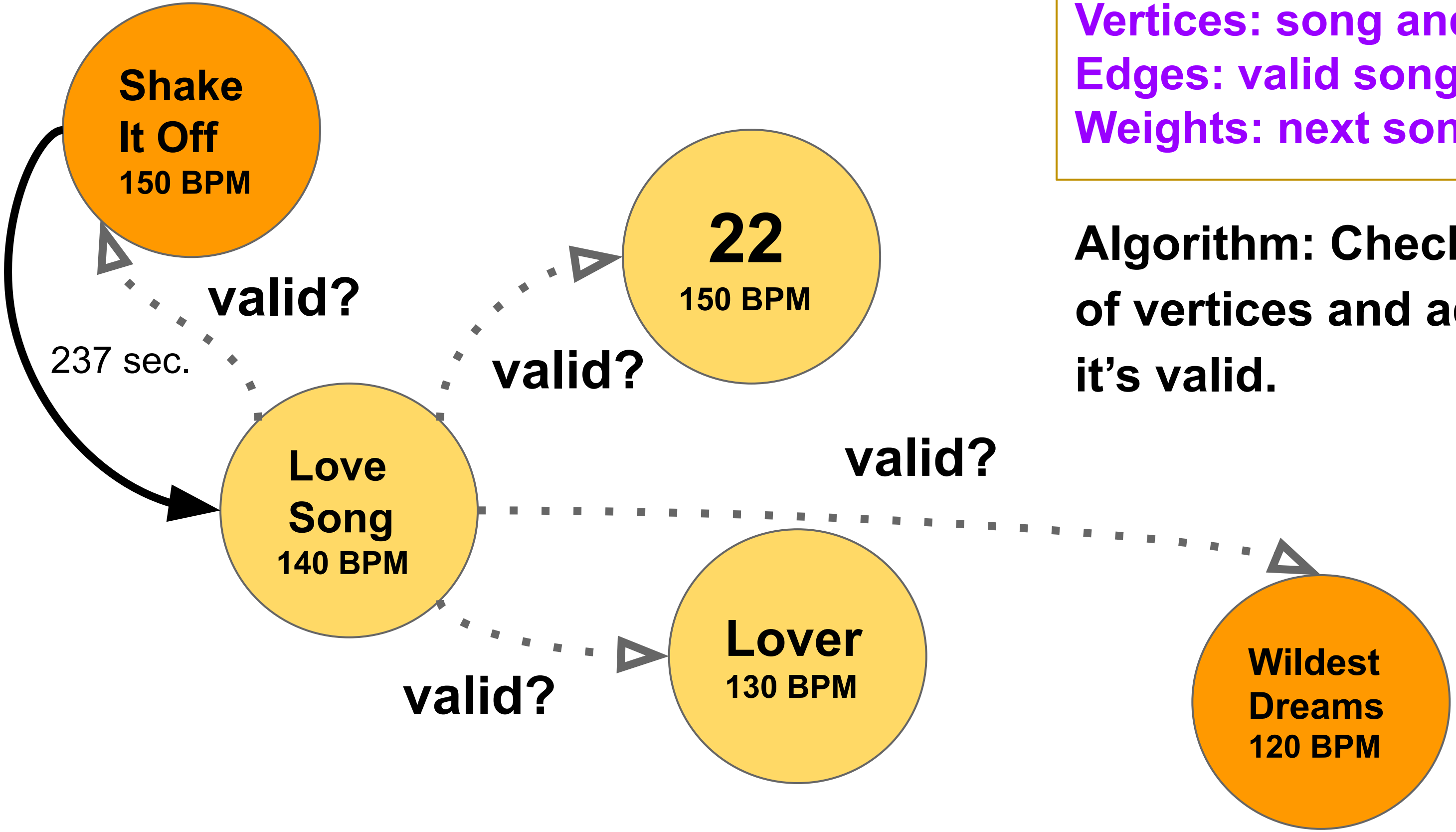
Problem 6B



Vertices: song and BPM
Edges: valid song transitions
Weights: next song length

Allow the DJ to play the next song only if its tempo is slower and within 10 BPM of the current song.

