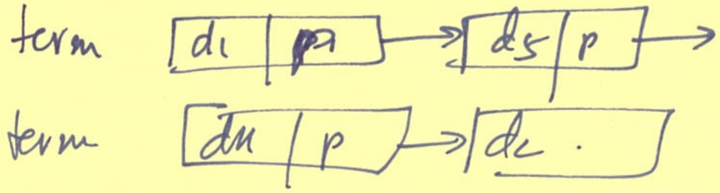
Web crawling : p. 72

provide one or more seed URLs as a string.
That will lead to many other web sites through their links. Obtain the content of these sites and store them.
How deep do you traverse : 2 or 3 deep depending on the application.
issues in web crawling : see p. 72

- An inverted index :
  - posting list one list for each term that appears in the collection.
- Posting list consists of individual postings: each of which consists of document id, payload
- Payload may contain a variety of information
  - most commonly term frequency,
  - position of every occurrence of the term
  - other properties of the term : was it highlighted appeared in the ~~table~~ title etc.
  - the ~~list~~ list could be sorted by term frequency or some rank of the document.



map (docid n , doc d)

H ← new AssociativeArray

for all terms t ∈ doc do

H{t} ← H{t} + 1

for all term t ∈ H do

EMIT (term t, posting ⟨n, H{t}⟩)

—

reduce (term t , posting list (n, H{t})

(n, f_1) (n, f_2) ..)

P ← new List

for all posting (a,f) p in posting List

Append (P, ⟨a, f⟩)

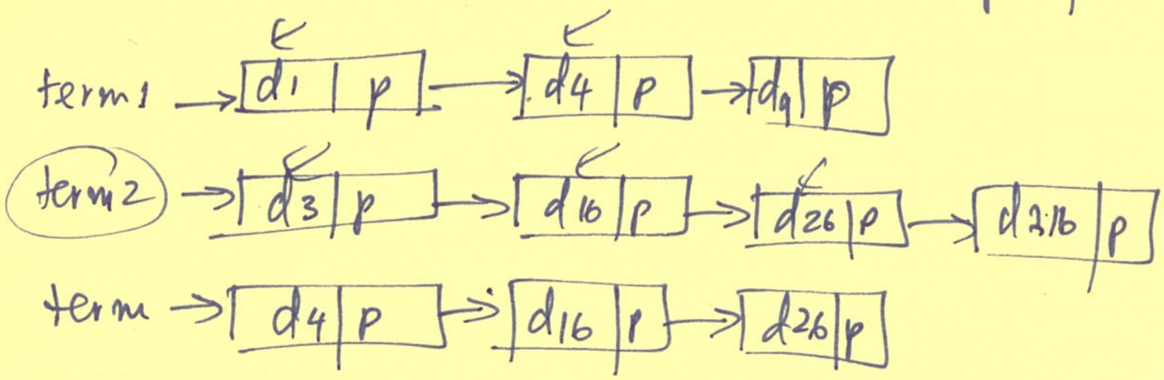
Sort P.

EMIT (term t, posting P)

Toy example

April 13, 2016
Spring 2016

CSE ~~4~~87/587



p : payload

- frequency — # of times it occurs
- list of positions
- attributes of term

term → list is sorted by the document "rank"

```
class Mapper
  ~~Mapper (docid n, doc d)~~
  method map (docid n, doc d)
    H ← new AssociativeArray
    for all term t ∈ doc d do
        H{t} ← H{t} + 1
    for all term t ∈ H do
        Emit (term t, posting <n, H{t}>)
```

d23 | ...

```
class Reducer
  method reduce (term t, posting [<n1, f1> ... <n?, f?>])
    P ← new List
    for all posting <a, f> in posting do
      Append (P, <a, f>)
!   SORT (P)
    Emit (term t, posting P)
```

term1 → d1 | .. → d20? | → ...

