Shortest Paths in Graphs and Digraphs

Textbook Reading:

- Review BFS and BFS out-tree for a digraph, Subsection 5.4.6, pp. 227-231. BFS out-tree is a shortest path tree.

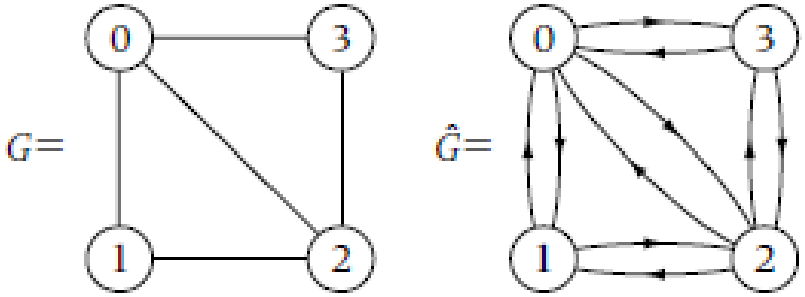
- Read Section 6.6, pp. 273-279, Dijkstra's Shortest Path algorithm for weighted graphs and digraphs.

Shortest paths in Unweighted Graphs and Digraphs

We first consider the unweighted problem, i.e., finding a shortest path in an graph or digraph without weights on the edges. A shortest path from vertex *u* to *v* is one that has minimum hop-length, i.e., has the fewest edges. In a digraph a shortest path is a minimum length directed path from *u* to *v*, i.e., the edges are consistently directed toward *v* and away from *u*.

Designing a shortest path algorithm for digraphs yields one for (undirected) graphs

The problem of finding a shortest path in a digraph D generalizes the problem of finding a shortest path in an undirected graph G. This is because as was pointed out in a previous lecture (see supplemental notes on Implementing graphs and digraphs). Given a any graph *G* we can associated the combinatorially equivalent symmetric digraph $\hat{G}$, where each undirected edge {*u*,*v*} is replace with the two directed edges (*u*,*v*) and (*v*,*u*).



For each path in *G* joining *a* and *b* there is a directed path in $\hat{G}$ joining a and b and vice versa. Thus, a shortest path from a to b in $\hat{G}$ corresponds to a shortest (directed) path from *a* to *b* in $\hat{G}$ and vice versa.

Computing Shortest Paths in Unweighted Digraphs

The most efficient way to find shortest paths from a given vertex *r* to all other vertices in a digraph for which there exists a path in the digraph from *r* to that vertex is using Breadth-First Search (BFS). We perform BFS starting at initial vertex *r* and keep track of the **BFS out-tree rooted at *r*** using the parent array implementation. When we dequeue a vertex *u* and enqueue vertices *v* in the neighborhood of *u* that are unvisited, we set

Parent*[*v*] = *u.

The BFS out-tree determined by *Parent*[0: *n* – 1] is a shortest-path tree rooted at *r*, i.e., contains a shortest directed path from *r* to *v* for all vertices *v* for which there exists a directed path in the digraph from *r* to *v*.

Complexity of finding a shortest-path out-tree

Computing a BFS-out tree has worst-case complexity

$$W(m,n) \in \Theta(m + n)$$

To achieve this complexity the digraph D should be implemented using **adjacency lists**. Clearly, this algorithm is optimal, because just to set up the adjacency lists take time $\Theta(m + n)$.

Shortest Paths in Weighted Graphs and Digraphs

Observe that a G together with a weighting w of the vertices is combinatorially equivalent to the symmetric graph $\hat{G}$ together with the weighting $\hat{w}$ where each weighted edge e = $\{u,v\}$ is replaced with the two directed edges (u,v) and (v,u) both having weight w(e). A shortest (directed) path in a weighted digraph from a to b is a path from a to b having minimum weight, where the weight of the path is the sum of the weight of its edges.

Thus, a shortest path algorithm for digraphs includes as a special case a shortest path algorithm for graphs.

Growing a Directed Tree and the Concept of a Directed Cut

Suppose we are growing an out-directed tree T starting with an initial or root vertex r. Then, at any stage of expanding T, we must choose and edge $e = (u,v)$ where its tail ***u*** **is in the tree (** $u \in V(T)$) and the head ***v*** **is not in the tree (** $v \in V - V(T)$). Otherwise, in the case where both u and v belong to the tree a cycle would be formed or in the case both u and v are not in the tree, the edge e would be disconnected from T. The set of edges having one end vertex in the tree and the other not in tree, which we denote by $Cut(T)$, is called a **directed cut** or simply a **cut** in graph theory.

Thus, in order to expand T we must choose an edge from the directed cut ***Cut*(*T*).**

Virtual edges

Vertex *r* may not be connected via a directed path to all the other vertices in the digraph *D*. For convenience, to make sure it is, we add the edge (*r*,*v*) for all vertices v not out-adjacent to *r* and give these edge the weight ∞ (in practice this would be a large weight). If (*r*,*v*) is chosen to be part of the final shortest-path tree, which now will necessarily be a spanning out-tree rooted at *r*, then we can conclude that there is no directed path from *r* to *v* in *D*.

High-Level Description of Dijkstra's Algorithm

Input: *D (V, E)* (a digraph), *r* (a vertex of *G*), *w* (a nonnegative weighting on *E*)

Output: *T* (a shortest-path out-tree rooted at *r*)

Start with an initial tree T_0 consisting of a single vertex *r* and no edges. Similar to Prim's algorithm we grow a spanning tree rooted at *r* by adding a sequence of edges

$$T_i = T_{i-1} + e_i \text{ , } i = 1, 2, \ldots, n-1,$$

where $e_i = (u_i, v_i)$ is chosen so that it has **minimum weight** over all the edges in $Cut(T_i)$, where the weight of edge e_i is taken to be

$$Dist[u_i] + w(e_i)$$

where $Dist[u_i]$ is the weighted distance from r to u_i, i.e., the weight of the path in the tree from *r* to u_i. The final tree $T = T_{j-1}$ is a shortest path out-tree rooted at *r*, i.e., the path in *T* from *r* to any other vertex v that is reachable from *r* has minimum weight over all paths in D from r to v, i.e., is a shortest path in *D* from *r* to *v*.

High-Level Pseudocode of Dijkstra's Algorithm

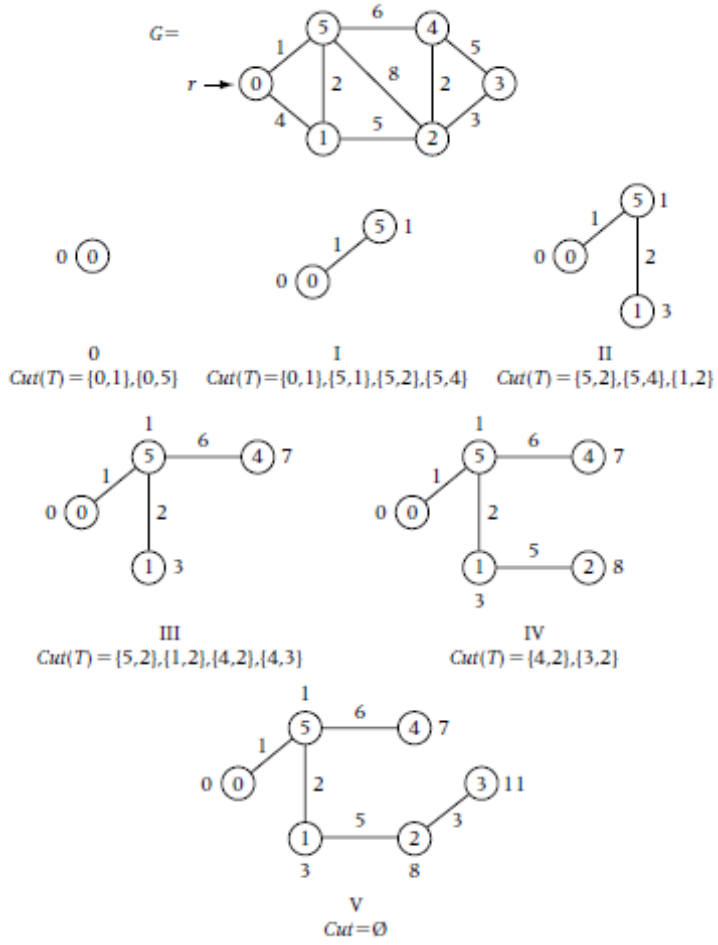
procedure *Dijkstra*(*G*, *r*, *w*, *T*)
Input: *D (V, E)* (a digraph)
r (a vertex of *D*)
w (a nonnegative weighting on *E*)
Output: *T* (a shortest path out-tree rooted at *r*)

dcl *Dist*[0:*n* – 1] (an array initialized to ∞)
T is initialized to be the tree consisting of the single vertex *r*
Dist[*r*] ← 0
while *Cut*(*T*) ≠ ∅ **do**
select an edge *uv* ∈ *Cut*(*T*) such that *Dist*[*u*] + *w*(*uv*) is minimized
add vertex *v* and edge *uv* to *T*
Dist[*v*] ← *Dist*[*u*] + *w*(*uv*)
endwhile
end *Dijkstra*

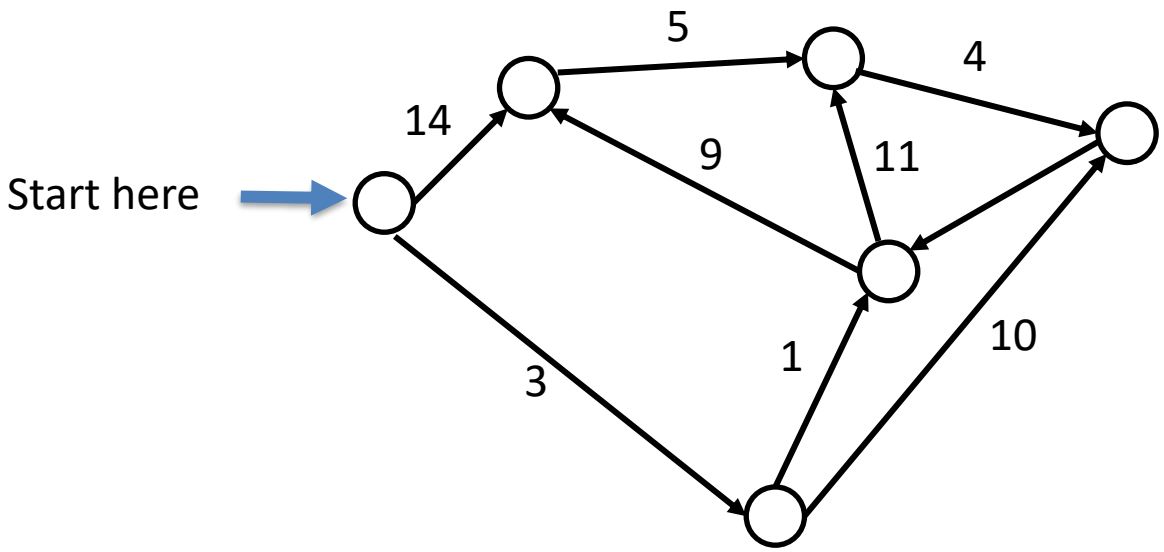
Dijkstra's algorithm for undirected graphs

As mentioned earlier an undirected graph G can be consider as a special case of a digraph. Thus, our high-level description of Dijkstra's algorithm for digraphs includes undirected graphs as a special. It is identical to Prim's algorithm except the weight $w(uv)$ of an edge $\{u,v\}$ in the cut is replace with $w(uv) + Dist[u]$.

High-level action of Dijkstra's Algorithm for a sample graph *G*



PSN. Show the action of Dijkstra's algorithm for the following digraph



Finding minimum edge in *Cut*(*T*) Naïve Algorithm

- Similar to Prim's algorithm using the naïve algorithm to compute the minimum edges in each cut by simply scanning all the edges in the cut would result in Dijkstra's algorithm having worst-case complexity $W(n) \in \Theta(n^3)$