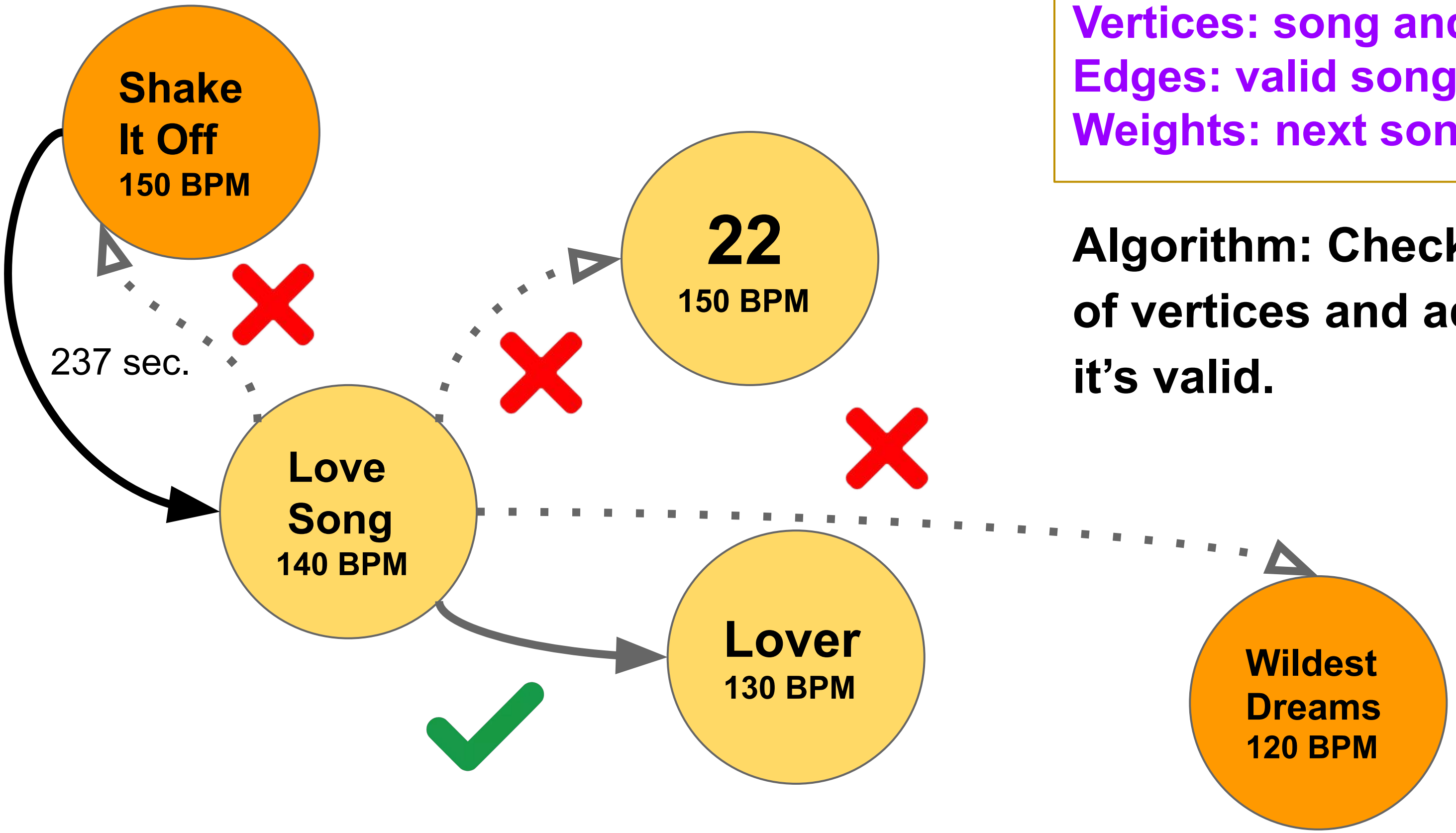
Problem 6B



Vertices: song and BPM
Edges: valid song transitions
Weights: next song length

Algorithm: Check every pair of vertices and add an edge if it's valid.

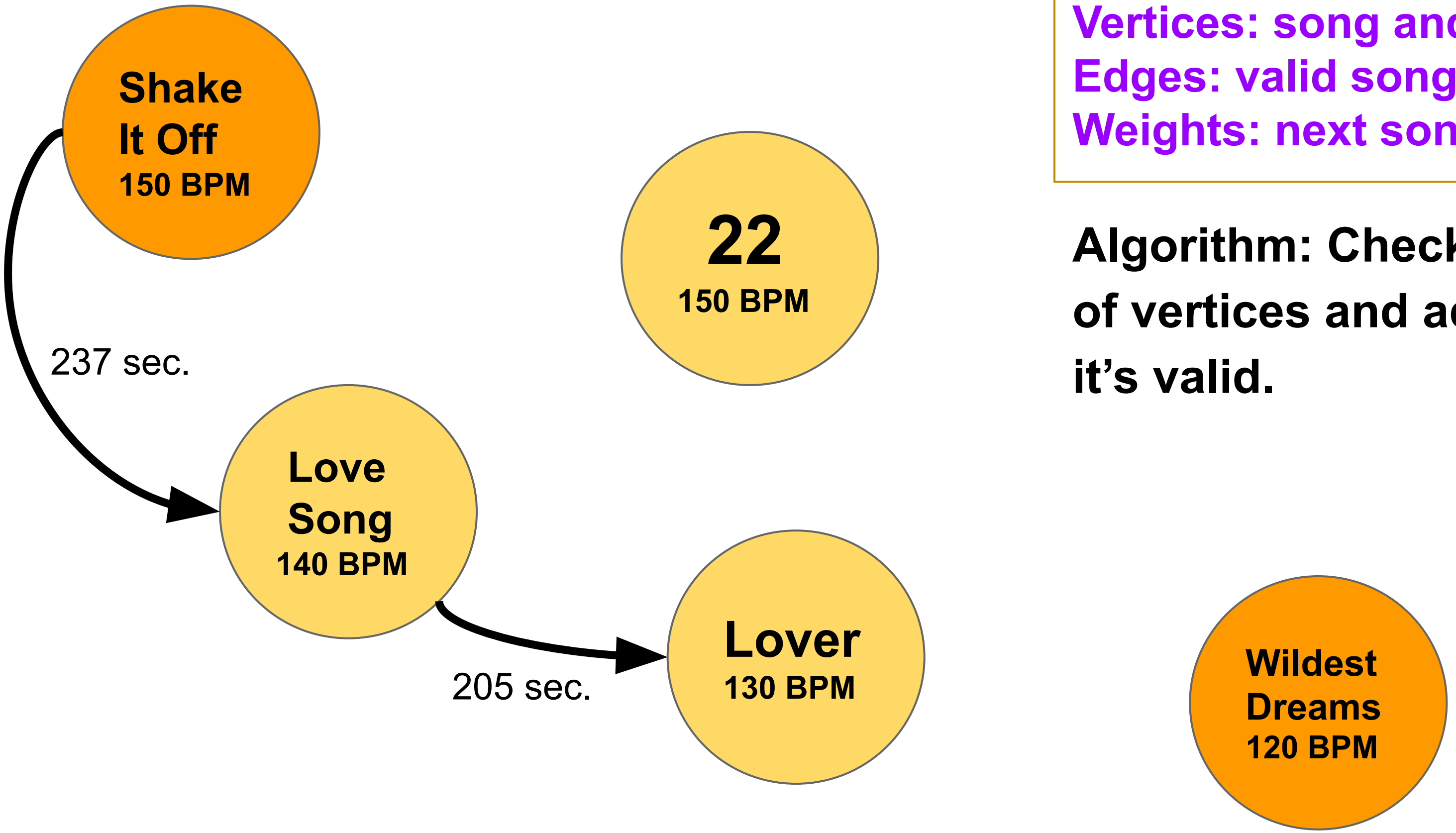
Problem 6B



Vertices: song and BPM
Edges: valid song transitions
Weights: next song length

Algorithm: Check every pair of vertices and add an edge if it's valid.

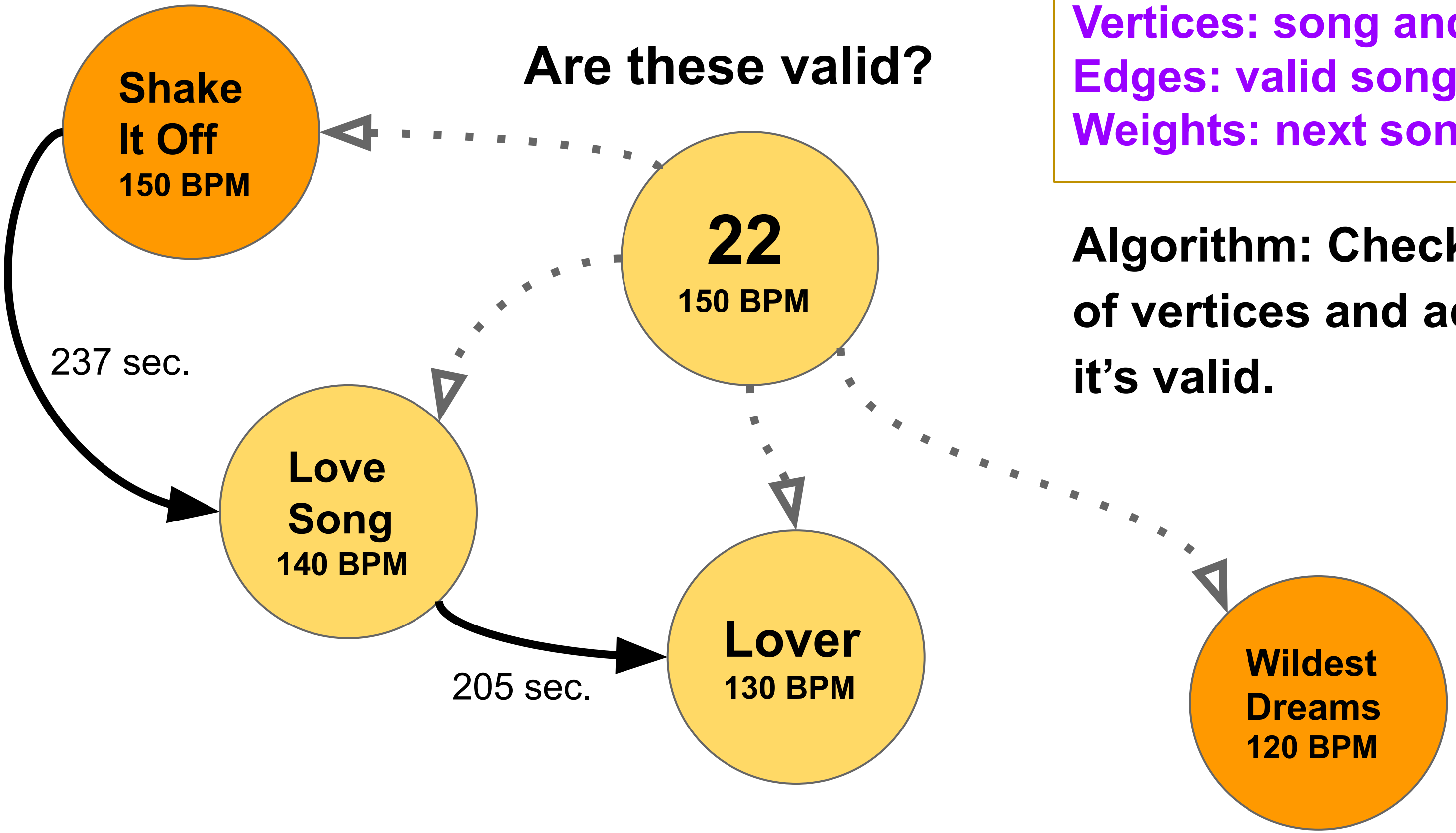
Problem 6B



Vertices: song and BPM
Edges: valid song transitions
Weights: next song length

Algorithm: Check every pair of vertices and add an edge if it's valid.

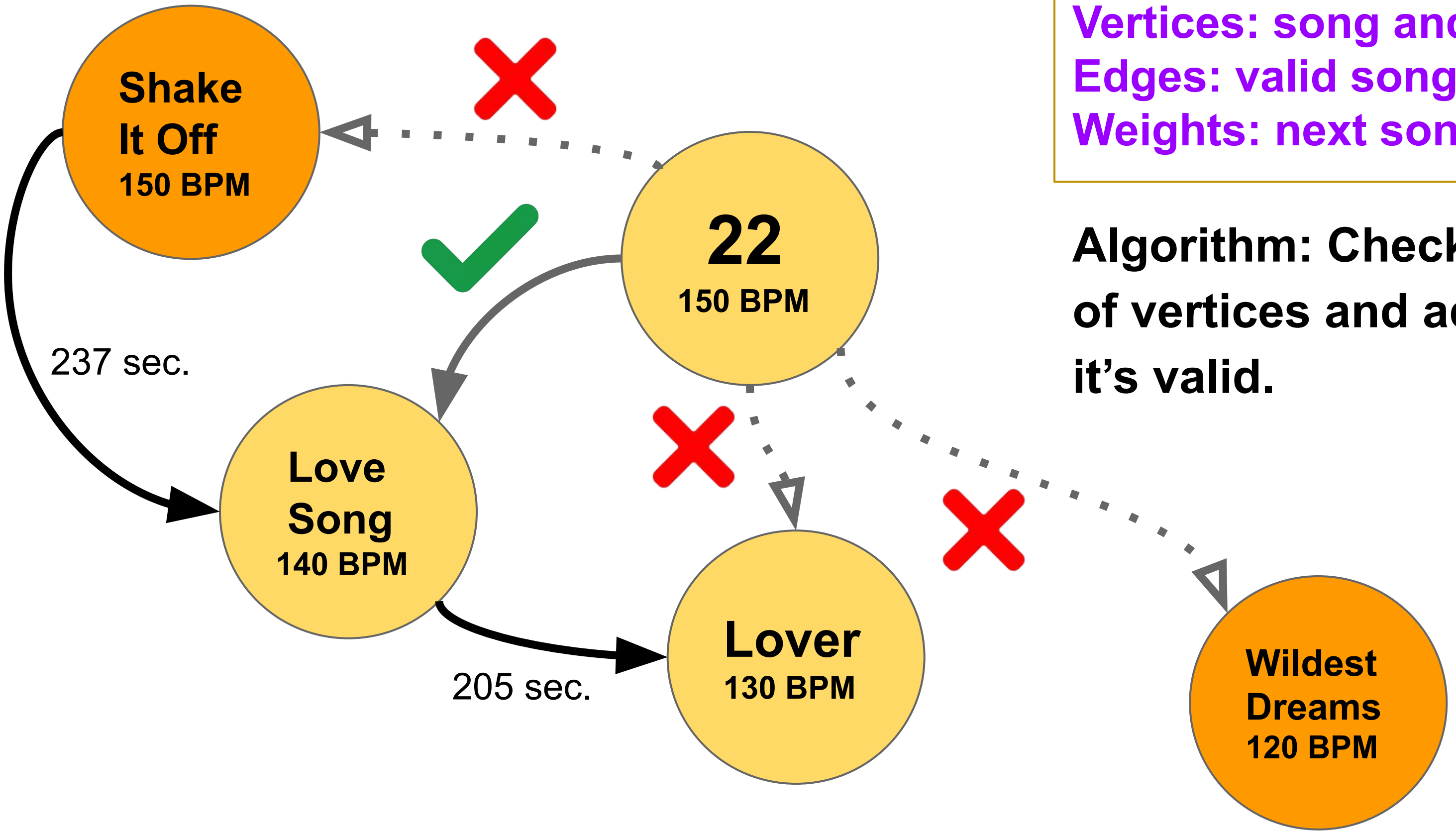
Problem 6B



Vertices: song and BPM
Edges: valid song transitions
Weights: next song length

Algorithm: Check every pair of vertices and add an edge if it's valid.

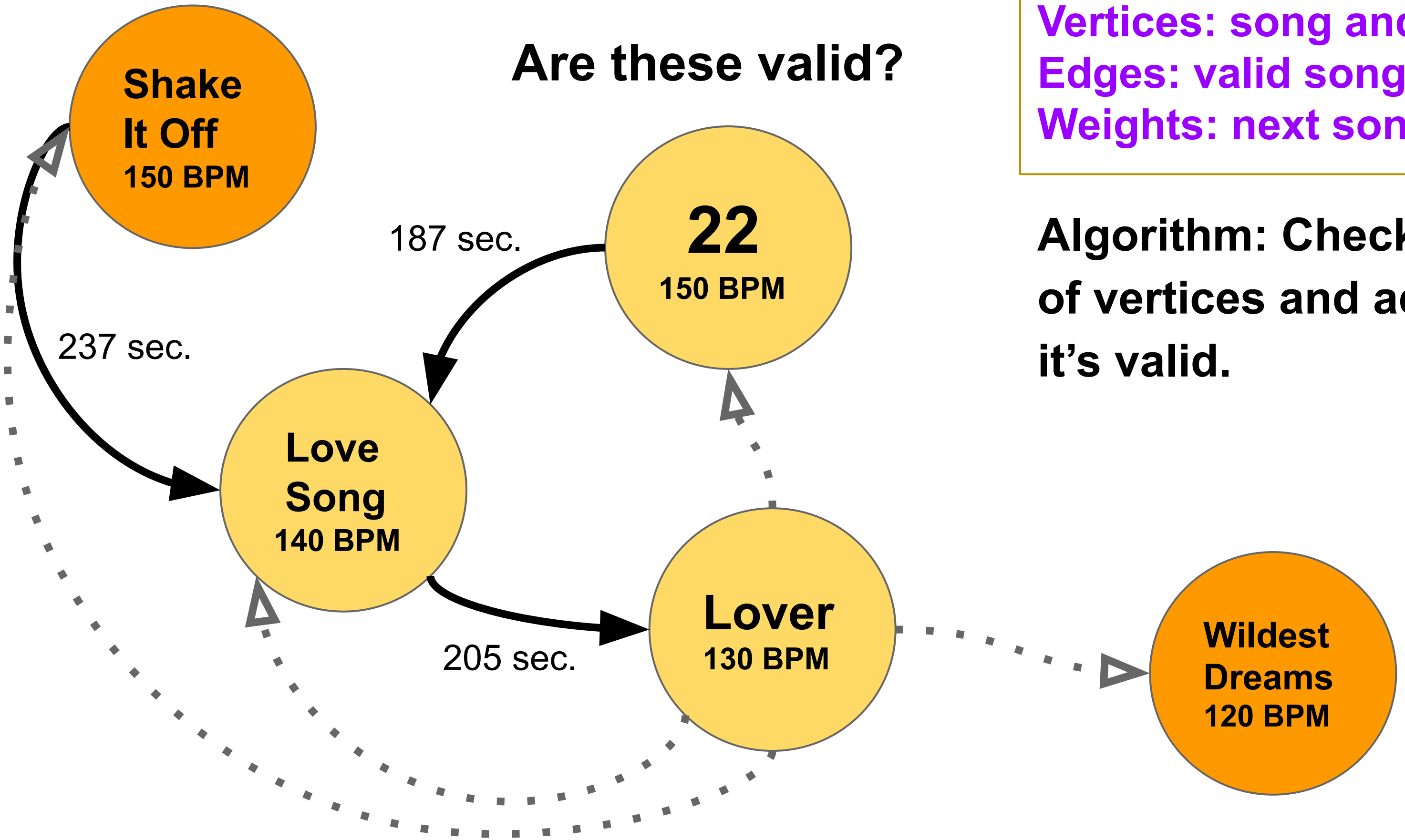
Problem 6B



Vertices: song and BPM
Edges: valid song transitions
Weights: next song length

Algorithm: Check every pair of vertices and add an edge if it's valid.

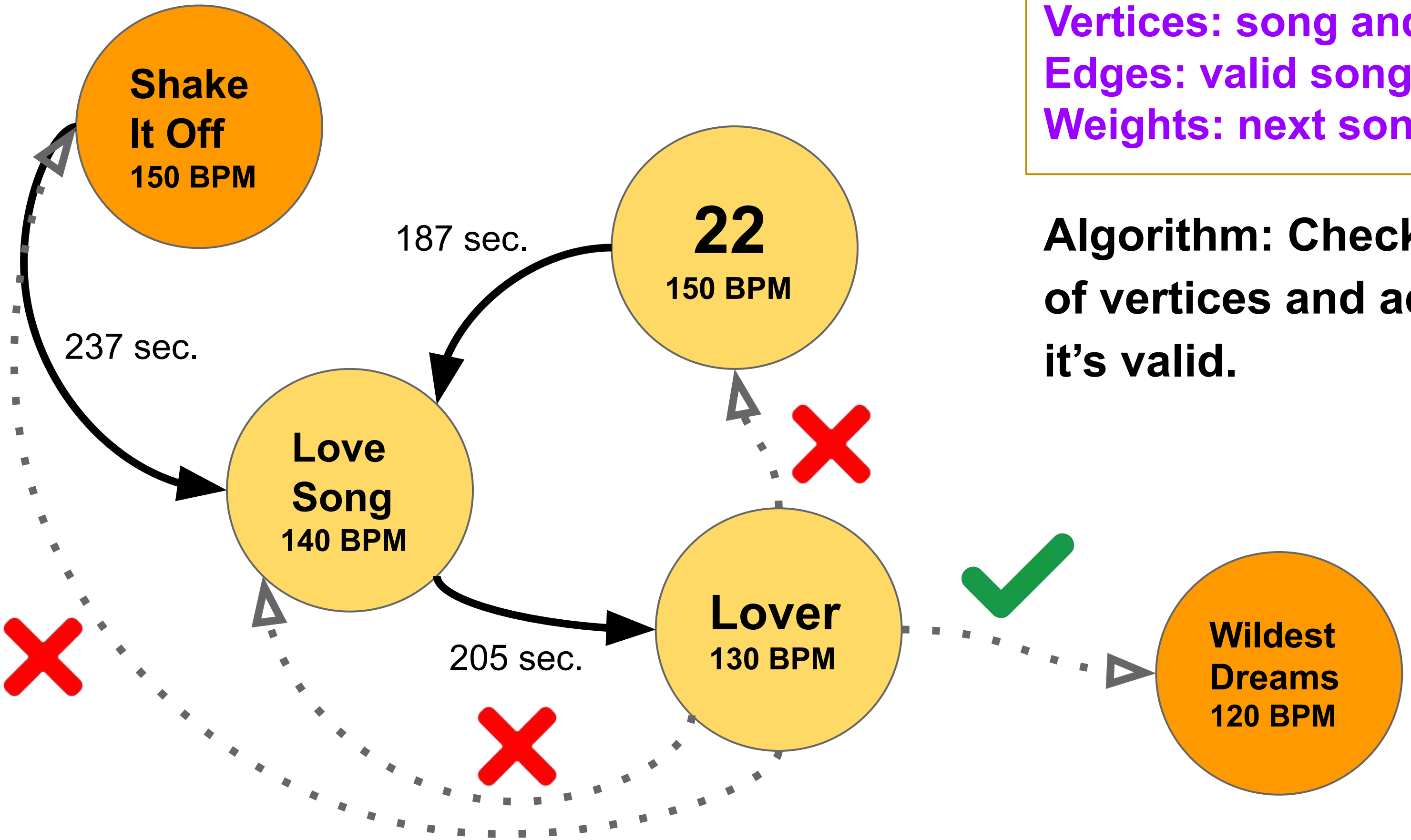
Problem 6B



Vertices: song and BPM
Edges: valid song transitions
Weights: next song length

Algorithm: Check every pair of vertices and add an edge if it's valid.

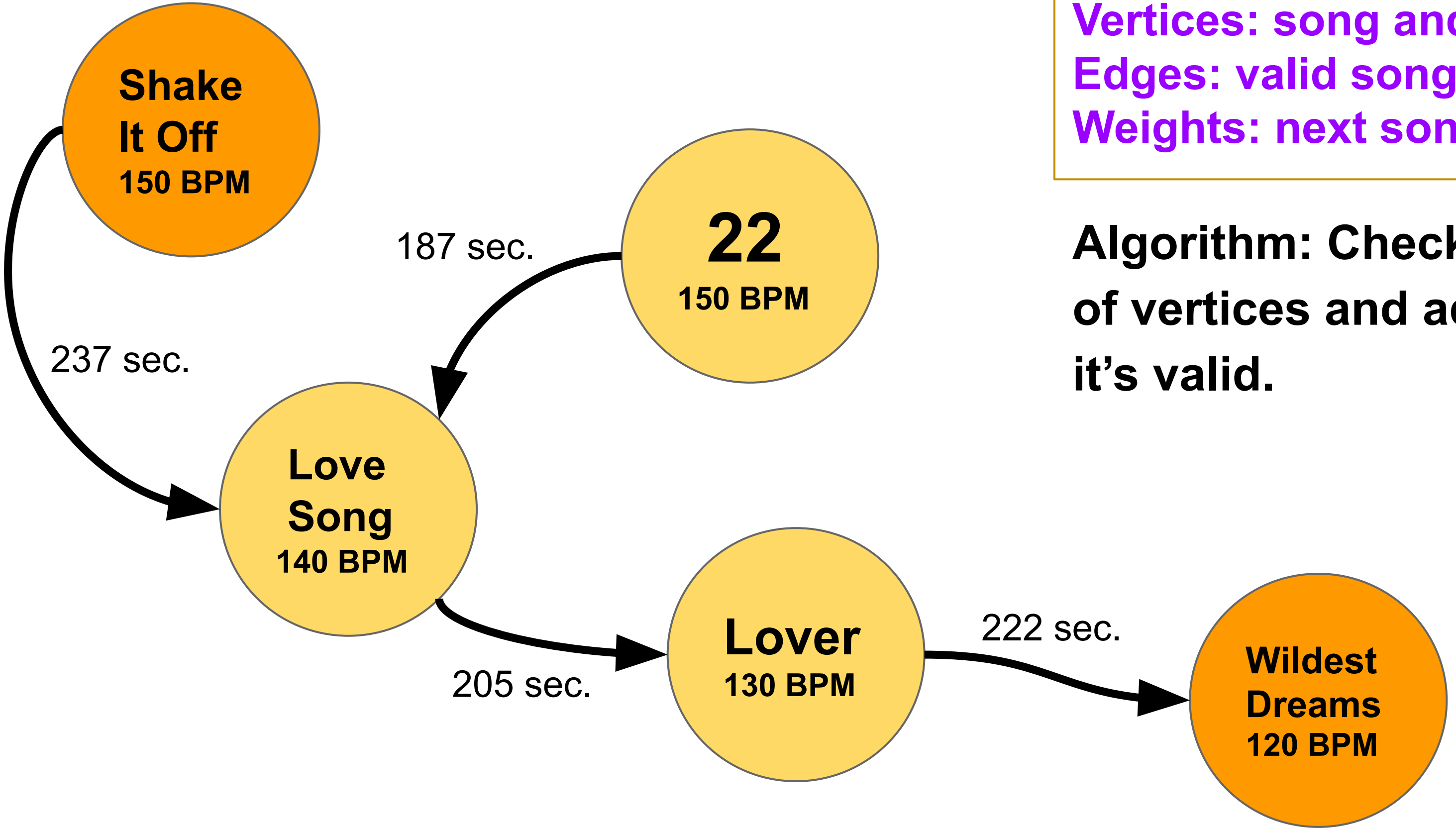
Problem 6B



Vertices: song and BPM
Edges: valid song transitions
Weights: next song length

Algorithm: Check every pair of vertices and add an edge if it's valid.

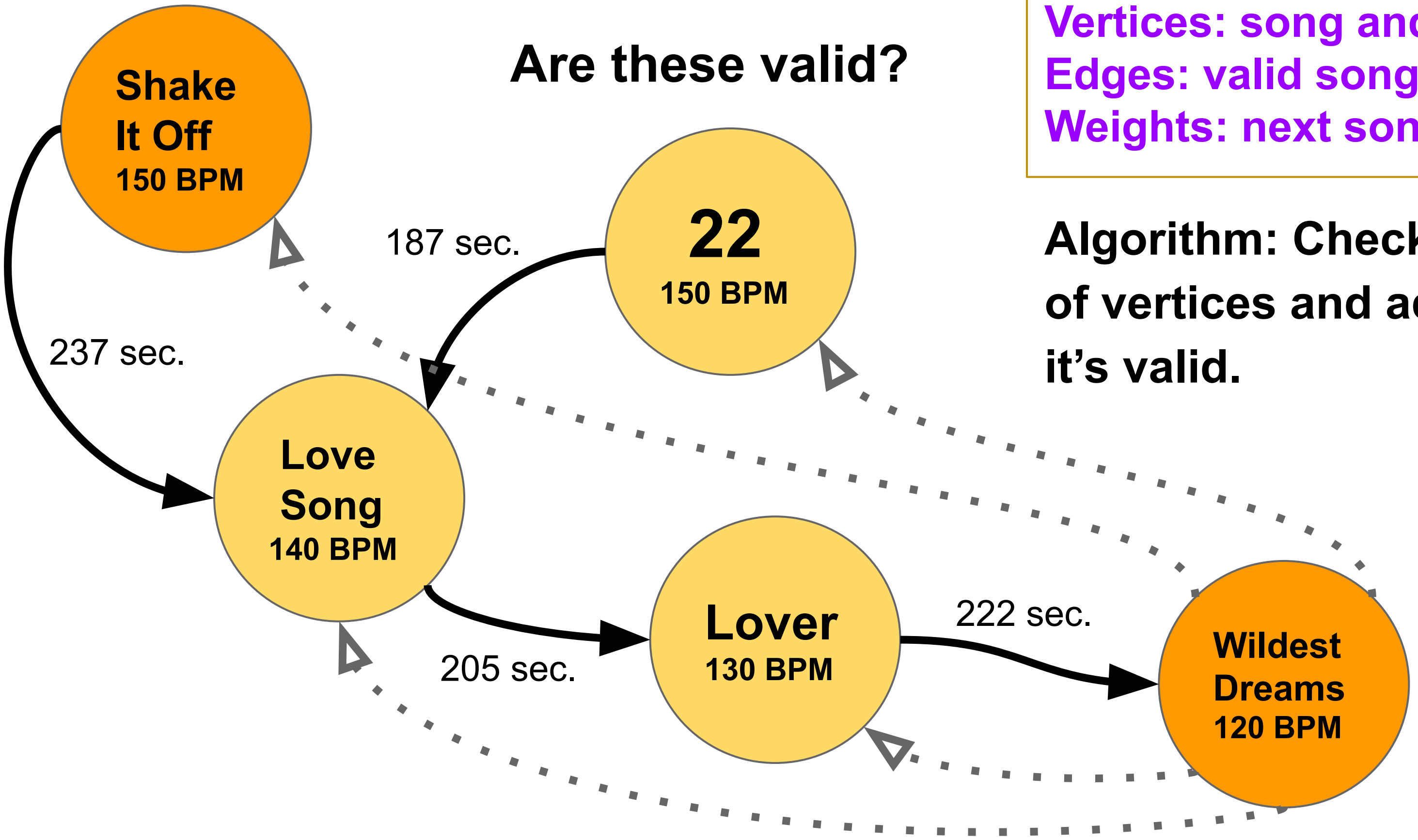
Problem 6B



Vertices: song and BPM
Edges: valid song transitions
Weights: next song length

Algorithm: Check every pair of vertices and add an edge if it's valid.

Problem 6B

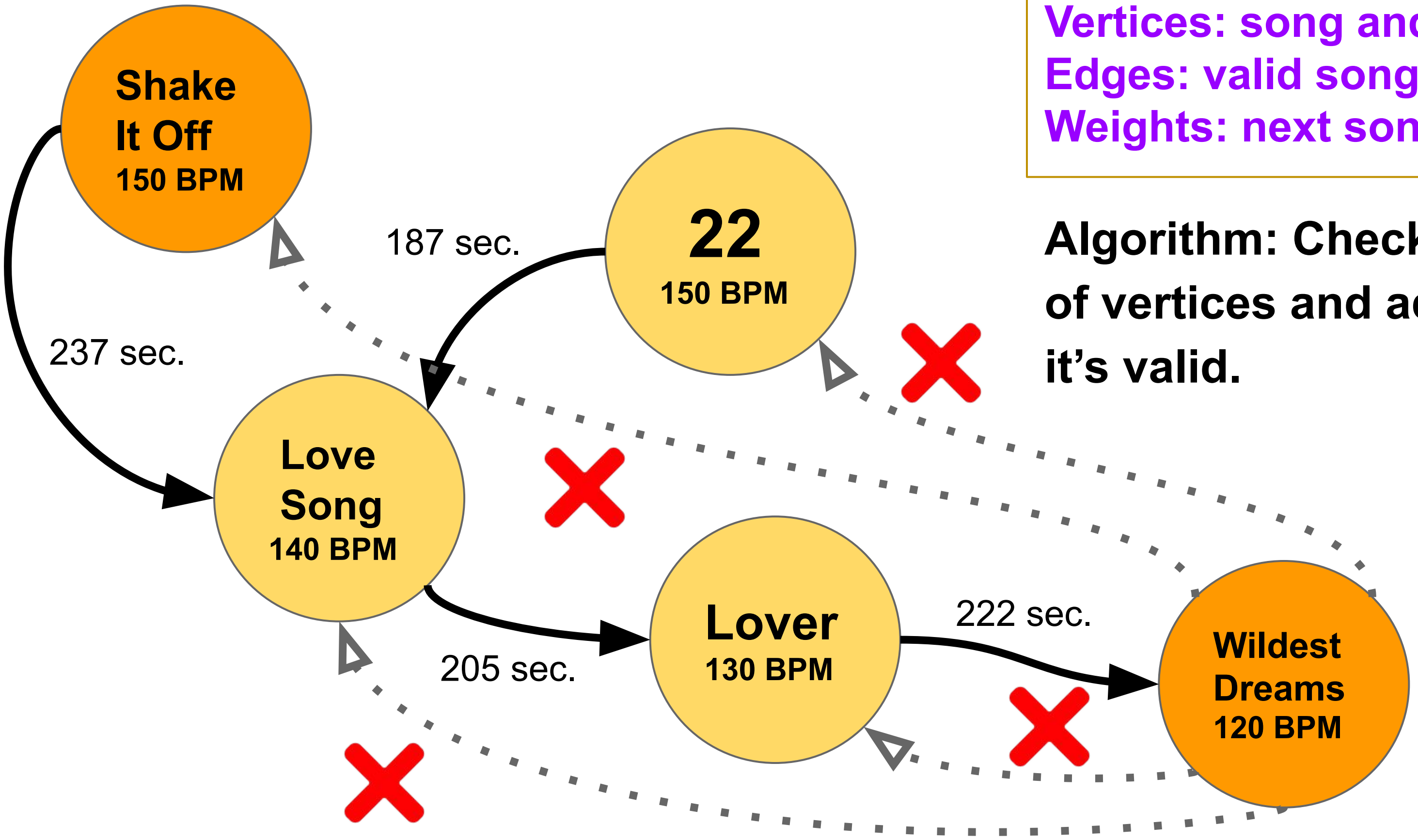


Are these valid?

Vertices: song and BPM
Edges: valid song transitions
Weights: next song length

Algorithm: Check every pair of vertices and add an edge if it's valid.

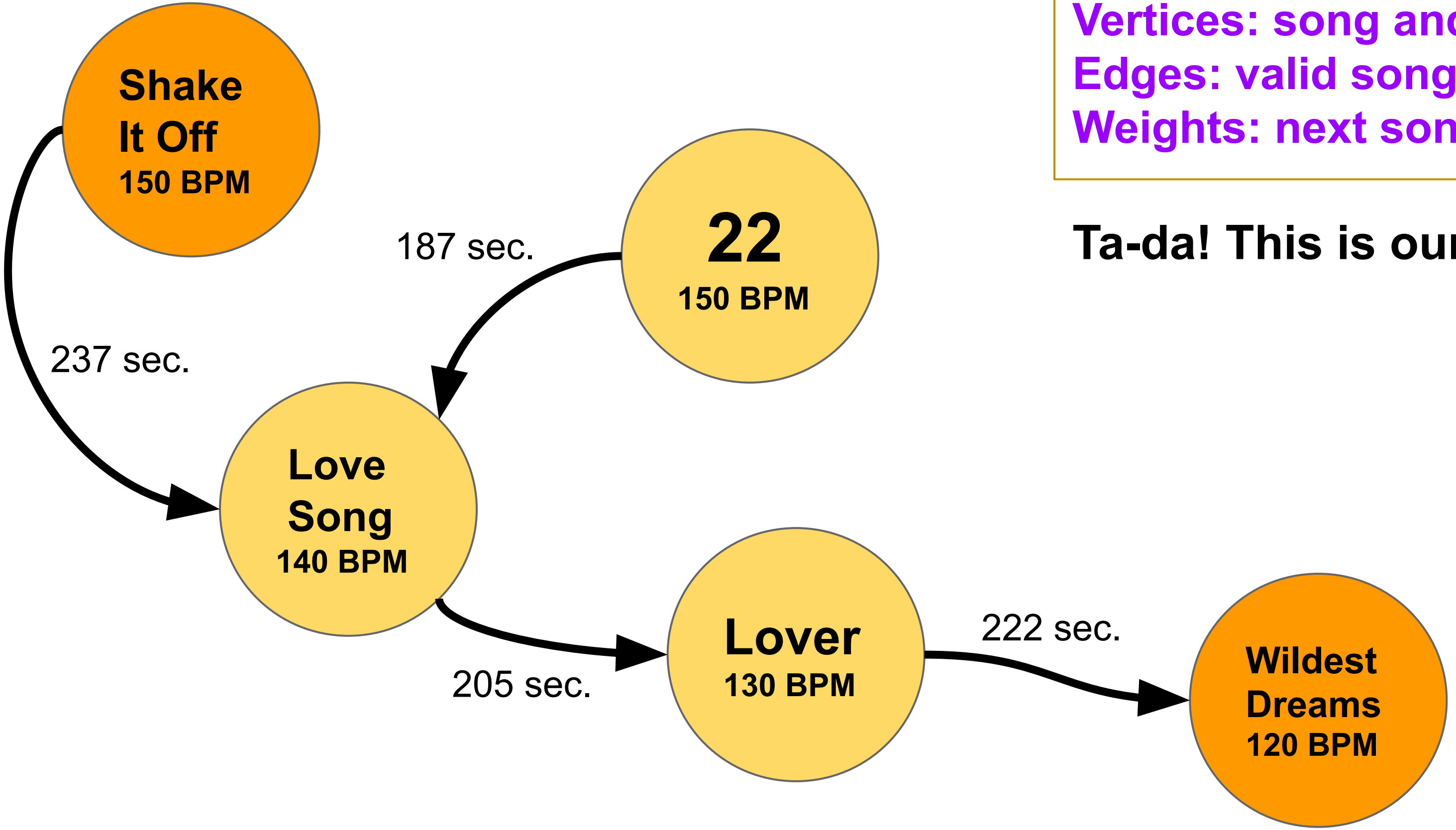
Problem 6B



Vertices: song and BPM
Edges: valid song transitions
Weights: next song length

Algorithm: Check every pair of vertices and add an edge if it's valid.

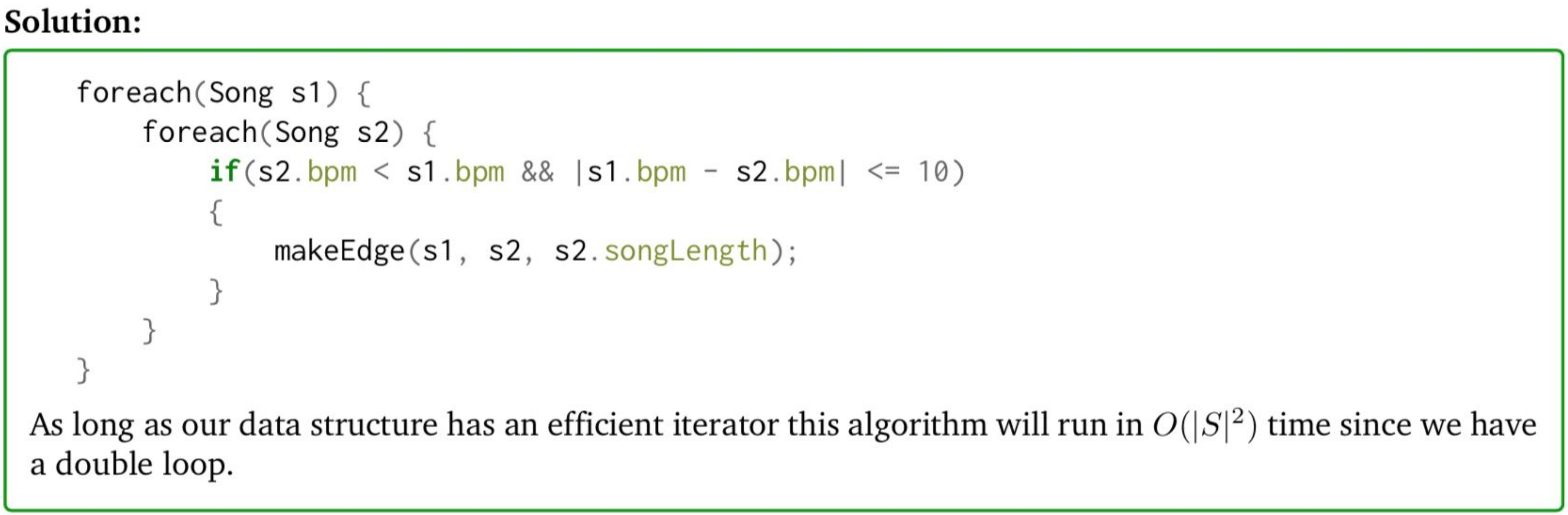
Problem 6B



Vertices: song and BPM
Edges: valid song transitions
Weights: next song length

Ta-da! This is our graph :)

Problem 6B



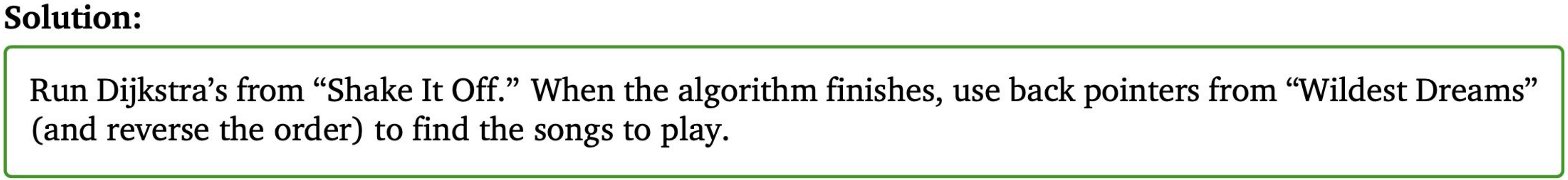
Solution:

```
foreach(Song s1) {
    foreach(Song s2) {
        if(s2.bpm < s1.bpm && |s1.bpm - s2.bpm| <= 10)
        {
            makeEdge(s1, s2, s2.songLength);
        }
    }
}
```

As long as our data structure has an efficient iterator this algorithm will run in $O(|S|^2)$ time since we have a double loop.

Problem 6C

(c) Describe an algorithm you could run on the graph you just constructed to find the list of songs you can play to get to "Wildest Dreams" the fastest without disappointing the crowd.



Solution:

Run Dijkstra's from "Shake It Off." When the algorithm finishes, use back pointers from "Wildest Dreams" (and reverse the order) to find the songs to play.

Problem 6D

(d) What is the running time of your plan to find the list of songs? You should include the time it would take to construct your graph and to find the list of songs. Give a simplified big-O running time in terms of whatever variables you need.

Problem 6D

How long did it take to construct our graph? → $O(S^2)$

We then run Dijkstra’s starting from Shake It Off. What’s the runtime of Dijkstra’s? → $O(E*log(S) + S*log(S))$

What’s the total runtime to find the list of songs the DJ should play?

$O(S^2 + E*log(S) + S*log(S))$

Is this simplified?

S^2 dominates $S*log(S)$ so we can ignore the smaller term!

Total runtime:
$O(S^2 + E*log(S))$