April 13, 2016

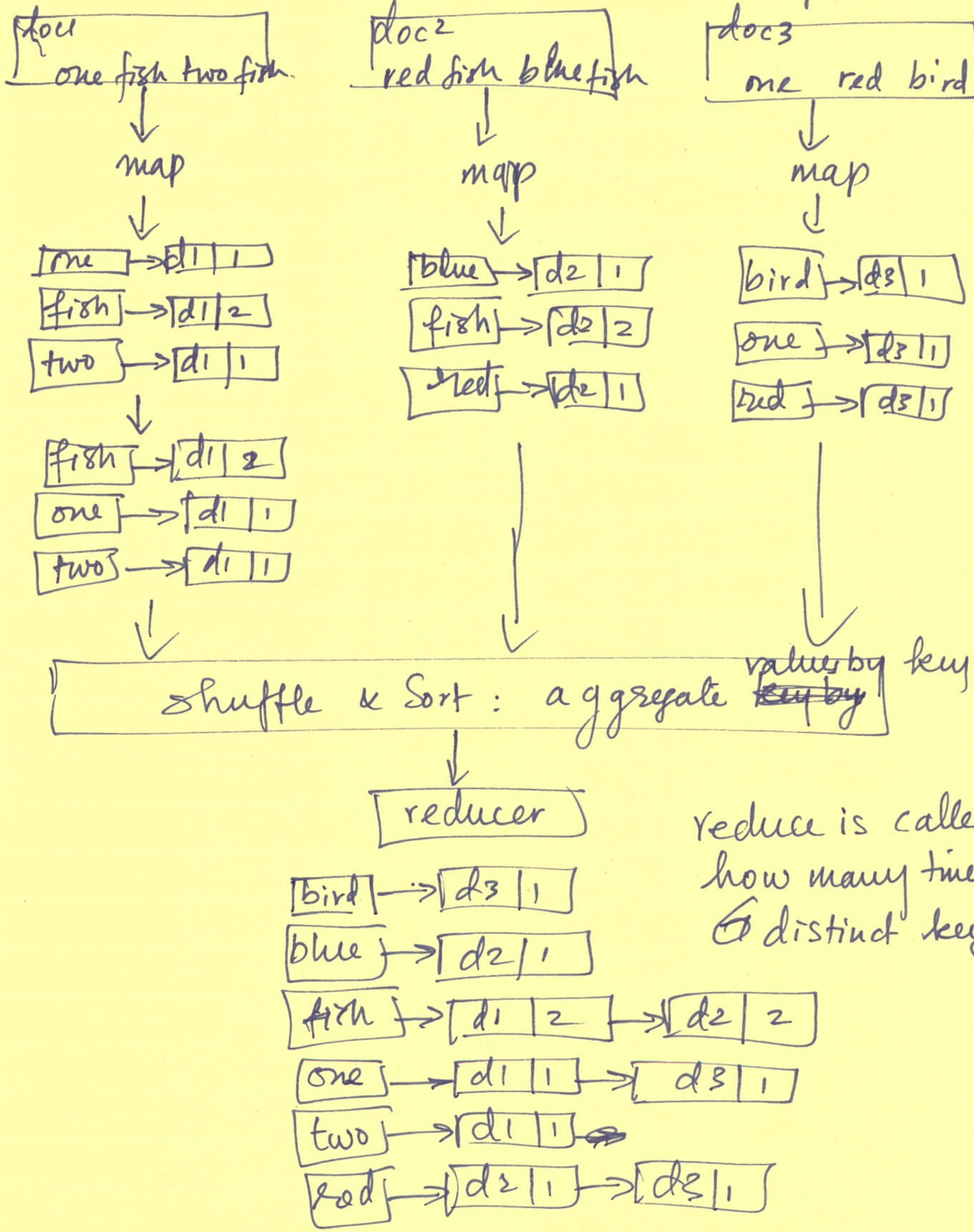
issue:
SORT is done by the reducer.
output is sorted by the "term" that is the key
not by document.

<u>solution</u>:
key-value switch pattern

$\langle \langle \text{term } t, \text{doc } n \rangle, \text{tf}(t) \rangle$

moving the "sort" out of the reducer.

