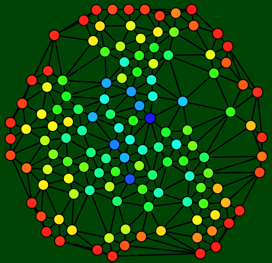
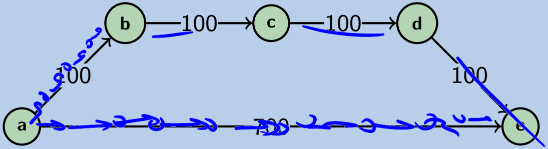
CSE 332

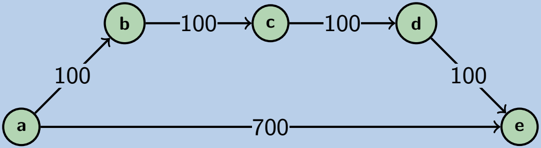
Data Structures and Parallelism

Graphs 3: Single-Source Shortest Paths

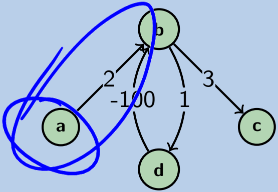


Some Initial Thoughts



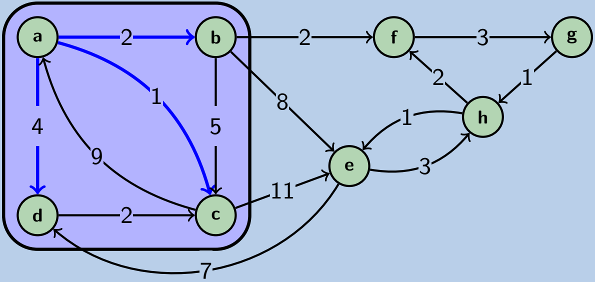


b
100
c
100
d
100
100
a
700
e



b
2
-100
1
3
a
c
d

The Idea



We will run a **simulation** of (infinitely many) ants exploring the graph.

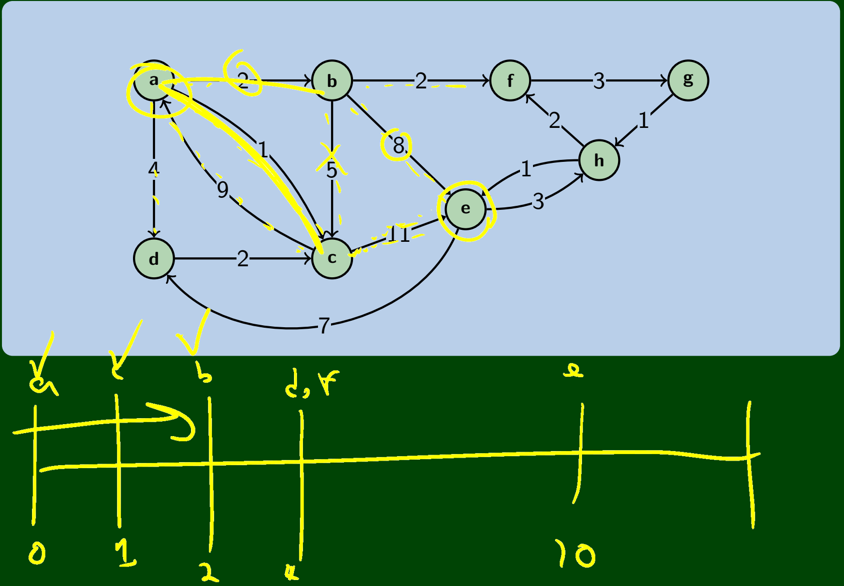
The ants all move at identical speeds.

We're interested in the **time step** that some ant first reaches each vertex.

