MT 2 'Cheat Sheet'

David Walter

Graph/ Search Material on next page.

Hashing: SUHA: s **$\Theta(1 + \alpha)$** — the 1 comes from applying the hash function and random access to the slot whereas the α comes from searching the list. This is equal to $O(1)$ if $\alpha = O(1)$, i.e., $m = \Omega(n)$.

Chaining: **$\alpha = n/m$** . We denote α as the load factor of the hash table. By ensuring m is large enough to keep α small, searches and deletions in a hash table using chaining take $O(1 + \alpha)$ or approximately $O(1)$ time.

Open addressing: alternative location in the hash table, uniform hashing assumption says that "Each key k is equally likely to hash to any of the m! permutations of h<0, 1, . . . ,(m − 1)i>, **, linear probing** tries consecutive locations one by one until it finds a free one, The expected number of lookups is $1/(1 - \alpha)$, which can become very large when α is close to 1.

Q: Consider an **open addressing** hash table with m slots, where collisions are handled using **linear probing**. Assume simple uniform hashing, and assume that the table is initially empty. The probability that the first two slots of the table are filled after the first two insertions is 3/m^2. TRUE There are m2 equally likely ways to map the first two keys into the table. Of those, the following three possibilities lead to the first two slots being filled: h(K1) = 0, h(K2) = 1; h(K1) = 1, h(K2) = 0; and h(K1) = 0, h(K2) = 0.

Double hashing: $hm = (h1(k) + i \cdot h2(k)) \bmod m$

Rolling hash: **Rabin-Karp**: hash() → computes the hash of the list. • append(val) → adds val to the end of the list. • skip(val) → removes the front element from the list, assuming it is val.

Bellman-Ford Analysis

```
for v in V:                      } O(V)          TOTAL: O(VE)
    v.d = ∞
    v.π = None
s.d = 0
for i from 1 to |V| − 1:                          } O(VE)
    for (u, v) in E:             } O(E)
        relax(u, v)  } O(1)
for (u, v) in E:                                  } O(E)
    if v.d > u.d + w(u, v):
        report that a negative-weight cycle exists
```

```
DIJKSTRA (G, W, s)      //uses priority queue Q
    Initialize (G, s)
    S ← φ
    Q ← V[G]      //Insert into Q
    while Q ≠ φ
        do u ← EXTRACT-MIN(Q)      //deletes u from Q
        S = S ∪ {u}
        for each vertex v ϵ Adj[u]
            do RELAX (u, v, w)    ← this is an implicit DECREASE_KEY operation
```

DFS: **not sorting**, run time: O(E + V), space: O(V), **stack**, maze, for DAGs, non-neg edges. **Not for shortest detection,** does: detect cycles(no back edges, no cycles), topological sort. DFS tree, V-1 edges in DFS tree. No cross-edges in dfs tree for undirected

BFS: **sort, unweighted** O(V + E), priority queue, undirected, thru 'levels', going from neighbors of s, to neighbors or neighbors, etc. BFS tree, V-1 edges in BFS tree

Triangle inequality: For any edge (u, v), we have δ(s, v) ≤ δ(s, u) +w(u, v). In English, the weight of the shortest path from s to v is no greater than the weight of the shortest path from s to u plus the weight of the edge from u to v

- v.d - The weight of the current shortest path from s to v.
- v.π - The parent vertex of v in the current shortest path.
- w(u, v) is the weight of the edge from vertex u to vertex v
- δ(u, v) is the weight of the shortest path from vertex u to vertex v

DAG-SP: **sorting DAGs,** O(V +E) , We know that there will be no path from a vertex v to another vertex u occurring before v in the topological order. The first step in finding the shortest path, therefore, should be to topologically sort the graph, in O(V +E) time. Now we go through each vertex starting from the beginning, and relax each outgoing edge. This will guarantee that each edge only need be relaxed once, making the total runtime O(V + E). A sample execution is shown in Figure 1.

Dijkstra: **sorting weighted, non-neg,** O(V logV + E), weighted- directed graphs, only guaranteed for non-negative weight edges. min-heap fibinacci (priority queue), calc from min & remove & put in 'visited' until, reach goal state/vertex. sorting, runtime: O(B + X|V| + D|E|), where B is the time to build the priority queue, X is the time for *EXTRACT-MIN* and D is the *DECREASE-KEY* operation, cant fix djikstra with making weights positive, Dijkstra's algorithm will return an incorrect answer only when a negative edge goes into an already processed node and there are edges coming out of that vertex

Fibinacci-heap: insert= θ(1), dec-key = θ(1), extract-min = θ(log n)

- **Bellman Ford**: **sorting, Negative weight edges**: For graphs that might have negative weight edges we use Bellman Ford's algorithm to find all shortest paths from a source vertex. Bellman Ford relaxes all O(|E|) edges of a graph exactly O(|V |) times, so the running time of the algorithm is O(|V ||E|). Bellman Ford can also detect if a graph has a negative weight cycle reachable from the source vertex or not, and if so, does not return a shortest path.