Chapter 9

Perfect Hashing and Graphs.

In the past week, we have seen how to store keys in hash tables so that the number of mapped keys in a specific cell is $O(1)$ in expectation. The table is constructed using a hashing function h : key space $\rightarrow$ $m$cells, randomly chosen from a universal hash function family. This function maps keys to cells, and in each cell, the keys are stored using a linked list. The cost of supported subroutines depends on the length of the list. We named any pair of different keys $x \neq y$ that are mapped to the same cell in the table, namely $h(x) = h(y)$, a collision.

Perfect hashing is a method to ensure that no collision occurs, it works only if all keys are given in advance and they are unique, meaning that the table doesn't support insertion. The idea is as follows, we sample an hash function, and then check if, for all x, y in the input, it holds that $h(x) \neq h(y)$. If so then we continue. Otherwise we repeat.

```
1  let collision ← True
2  while collision do
3  |   collision ← False
4  |   let T be array at length m
5  |   h ← sample uniformly random from universal hash family H
6  |   for x ∈ x_1, x_2..x_n do
7  |   |   if T_{h(x)} is not empty then
8  |   |   |   collision ← True
9  |   |   |   break the for-loop
10 |   |   end
11 |   |   else
12 |   |   |   T_{h(x)} ← x
13 |   |   end
14 |   end
15 end
16 return T, h
```

Algorithm 1: perfect-hashing($x_1, x_2, ..x_n$)

Question. What is the probability of choosing h with no collisions on the first trial? Notice that the answer depends on m. (To see this, imagine the case where $m = 1$. In this case, there must be collisions.) Therefore, the correct question is:

for what values of m do we succeed in finding a hash function with no collisions on the first trial? Let $X_{x,y}$ be the indicator of the event $h(x) = h(y)$. The expected number of collisions is then:

$$\mathbf{E}\left[\sum_{x\neq y} X_{x,y}\right] = \sum_{x\neq y} \mathbf{E}\left[X_{x,y}\right] = \binom{n}{2}\frac{1}{m}$$

Now, we would like to answer for what value of m there is no collision. Therefore, if we take $m = n^2$, then the expected number of collisions is less than $1/2$. By the Markov inequality, the probability of having more than one collision is less than:

$$P\left(\sum_{x\neq y} X_{x,y} > 1\right) \leq \mathbf{E}\left[\sum_{x\neq y} X_{x,y}\right] = \frac{1}{2}$$

And therefore the expected number of rounds is less than:

$$\mathbf{E}\left[\text{ rounds }\right] = \sum_{t=0}^{\infty} tP(t \text{ rounds }) \leq \sum_{t=0}^{\infty} t\frac{1}{2^{t-1}} = O(1)$$

Question. What is the space complexities? We have to allocate an array at length m which is $\Theta(n^2)$ memory. Is that good? So remember that in standard hash tables, the expected number of elements that were hashed into the same cell as the key x is

$$1 + \frac{n-1}{m}$$

Taking $m = \Theta(n)$ is enough to ensure that the expected running time of in insertion/deletion/access is $O(1)$. This raises the question of whether the space complexity of perfect hashing can be reduced to linear.

9.0.1 Perfect Hashing in Linear Space.

The idea is as follows: we will use a two-stage hashing process. In the first stage, keys will be mapped to hash tables instead of cells. Each hash table will be constructed using perfect hashing and may require a space that is quadratic in the number of elements stored in it (which were mapped to it in the first stage). Therefore, if we denote by n_i the number of elements mapped to the ith hash table, the space cost will be $\sum_i n_i^2$. Instead of starting over when a collision occurs, we will do so when $\sum_i n_i^2 > 4n$. So, now it's left to show that we expect $\sum_i n_i^2$ to be linear, which implies that the expected number of rounds is constant.

$$n_i^2 = 2\binom{n_i}{2} + n_i$$

On the other hand, $\sum_i \binom{n_i}{2}$ is exactly the number of collisions, as for any i, $\binom{n_i}{2}$ counts the number of distinct pairs in the ith table, which is equivalent to counting

```
let toomanycollisions ← True
while toomanycollisions do
    toomanycollisions ← False
    let T be array at length m
    initialize any T_i to be an empty linked list.
    h ← sample uniformly random from universal hash family H
    for x ∈ x_1, x_2..x_n do
        T_{h(x)}.insert(x)
        T_{h(x)}.size = T_{h(x)}.size +1
    end
    if Σ_i T_{h(i)}.size^2 ≥ μ then
        toomanycollisions ← True
    end
end
let H be an array at length m
for i ∈ [m] do
    T_i, h_i ← hash the elements in T_i using
        perfect hashing.
end
return T, h
```

