**Breadth-First traversal**

BFS(*G*,*v*) **is**

let *Q* be a queue

*Q*.enqueue(*v*)

label *v* as visited

**while** *Q* is not empty

*v* = Q.peekFront()

v.dequeue()

**for all** edges from *v* to *w* **in** *G*.adjacentEdges(*v*) **do**

**if** *w* is not labeled as visited

*Q*.enqueue(*w*)

label *w* as visited

**Depth-First traversal**

DFS-iterative(*G*,*v*):

let *S* be a stack

*S*.push(*v*)

**while** *S* is not empty

*v* = *S*.peek() (then S.pop())

**if** *v* is not labeled as discovered:

label *v* as discovered

**for all** edges from *v* to *w* **in** *G*.adjacentEdges(*v*) **do**

*S*.push(*w*)

**Kruskal's algorithm:**

sort the edges of G in increasing order by length

keep a subgraph S of G, initially empty

for each edge e in sorted order *(OR until you have n-1 edges)*

*if the endpoints of e are disconnected in S)*

add e to S (if no cycles are created)

return S

// Determines a minimum spanning tree for a weighted, connected

// undirected graph whose weights are nonnegative, beginning with any v

**Prim’s algorithm (v: Vertex)**

mark vertex v as visited and include it in minimum spanning tree

**while** (there are unvisited vertices)

find the least-cost edge (v,u) from a visited vertex to some u

mark as visited

add the vertex u and edge (v, u) to the minimum spanning tree

**Dijkstra’s algorithm:**

Given a graph, G, with edges E of the form (v1, v2) and vertices V, and a

source vertex, s

dist : array of distances from the source to each vertex

prev : array of pointers to preceding vertices

i    : loop index

F    : list of finished vertices

U    : list or heap unfinished vertices

/\* Initialization: set every distance to INFINITY until we discover a path \*/

for i = 0 to |V| - 1

    dist[i] = INFINITY

    prev[i] = NULL

end

/\* The distance from the source to the source is defined to be zero \*/

dist[s] = 0

/\* This loop corresponds to sending out the explorers walking the paths, where

\* the step of picking "the vertex, v, with the shortest path to s" corresponds

\* to an explorer arriving at an unexplored vertex \*/

**while**(F is missing a vertex)

   pick the vertex, v, in U with the shortest path to s

   add v to F

**for each edge of v**, (v1, v2)

        /\* The next step is sometimes given the confusing name "relaxation"

**if**(dist[v1] + length(v1, v2) < dist[v2])

            dist[v2] = dist[v1] + length(v1, v2)

            prev[v2] = v1

            possibly update U, depending on implementation

**External Sorting**

All of the algorithms we have studied so far have been internal sorts, that is, sorts that require the data to be entirely sorted in primary memory during the sorting process. We now turn our attention to external sorting, sorts that allow portions of the data to be stored in secondary memory (for example, a disk) during the sorting process.  External sorting is necessary when the files are too large to fit in memory.

**Merging Ordered Files**

A merge is the process that, given two files ordered on a given key, combine the files into one ordered file on the same given key.

File 1:  1, 3, 5                     File 2:  2, 4, 6, 8, 10  (input)

File 3:  1, 2, 3, 4, 5, 6, 8, 10                                (output)

**Merging Unordered Files**

Sort Merge Strategy

1. Divide the file into runs such that the size of a run is small enough to fit into main memory
2. Sort each run in main memory using a fast in-memory sorting algorithm
3. Merge the resulting runs together into successively bigger runs, until the file is sorted.

The series of consecutively ordered data in a file is known as a merge run. Many different merge concepts have been developed over the years. Here are three that are representative: