FIT2004

Algorithms and Data Structures

Weeks 4 – 7 Study Notes

Covers: Week 4 — Order Statistics & Quickselect | Weeks 5–6 — Graphs Basics & Traversal | Week 7 — Greedy Algorithms (Dijkstra, Prim, Kruskal)

Week 4 · Order Statistics and Selection

The k-th order statistic of a sequence of n elements is the k-th smallest element. The median is the ⌈(n+1)/2⌉-th order statistic. The selection problem asks us to find the k-th order statistic without fully sorting.

4.1 Finding Minimums and Maximums

Finding the minimum requires n − 1 comparisons — each comparison eliminates at most one candidate. Finding minimum and maximum separately requires 2n − 2 comparisons, but we can do better.

Trick: Find min and max simultaneously in ~1.5n comparisons

Process elements in pairs. For each pair:

1. Compare the two elements of the pair against each other (1 comparison).
2. Compare the larger with the running maximum (1 comparison).
3. Compare the smaller with the running minimum (1 comparison).

Each pair costs 3 comparisons instead of 4. Total: roughly 3n/2 comparisons.

This is provably optimal — 1.5n is the lower bound for this problem.

4.2 Quickselect

Quickselect finds the k-th smallest element using the same partition idea as Quicksort. After partitioning, only recurse on the side that contains position k — the other side is irrelevant and skipped entirely.

```
function QUICKSELECT(array[lo..hi], k)
  if hi > lo then
    pivot = array[lo]
    mid = PARTITION(array[lo..hi], pivot)  // e.g. DNF partition
    if k < mid then
      return QUICKSELECT(array[lo..mid-1], k)
    else if k > mid then
      return QUICKSELECT(array[mid+1..hi], k)
    else
      return array[k]
```

```
  else
    return array[k]
```

Case	Time	Note
Best	O(n)	Pivot always median — one recursive call on n/2
Average	O(n)	Expected O(n) with random pivot selection
Worst	O(n²)	Pivot always min or max — degrades like Quicksort

4.3 Randomised Pivot Selection

Selecting the pivot uniformly at random gives expected O(n) run time. The formal proof shows $E[T(n)] \leq 4n$ by induction: choosing $C = 4$, the inductive step gives $T(n) \leq (3C/4 + 1)n < Cn$ for large enough n.

4.4 Median of Medians

Gives a deterministic worst-case O(n) Quickselect, at the cost of a larger constant than random pivots. The key insight: choose an approximate median good enough to guarantee a 30:70 worst-case split.

Algorithm: Median of Medians

1. Divide array into groups of 5 (last group may be smaller).
2. Find the exact median of each group using insertion sort (O(1) per group since size ≤ 5).
3. Recursively call Quickselect on the list of n/5 medians to find their median.
4. Use this median of medians as the pivot for the main partition.

```
function MEDIAN_OF_MEDIANS(array[lo..hi])
  if hi - lo < 5 then
    return MEDIAN_OF_FIVE(array[lo..hi])  // base: sort and return middle
  medians = []
  for i = lo to hi step 5 do
    j = min(i+4, hi)
    medians.append(MEDIAN_OF_FIVE(array[i..j]))
  return QUICKSELECT(medians, floor((length(medians)+1)/2))
```

Why this guarantees O(n) worst-case

Half the group medians are ≤ pivot (by definition of median of medians).

Half the group medians are ≥ pivot.

Each group has 2 elements on each side of its median, so:

- At least 3n/10 elements are ≤ pivot.
- At least 3n/10 elements are ≥ pivot.

So the pivot lies in the 30th–70th percentile — worst-case split is 70:30.

Recurrence: $T(n) = T(0.7n) + T(0.2n) + an \rightarrow T(n) = O(n)$ by induction.

The T(0.2n) term is the recursive call on the list of n/5 medians.

Using median of medians as the pivot for Quicksort gives worst-case O(n log n) Quicksort (called Balanced Quicksort). In practice, random pivots outperform it due to the large constant.

