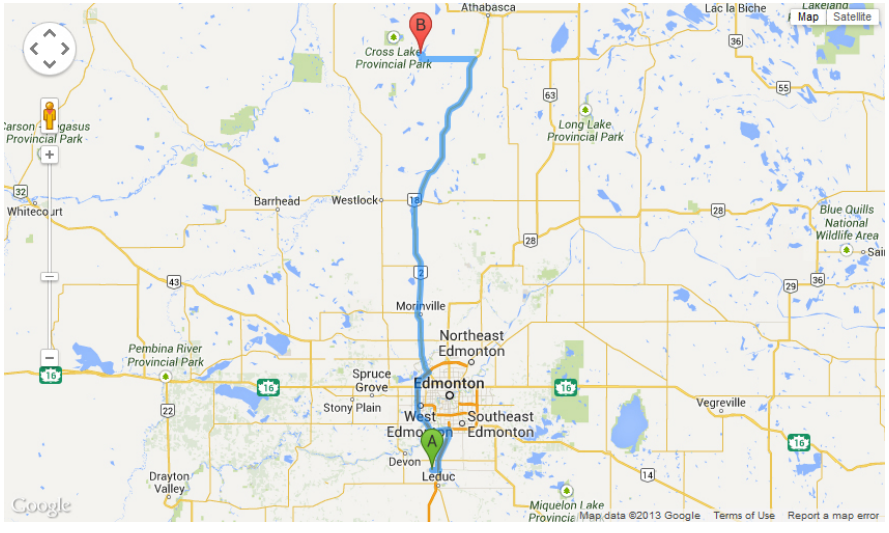
Graph Searches - Dijkstra's

CMPUT 275 - Winter 2018

University of Alberta

Many graphs have a cost associated with edges.

In the Edmonton road network, an edge connects two geographic locations. A natural cost is the distance between these points.



Thus, it is interesting to find $s - t$ paths with minimum possible edge cost. Example: minimize driving distance.

This Lecture: All costs are nonnegative! Handling negative-cost edges is trickier and can only be handled in some cases.

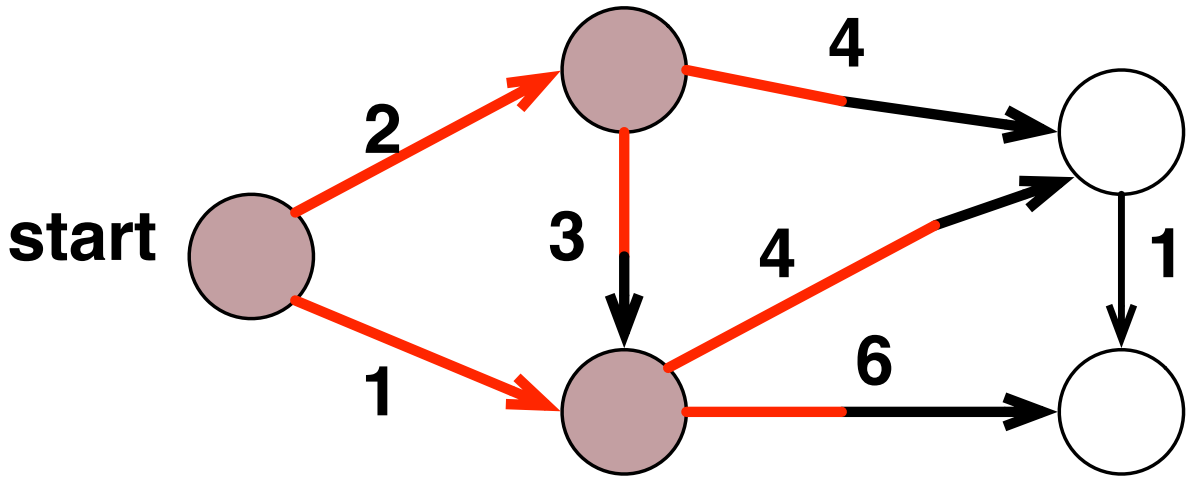
Dijkstra's Algorithm

By far the most popular and useful algorithm for this is Dijkstra's algorithm (and its variants).

It builds off of the same graph search paradigm from last lecture:

- Incrementally build a set of reachable nodes.
- The order nodes are *processed* will affect the search tree, we use a particular order to get minimum-cost paths.

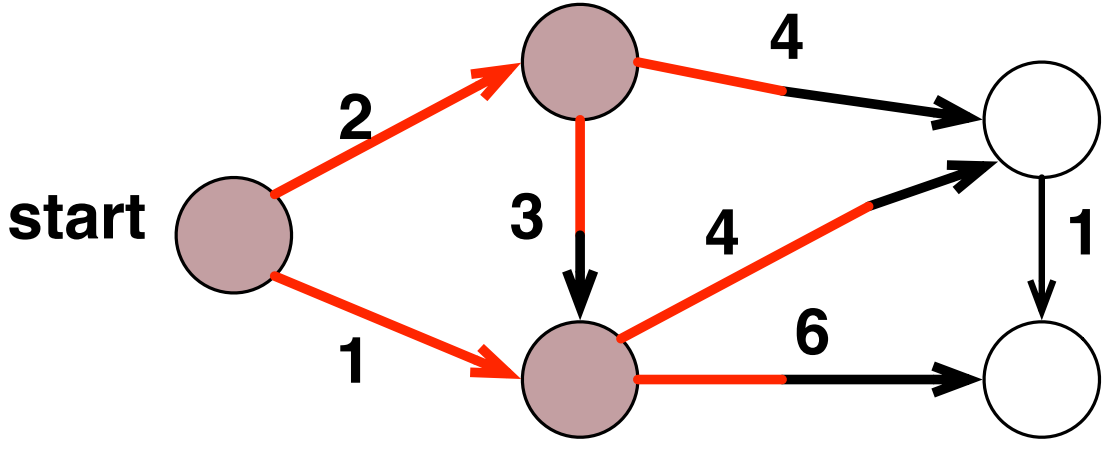
If all edge costs are 1, we already know what to do. Use a `queue` to get $O(|E|)$ running time.



Suppose we light a fire at the start, which then spreads across edges. The cost of an edge is the time it takes the fire to spread across it.

Rules:

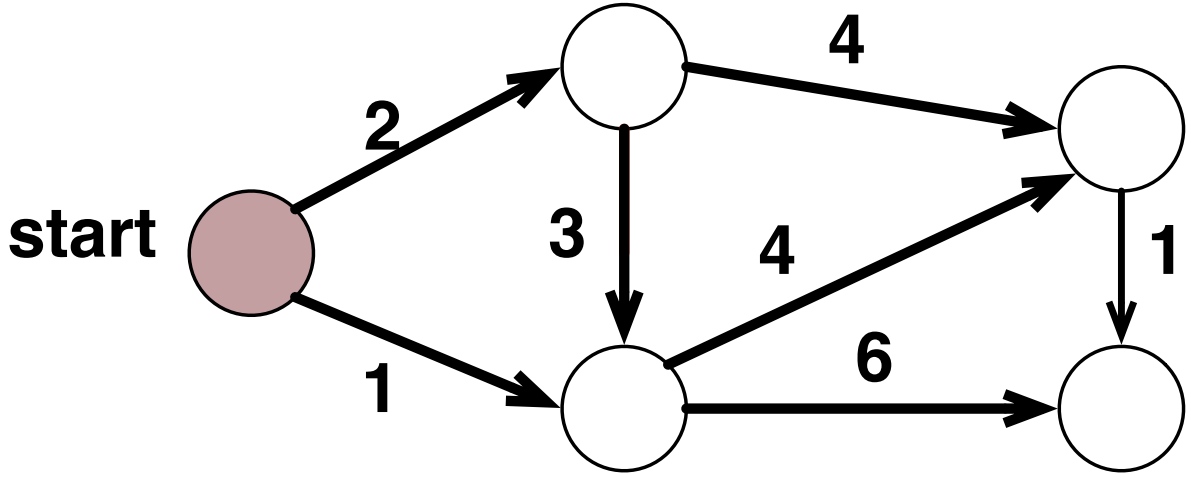
- The fire only burns in the direction of an edge (not against it).
- Once fire reaches a vertex for the first time, it lights all edges exiting the vertex.
- If fire reaches a vertex that is already burned, it does nothing.



The time each vertex is burned is the cost of the cheapest path to it!

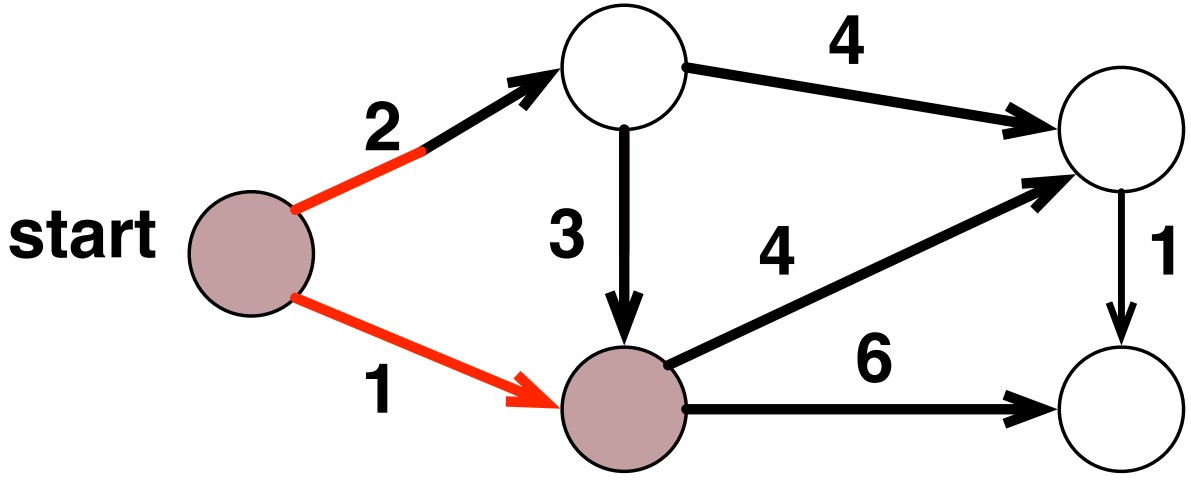
Dijkstra's algorithm essentially simulates this process. Let's step through an example first.

Time = 0



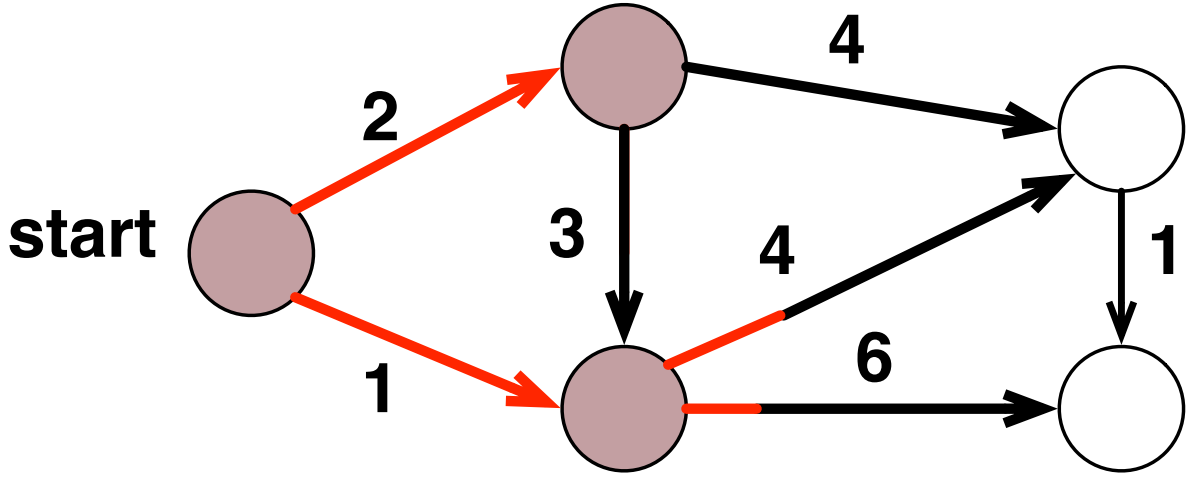
Light the start on fire.

Time = 1



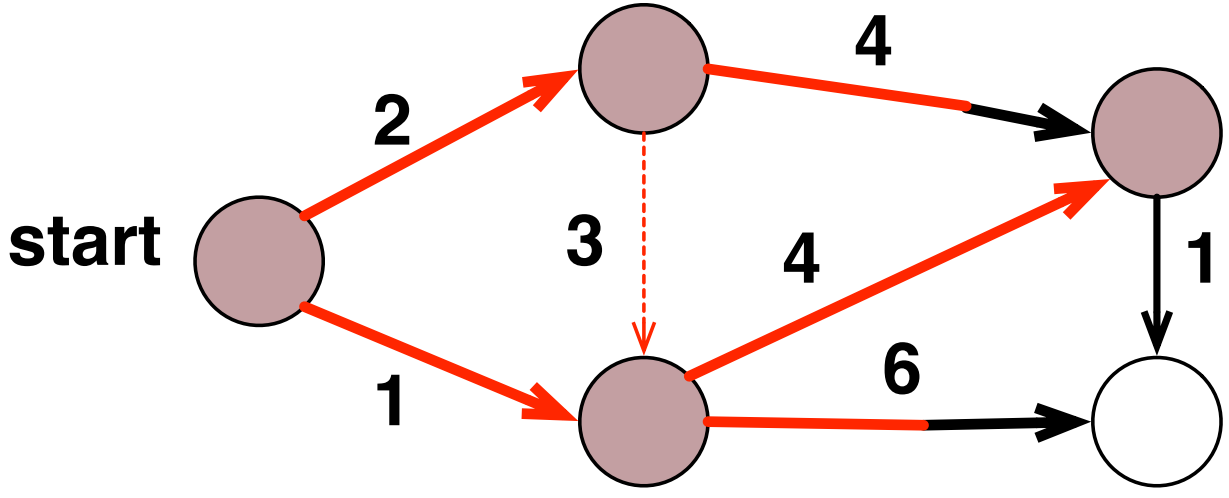
The fire reaches the bottom-left vertex, which catches fire.

Time = 2



The fire reaches the top-left vertex, which catches fire. Note the fire has spread out from the bottom-left vertex for 1 unit of time.

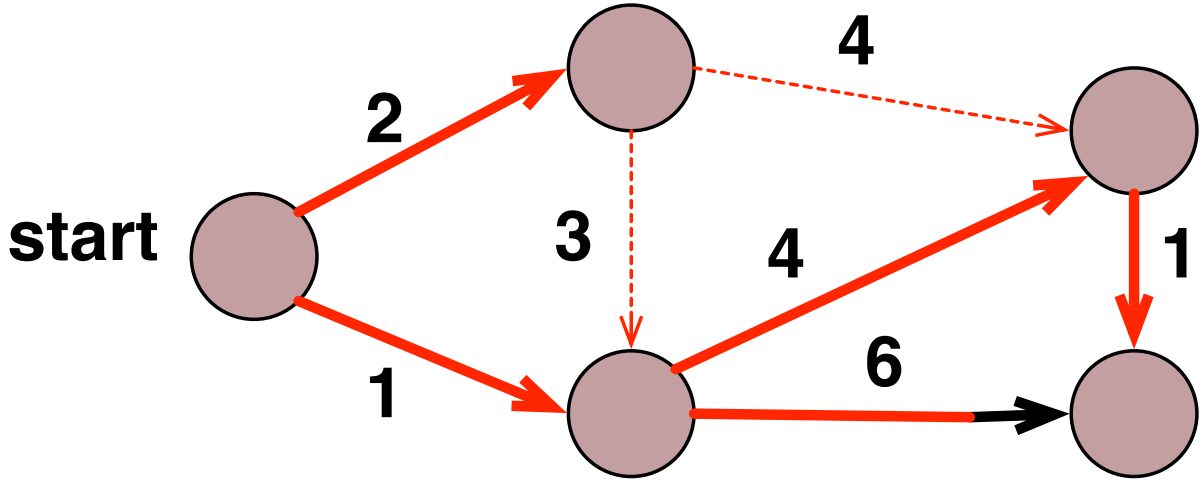
Time = 5



After 3 more units of time, the fire reaches the top-right vertex, which catches fire.

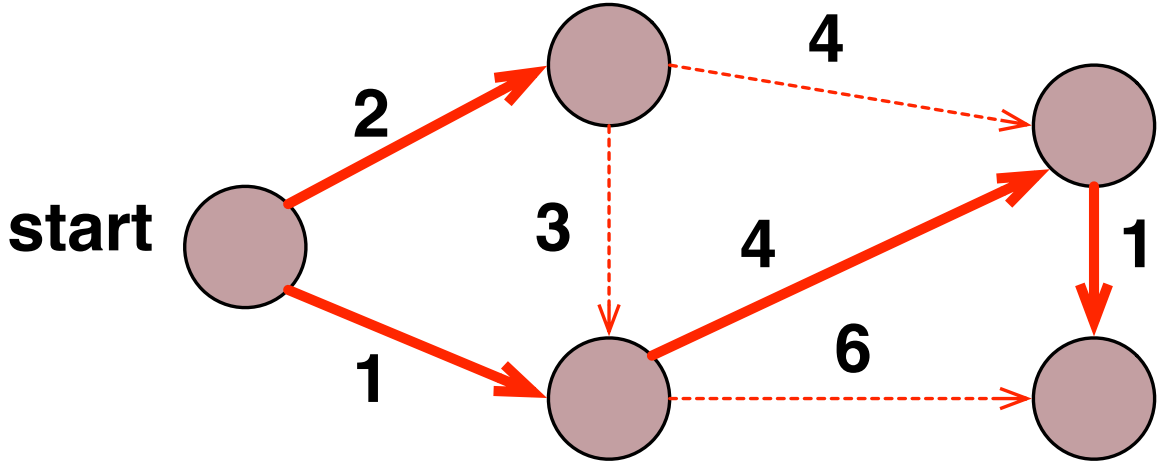
Note the middle vertical edge also had its fire reach the bottom vertex, but it was already burned so that edge was useless in finding a shortest path.

Time = 6



The bottom-right vertex catches fire. Note another edge had its fire reach an already burned vertex at this time also.

Time = 7



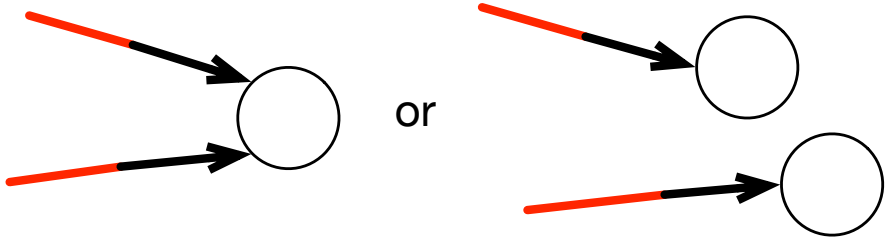
Finally, the last burning edge reaches its endpoint. It was already burned; we already have a shortest path to it.

The thick edges are a search tree!
They give shortest paths to each burned/reached vertex.

Comments

- Think of when an edge stops burning as an **event**. At each event, we do the following:
  - If the endpoint of the edge is not yet burned, burn it and spawn new events (one per outgoing edge).
  - If the endpoint of the edge is already burned, do nothing.

- If two events happen at the same time, we can process them in either order in our algorithm (i.e. no specific tiebreaking rules).



Heaps

We need a container that will hold **events** along with the time the event happens. It should support the removal of the next event efficiently. With foresight, call such a container a **heap**.

A heap will store items, each with a **key** which you can think of as **time** in the context of the graph search.

A heap supports the following operations:

- `insert(x, key)`: add `x` to the heap with the given key value.
- `min()`: returns the pair `(x, key)` with the smallest key.
- `popmin()`: removes and returns the pair `(x, key)` with the smallest key.
- `size()`: number of items being held.

Pseudocode

Algorithm 1 Dijkstra's Algorithm

reached $\leftarrow$ empty dictionary
events $\leftarrow$ empty heap
events.*insert*$((s, s), 0)$ # vertex s burns at time 0
while *len*(*events*) > 0 **do**
  (u, v), *time* $\leftarrow$ *events*.*popmin*()
  if $v \notin$ *reached* **then**
    reached$[v] \leftarrow u$ # burn vertex v, record predecessor u
    for each neighbour w of v **do**
      # new event: edge (v,w) started burning
      events.*insert*$((v, w), time + cost(v, w))$
return *reached*

Does This Work

The appeal to the intuition about fire is helpful, but is this really a tight argument that this works?

We can prove that when each vertex is added to the reached dictionary, the time (i.e. key value for the extracted $((u, v), time)$) is equal to the length of the cheapest path to v.

Details Worked Out in Class

Running Time Analysis

It really depends on how efficient the heap operations are. But the running time is dominated by:

- # Insertions: $O(|E|)$
- # Pop operations: $O(|E|)$

Why?
Each iteration of the inner loop inserts an edge into the heap. Each edge is considered by this iteration at most once, so is inserted at most once.

Also # insertions = # pops (each edge inserted will eventually be popped).

Running Time Analysis

Next Lecture: We will discuss and implement a heap that supports insertions and pops in $O(\log n)$ time where n is the number of items in the heap.

As the heap holds each edge at most once (and the fictitious edge (s, s)), a total of $\leq |E| + 1$ pass through the heap during Dijkstra's algorithm.

Running Time of Dijkstra's: $O(|E| \log |E|)$.

Essentially the Best Possible

A more sophisticated use of heaps can get the running time down to $O(|E| + |V| \cdot \log |V|)$, but this small improvement takes a lot of effort and is only helpful in graphs with many more edges than vertices.