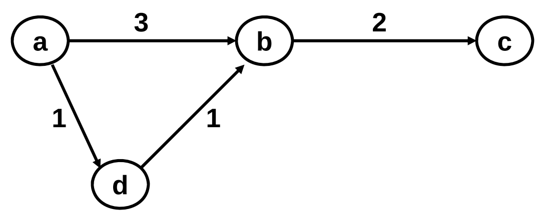
1.204 Lecture 12

Greedy/dynamic programming algorithms:
Shortest paths

Shortest paths in networks

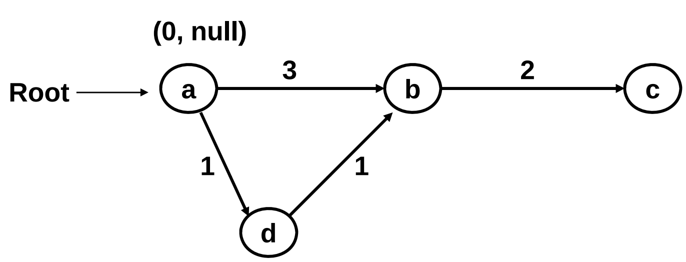
- Shortest path algorithm:
  - Builds shortest path tree
  - From a root node
  - To all other nodes in the network.
- All shortest path algorithms are labeling algorithms
  - Labeling is process of finding:
    - Cost from root at each node (its label), and
    - Predecessor node on path from root to node
- Algorithm needs two data structures:
  - Find arcs out of each node
    - Array-based representation of graph itself
  - Keep track of candidate nodes to add to shortest path tree
    - Candidate list (queue) of nodes as they are:
      - Discovered and/or
      - Revisited

Example

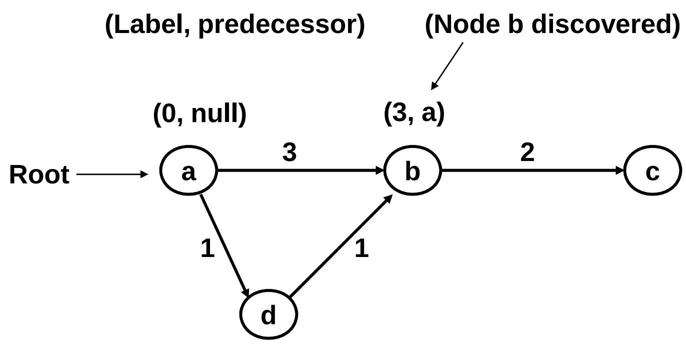


Example

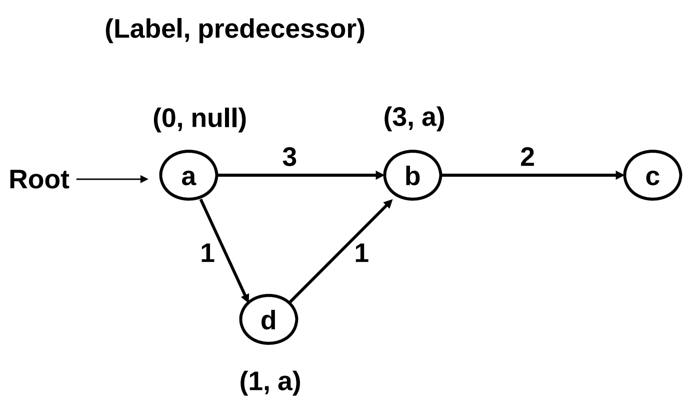
(Label, predecessor)



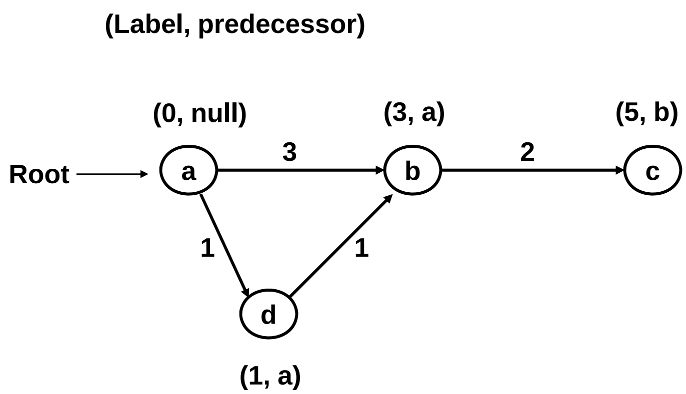
Example



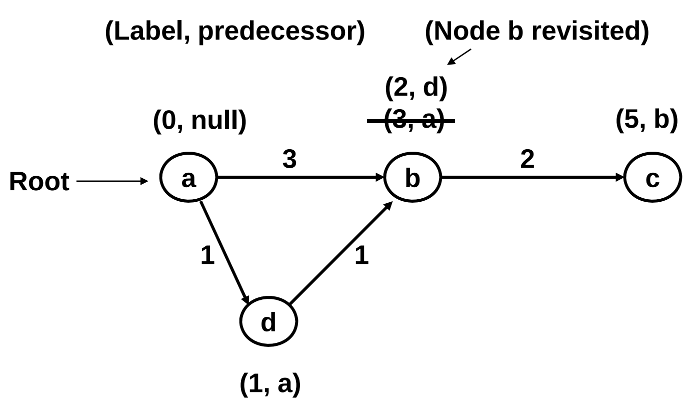
Example



Example

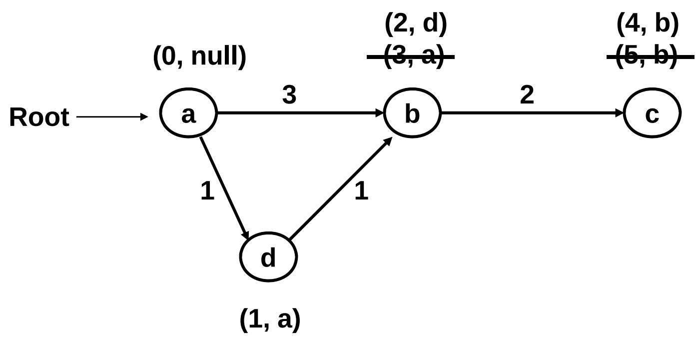


Example



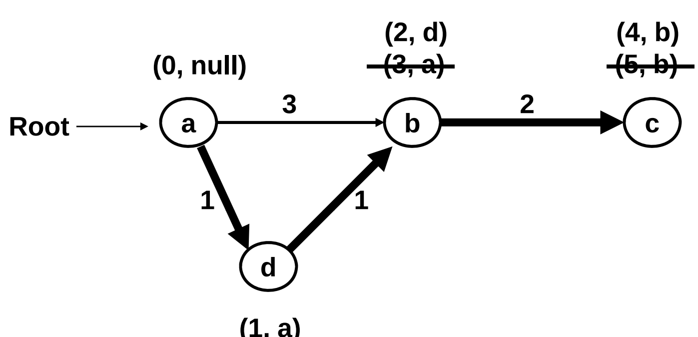
Example

(Label, predecessor)



Example

(Label, predecessor)



Orange (thick) arcs are shortest path tree with distances and predecessors

Types of shortest path algorithms

- Label setting. If arc is added to shortest path tree, it is permanent.
  - Dijkstra (1959) is standard label setting algorithm.
  - Fastest for dense networks with average out-degree ~> 30
  - Requires heap or sorted arcs
- Label correcting. If arc is added to tree, it may be altered later if better path if found.
  - Series of algorithms, each faster, depending on how candidate list is managed. Fastest when out-degree ~< 30
    - Bellman-Ford (1958). New node discovered always put on back of candidate list and next node taken from front of list. (Queue)
    - D'Esopo-Pape (1974). New node put on front of candidate list if it has been on list before, otherwise on back ('Sharp labels')
    - Bertsekas (1992). New node put on front of candidate list if its label smaller than current front node, otherwise on back
    - Hao-Kocur (1992). New node is put on front of list if it has been on list before. Otherwise it is put on back of list if label > front node and on front of list if smaller. ('Sharp labels')
- Previous example was label correcting
  - Label setting requires looking at shortest arc at every step