Algorithm 2: perfect-hashing-linear-space($x_1, x_2, ..x_n$)

the number of $x \neq y$ such that $h(x) = h(y) = i$. Thus,

$$\mathbf{E}\left[\sum_i n_i^2\right] = \mathbf{E}\left[\sum_i 2\binom{n_i}{2} + n_i\right] = 2 \cdot \mathbf{E}\left[\text{collisions}\right] + \mathbf{E}\left[\overbrace{\sum_i n_i}^{n}\right]$$
$$= 2 \cdot \binom{n}{2}\frac{1}{m} + n$$

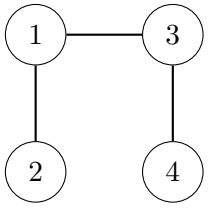
Therefore, by choosing $m = 4n$ for the first stage, the probability of failing to choose a proper hash function is less than $1/2$.

9.1 Graphs

9.1.1 Definitions, Examples, and Basics

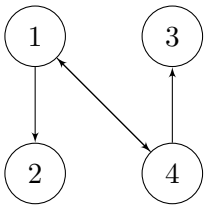
Definition 9.1.1. *A **non-directed graph** G is a pair of two sets - $G = (V, E)$ - V being a set of vertices and E being a set of couples of vertices, which represent edges ("links") between vertices.*

Example: $G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{3, 4\}\})$ is the following graph:



Definition 9.1.2. *A **directed graph** G is a pair of two sets - $G = (V, E)$ - V being a set of vertices and $E \subseteq V \times V$ being a set of directed edges ("arrows") between vertices.*

Example: $G = (\{1,2,3,4\}, \{(1,2),(1,4),(4,1),(4,3)\})$ is the following graph (note that it has arrows):



Definition 9.1.3. *A **weighted graph** composed by a graph $G = (V, E)$ (either non-directed or directed) and a weight function $w : E \to \mathbb{R}$. Usually (but not necessary), we will think about the quantity $w(e)$, where $e \in E$, as the length of the edge.*

Now that we see graphs with our eyes, we can imagine all sorts of uses for them... For example, they can represent the structure of the connections between friends on Facebook, or they can even represent which rooms in your house have doors between them.

Remark 9.1.1. *Note that directed graphs are a **generalization** of non-directed graphs, in the sense that every non-directed graph can be represented as a directed graph. Simply take every non-directed edge $\{v, u\}$ and turn it into two directed edges $(v, u), (u, v)$.*

Remark 9.1.2. *Note that most of the data structures we discussed so far - Stack, Queue, Heap, BST - can all be implemented using graphs.*

Now let's define some things in graphs:

Definition 9.1.4. *(Path, circle, degree)*

1. *A **simple path** in the graph G is a series of unique vertices (that is, no vertex appears twice in the series) $v_1, v_2, ..., v_n$ that are connected with edges in that order.*
2. *A **simple circle** in the graph G is a simple path such that $v_1 = v_n$.*
3. *The **distance** between two vertices $v, u \in V$ is the length of the shortest path between them (∞ if there is no such path).*

Remark 9.1.3. *Note that for all $u, v, w \in V$ the triangle inequality holds regarding path lengths. That is:*

$$dist(u, w) \leq dist(u, v) + dist(v, w)$$

Definition 9.1.5. *(connectivity)*

1. *Let $G = (V, E)$ be a non-directed graph. A **connected component** of G is a subset $U \subseteq V$ of maximal size in which there exists a path between every two vertices.*
2. *A non-directed graph G is said to be a **connected** graph if it only has one connected component.*

3. *Let $G = (V, E)$ be a directed graph. A **strongly connected component** of G is a subset $U \subseteq V$ of maximal size in which for any pair of vertices $u, v \in U$ there exist both directed path from u to v and a directed path form v to u.*

Claim 9.1.1. *Let $G = (V, E)$ be some graph. If G is connected, then $|E| \geq |V| - 1$*

Proof. We will perform the following process: Let $\{e_1, ..., e_m\}$ be an enumeration of E, and let $G_0 = (V, \emptyset)$. We will build the graphs $G_1, G_2, ...G_m = G$ by adding edges one by one. Formally, we define -

$$\forall i \in [m] \;\; G_i = (V, \{e_1, ..., e_i\})$$

G_0 has exactly $|V|$ connected components, as it has no edges at all. Then G_1 has $|V| - 1$. From there on, any edges do one of the following:

