1 Graph Overview

1.1 DFS

1.1.1 Recall

- DFS is $O(|E| + |V|)$ search that goes through all the vertices in a graph.
- DFS gives each node a pre-order and post-order number.
- We can use these numbers, or modifications to the graph during pre-visit and post-visit operations to produce useful algorithms.
- Remember forward edges, back edges, and cross edges.
- Uses from Class: Topological sort, finding Strongly Connected Components.

1.1.2 Pseudo-code

```
SEARCH(v)
(1)   EXPLORED(v) ← 1
(2)   PREVISIT(v)
(3)   foreach (v, w) ∈ E
(4)     if EXPLORED(w) = 0 then SEARCH(w)
(5)   POSTVISIT(v)
```

```
DFS(G(V, E))
(1)   foreach v ∈ V
(2)     EXPLORED(v) ← 0
(3)   foreach v ∈ V
(4)     if EXPLORED(v) = 0 then SEARCH(v)
```

- The functions PREVISIT and POSTVISIT can be modified to solve several interesting problems.
- Linear-time algorithm for computing *strongly connected components*.

1.2 Strongly Connected Components

1.2.1 The Basics

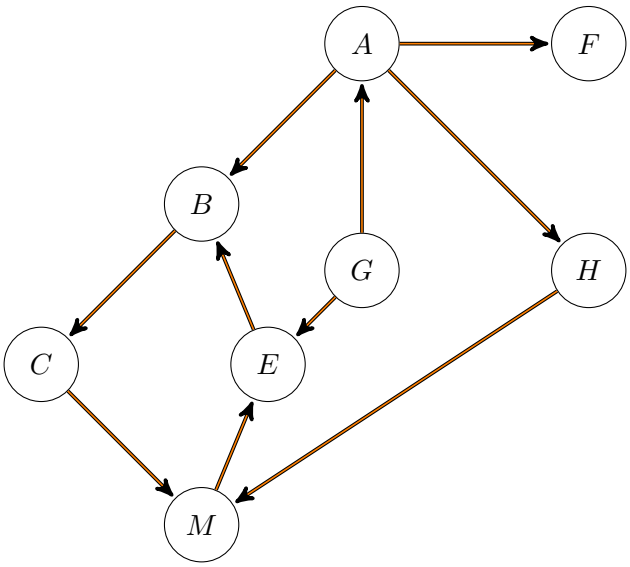
- In a strongly connected component, all vertices have paths to each other.
- You can convert an arbitrary directed graph to a DAG by finding SCCs, and treating each SCC as a "super-node".
- SCCs are equivalence classes. The relation: u and v related if there exists paths from u to v and v to u.

1.3 Topological Sort

1.3.1 The Basics

- Given a Directed Acyclic Graph, order such that for u has a path to v implies that u appears before v.

Exercise. *Run dfs on the graph and then reduce it to a DAG by finding the SCCs*



Solution.
Essentially, we have a DAG that has 5 components, G, A, F, H, and BCEM. The G is at the top, pointing towards A and BCEDM, the A points to F and H, and the H points to the BCEM.

Exercise. *Give an algorithm to find an edge that is present in all the cycles of a given graph.*

Solution.
Begin by identifying all the cycles in a graph via DFS. Suppose we find a cycle C with $|V|$ edges. We then run DFS again on $G - C$ to look for another cycle. If the resulting search yields another cycle, then we know we just found two disjoint cycles with no edges in common, meaning there is no edge that is present in all cycles. Otherwise, we know that we have a cycle C with an edge e that is in all the cycles. We can then just alternate removing edges of C from G to check if the graph is acyclic. If so, we have just found the common edge.

1.4 BFS

1.4.1 Recall

- BFS is also a $O(|E| + |V|)$ search, often done from just one node in a graph.

- BFS gives you a list of nodes in increasing path-length from source.
- Uses from Class: Shortest paths.

1.5 Pseudo-code

```
BFS(G(V, E), s ∈ V)
(1)    foreach v ∈ V − s
(2)        v.DIST = ∞
(3)        v.PREV = nil
(4)        v.PLACED = 0
(5)    s.DIST = 0
(6)    s.PREV = nil
(7)    s.PLACED = 1
(8)    QUEUE q
(9)    ENQUEUE(q, s)
(10)   while q ≠ ∅
(11)       u =DEQUEUE(q)
(12)       foreach (u, v) ∈ E
(13)           if v.placed = 0
(14)               v.DIST = u.DIST + 1
(15)               v.PREV = u
(16)               v.PLACED = 1
(17)               ENQUEUE(q, v)
```

2 Heaps

Heaps are data structures that make it easy to find the element with the most extreme value in a collection of elements. A MIN-HEAP prioritizes the element with the smallest value, while a MAX-HEAP prioritizes the element with the largest value. Because of this property, heaps are often used to implement priority queues.

You can find more about heaps by reading pages 151–169 in CLRS.

2.1 Representing a Heap

While a heap can be represented as a regular tree, it is often more efficient to store a binary heap as an array. We call the first element in the heap element 1. Now, given an element i, we can find its left and right children with a little arithmetic:

Exercise.

- $\textsc{Parent}(i) =$
- $\textsc{Left}(i) =$
- $\textsc{Right}(i) =$

Solution.
The array representation is simplified by calling the first element in the heap 1:

- $\textsc{Parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor$
- $\textsc{Left}(i) = 2i$
- $\textsc{Right}(i) = 2i + 1$

The completeness requirement makes sure this representation of heaps is compact.

2.2 Heap operations

2.2.1 Max-Heapify

Max-Heapify(H, N): Given that the children of the node N in the MAX-HEAP H are each the root of a MAX-HEAP, rearranges the tree rooted at N to be a MAX-HEAP.

MAX-HEAPIFY(H, N):
Require: LEFT(N), RIGHT(N) are each the root of a MAX-HEAP
  $(l, r) \leftarrow (\text{LEFT}(N), \text{RIGHT}(N))$
  if EXISTS(l) and $H[l] > H[N]$ **then**
    $largest \leftarrow l$
  else
    $largest \leftarrow N$
  end if
  if EXISTS(r) and $H[r] > H[largest]$ **then**
    $largest \leftarrow r$
  end if
  if $largest \neq N$ **then**
    SWAP($H[N], H[largest]$)
    MAX-HEAPIFY($H, largest$)
  end if
Ensure: N is the root of a MAX-HEAP

Exercise.

- *Run* MAX-HEAPIFY *with* $N = 1$ *on*

$$H = [14, 16, 10, 8, 7, 9, 6, 2, 4, 1]$$

- *What is* MAX-HEAPIFY*'s run-time?*

Solution.

- MAX-HEAPIFY(H, N) returns a max-heap

$$[16, 14, 10, 8, 7, 9, 6, 2, 4, 1]$$

- Runs in $O(\log n)$ time because node N is moved down at most $\log n$ times.

2.2.2 Build-Heap

Build-Heap(A): Given an unordered array, makes it into a max-heap.

BUILD-HEAP(A):
Require: A is an array.
  for $i = \lfloor$ length$(A)/2 \rfloor$ downto 1 **do**
    MAX-HEAPIFY(A, i)
  end for

Exercise.

- *Run* BUILD-HEAP *on* $A = [2, 1, 4, 3, 6, 5]$
- *Running time (loose upper bound):*
- *Running time (tight upper bound):*

Solution.

- $$[2, 1, 4, 3, 6, 5] \rightarrow$$
  $$[2, 1, 5, 3, 6, 4] \rightarrow$$
  $$[2, 6, 5, 3, 1, 4] \rightarrow$$
  $$[6, 3, 5, 2, 1, 4]$$
- $O(n \log n)$
- $O(n)$. Intuitively, we might be able to get a tighter bound because MAX-HEAPIFY is run more often at lower points on the tree, when subtrees are shallow. Making use of the fact that an n-element heap has height $\lfloor \log n \rfloor$ and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h, the total number of comparisons that will have to be made is

  $$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lg n} \frac{h}{2^h} \right)$$

  It is not hard to see that $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$. So the runtime is $O(n)$.

2.2.3 Extract-Max

Extract-Max(H): Remove the element with the largest value from the heap.

EXTRACT-MAX(H):
Require: H is a non-empty MAX-HEAP
  $max \leftarrow H[root]$
  $H[root] \leftarrow H[\text{SIZE}(H)]$ {last element of the heap.}
  $\text{SIZE}(H) -= 1$
  MAX-HEAPIFY($H, root$)
  return max

Exercise.

- *Run* EXTRACT-MAX *on* $H = [6, 3, 5, 2, 1, 4]$.
- *What is* EXTRACT-MAX*'s run time?*

Solution.

- EXTRACT-MAX first returns 6. It then moves the last element, 4, to the head (why the last element?) and MAX-HEAPIFY is used to maintain the heap structure:

$$[4, 3, 5, 2, 1] \rightarrow [5, 3, 4, 2, 1]$$

- $O(\log n)$

2.2.4 Insert

Insert(H, v): Add the value v to the heap H.

INSERT(H, v):
Require: H is a MAX-HEAP, v is a new value.
  SIZE(H) $+= 1$
  $H[\text{SIZE}(H)] \leftarrow v$ {Set v to be in the next empty slot.}
  $N \leftarrow \text{SIZE}(H)$ {Keep track of the node currently containing v.}
  while N is not the root and $H[\text{PARENT}(N)] < H[N]$ **do**
    SWAP$(H[\text{PARENT}(N)], H[N])$
    $N \leftarrow \text{PARENT}(N)$
  end while

Exercise.

- *Run* INSERT*(H, v) with $v = 8$ and*
$$H = [6, 3, 5, 2, 1, 4]$$
- *What is* INSERT*'s runtime?*

Solution.

- Insert v into H as follows:
$$[6, 3, 5, 2, 1, 4, 8] \rightarrow$$
$$[6, 3, 8, 2, 1, 4, 5] \rightarrow$$
$$[8, 3, 6, 2, 1, 4, 5]$$
  Here, the while loop is halted by the condition that N is the root.
- INSERT's runtime is $O(\log n)$.

Increase-Key(H, N, v): Increase the value of node N to a new, higher value v in the heap H.

INCREASE-KEY(H, N, v):
Require: H is a MAX-HEAP, N is an element, and v is its new value.
  $H[N] \leftarrow v$ {Update the key of N-th element to v.}
  $C \leftarrow N$ {Keep track of the node currently containing v.}
  while C is not the root and $H[\text{PARENT}(C)] < H[C]$ **do**
    SWAP$(H[\text{PARENT}(C)], H[C])$
    $C \leftarrow \text{PARENT}(C)$
  end while

Exercise. *Using heaps, how would we implement:*

- *A priority queue?*

- *Heapsort?*

Solution.

- In a priority queue, elements with a higher priority are dequeued before elements with a lower priority. Therefore, the *queue* operation is simply the insert operation of a MAX-HEAP (with the priority as the value of the node containing the element in the heap). The *dequeue* operation is the EXTRACT-MAX operation.
- Given a heap H and an array to sort A_n where $|A_n| = N$, simply insert each element of A_n into H. Then EXTRACT-MAX until $H = \emptyset$. The elements are extracted in decreasing order, thereby sorting the array. The running time is $\Theta(n \log n)$ because for each element, we insert ($\Theta(\log n)$) and extract ($\Theta(\log n)$) once.

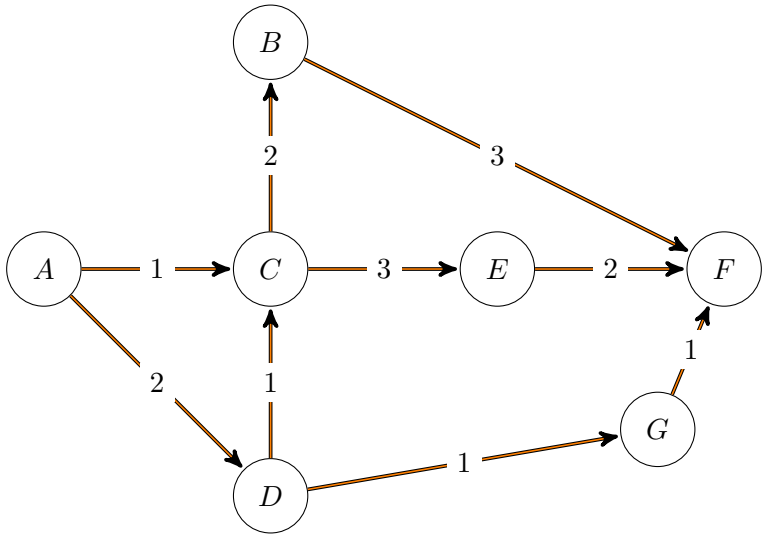
3 Advanced Graph Algorithms - Dijkstra's algorithm

- Dijkstra's algorithm solves the *single-source shortest path problem*: given a graph $G = (V, E, \omega)$ with nonnegative weights, determine the shortest path from a *source vertex* $s \in V$ to every $v \in V$.

DIJKSTRA$(G(V, E, \omega), s \in V)$
(1) **foreach** $v \in V$
(2) DIST$[v] \leftarrow \infty$, PREV$[v] \leftarrow$ NIL
(3) DIST$[s] \leftarrow 0$
(4) $H \leftarrow \{(s, 0)\}$
(5) **while** $H \neq \emptyset$
(6) $v \leftarrow$ DELETEMIN(H)
(7) **foreach** $(v, w) \in E$
(8) **if** DIST$[w] >$ DIST$[v] + \omega(v, w)$
(9) DIST$[w] \leftarrow$ DIST$[v] + \omega(v, w)$, PREV$[w] \leftarrow v$, INSERT$((w,$ DIST$[w]), H)$

- The running time of Dijkstra's algorithm depends on the implementation of the heap H.
- Note that Dijkstra's algorithm does not work for negative edge weights!

Exercise. *Run Dijkstra on this graph from A to F.*



Solution.

Looking at	Heap
A	C(1) D(2)
C	D(2) B(3) E(4)
D	B(3) G(3) E(4)
B	G(3) E(4) F(6)
G	E(4) F(4)
F	

Exercise. *In a strongly-connected graph with positive edge weights, find the shortest path between all pairs of vertices such that the path goes through a particular vertex v.*

Solution.
Essentially, the solution is to run Dijkstra on v for G and G^R and create the shortest path tree for both. This should take around $O(|V|log|V| + |E|)$ using a fibonacci heap. Afterwards, any shortest path (u, w) can be computed by looking up the computed shortest path from u to v and v to w.