Week 5–6 · Graphs: Basics, Traversal, and Applications

Graphs model relationships between objects. Two of the largest technology companies in the world (Google and Facebook/Meta) are built around graphs. Graph algorithms are ubiquitous in CS, science, and industry.

5.1 Graph Definitions

Graph G = (V, E)

V = set of vertices (nodes). Usually n = |V|.

E = set of edges (arcs). Usually m = |E|.

An edge e connects two vertices u and v, written (u,v).

Types of graphs

Property	Options	Notes
Direction	`Directed / Undirected`	`(u,v) ≠ (v,u)` in directed; equal in undirected
Weights	`Weighted / Unweighted`	Weighted edge has value `w(u,v)`
Loops	`Allowed / Disallowed`	A loop is an edge from a vertex to itself
Multiplicity	`Simple / Multigraph`	Multigraph allows multiple edges between same pair
Density	`Dense / Sparse`	Dense: $\|E\| \approx \|V\|^2$ Sparse: $\|E\| \ll \|V\|^2$

Directed Acyclic Graph (DAG)

A directed graph with no directed cycles.

Used to represent tasks with prerequisites (courses, project management).

Key problems on DAGs: topological sorting and critical path.

5.2 Graph Representations

Adjacency Matrix

A |V| × |V| matrix. Entry a[i][j] = weight of edge (i,j), or 0/∞ if no edge exists. Space: O(|V|²). Adjacency check: O(1). Best for dense graphs.

Adjacency List

Array of lists: each vertex stores its neighbours (with weights if weighted). Space: O(|V| + |E|). Iterating over neighbours is efficient. Best for sparse graphs and most algorithms.

Edge List

Simple list of all edges. Space: O(|E|). Cannot efficiently check adjacency or iterate over a vertex's neighbours. Primarily used in Kruskal's algorithm where we need edges sorted by weight.

Representation	Space	Adjacency check	Best for
Adjacency Matrix	`O(|V|²)`	`O(1)`	`Dense graphs, frequent adjacency queries`
Adjacency List	`O(|V|+|E|)`	`O(degree)`	`Sparse graphs, most algorithms`
Edge List	`O(|E|)`	`O(|E|)`	`Kruskal's MST (needs sorted edges)`

5.3 Depth-First Search (DFS)

Explores as deep as possible from each vertex before backtracking. Implemented recursively using the call stack. Marks vertices as visited to avoid cycles. Time: O(|V| + |E|).

```
// Driver: call DFS on every unvisited vertex
function TRAVERSE(G = (V,E))
  visited[1..n] = false
  for each vertex u = 1 to n do
    if not visited[u] then DFS(u)

function DFS(u)
  visited[u] = true
  for each vertex v adjacent to u do
    if not visited[v] then DFS(v)
```

Applications of DFS (all run in O(|V|+|E|))

Connected components: Each new DFS call from an unvisited vertex finds one complete component.

Cycle detection: If DFS finds a visited vertex that is not its parent, there is a cycle.

Topological sort: Append each vertex AFTER visiting all its descendants; reverse the result.

Two-colouring/bipartite: Assign alternating colours; fail if two adjacent vertices share a colour.

Bridges and cut-points: Edges/vertices whose removal disconnects the graph.

Finding Connected Components — Algorithm 26

```
function CONNECTED_COMPONENTS(G = (V,E))
  component[1..n] = null;  num_comp = 0
  for each u = 1 to n do
    if component[u] = null then
      num_comp += 1;  DFS(u, num_comp)
  return num_comp, component
```

```
function DFS(u, comp_num)
  component[u] = comp_num
  for each v adjacent to u do
    if component[v] = null then DFS(v, comp_num)
```

Cycle Detection (undirected) — Algorithm 27

Pass the parent vertex p to DFS. If we find a visited vertex that is not p, it is a cycle. (We exclude p because the edge we arrived from must not count as a cycle.)

```
function DFS(u, p)   // p = vertex we came from
  visited[u] = true
  for each v adjacent to u do
    if visited[v] and v ≠ p then return true   // cycle found
    else if v ≠ p and DFS(v, u) = true then return true
  return false
```