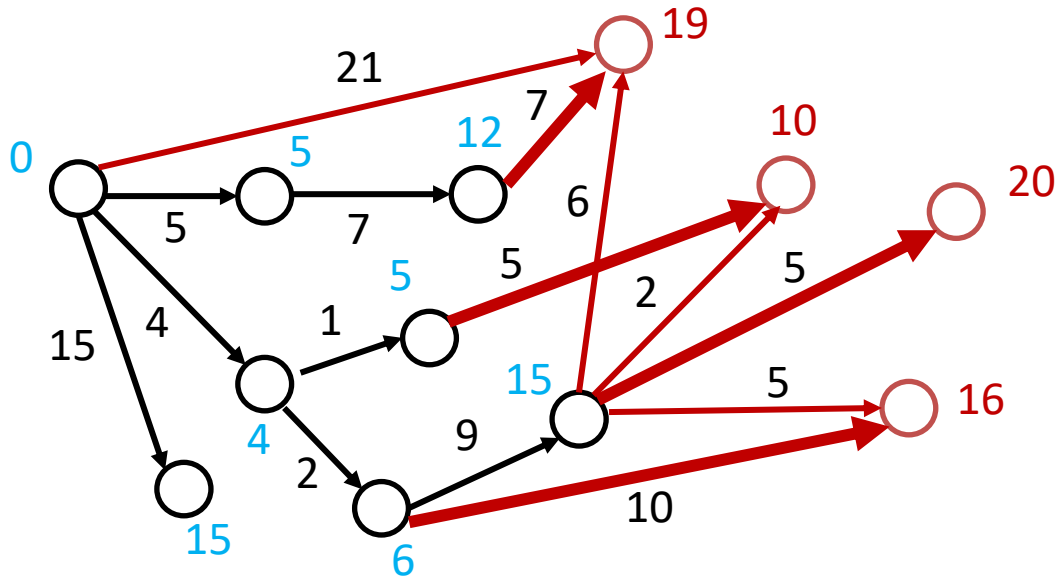
- As with Prim's algorithm we can do much better than this.

Improvement using a Priority Queue

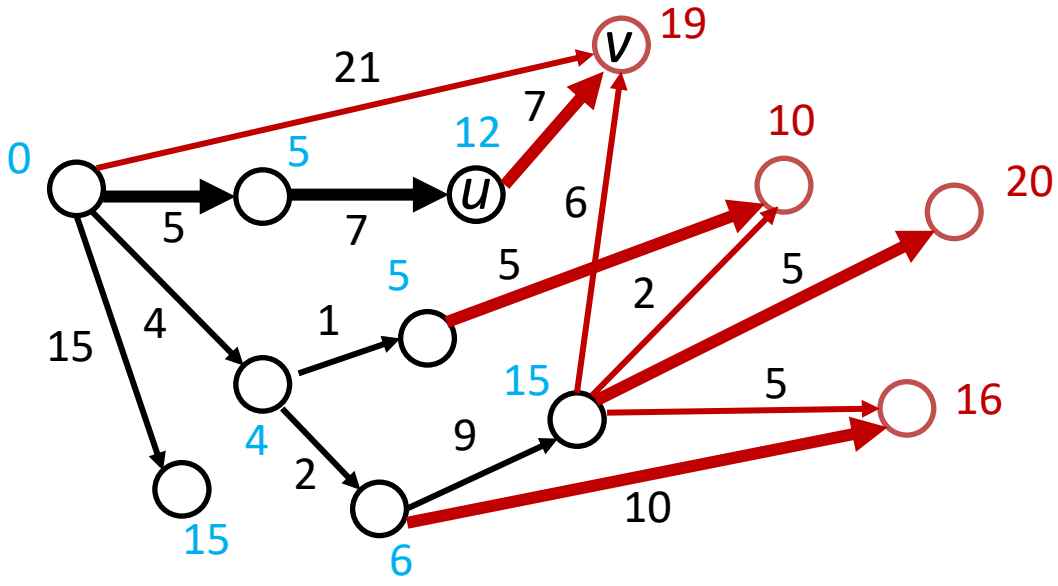
Similar to Prim's algorithm we obtain an improvement by maintaining a priority queue of vertices not in the current tree *T,* the only difference is the **priority of a vertex *v*** is

***Dist*[*v*],**

where *Dist*[*v*] is the minimum weight over all paths from *r* to *v* where all but the last edge of the path belongs to *T*.



Letting *uv* denote the last edge in such a path having minimum weight, note that *uv* has minimum weight over all the edges in the *Cut*(*T*) having head *v*, where the weight of an edge *uv* in *Cut*(*T*) is given by *Dist*[*u*] + *w*(*uv*).



Minimum Weight Edge in *Cut*(*T*)

Based on our last observation, it follows that

minimum weight of the an edge in *Cut*(*T*)
= minimum of *Dist*[*v*] over all *v* not in *T*
= minimum over all *v* in the priority queue.

Thus, this minimum weight over all the edges in *Cut*(*T*) can be computed simply by performing one **dequeue** operation.