1. Keeps the number of connected components the same (the edge closes a cycle) '
2. Lowers the number of connected components by 1 (the edges does not close a cycle)

So in general, the number of connected components of G_i is $\geq |V| - i$. Now, if $G_m = G$ is connected, it has just one connected component! This means:

$$1 \geq |V| - |E| \implies |E| \geq |V| - 1$$

□

9.1.2 Graph Representation

Okay, so now we know what graphs are. But how can we represent them in a computer? There are two main options. The first one is by **array of adjacency lists**. Given some graph G, every slot in the array will contain a linked list. Each linked list will represent a list of some node's neighbors. The second option is to store edges in an **adjacency matrix**, a $|V| \times |V|$ binary matrix in which the v, u-cell equals 1 if there is an edge connecting u to v. That matrix is denoted by A_G in the example below. Note that the running time analysis might depend on the underline representation.

Question. What is the memory cost of each of the representations? Note that while holding an adjacency matrix requires storing $|V|^2$ bits regardless of the size of E, Maintaining the edges by adjacency lists costs linear memory at the number of edges and, therefore, only $\Theta(|V| + |E|)$ bits.

Example. Consider the following directed graph:

9.1.3 Breadth First Search (BFS)

One natural thing we might want to do is to travel around inside a graph. That is, we would like to visit all of the vertices in a graph in some order that relates to the edges. Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex s. Whenever the search discovers a white

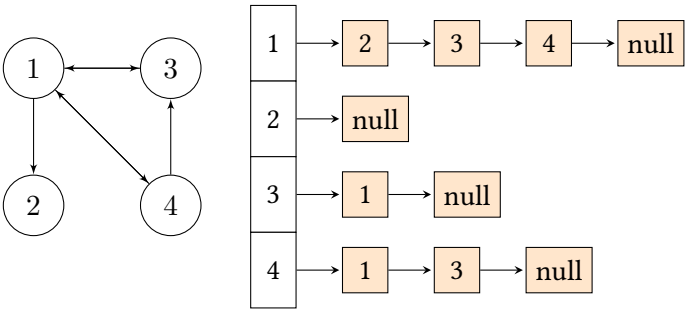


Figure 9.3: Persenting G by array of adjacency lists.

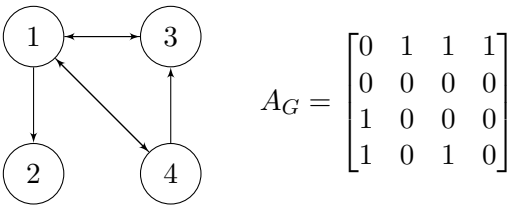


Figure 9.4: Presenting G by adjacency matrix.

vertex v in the course of scanning the adjacency list of a gray vertex u, the vertex v and the edge (u, v) are added to the tree. We say that u is the predecessor or parent of v in the breadth-first tree. Since every vertex reachable from s is discovered at most once, each vertex reachable from s has exactly one parent. (There is one exception: because s is the root of the breadth-first tree, it has no parent.) Ancestor and descendant relationships in the breadth-first tree are defined relative to the root s as usual: if u is on the simple path in the tree from the root s to vertex v, then u is an ancestor of v, and v is a descendant of u. The breadth-first-search procedure BFS on the following page assumes that the graph $G = (V, E)$ is represented using adjacency lists. It denotes the queue by Q, and it attaches three additional attributes to each vertex v in the graph:

1. v.visited is a boolean flag which indicate wheter v was allready visited.
2. $\pi(v)$ is v's predecessor in the breadth-first tree. If v has no predecessor because it is the source vertex or is undiscovered, then $\pi(v)$ is None/NULL.

Correctness: The example should be enough to explain the correctness. A concrete proof can be found in the book, page 597.

Runtime: We can analyse the runtime line-by-line:

- Lines 1-2: $|V|$ operations, all in $O(1)$ runtime, for a total of $O(|V|)$.
- Lines 3-6: $O(1)$
- Lines 7-8: First we need to understand the number of times the $while$ loop iterates. We can see that every vertex can only enter the queue ONCE (since it is then tagged as "visited"), and therefore it runs $\leq |V|$ times. All operations are $O(1)$, and we get a total of $O(|V|)$.

```
1  for v ∈ V do
2  |  v.visited ← False
3  end
4  Q ← new Queue
5  Q.Enqueue(s)
6  s.visited ← True
7  while Q is not empty do
8  |  u ← Q.Dequeue()
9  |  for neighbor w of u do
10 |  |  if w.visited is False then
11 |  |  |  w.visited← True
12 |  |  |  π(w) ← u
13 |  |  |  Q.Enqueue(w)
14 |  |  end
15 |  end
16 end
```

Algorithm 3: BFS(G, s)

- Lines 9-13: Next, we want to understand the number of times this for loop iterates. The for loop starts iterating once per vertex, and then the number of its iterations is the same as the number of neighbors that this vertex has. Thus, it runs $O(|E|)$ times.