5.3.4 Breadth-First Search (BFS)

Visits all vertices at distance k from the source before any at distance k+1. Uses a queue. Produces a shortest-path tree in unweighted graphs. Time: O(|V| + |E|).

```
function BFS(G = (V,E), s)
  visited[1..n] = false;  visited[s] = true
  queue = Queue();  queue.push(s)
  while queue is not empty do
    u = queue.pop()
    for each v adjacent to u do
      if not visited[v] then
        visited[v] = true;  queue.push(v)
```

BFS for unweighted shortest paths — Algorithm 29

Maintain dist[u] (hop distance from source) and pred[u] (predecessor on shortest path). Use pred[] to reconstruct any specific path afterwards.

```
function BFS_SHORTEST(G, s)
  dist[1..n] = ∞;  pred[1..n] = null
  queue = Queue();  queue.push(s);  dist[s] = 0
  while queue is not empty do
    u = queue.pop()
    for each v adjacent to u do
      if dist[v] = ∞ then
        dist[v] = dist[u] + 1;  pred[v] = u;  queue.push(v)

function GET_PATH(s, u, pred)
  path = [u]
  while u ≠ s do
    path.append(pred[u]);  u = pred[u]
  return reverse(path)
```

5.4 Shortest Path Properties

- Optimal substructure: every sub-path of a shortest path is itself a shortest path.
- Triangle inequality: $\delta(s,v) \leq \delta(s,u) + w(u,v)$ for any edge (u,v). You cannot beat the shortest path.

- Shortest paths are simple (no cycles) when all cycles have positive total weight.
- Negative-weight cycles make shortest paths undefined (can loop forever to decrease cost).

Three shortest path problem variants

Single-pair: Given u and v, find the shortest path between them.

Single-source: Given source s, find shortest paths from s to all other vertices.

All-pairs: Find shortest paths between all pairs u,v ∈ V.

In practice: single-pair is no easier than single-source. Single-source algorithms solve both.

5.5 Topological Sorting

Only defined for DAGs. A topological ordering places every vertex u before all vertices v such that (u,v) is an edge — satisfying all prerequisites. A DAG always has at least one topological ordering.

Kahn's Algorithm — Algorithm 31

Greedily process vertices with no remaining prerequisites. Maintain in-degree counts; when a vertex's count hits 0, it is ready.

```
function TOPOLOGICAL_SORT(G = (V,E))
  order = []
  ready = queue of all vertices with in-degree 0
  while ready is not empty do
    u = ready.pop();  order.append(u)
    for each edge (u,v) adjacent to u do
      decrement in-degree of v
      if in-degree of v = 0 then ready.push(v)
  return order
```

Time: O(|V| + |E|). If the order contains fewer than |V| vertices, the graph contains a cycle (not a DAG).

DFS-based Topological Sort — Algorithm 32

Append each vertex to an output array after all its descendants have been visited. Reverse the array at the end. Time: O(|V| + |E|).

```
function TOPOLOGICAL_SORT(G = (V,E))
  order = [];  visited[1..n] = false
  for each v = 1 to n do
    if not visited[v] then DFS(v)
  return reverse(order)

function DFS(u)
  visited[u] = true
  for each v adjacent to u do
    if not visited[v] then DFS(v)
  order.append(u)   // append AFTER visiting all descendants
```

5.6 Union-Find (Disjoint Set) Data Structure

Maintains a collection of disjoint sets, supporting two operations efficiently. Used for incremental connectivity: adding edges and answering whether two vertices are connected.

Operations

FIND(u): Return the representative (root) of the set containing u.

UNION(u,v): Merge the sets containing u and v.

CONNECTED(u,v) = (FIND(u) = FIND(v)).

Implementation: Disjoint Set Forest

Each set is a rooted tree. The root is the representative. parent[x] stores x's parent; root satisfies parent[x] = x.

```
function FIND(x)                          // Basic (no optimisations)
  if parent[x] = x then return x
  else return FIND(parent[x])

function UNION(x, y)
  parent[FIND(x)] = FIND(y)
```

Optimisation 1: Path Compression

During FIND, make every node on the path point directly to the root.

Flattens the tree — future FIND operations on the same path become O(1).

Implementation: one elegant recursive line:

```
function FIND(x)
  if parent[x] ≠ x then parent[x] = FIND(parent[x])
  return parent[x]
```

With path compression alone: O(m log n) for m operations.

Optimisation 2: Union by Rank

Maintain rank[x] = upper bound on height of tree rooted at x.

Always attach the smaller-rank tree under the larger-rank tree.

Prevents tall chains — trees stay balanced.

With union by rank alone: O(m log n) for m operations.

Combined: Path Compression + Union by Rank

Amortised cost: O(m α(n)) for m operations, where α(n) is the inverse Ackermann function.

$\alpha(n) \leq 4$ for any n that fits in the observable universe — effectively constant.

This is provably optimal — no disjoint-set data structure can do better.

```
function UNION(x, y)                      // Union by rank
  x = FIND(x);  y = FIND(y)
  if rank[x] < rank[y] then parent[x] = y
  else
    parent[y] = x
```

```
if rank[x] = rank[y] then rank[x] += 1
```

Method	Cost per operation	Notes
No optimisations	O(n) worst case	Long chains possible
Path compression only	O(log n) amortised	Flattens tree on each FIND
Union by rank only	O(log n) amortised	Balances tree on each UNION
Both (path + rank)	O(α(n)) amortised	Optimal — effectively O(1)

Week 7 · Greedy Algorithms

Greedy algorithms make locally optimal choices at each step and never reconsider them. Correctness is not automatic — it must be proved (typically via an invariant or exchange argument). This week covers Dijkstra's algorithm and two MST algorithms.

6.1 Edge Relaxation

Most shortest-path algorithms use relaxation as their core operation. We maintain a distance estimate dist[v] for each vertex — the length of the shortest path found so far. Relaxation updates this estimate when we find a better path.

```
function RELAX(edge (u,v))
  if dist[v] > dist[u] + w(u,v) then
    dist[v] = dist[u] + w(u,v)
    pred[v] = u
```

The predecessor array pred[] records the path. Once all distances are finalised, call GET_PATH(s, v, pred) (Algorithm 30) to reconstruct any specific path.

6.1 Dijkstra's Algorithm

Finds shortest paths from a single source s to all other vertices. Requires all edge weights to be non-negative.

Key idea

Greedy: always process the unvisited vertex with the smallest current distance estimate.

Correctness: because all weights are ≥ 0, once a vertex is popped from the priority queue, its distance is final — no future path can be shorter.

This is NOT true with negative weights (a later negative edge could create a shorter path).

```
function DIJKSTRA(G = (V,E), s)
  dist[1..n] = ∞;  pred[1..n] = 0;  dist[s] = 0
  Q = priority_queue();  Q.push(s, key=0)
```

```
while Q is not empty do
  u, key = Q.pop_min()
  if dist[u] = key then          // skip stale entries
    for each edge (u,v) adjacent to u do
      if dist[v] > dist[u] + w(u,v) then
        dist[v] = dist[u] + w(u,v)
        pred[v] = u
        Q.push(v, key=dist[v])   // insert updated entry
return dist[1..n], pred[1..n]
```

Implementation note: instead of decreasing keys in the priority queue, we insert a new entry and skip stale ones when popped. This avoids the need for a decrease-key operation and is simpler to implement.

Priority queue	Time complexity	Notes
Array (linear search)	$O(\|V\|^2)$	Simple; best for very dense graphs where $\|E\| \approx \|V\|^2$
Binary heap	$O((\|E\|+\|V\|) \log \|V\|)$	Best for sparse graphs; standard implementation
Fibonacci heap	$O(\|E\| + \|V\| \log \|V\|)$	Theoretically best; complex to implement

Correctness of Dijkstra's Algorithm

Claim: When Dijkstra's terminates, dist[v] = δ(s,v) for all v.

Proof by induction on the set S of vertices removed from the queue.

Base: dist[s] = 0 is correct (no negative weights).

Inductive step: suppose all distances in S are correct. Let u be the next vertex popped.