Maintaining *Dist*

After we dequeue a vertex *u* and add it to the tree, we must update *Dist*. The vertices for which *Dist* will be affected are those vertices *v* **not in the tree** for which **(*u*,*v*)** $\in$ ***E***. They can be updated by simply scanning the neighborhood of *u* and performing the operation

if *Dist*[*u*] + *w*(*uv*) < *Dist*[*v*] **then**

Dist[*v*] ← *Dist*[*u*] + *w*(*uv*)

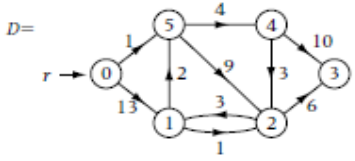
Parent[*v*] ← *u*

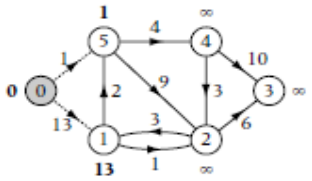
endif

Pseudocode for Dijkstra's Algorithm

```
procedure Dijkstra(D,w,r,Parent[0:n – 1],Dist)
Input:   D (digraph with vertex set V and edge set E)
         r (a vertex of D)
         w (a weighting of E with nonnegative real numbers)
Output:  Parent[0:n – 1] (parent array of a shortest-path spanning out-tree – with virtual edges added)
         Dist[0:n – 1] (array of weights of shortest paths from r)
    for v ← 0 to n – 1  do                  // initialize Dist[0:n – 1] and InTheTree[0:n – 1]
            Dist[v] ← ∞
            InTheTree[v] ← .false.
    endfor
    Dist[r] ← 0
    Parent[r] ← -1
    for Stage ← 1 to n – 1 do
        Select vertex u that minimizes Dist[u] over all u such that InTheTree[u] = .false.
        InTheTree[u] ← .true.
        for each vertex v such that uv ∈ E do       // update Dist[v] and Parent[v]
            if .not. InTheTree[v] then
                if Dist[u] + w(uv) < Dist[v] then
                    Dist[v] ← Dist[u] + w(uv)
                    Parent[v] ← u
                endif
            endif
        endfor
    endfor
end Dijkstra
```

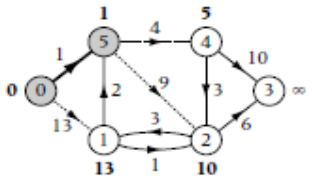
Action of Dijkstra's Algorithm for Sample Digraph *D*





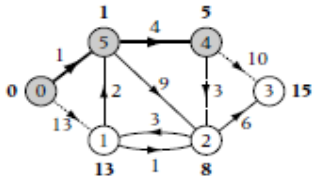
i	0	1	2	3	4	5
Dist[*i*]	0	13	∞	∞	∞	1
Parent[*i*]	−1	0	–	–	–	0
InTheTree[*i*]	T	F	F	F	F	F

Stage 1



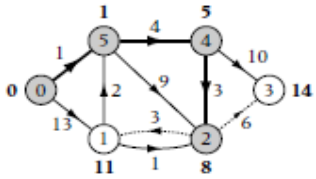
i	0	1	2	3	4	5
Dist[*i*]	0	13	10	∞	5	1
Parent[*i*]	−1	0	5	–	5	0
InTheTree[*i*]	T	F	F	F	F	T

Stage 2



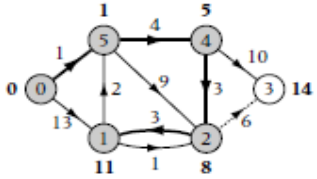
i	0	1	2	3	4	5
Dist[*i*]	0	13	8	15	5	1
Parent[*i*]	−1	0	4	4	5	0
InTheTree[*i*]	T	F	F	F	T	T

Stage 3



i	0	1	2	3	4	5
Dist[*i*]	0	11	8	14	5	1
Parent[*i*]	−1	2	4	2	5	0
InTheTree[*i*]	T	F	T	F	T	T

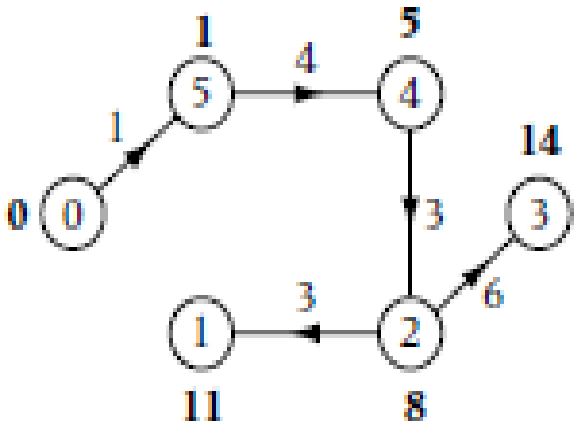
Stage 4



i	0	1	2	3	4	5
Dist[*i*]	0	11	8	14	5	1
Parent[*i*]	−1	2	4	2	5	0
InTheTree[*i*]	T	T	T	F	T	T

continued

Action of Dijkstra's Algorithm for Sample Digraph D cont'd



Resulting shortest path spanning out-tree rooted at vertex $r = 0$

Slight Savings

Similar to procedure *Prim, procedure Dijkstra* terminates after only $n - 1$ stages, even though there are n vertices in the final minimum spanning tree. The reason is simple: After stage $n - 1$ has been completed, *InTheTree*[0:$n - 1$] is **.false.** for only one vertex w. Thus, another iteration of the **for** loop controlled by *Stage* would result in no change to the arrays *Nearest*[0:$n - 1$] and *Parent*[0:$n - 1$]. In other words, the last vertex and edge in the minimum spanning tree come in for free.

Complexity Analysis of Procedure Dijkstra's is identical to Prim's

There are $n - 1$ stages and each stage involves at most $n - 2$ comparisons of edge weights to find the minimum value of Nearest and at most $n - 1$ comparisons to update Nearest. Thus, Procedure Prim has complexity (computing time) $O(n^2)$. Further, this result is sharp, i.e.,

$$W(n) \in \Theta(n^2) .$$

Using min-heap to implement priority queue

The dequeue operation for each vertex takes time $O(\log n)$, so

the total time for dequeuing is $O(n \log n)$.

Updating the priorities when vertex v is added to the out-directed tree T takes time $O(d_{\text{out}}(v)\log n)$ where $d_{\text{out}}(v)$ denotes the out-degree of v. That is because the priority may need to be updated for each vertex in the out-neighborhood of v and each update takes time $O(\log n)$. Thus,

the total time for updates over the entire algorithm is $O(m \log n)$.

This follows from the observation that the sum of $d_{\text{out}}(v)$ over all the vertices v equals the number of edges m.

Min-heap implementation of priority queue, cont'd

Thus, the time for updates dominates the time for dequeuing, so that

the computing time of Dijkstra's algorithm is $O(m \log n)$.

Note that to achieve this, it is necessary to implement the digraph D using **adjacency lists**. Also, this result is sharp, i.e., the worst-case complexity of Dijkstra's algorithm implemented in this way is given by

$$W(m,n) \in \Theta(m \log n).$$

Prim's vs. Dijkstra's

Dijkstra's algorithm is very similar to Prim's algorithm. However, Dijkstra's works for digraphs, whereas Prim's only works for undirected graphs. We can apply Prim's to a digraph, similar to the way we did Dijkstra's, except with the weight of an edge uv in the directed cut $Cut(T)$ being $w(uv)$ instead of $Dist[u] + w(uv)$, but it won't always yield a minimum-weight out-directed spanning tree rooted at r (where virtual edges having tail r are added so digraph contains at least one out-directed spanning tree).

Exercise (optional). See if you can find a counterexample showing that Prim's algorithm fails for digraphs. That is, applying Prim's algorithm to your counterexample weighted digraph D and root vertex r yields an out-directed spanning tree rooted at r whose weight is strictly greater than another out-directed spanning tree in D.

There is an efficient algorithm for finding a minimum-weight out-directed spanning tree rooted at a given vertex r, but it is a sophisticated algorithm that is beyond the scope of this course.

Other Shortest-Path Algorithms

- There are a number of other shortest-path algorithms in the literature. When we get to the Chapter 8 on Dynamic Programming, we will discuss another shortest-path algorithm called the Floyd-Warshall algorithm, which computes all-pairs shortest paths, i.e., shortest paths between every pair of vertices. It works for negative weights as long as there are no *negative directed cycles*, i.e., the sum of the weights of the edges in the directed cycle is negative.

- Dijkstra's algorithm fails if negative weights are permitted, even if there are no negative cycles. It is not difficult to come up with a counterexample DAG on 3 vertices where Dijkstra's algorithm fails. Further, there are counterexamples for any $n \geq 3$ vertices. See if you can come up with a counterexample (optional). Hint: take DAG to consist of a single path from r to s, together with the edge (r,s).

- The problem of finding a shortest path where negative edge weights are permitted is NP-hard, i.e., there is no known polynomial-time algorithm for solving the problem in the worst-case.

How are dogs able to estimate the flight path of a ball?

They ballbark it.