Computational results

CPU times (in milliseconds) on road networks
(HP9000-720 workstation, 1992)

Nodes	*Arcs*	*Visit*	*Dijkstra*	*Bellman*	*D'Esopo*	*Bertsekas*	*Hao-Kocur*
5199	14642	13	98	42	37	21	19
28917	64844	96	1192	590	125	144	104
115812	250808	459	9007	5644	619	789	497
119995	271562	488	13352	7651	708	1183	596
187152	410338	779	27483	15067	1184	1713	926

Times are 300x faster today (hardware- Moore's Law).
Also, slow implementations run 100x slower (lists, sorts, etc.)

Worst case, average performance

Algorithm	Worst case	Average case
Label-correcting	$O(2^a)$ Bellman-Ford is O(an)	~O(a)
Label-setting	$O(a^2)$ in simple version O(a lg n) with heap	O(a lg n) with heap

It takes a real sense of humor to use an $O(2^n)$ algorithm in 'hard real-time' applications in telecom, but it works!

Label correctors with an appropriate candidate list data structure in fact make very few corrections and run fast

Tree (D,P) and list (CL) arrays

Array	Definition	Description
D	Distance (output)	Current best distance from root to node i
P	Predecessor (output)	Predecessor of node in shortest path (so far) from root to node i
CL	Candidate list (internal)	List of nodes that are eligible to be added to the growing shortest path tree. CL[i]= NEVER_ON_CL if node has never been on CL ON_CL_BEFORE if node has been on CL before j if node i is now on CL and j next END_OF_LIST if node is last on CL

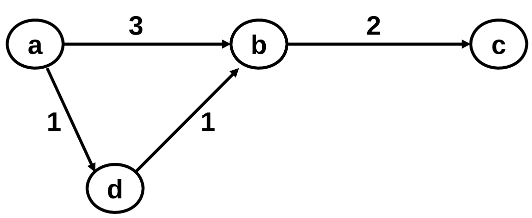
6 1-D arrays for input, output, data structures:

Graph input and data structure:	Head, To, Dist
Tree output and data structure:	D, P
Candidate list to control algorithm:	CL

Label correcting algorithm: Hao-Kocur

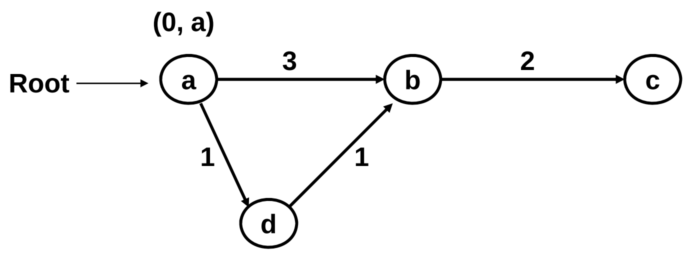
- Initialize:
  - P: Shortest path tree= {root}
  - D: Distance from root to all nodes= “infinity”
  - CL: Candidate list= {root}, at end of list
- At each step:
  - A node i is removed from front of CL
  - For each arc ij leaving node i where the distance from the root to node j is shortened by going via node i, add node j to CL:
    - If CL[j] == ON_CL_BEFORE, add j to front of CL
    - If CL[j] == NEVER_ON_CL:
      - If D[j] < D[front node on CL], add j to front of CL
      - Else add j to end of CL
    - If CL[j] > 0, j is now on CL. Do nothing.
    - If CL[j] == END_OF_LIST, terminate algorithm

Example

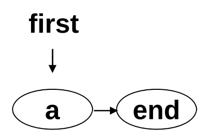


i	*P*	*D*	*CL*
a	EMPTY	MAXCOST	NEVER
b	EMPTY	MAXCOST	NEVER
c	EMPTY	MAXCOST	NEVER
d	EMPTY	MAXCOST	NEVER

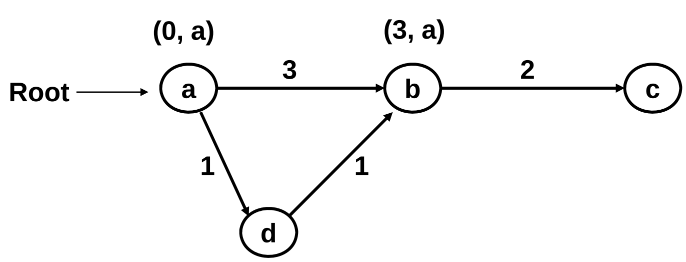
Example



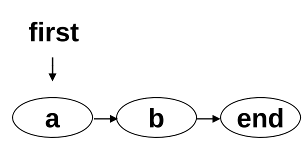
i	*P*	*D*	*CL*
a	a	0	END
b	EMPTY	MAXCOST	NEVER
c	EMPTY	MAXCOST	NEVER
d	EMPTY	MAXCOST	NEVER



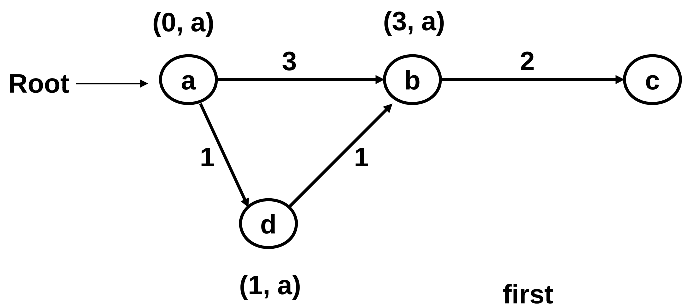
Example



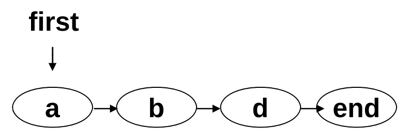
i	*P*	*D*	*CL*
a	a	0	b
b	a	3	END
c	EMPTY	MAXCOST	NEVER
d	EMPTY	MAXCOST	NEVER



Example

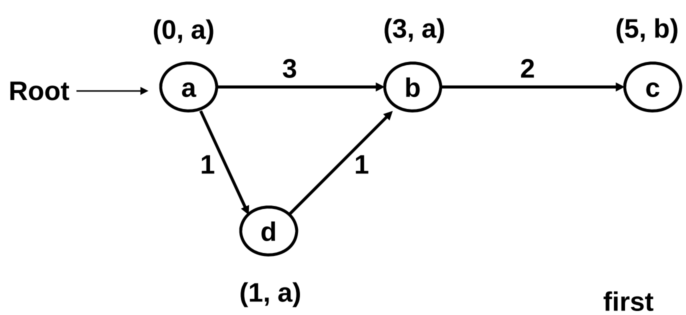


i	*P*	*D*	*CL*
a	a	0	b
b	a	3	d
c	EMPTY	MAXCOST	NEVER
d	a	1	END

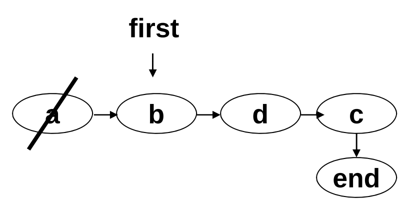


Node d on rear because
D[d] > D[first] = D[a]

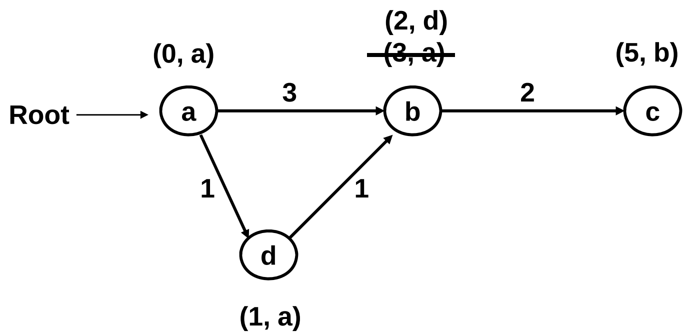
Example



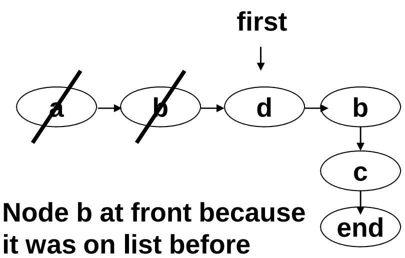
i	*P*	*D*	*CL*
a	a	0	ON_BEF
b	a	3	d
c	b	5	END
d	a	1	c



Example

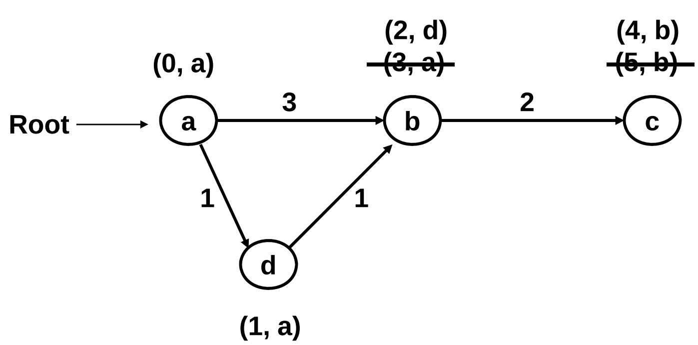


i	*P*	*D*	*CL*
a	a	0	ON_BEF
b	d	2	c
c	b	5	END
d	a	1	b

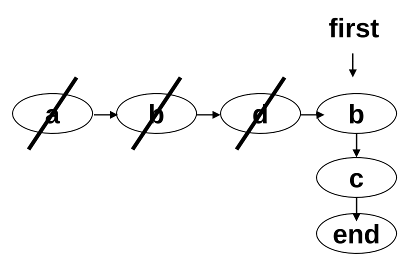


Node b at front because
it was on list before

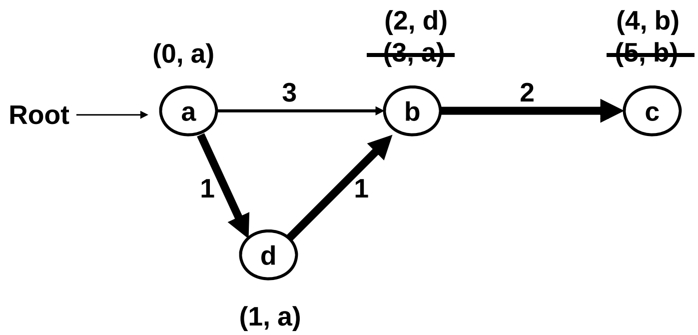
Example



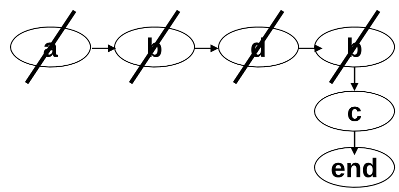
i	*P*	*D*	*CL*
a	a	0	ON_BEF
b	d	2	c
c	b	4	END
d	a	1	ON_BEF



Example



i	*P*	*D*	*CL*
a	a	0	ON_BEF
b	d	2	ON_BEF
c	b	4	END
d	a	1	ON_BEF



Code, p.1

```
public class Graph {   // Same as before, except add P, D data members
   private int to[];
   private int dist[];
   private int H[];
   private int nodes;
   private int arcs;
   private int[] D;    // Distance from root to node.
   private int[] P;    // Predecessor node on path from root

   // Constructor, readData() methods same as before
```

Code, p.2

```
public void shortHK(int root) {
      // Constants—could be in Graph as static
      final int MAX_COST= Integer.MAX_VALUE/2;
      final int EMPTY= Short.MIN_VALUE;
      final int NEVER_ON_CL= -1;
      final int ON_CL_BEFORE= -2;
      final int END_OF_CL= Integer.MAX_VALUE;
      D= new int[nodes];
      P= new int[nodes];
      int[] CL= new int[nodes];
      // Initialize
      for (int i=0; i < nodes; i++) {
        D[i]= MAX_COST;
        P[i]= EMPTY;
        CL[i]= NEVER_ON_CL; }
      D[root]= 0;
      CL[root]=  END_OF_CL;
      int lastOnList= root;
      int firstNode= root;

      // Continued on next page
```

Code, p.3

Manage CL

```
  do {
    int Dfirst= D[firstNode];
    for(int link=head[firstNode]; link<head[firstNode+1]; link++){
      int outNode= to[link];            // Loop thru arcs out of node
      int DoutNode= Dfirst + dist[link];
      if (DoutNode < D[outNode]) {      // Do something only if impvt
        P[outNode]= firstNode;
        D[outNode]= DoutNode;
        int CLoutNode= CL[outNode];
        if (CLoutNode==NEVER_ON_CL || CLoutNode==ON_CL_BEFORE) {
          int CLfirstNode= CL[firstNode];
          if (CLfirstNode != END_OF_CL &&       // Front of CL
            (CLoutNode==ON_CL_BEFORE || DoutNode<D[CLfirstNode])){
            CL[outNode]= CLfirstNode;
            CL[firstNode]= outNode; }
          else {                                // Back of CL
            CL[lastOnList]= outNode;
            lastOnList= outNode;
            CL[outNode]= END_OF_CL;  }  }  }  } // End for loop
    int nextCL= CL[firstNode];                // Go to next node
    CL[firstNode]= ON_CL_BEFORE;
    firstNode= nextCL;
  } while (firstNode < END_OF_CL); }  } // End do loop
```

Code, p.4

```
public void print() {
      System.out.println("i \tP \tD");
      for (int i=0; i < nodes; i++) {
        if (P[i] == Short.MIN_VALUE)
          System.out.println(i + "\t- \t" + D[i]);
        else
          System.out.println(i + "\t" + P[i] + "\t" + D[i]);
      }
  }

public static void main(String[] args) {
      Graph network= new Graph("src/dataStructures/graph.txt");
      System.out.println("\nDEP shortest path, root 0");
      network.shortDEP(0);
      network.print();
      System.out.println("\nHK shortest path, root 0");
      network.shortHK(0);
      network.print();
      System.out.println("\nDijkstra shortest path, root 0");
      network.shortDijkstra(0);
      network.print();
  }
```

Summary: Label correctors

- **Shortest path algorithm**
  - **22 lines of code, after initialization**
    - **Down from 200+ lines 25 years ago for d'Esopo-Pape**
  - **One addition operation, otherwise only increment, compare**
  - **3 data structures (queue/candidate list, network, tree) as arrays**
    - **They control the very simple algorithm very efficiently**
  - **Linked list would be too expensive**
    - **Memory allocation in small chunks is very slow**
  - **Separate data structures and algorithm would be too expensive**
    - **Method call overhead noticeable in real time algorithms**
  - **One preprocessing trick used by Hao-Kocur:**
    - **Sort arcs out of node by distance. Get a bit of 'Dijkstra effect'**

Label setting algorithm: Dijkstra

- **Dijkstra labels are permanent**
  - **Once set, they do not need to be corrected**
- **Greedy algorithm**
  - **Starts at an arbitrary node, which is the root of the tree**
  - **Puts arcs on a heap as they are discovered**
    - **Each arc's distance=**
      distance to its 'from node' from root + arc distance
  - **The algorithm deletes the top arc from the heap**
    - **If the 'to node' of the arc is not labeled, the arc becomes part of the shortest path tree**
    - **If the 'to node' is labeled, its destination node has already been labeled by a shorter path, and this arc is discarded**
  - **The algorithm terminates when all nodes are labeled**
    - **When (nodes -1) arcs have been added to the shortest path tree**
    - **Or when the heap is empty (if graph is not connected and all nodes are not reachable)**

Dijkstra code, p.1

```
public void shortDijkstra(int root) {
  final int MAX_COST= Integer.MAX_VALUE/2; // 'Infinite' initial
  final int EMPTY= Short.MIN_VALUE; // Flag for no value: -32767
  Heap g= new Heap(arcs);

  D= new int[nodes];               // Distance from root
  P= new int[nodes];               // Predecessor node from root

  for (int i=0; i < nodes; i++) {  // Initialize all nodes
    D[i]= MAX_COST;                // Initial label-> infinity
    P[i]= EMPTY;                   // No predecessor on path
  }

  MSTArc inArc= null;
  D[root]= 0;                    // Root is 0 distance from root
  P[root]= 0;                    // Root is its own predecessor
  for (int arc= head[root]; arc< head[root+1]; arc++)
    g.insert(new MSTArc(root, to[arc], dist[arc]));

  // Continued on next slide
```

Dijkstra code, p.2

```
    for (int i = 0; i < nodes-1; i++) {
      do {                            // Find arc to add to tree
        if (g.isEmpty()) return;      // Heap empty; done
        inArc= (MSTArc) g.delete();
      } while (P[inArc.to]!= EMPTY); // 'To' can't be in tree
      int inNode= inArc.to;           // Node added to tree
      P[inNode]= inArc.from;          // Predecessor is "from"
      D[inNode]= inArc.dist;          // Distance from root
      // Add arcs to heap from newly added node
      for (int arc= head[inNode]; arc< head[inNode+1]; arc++)
         g.insert(new MSTArc(inNode, to[arc],
               D[inNode] + dist[arc]));
    }
  }

// MSTArc class same as in PrimHeap
// print(), main() methods essentially the same
```

Shortest path algorithm usage

- Quicker to recompute than to retrieve from disk storage
- Label correcting algorithms are fastest for most problems
  - If average node degree > about 30, use Dijkstra
  - Dijkstra best in a few other special cases
- All pairs shortest path algorithms require a lot of storage
  - Usually you don't need all pairs (you may need many)
  - Label correcting algorithm is typically used, in a loop
- Can terminate early if looking for just one O-D path
  - Obvious in Dijkstra, requires care in label correctors
- Building blocks:
  - Shortest path uses graph, heap, set data structures
  - Network equilibrium (future topic) uses shortest path as building block
  - Branch and bound can use shortest paths as component, etc.
- Can integrate graphs, shortest paths with GIS (display, pan, zoom)
  - Integrate graphs and quadtrees
  - Can also integrate address lookups, etc.

Shortest path algorithm usage, p.2

- Negative edges (but no negative cycles)
  - Simple algorithm to convert to all costs> 0
    - Do one pass with label corrector
    - If negative cycle found, terminate
    - Add label difference between origin and destination nodes to negative arc costs
- Negative cycles
  - Use label corrector variation to detect
  - This is a different problem (e.g., arbitrage)!
- Kth shortest path, longest path problems, others
  - Combinatorial; often use dynamic programming