April 13, 2016



reduce is called how many times?
6 distinct keys

April 13, 2016

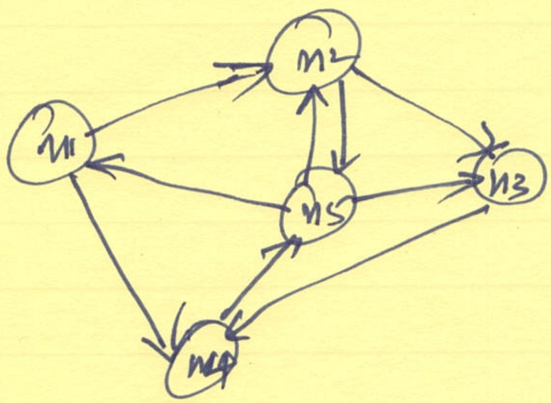
Graph: G (V, E)
nodes edges

connections can be directed or undirected

social network
path planning
roadways
Cable layout   Read the book

simple connected ~~gra~~ directed graph.

matrix: adjacency matrix



	n1	n2	n3	n4	n5
n1	0	1	0	[illegible]	0
n2	0	0	1	0	[illegible]
n3	0	0	0	1	0
n4	0	0	0	0	1
n5	1	1	1	0	

representation ⇓

million nodes billion nodes

matrix : sparse matrix ⟹ lists

n1	[n2 n4]
n2	[n3 n5]
n3	[n4]
n4	[n5]
n5	[n1 n2 n3]

adjacency lists

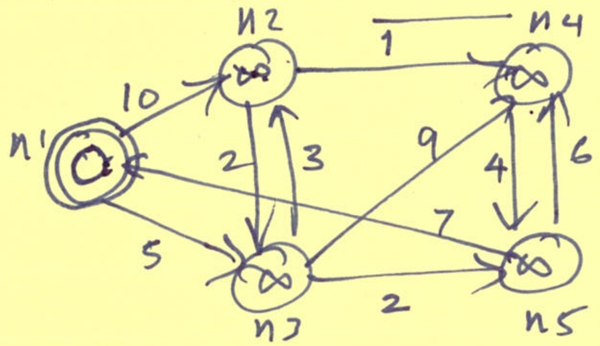
Breadth first search

April 13, 2016

Dijkstra(G, w, s)
  d[s] = 0;   d -> distance   G[V, E]
  for all vertex v in V (except first)
    d[v] ← ∞
  Q ← {V}
  while (Q != ~~empty~~)
    u ← ExtractMin(Q);
    for all vertex v ∈ u.adjacencylist do
      if d[v] > d[u] + w(u, v) then
        d[v] ← d[u] + w(u, v)

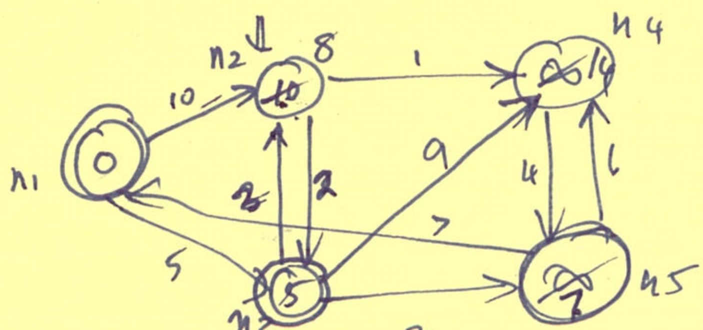
d are lables on the "nodes" computed by the algorithm