So all in all we get a runtime of $O(|V| + |E|)$

9.1.4 Usage of BFS

Now we have a way to travel through a graph using the edges. How else can we use it?

Exercise: Present and analyse an algorithm $CC(G)$ which receives some undirected graph G and outputs the number of connected components in G in $O(|V|+|E|)$.

Solution: Consider the following algorithm:

```
1  count ← 0
2  for v ∈ V do
3  |  if v.visited = False then
4  |  |  count←count+1
5  |  |  BFS(G, v)
6  |  end
7  end
8  return count
```

Algorithm 4: CC(G)

9.1.5 Depth First Search (DFS)

As its name implies, depth-first search searches "deeper" in the graph whenever possible. Depth-first search explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until all vertices that are reachable from the original source vertex have been discovered. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, repeating the search from that source. The algorithm repeats this entire process until it has discovered every vertex.

```
1  DFS( G):
2  for v ∈ V do
3  |  vi.visited ← False
4  end
5  time ← 1
6  for v ∈ V do
7  |  if not v.visited then
8  |  |  π (v) ← null
9  |  |  Explore( G, v )
10 |  end
11 end
```

Algorithm 5: DFS(G)

```
1  Explore(G, v):
2  Previsit(v) for (v, u) ∈ E do
3  |  if not u.visited then
4  |  |  π (u) ← v
5  |  |  Explore( G, u )
6  |  end
7  end
8  Postvisit(v)
```

```
1  Previsit(v):
2  pre(v) ← time
3  time ← time +1
```

```
1  Postvisit (v):
2  post(v) ← time
3  time ← time +1
```

Properties of depth-first search. Depth-first search yields valuable information about the structure of a graph. Perhaps the most basic property of depth-first search is that the predecessor subgraph G_π does indeed form a forest of trees since the structure of the depth-first trees exactly mirrors the structure of recursive calls of explore-function. That is, $u = \pi(v)$ if and only if explore(G, v) was called during a search of u's adjacency list. Additionally, vertex v is a descendant of vertex

u in the depth-first forest if and only if v is discovered during the time in which u is gray. Another important property of depth-first search is that discovery and finish times have a parenthesis structure. If the explore procedure were to print a left parenthesis "(u" when it discovers vertex u and to print a right parenthesis r"u)" when it finishes u, then the printed expression would be well-formed in the sense that the parentheses are properly nested.

The following theorem provides another way to characterize the parenthesis structure.

Claim 9.1.2 (Parenthesis theorem)**.** *In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:*

1. *the intervals [pre(u), post(u)] and [pre(v), post(v)] are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest.*
2. *the interval [pre(u), post(u)] is contained entirely within the interval [pre(v), post(v)], and u is a descendant of v in a depth-first tree, or*
3. *the interval [pre(v), post(v)] is contained entirely within the interval [pre(u), post(u)], and v is a descendant of u in a depth-first tree.*

Proof. We begin with the case in which pre(u) $<$ pre(v). We consider two subcases, according to whether pre(v) $<$ post(u). The first subcase occurs when pre(v) $<$ post(u), so that v was discovered while u was still gray, which implies that v is a descendant of u. Moreover, since v was discovered after u, all of its outgoing edges are explored, and v is finished before the search returns to and finishes u. In this case, therefore, the interval [pre(v), post(v)] is entirely contained within the interval [pre(u), post(u)]. In the other subcase, post(u) $<$ pre(v), and by defintion, pre(u) $<$ post(u) $<$ pre(v) $<$ post(v), and thus the intervals [pre(u), post(u)] and [pre(v), post(v)] are disjoint. Because the intervals are disjoint, neither vertex was discovered while the other was gray, and so neither vertex is a descendant of the other. □

Corollary. Nesting of descendants' intervals. Vertex v is a proper descendant of vertex u in the depth-first forest for a (directed or undirected) graph G if and only if pre(u) $<$ pre(v) $<$ post(v) $<$ post(u).