Suppose for contradiction that dist[u] > δ(s,u).

Then there is a shorter path s ⇝ u. Let x be the last vertex on this path in S, y the next.

Edge (x,y) was relaxed when x was removed. So dist[y] = δ(s,y) ≤ δ(s,u) < dist[u].

But then Dijkstra's would have popped y before u — contradiction.

Therefore dist[u] is correct. By induction, all distances are correct.

Critical step requires weights ≥ 0: otherwise δ(s,y) ≤ δ(s,u) might not hold.

6.2 Minimum Spanning Trees

Minimum Spanning Tree (MST)

A spanning tree of G = (V,E) is a connected acyclic subgraph containing all |V| vertices.

A spanning tree has exactly |V|−1 edges.

A minimum spanning tree (MST) is a spanning tree with minimum total edge weight.

MSTs are used in: network cabling, clustering, approximation algorithms.

Both Prim's and Kruskal's algorithms maintain the same invariant:

MST Invariant (shared by Prim's and Kruskal's)

At every iteration, the current set of selected edges T is a subset of some MST.

Proof (exchange argument): if the greedy choice e is not in any MST M,

adding e to M creates a cycle; removing any other edge on that cycle gives a new spanning tree.

Since e is the minimum such edge, the new spanning tree has weight ≤ w(M), so it is also an MST.

Hence T ∪ {e} is a subset of this new MST.

6.2.1 Prim's Algorithm

Grows a single tree from an arbitrary root. At each step, adds the minimum-weight edge connecting the current tree to a new vertex. Essentially Dijkstra's with a different key: edge weight rather than total distance from source.

```
function PRIM(G = (V,E), r)    // r = any root vertex
  dist[1..n] = ∞;  parent[1..n] = null;  dist[r] = 0
  Q = priority_queue(V, key(v) = dist[v])
  while Q is not empty do
    u = Q.pop_min()
    T.add(u, parent[u])          // add vertex and its cheapest edge
    for each edge (u,v) adjacent to u do
      if v not in T and dist[v] > w(u,v) then
        dist[v] = w(u,v)         // KEY: edge weight, not total distance!
        parent[v] = u
  return T
```

Dijkstra vs Prim: The only difference is the key. Dijkstra uses dist[u] + w(u,v) (total distance from source). Prim uses w(u,v) alone (cost of the single edge to the tree). Everything else is identical.

	Dijkstra's	Prim's
Builds	`Shortest path tree from source s`	`Minimum spanning tree`
Key used	`dist[u] + w(u,v) — total path cost`	`w(u,v) — single edge weight`
Result	`dist[v] = shortest path length`	`T = minimum spanning tree`
Time	`O((|E|+|V|) log |V|)`	`O(|E| log |V|) same`

6.2.2 Kruskal's Algorithm

Builds the MST by greedily adding edges in order of increasing weight, skipping any edge that would create a cycle. Uses union-find to check connectivity in near-constant time.

```
function KRUSKAL(G = (V,E))
  sort edges E by weight ascending
  forest = UnionFind.initialise(n)     // each vertex is its own component
  T = (V, {})                          // MST starts with no edges
  for each edge (u,v) in E do
    if forest.FIND(u) ≠ forest.FIND(v) then   // different components → no
cycle
      forest.UNION(u, v)
      T.add_edge(u, v)
  return T
```

Time: O(|E| log |E|) for sorting. Union-find operations take O(|E| α(|V|)) ≈ O(|E|). Total: O(|E| log |E|) = O(|E| log |V|) since log|E| = O(log|V|) in a simple graph.

Prim's vs Kruskal's

	Prim's	Kruskal's
Strategy	`Grow a single tree from a root`	`Merge forest components by cheapest edge`
Data structure	`Priority queue (min-heap)`	`Sorted edge list + union-find`
Time	`O(|E| log |V|)`	`O(|E| log |V|)`
Better for	`Dense graphs`	`Sparse graphs (union-find is cheap)`
Requires	`Connected graph`	`Works on disconnected graphs too`

FIT2004 · Weeks 4–7 Notes · End of Document