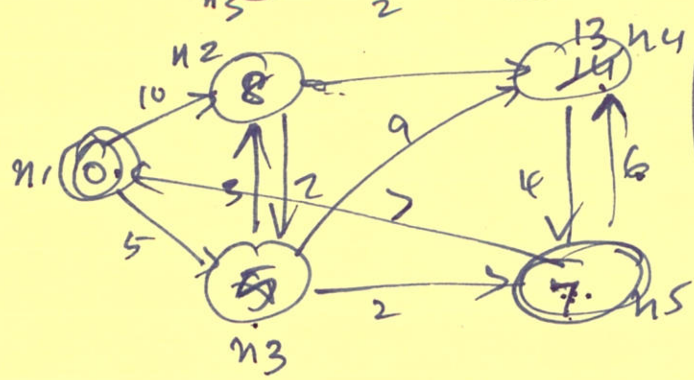
revisit

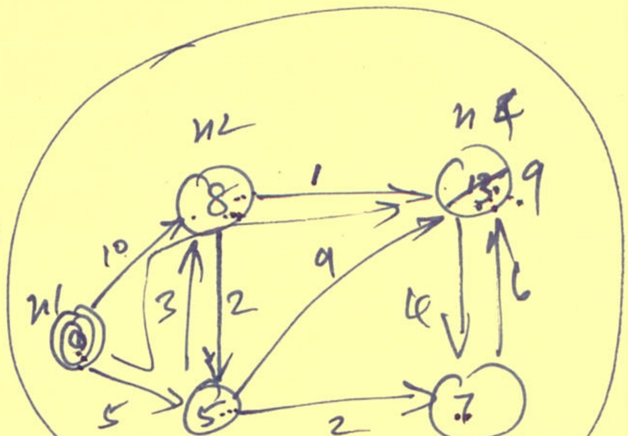


⟹ n1 Starting point

label all vertices with minimum weight from starting node







min cost ~~weightage~~ of from n1 → othernodes

```
DIJKSTRA(G, w, s)
    d[s] ← 0
    for all vertex v ∈ V do
        d[v] ← ∞
    Q ← {V}
    while Q ≠ ∅ do
        u ← EXTRACTMIN(Q)
        for all vertex v ∈ u.ADJACENCYLIST do
            if d[v] > d[u] + w(u, v) then
                d[v] ← d[u] + w(u, v)
```

except some vertex

Figure 5.2: Pseudo-code for Dijkstra's algorithm, which is based on maintaining a global priority queue of nodes with priorities equal to their distances from the source node. At each iteration, the algorithm expands the node with the shortest distance and updates distances to all reachable nodes.

As a refresher and also to serve as a point of comparison, Dijkstra's algorithm is shown in Figure 5.2, adapted from Cormen, Leiserson, and Rivest's classic algorithms textbook [41] (often simply known as *CLR*). The input to the algorithm is a directed, connected graph $G = (V, E)$ represented with adjacency lists, w containing edge distances such that $w(u, v) \geq 0$, and the source node s. The algorithm begins by first setting distances to all vertices $d[v], v \in V$ to ∞, except for the source node, whose distance to itself is zero. The algorithm maintains Q, a global priority queue of vertices with priorities equal to their distance values d.

Dijkstra's algorithm operates by iteratively selecting the node with the lowest current distance from the priority queue (initially, this is the source node). At each iteration, the algorithm "expands" that node by traversing the adjacency list of the selected node to see if any of those nodes can be reached with a path of a shorter distance. The algorithm terminates when the priority queue Q is empty, or equivalently, when all nodes have been considered. Note that the algorithm as presented in Figure 5.2 only computes the shortest distances. The actual paths can be recovered by storing "backpointers" for every node indicating a fragment of the shortest path.

A sample trace of the algorithm running on a simple graph is shown in Figure 5.3 (example also adapted from *CLR*). We start out in (a) with n_1 having a distance of zero (since it's the source) and all other nodes having a distance of ∞. In the first iteration (a), n_1 is selected as the node to expand (indicated by the thicker border). After the expansion, we see in (b) that n_2 and n_3 can be reached at a distance of 10 and 5, respectively. Also, we see in (b) that n_3 is the next node selected for expansion. Nodes we have already considered for expansion are shown in black. Expanding n_3, we see in

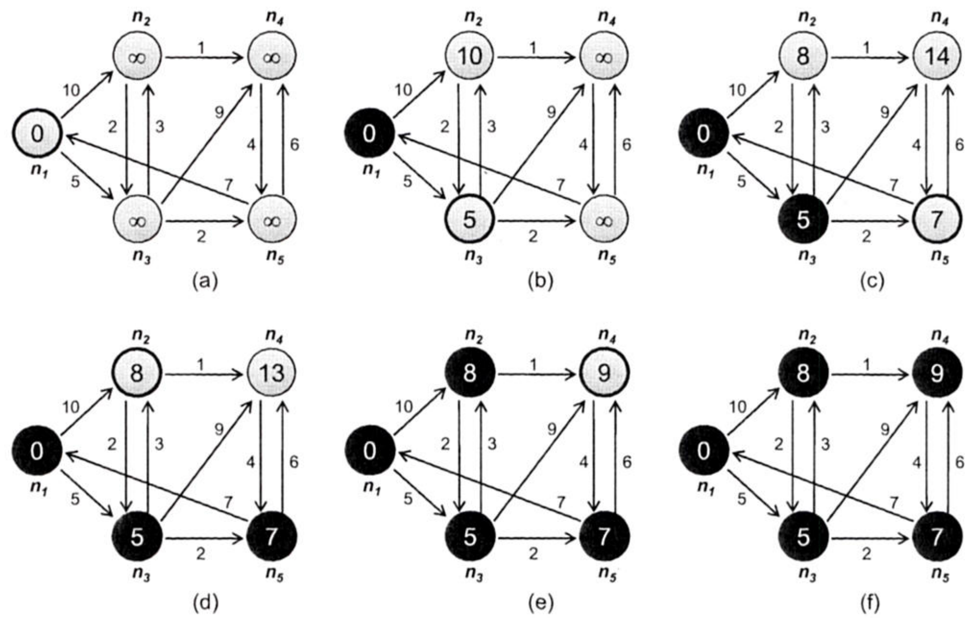


Figure 5.3: Example of Dijkstra's algorithm applied to a simple graph with five nodes, with n_1 as the source and edge distances as indicated. Parts (a)–(e) show the running of the algorithm at each iteration, with the current distance inside the node. Nodes with thicker borders are those being expanded; nodes that have already been expanded are shown in black.

(c) that the distance to n_2 has decreased because we've found a shorter path. The nodes that will be expanded next, in order, are n_5, n_2, and n_4. The algorithm terminates with the end state shown in (f), where we've discovered the shortest distance to all nodes.

The key to Dijkstra's algorithm is the priority queue that maintains a globally-sorted list of nodes by current distance. This is not possible in MapReduce, as the programming model does not provide a mechanism for exchanging global data. Instead, we adopt a brute force approach known as parallel breadth-first search. First, as a simplification let us assume that all edges have unit distance (modeling, for example, hyperlinks on the web). This makes the algorithm easier to understand, but we'll relax this restriction later.

The intuition behind the algorithm is this: the distance of all nodes connected directly to the source node is one; the distance of all nodes directly connected to those is two; and so on. Imagine water rippling away from a rock dropped into a pond—that's a good image of how parallel breadth-first search works. However, what if there are multiple paths to the same node? Suppose we wish to compute the shortest distance

```
class Mapper
    method Map(nid n, node N)
        d ← N.Distance
        Emit(nid n, N)                              ▷ Pass along graph structure
        for all nodeid m ∈ N.AdjacencyList do
            Emit(nid m, d + 1)                      ▷ Emit distances to reachable nodes

class Reducer
    method Reduce(nid m, [d1, d2, ...])
        d_min ← ∞
        M ← ∅
        for all d ∈ counts [d1, d2, ...] do
            if IsNode(d) then
                M ← d                               ▷ Recover graph structure
            else if d < d_min then                  ▷ Look for shorter distance
                d_min ← d
        M.Distance ← d_min                          ▷ Update shortest distance
        Emit(nid m, node M)
```

Figure 5.4: Pseudo-code for parallel breath-first search in MapReduce: the mappers emit distances to reachable nodes, while the reducers select the minimum of those distances for each destination node. Each iteration (one MapReduce job) of the algorithm expands the "search frontier" by one hop.

For more serious academic studies of "small world" phenomena in networks, we refer the reader to a number of publications [61, 62, 152, 2]. In practical terms, we iterate the algorithm until there are no more node distances that are ∞. Since the graph is connected, all nodes are reachable, and since all edge distances are one, all discovered nodes are guaranteed to have the shortest distances (i.e., there is not a shorter path that goes through a node that hasn't been discovered).