- At each step...
  - The ants try to move along some new edge
  - We "process" a vertex at the timestep that an ant arrives there
  - When an ant arrives, they dispatch new ants to every out-edge
- We're done!

a:0, c:1, b:2

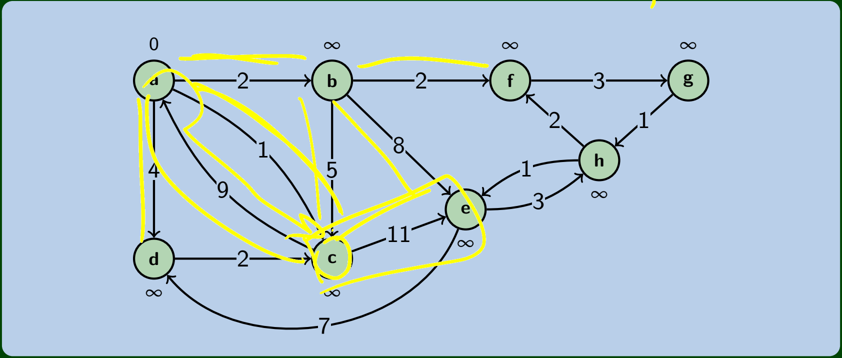
worklist ←



Example

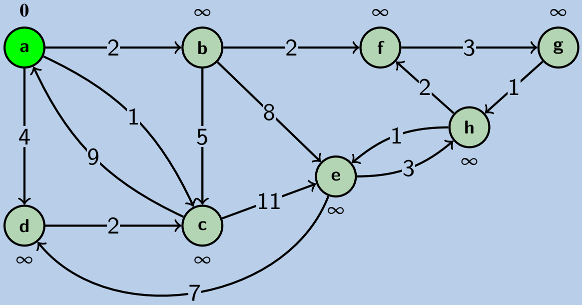
worklist ←

min(∞, 1 + 11)
min(12, 10)



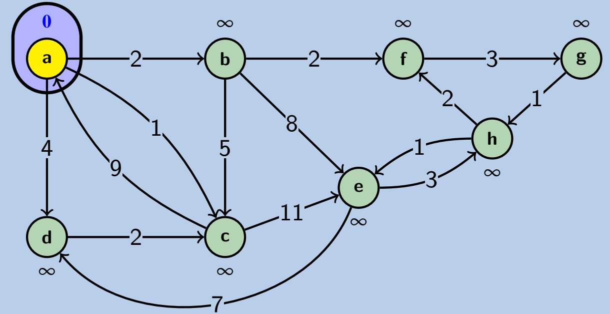
a	b	c	d	e	f	g	h
0	2	1	4	10	4	∞	∞

worklist ← | a ≤ 0 | ←



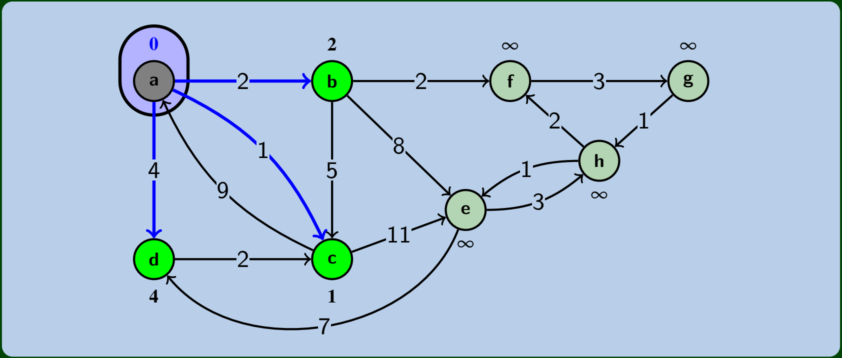
Example

worklist ←

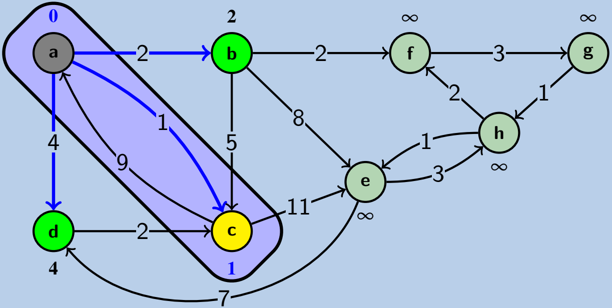


Example

worklist ← | c ≤ 1 | b ≤ 2 | d ≤ 4 | ←

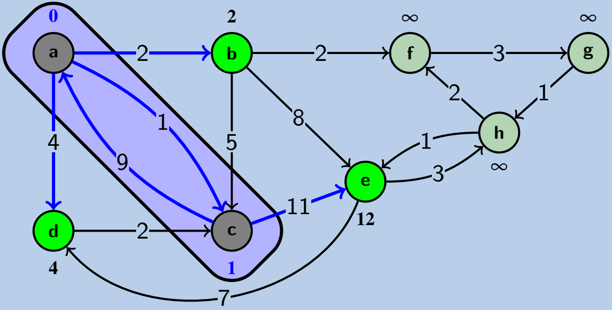


worklist $\leftarrow$ | b ≤ 2 | d ≤ 4 | $\leftarrow$



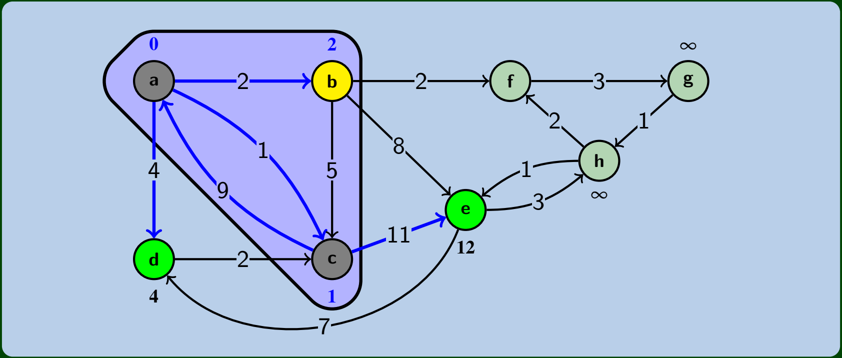
Example

worklist ← | b ≤ 2 | d ≤ 4 | e ≤ 12 | ←



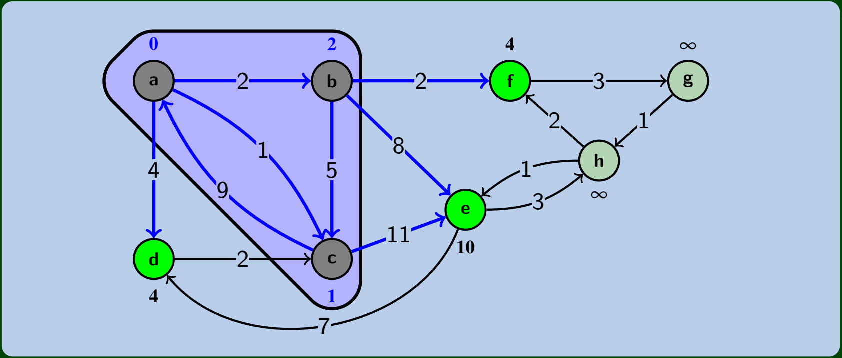
Example

worklist ← | d ≤ 4 | e ≤ 12 | ←



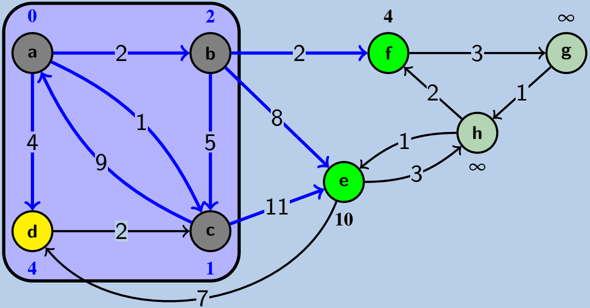
Example

worklist ← | d ≤ 4 | f ≤ 4 | e ≤ 10 | ←



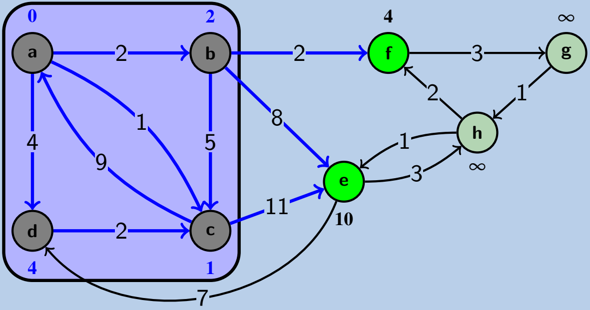
Example

worklist ← | f ≤ 4 | e ≤ 10 | ←



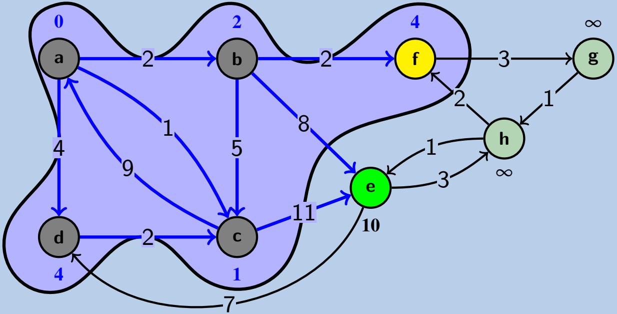
Example

worklist ← | f ≤ 4 | e ≤ 10 | ←



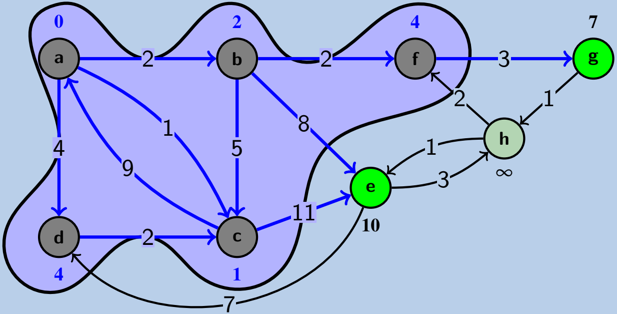
Example

worklist ← $e \leq 10$ ←



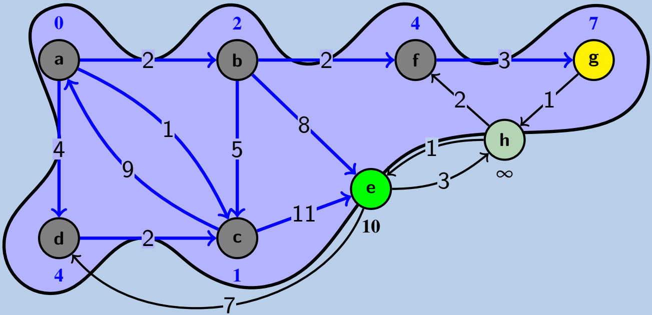
Example

worklist $\leftarrow$ | g ≤ 7 | e ≤ 10 | $\leftarrow$

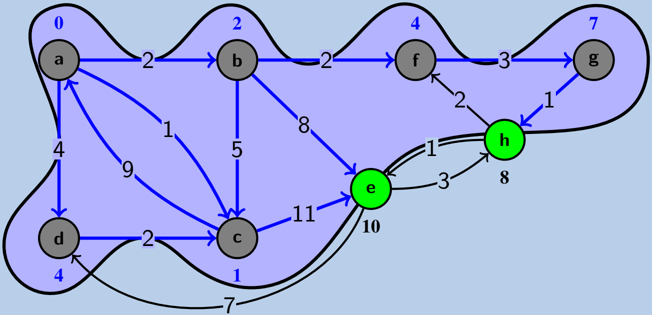


Example

worklist ← $\boxed{\text{e} \leq 10}$ ←

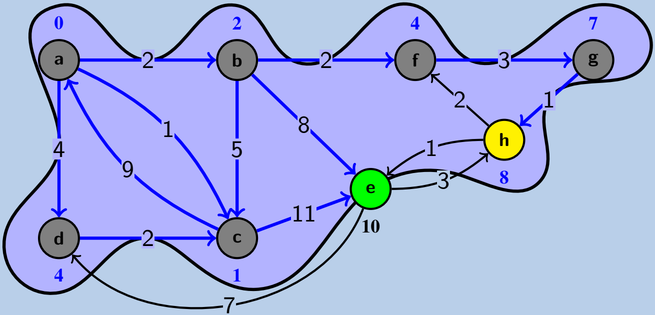


worklist ← | h ≤ 8 | e ≤ 10 | ←



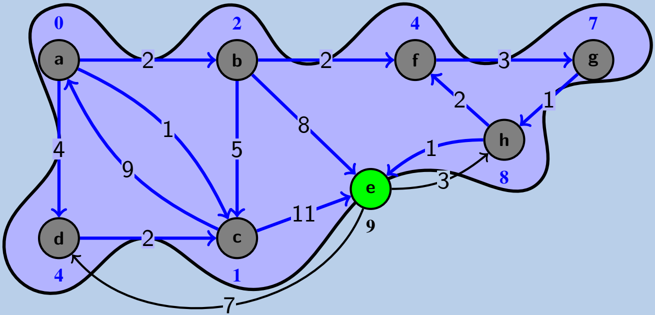
Example

worklist ← $\boxed{e \leq 10}$ ←

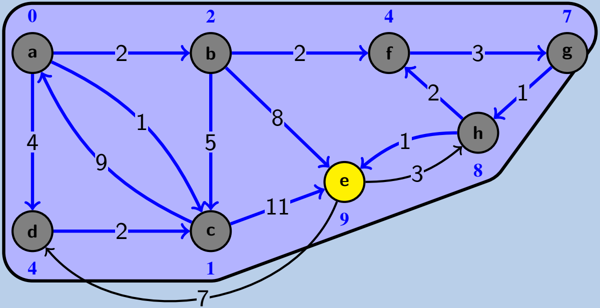


Example

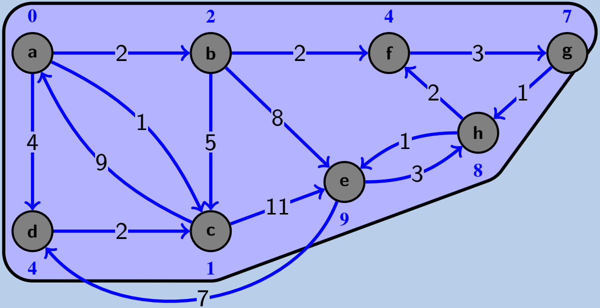
worklist ← e ≤ 9 ←



worklist ←



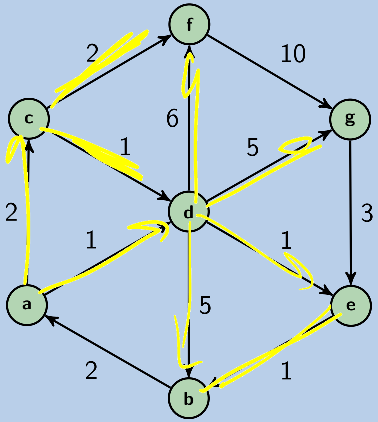
worklist ←



The Algorithm


```
dijkstra(G, source) {
   // We will use a "sorted list" as our worklist, because the items
   // in the work list are "events" which are processed in order

   // (v, timestep) in worklist, where v is a vertex and timestep
   // is the "time" the first ant got there
   worklist = [];     // These ants are "currently moving"

   // All the ants begin at vertex v at time step zero
   worklist.add((source, 0));

   while (worklist.hasWork()) {
      (v, time_to_v) = next();

      // Since a cluster of ants got to v, we dispatch new ants
      for (u : v.neighbors()) {
         // When does a cluster of ants get to u? How does it change?
         (u, time_to_u) = worklist.get(u);
         // w(v, u) is the edge weight from v to u
         time_from_v_to_u = w(v, u);
         to_u = min(time_to_u, time_to_v + time_from_v_to_u);
         worklist.add((u, to_u));
      }
   }
   return dist;
}
```

Example 2



a	b	c	d	e	f	g
0	6	2	1	2	4	6

Okay, and to implement this?

- Our sorted list is slow; so, replace it with a **priority queue**.

Okay, and to implement this?

- Our sorted list is slow; so, replace it with a **priority queue**.
- We need a way of “changing the priority of an element”

Okay, and to implement this?

- Our sorted list is slow; so, replace it with a **priority queue**.
- We need a way of “changing the priority of an element”

Remember, decreaseKey? That’s exactly what it does!

To make that work, we need to store a reference to the index/vertex in some dictionary.

```
dijkstra(G, source) {
   dist = new Dictionary();
   worklist = [];
   for (v : V) {
      if (v == source) { dist[v] = 0; }
      else             { dist[v] = ∞; }
      worklist.add((v, dist[v]));
   }

   while (worklist.hasWork()) {
      v = next();
      for (u : v.neighbors()) {
         dist[u] = min(dist[u], dist[v] + w(v, u));
         worklist.decreaseKey(u, dist[u]);
      }
   }

   return dist;
}
```

Example 3