The actual checking of the termination condition must occur outside of MapReduce. Typically, execution of an iterative MapReduce algorithm requires a non-MapReduce "driver" program, which submits a MapReduce job to iterate the algorithm, checks to see if a termination condition has been met, and if not, repeats. Hadoop provides a lightweight API for constructs called "counters", which, as the name suggests, can be used for counting events that occur during execution, e.g., number of corrupt records, number of times a certain condition is met, or anything that the programmer desires. Counters can be defined to count the number of nodes that have distances of ∞: at the end of the job, the driver program can access the final counter value and check to see if another iteration is necessary.

Graphs are ubiquitous: ~~bio~~ pathways, highways, friends, links of web sites.
flow of emails, messages

Graphs are characterized by nodes and ~~of~~ edges.
Vertices links

Connected: connection can be directed or undirected.

Graphs are applied to many real-world problems:

- graph search and path planning
- graph clustering: identifying communities in social networks
- Minimum spanning trees: subset of the ~~of~~ edges that minimises the sum of edge weights.
- Bipartite graph matching: matching two graphs: employers with potential employees
- Maximum flow:
- identify special nodes: ~~single~~

Simple directed graph:

	n1	n2	n3	n4	n5
n1	0	1	0	1	0
n2	0	0	1	0	1
n3	0	0	0	1	0
n4	0	0	0	0	1
n5	1	1	1	0	0

Dijkstra's algorithm:

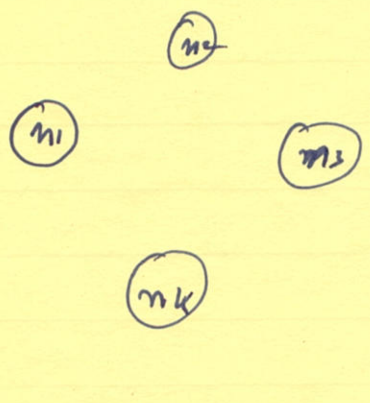
```
Dijkstra(G, w, s)      s: source node

  d(s) ← 0
  for all vertex v ∈ V do
      d(s) ← ∞
  Q ← {V}
  ~~which~~ while Q ≠ 0 do
      u ← ExtractMin(Q)
      for all vertex v ∈ u.AdjacentList do
          if d[v] > d[u] + w(u,v) then
              d[v] ← d[u] + w(u,v)
```

Key:
Priority queue of nodes sorted by distances

$G(V, E)$
s source node
w $\quad w(u,v) \geq 0$

Baseline algorithm: +1 source node → adjacent node



1 Starting point

output of mapper

```
1 0 2:3:
2 10000 3:4:
3 10000 2:4:5:
4 10000 5:
5 10000 1:4:
=================
1       0 2:3:
2       1 3:4:
3       1 2:4:5:
4       10000 5:
5       10000 1:4:
=====================
1       0 2:3:
2       1 3:4:
3       1 2:4:5:
4       2 5:
5       2 1:4:
===========================
1       0 2:3:
2       1 3:4:
3       1 2:4:5:
4       2 5:
5       2 1:4:
=================================
```

<1 [1 0 2:3:] <2 1> <3 1>
<2 [10000 3:4:] <3 10001> <4 10001>
<3 [10000 2:4:5] <2 10001> <4 10001> <5 10001>
<4 [10000 5:] <5 10001>
<5 [10000 1:4:] <1 10001> <4 10001>

input of reducer

<1 [1 0 2:3:] 1 1001>
<2 [10000 3:4:] 1 10001>
<3 [10000 2:4:5] 1 10001>
<4 [10001 5:] 10001 10001 10001>
<5 [10000 1:4:] 10001 10001>

key

1 [1 0 2:3:] <2 1> <3 1>
2 [1 3:4:] <3 2> <4 2>
3 [1 2:4:5:] <2 2> <4 2> <5 2>
4 [10000 5:] <5 10001>
5 [10000 1:4] <1 10001> <4 10001>

<1 [1 0 2 3]>
<2 [1 3:4:]>
<3 [1 2:4:5]>
<4 [10000 5:]>
<5 [10000 1:4:]>

4 2   4 10001

1 [1 0 2:3:]
2 [1 3:4:]
3 [1 2:4:5:]
4 2 5:
5 2 1:4: