Theoretische Informatik 1
Algorithmen und Datenstrukturen

Inofficial lecture notes

Marvin Borner

Vorlesung gehalten von

Ulrike von Luxburg

Wintersemester 2022/23

Content

1 Tricks

1.1 Logarithms

$$\log_a(x) = \frac{\log_b(x)}{\log_b(a)}$$

2 Big-O-Notation

- $f \in \mathcal{O}(g)$: f is of order at most g:

$$0 \geq \limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

- $f \in \Omega(g)$: f is of order at least g:

$$0 < \liminf_{n\to\infty} \frac{f(n)}{g(n)} \leq \infty \iff g \in \mathcal{O}(f)$$

- $f \in o(g)$: f is of order strictly smaller than g:

$$0 \geq \limsup_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

- $f \in \omega(g)$: f is of order strictly larger than g:

$$\liminf_{n\to\infty} \frac{f(n)}{g(n)} = \infty \iff g \in o(f)$$

- $f \in \Theta(g)$: f has exactly the same order as g:

$$0 < \liminf_{n\to\infty} \frac{f(n)}{g(n)} \leq \limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty \iff f \in \mathcal{O}(g) \land f \in \Omega(g)$$

2.1 Naming

- linear $\implies \Theta(n)$
- sublinear $\implies o(n)$
- superlinear $\implies \omega(n)$
- polynomial $\implies \Theta(n^a)$
- exponential $\implies \Theta(2^n)$

2.2 Rules

- $f \in \mathcal{O}(g_1 + g_2) \land g_1 \in \mathcal{O}(g_2) \implies f \in \mathcal{O}(g_2)$
- $f_1 \in \mathcal{O}(g_1) \land f_2 \in \mathcal{O}(g_2) \implies f_1 + f_2 \in \mathcal{O}(g_1 + g_2)$
- $f \in g_1\mathcal{O}(g_2) \implies f \in \mathcal{O}(g_1 g_2)$
- $f \in \mathcal{O}(g_1), g_1 \in \mathcal{O}(g_2) \implies f \in \mathcal{O}(g_2)$

3 Divide and conquer

Problem

Given two integers x and y, compute their product $x \cdot y$.

We know that $x = 2^{n/2}x_l + x_r$ and $y = 2^{n/2}y_l + y_r$.

We use the following equality:

$$(x_l y_r + x_r y_l) = (x_l + x_r)(y_l + y_r) - x_l y_l - x_r y_r$$

leading to

$$\begin{aligned} x \cdot y &= 2^n x_l y_l + 2^{n/2}(x_l y_r + x_r y_l) + x_r y_r \\ &= 2^n x_l y_l + 2^{n/2}((x_l + x_r)(y_l + y_r) - x_l y_l - x_r y_r) + x_r y_r \\ &= 2^{n/2} x_l y_l + (1 - 2^{n/2}) x_r y_r + 2^{n/2}(x_l + x_r)(y_l y_r). \end{aligned}$$

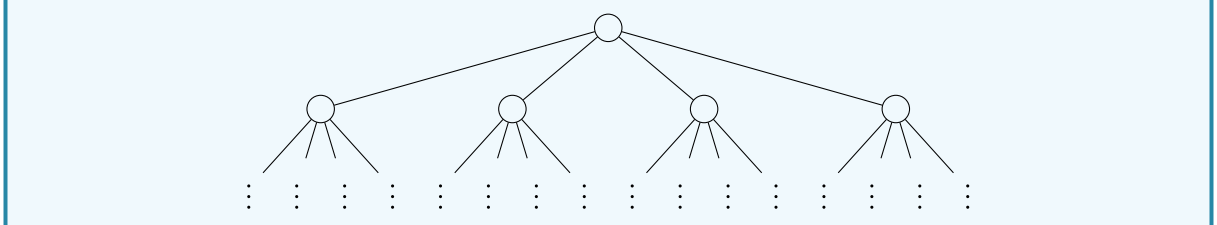
Therefore we get the same result with 3 instead of 4 multiplications.

If we apply this principle once: Running time of $(3/4)n^2$ instead of n^2.

If we apply this principle recursively: Running time of $\mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.59})$ instead of n^2 (calculated using the height of a recursion tree).

3.1 Recursion tree

Visualisation



- Level k has a^k problems of size $\frac{n}{b^k}$
- Total height of tree: $\lceil \log_b n \rceil$
- Number of problems at the bottom of the tree is $a^{\log_b n} = n^{\log_b a}$
- Time spent at the bottom is $\Theta(n^{\log_b a})$

3.2 Master theorem

If $T(n) = aT(\lceil n/b \rceil) + \mathcal{O}(n^d)$ for constants $a > 0$, $b > 1$ and $d \geq 0$, then

$$T(n) = \begin{cases} \mathcal{O}(n^d) & d > \log_b a \\ \mathcal{O}(n^d \log_n) & d = log_b a \\ \mathcal{O}(n^{log_b a}) & d < \log_b a \end{cases}$$

Example

Previous example of clever integer multiplication:

$$T(n) = 3T(n/2) + \mathcal{O}(n) \implies T(n) = \mathcal{O}(n^{\log_2 3})$$

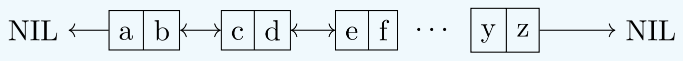
4 Arrays and lists

4.1 Array

- needs to be allocated in advance
- read/write happens in constant time (using memory address)

4.2 Doubly linked list

Visualisation



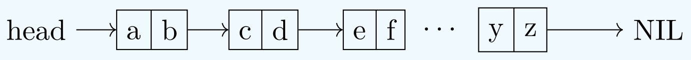
NIL can be replaced by a sentinel element, basically linking the list to form a loop.

4.2.1 Basic operations

- **Insert**: If the current pointer is at e, inserting x after e is possible in $\mathcal{O}(1)$.
- **Delete**: If the current pointer is at e, deleting x before e is possible in $\mathcal{O}(1)$.
- **Find element with key**: We need to walk through the whole list $\implies \mathcal{O}(n)$
- **Delete a whole sublist**: If you know the first and last element of the sublist: $\mathcal{O}(1)$
- **Insert a list after element**: Obviously also $\mathcal{O}(1)$

4.3 Singly linked list

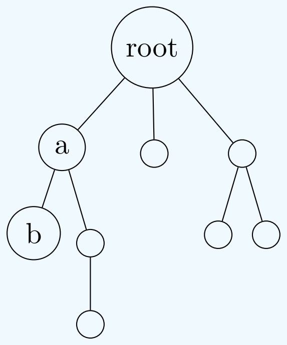
Visualisation



- needs less storage
- no constant time deletion $\implies$ not good

5 Trees

Visualisation



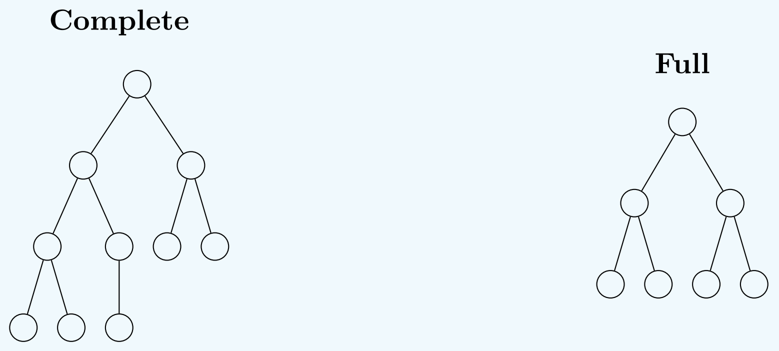
- (a) is the parent/predecessor of (b)
- (b) is a child of (a)

- *Height of a vertex*: length of the shortest path from the vertex to the root
- *Height of the tree*: maximum vertex height in the tree

5.1 Binary tree

- each vertex has at most 2 children
- *complete* if all layers except the last one are filled
- *full* if the last level is filled completely

Visualisation



5.1.1 Height of binary tree

- Full binary tree with n vertices: $\log_2(n+1) - 1 \in \Theta(\log n)$
- Complete binary tree with n vertices: $\lceil \log_2(n+1) - 1 \rceil \in \Theta(\log n)$

5.1.2 Representation

- Complete binary tree: Array with entries layer by layer
- Arbitrary binary tree: Each vertex contains the key value and pointers to left, right, and parent vertices
  - Elegant: Each vertex has three pointers: A pointer to its parent, leftmost child, and right sibling

6 Stack and Queue

6.1 Stack

- Analogy: Stack of books to read
- `Push(x)` inserts the new element x to the stack
- `Pop()` removes the next element from the stack (LIFO)

6.1.1 Implementation

- Array with a pointer to the current element, `Push` and `Pop` in $\mathcal{O}(1)$
- Doubly linked list with pointer to the end of the list
- Singly linked list and insert elements at the beginning of the list

6.2 Queue

- Analogy: waiting line of customers
- `Enqueue(x)` insertes the new element x to the end of the queue
- `Dequeue()` removes the next element from the queue (FIFO)

6.2.1 Implementation

- Array with two pointers, one to the head and one to the tail $\implies$ `Enqueue`/`Dequeue` in $\mathcal{O}(1)$
- Linked lists

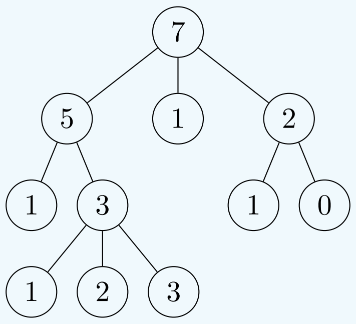
7 Heaps and priority queues

7.1 Heaps

- Data structure that stores elements as vertices in a tree
- Each element has a key value assigned to it
- Max-heap property: all vertices in the tree satisfy

$$\text{key}(\text{parent}(v)) \geq \text{key}(v)$$

Visualisation



- Binary heap:
  - Each vertex has at most two children
  - Layers must be finished before starting a new one (left to right insertion)
  - Advantage:
    - Control over height/width of tree
    - easy storage in array without any pointers

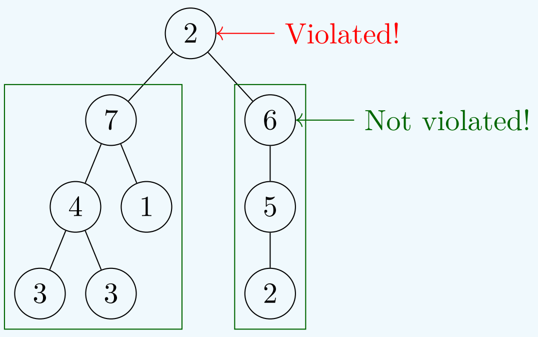
7.1.1 Usage

- Compromise between a completely unsorted and completely sorted array
- Easier to maintain/build than a sorted array
- Useful for many other data structures (e.g. priority queue)

7.1.2 `Heapify`

The `Heapify` operation can repair a heap with a violated heap property ($\text{key}(i) < \text{key}(\text{child}(i))$ for some vertex i and at least one child).

Visualisation

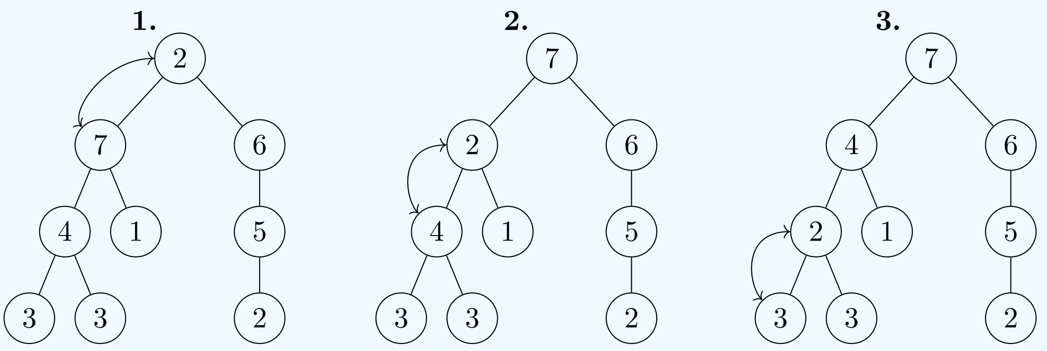


Procedure: "Let key(i) float down"

- Swap i with the larger of its children

- Recursively call `heapify` on this child
- Stop when heap condition is no longer violated

Visualisation



Worst case running time:

- Number of swapping operations is at most the height of the tree
- Height of tree is at most $h = \lceil \log(n) \rceil = \mathcal{O}(\log n)$
- Swapping is in $\mathcal{O}(1) \implies$ worst case running time is $\mathcal{O}(\log n)$

7.1.3 `DecreaseKey`

The `DecreseKey` operation *decreases* the key value of a particular element in a correct heap.

Procedure:

- Decrease the value of the key at index i to new value b
- Call `heapify` at i to let it bubble down

Running time: $\mathcal{O}(\log n)$

7.1.4 `IncreaseKey`

The `IncreseKey` operation *increases* the key value of a particular element in a correct heap.

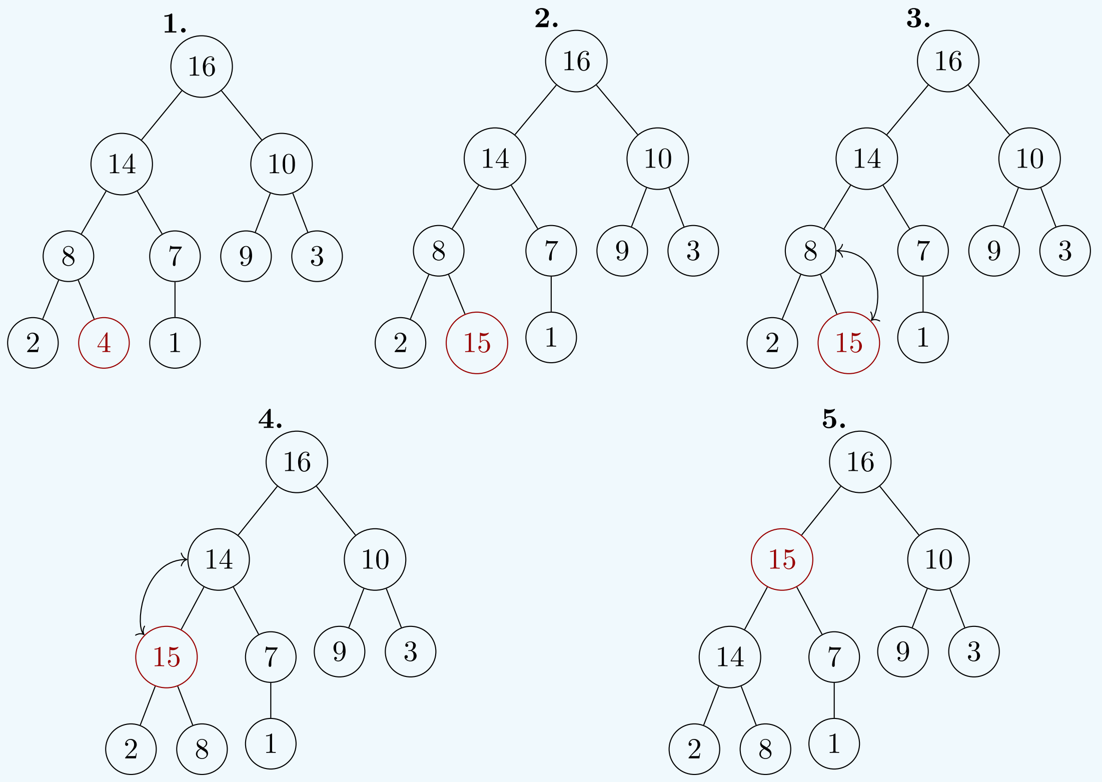
Procedure:

- Increase the value of the key at index i to new value b
- Walk upwards to the root, exchaning the key values of a vertex and its parent if the heap property is violated

Running time: $\mathcal{O}(\log n)$

Visualisation

`IncreaseKey` from 4 to 15:



7.1.5 `ExtractMax`

The `ExtractMax` operation *removes* the largest element in a correct heap.

Procedure:

- Extract the root element (the largest element)
- Replace the root element by the last leaf in the tree and remove that leaf
- Call `heapify(root)`

Running time: $\mathcal{O}(\log n)$

7.1.6 `InsertElement`

The `InsertElement` operation *inserts* a new element in a correct heap.

Procedure:

- Insert it at the next free position as a leaf, asiign it the key $-\infty$
- Call `IncreaseKey` to set the key to the given value

Running time: $\mathcal{O}(\log n)$

7.1.7 `BuildMaxHeap`

The `BuildMaxHeap` operation makes a heap out of an unsorted array A of n elements.

Procedure:

- Write all elements in the tree in any order

- Then, starting from the leafs, call `heapify` on each vertex

Running time: $\mathcal{O}(n)$

8 Priority queue

Maintains a set of prioritized elements. The `Dequeue` operation returns the element with the largest priority value. `Enqueue` and `IncreaseKey` work as normal.

8.1 Implementation

Typically using a heap:

- Building the heap is $\mathcal{O}(n)$
- Enqueue: heap `InsertElement`, $\mathcal{O}(\log n)$
- Dequeue: heap `ExtractMax`, $\mathcal{O}(\log n)$
- `IncreaseKey`, `DecreaseKey`: $\mathcal{O}(\log n)$

"Fibonacci heaps" can achieve `DecreaseKey` in $\mathcal{O}(1)$.

9 Hashing

Idea:

- Store data that is assigned to particular key values
- Give a "nickname" to each of the key values
- Choose the space of nicknames reasonably small
- Have a way to compute "nicknames" from the keys themselves
- Store the information in an array (size = #nicknames)

Formally:

- **Universe** U: All possible keys, actually used key values are much less ($m < |U|$)
- **Hash function**: $h : U \to \{1, ..., m\}$
- **Hash values**: $h(k)$ (slot)
- **Collision**: $h(k_1) = h(k_2),\ k_1 \neq k_2$

9.1 Simple hash function

If we want to hash m elements in universe $\mathbb{N}$:

$$h(k) = k \pmod{m}$$

For n slots generally choose m using a prime number $m_p > n$

9.2 Hashing with chaining

Method to cope with collisions:

Each hash table entry points to a linked list containing all elements with this particular hash key - collisions make the list longer.

We might need to traverse this list to retrieve a particular element.

9.3 Hashing with open addressing

(**Linear probing**)

All empty slots get marked as empty

Inserting a new key into $h(k)$:

- If unused, insert at $h(k)$
- If used, try insert at $h(k) + 1$

Retrieving elements: Walk from $h(k)$

- If we find the key: Yay
- If we hit the empty marker: Nay

Removing elements:

- Another special symbol marker..
- Or move entries up that would be affected by the “hole” in the array

10 Graph algorithms

10.1 Graphs

A graph $G = (V, E)$ consists of a set of vertices V and a set of edges $E \subset V \times V$.

- edges can be **directed** (u, v) or **undirected** $\{u, v\}$
- u is **adjacent** to v if there exists an edge between u and v: $u \sim v$ or $u \to v$
- edges can be **weighted**: $w(u, v)$
- undirected **degree of a vertex**:

$$d_v := d(v) := \sum_{v \sim u} w_{vu}$$

- directed **in-/out-degree of a vertex**:

$$d_i n = \sum_{\{u : u \to v\}} w(u, v)$$

$$d_o ut = \sum_{\{u : v \to u\}} w(v, y)$$

- number of vertices: $n = |V|$
- number of edges: $m = |E|$
- **simple** path if each vertex occurs at most once
- **cycle** path if it end in the vertex where it started from and uses each edge at most once
- **strongly connected** directed graph if for all $u, v \in V,\ u \neq v$ exists a directed path from u to v and a directed path from v to u
- **acyclic** graph if it does not contain any cycles (**DAG** if directed)
- **bipartite** graph if its vertex set can be decomposed into two disjoint subsets such that all edges are only between them

10.1.1 Representation

- Unordered edge list: For each edge, encode start and end point
- Adjacency matrix:
  - $n \times n$ matrix that contains entries $a_{ij} = 1$ if there is a directed edge from vertex i to vertex j

  - if weighted, $a_{ij} = wij$
  - **implementation** using n arrays of length n
  - adjacency test in $\mathcal{O}(1)$
  - space usage n^2
- Adjacency list:
  - for each vertex, store a list of all outgoing edges
  - if the edges are weighted, store the weight additionally in the list
  - sometimes store both incoming and outgoing edges
  - **implementation** using an array with list pointers or using a list for each vertex that encodes outgoing edges

Typical choice:

- *dense* graphs: adjacency matrices tend to be easier.
- *sparse* graphs: adjacency lists

10.2 Depth first search

Idea: Starting at a arbitrary vertex, jump to one of its neighbors, then one of his neighbors etc., never visiting a vertex twice. At the end of the chain we backtrack and walk along another chain.

Running time

- graph: $\mathcal{O}(|V| + |E|)$
- adjacency matrix: $\mathcal{O}(|V|^2)$

Algorithm:

```
function DFS(G)
  for all $u \in V$
    u.color = white # not visited yet
  for all $u \in V$
    if u.color == white
      DFS-Visit(G, u)

function DFS-Visit(G, u)
  u.color = grey # in process
  for all $v \in \text{Adj}(u)$
    if v.color == white
      v.pre = u # just for analysis
      DFS-Visit(G, v)
  u.color = black # done!
```

10.3 Strongly connected components

Component graph G^{SCC} of a directed graph:

- vertices of G^{SCC} correspond to the components of G
- edge between vertices A and B in G^{GCC} if vertices u and v in connected components represented by A and B such that there is an edge from u to v
- G^{SCC} is a DAG for any directed graph G
- **sink** component if the vertex in G^{SCC} does not have an out-edge
- **source** component if the vertex in G^{SCC} does not have an in-edge

10.4 DFS in sink components

With sink component B:

- `DFS` on G in vertex $u \in B$: `DFS-Visit` tree covers the whole component B
- `DFS` on G in vertex u non-sink: `DFS-Visit` tree covers more than this component

$\Longrightarrow$ use DFS to discover SCCs

10.5 Finding sources

- **discovery time** $d(u)$: time when DFS first visits u
- **finishing time** $f(u)$: time when DFS is done with u

Also: $d(A) = \min_{u \in A} d(u)$ and $f(A) = \max_{u \in A} f(u)$.

Let A and B be two SCCs of G and assume that B is a descendent of A in G^{SCC}. Then $f(B) < f(A)$ always.

Assume we run DFS on G (with any starting vertex) and record the finishing times of all vertices. Then the vertex with the largest finishing time is in a source component.

10.6 Converting sources to sinks

Reversing the graph: we consider the graph G^t which has the same vertices as G but all edges with reversed directions. Note that G^t has the same SCCs as G.

We can then use the source-finding algorithm to find sinks by first reversing the graph.

10.7 Finding SCCs

- run DFS on G with any arbitrary starting vertex. The vertex u^* with the largest $f(u)$ is in a source of G^{SCC}
- the vertex u^* is in a sink of $(G^t)^{SCC}$
- start a second DFS on $u*$ in G^t. The tree discovered by DFS(G^t, u^*) is the first SCC
- continue with DFS on the remaining vertices $V = v*$ with the highest $f(u)$
- etc.

Running time:

- DFS twice: $\mathcal{O}(|V| + |E|)$
- reverse: $\mathcal{O}(|E|)$
- order the vertices by $f(u)$: $\mathcal{O}(|V|)$
- $\Longrightarrow \mathcal{O}(|V| + |E|)$

10.8 Cycle detection

A directed graph has a cycle iff its DFS reveals a back edge (to a previously visited vertex).

10.9 Topological sort

A **topological sort** of a directed graph is a linear ordering of its vertices such that whenever there exists a directed edge from vertex u to vertex v, u comes before v in the ordering.

Every DAG has a topological sort.

Procedure:

- run DFS with an arbitrary starting vertex

- if the DFS reveals a back edge, topological sort doesn't exist
- otherwise, sort the vertices by decreasing finishing times

10.10 Breadth first search

DFS has inefficiency problems with some specific graph structures.

BFS explores the local neighborhood first.

DFS uses a stack, BFS uses a queue.

Algorithm:

```
function BFS(G)
  for all $u \in V$
    u.color = white # not visited yet
  for all $s \in V$
    if s.color == white
      BFS-Visit(G, s)

function BFS-Visit(G, s)
  u.color = grey # in process
  Q = [s] # queue containing s
  while Q $\neq \emptyset$
    u = dequeue(Q)
    for all $v \in \text{Adj}(u)$
      if v.color == white
        v.color = grey
        enqueue(Q, v)
    u.color = black
```

Running time:

- $\mathcal{O}(|E| + |V|)$ in adjacency list
- $\mathcal{O}(|V|^2)$ in adjacency matrix

10.11 Shortest path problem (unweighted)

$$d(u, v) = \min\{l(\pi) \mid \pi \text{ path between } u \text{ and } v\}$$

Simple **algorithm** using BFS:

```
function BFS(G)
  for all $u \in V \setminus \{s\}$
    u.color = white # not visited yet
    u.dist = $\infty$
  s.dist = 0
  s.color = grey # in process
  Q = [s] # queue containing s
  while Q $\neq \emptyset$
    u = dequeue(Q)
    for all $v \in \text{Adj}(u)$
      if v.color == white
        v.color = grey
        v.dist = u.dist + 1
        enqueue(Q, v)
    u.color = black
```

Other **algorithm** using BFS, easily provable:

```
function BFS(s)
    d = [$\infty$, ...,$\infty$]
    parent = [$bot$, ..., $bot$]
    d[s] = s
    Q = {s}
    Q' = {s}
    for l = 0 to $\infty$ while $Q \neq \emptyset$ do
      for each $u \in Q$ do
        for each $(u, v) \in E$ do
          if parent(v) = $\perp$ then
            Q' = Q' $\cup \{v\}$
            d[v] = l + 1
            parent[v] = u
      (Q,Q') = (Q',$\emptyset$)
    return (d, parent)
```

10.12 Testing whether a graph is bipartite

Algorithm:

- assume the graph is connected or run on each component
- start BFS with arbitrary vertex, color start red
- neighbors of a red vertex become blue
- neighbors of a blue vertex become red
- bipartite iff there's no color conflict

10.13 Shortest path problems

- **Single Source Shortest Paths**: Shortest path distance of one particular vertex s to all other vertices
- **All Pairs Shortest Paths**: Shortest path distance between all pairs of points
- **Point to Point Shortest Paths**: Shortest path distance between a particular start vertex s and a particular target vertex t

10.13.1 Storing paths efficiently

Keep track of the predecessors in the shortest paths with the help of a **predecessor matrix** $\Pi = (\pi_{ij})_{i,j=1,\dots,n}$:

- If $i = j$ or there is no path from i to j, set $\pi_{ij} = \text{NIL}$
- Else set π_{ij} as the predecessor of j on a shortest path from i to j

Space requirement:

- SSSP: $\mathcal{O}(|V|)$
- APSP: $\mathcal{O}(|V|^2)$

10.14 Relaxation

- for each vertex, keep an attribute $v.dist$ that is the current estimate of the shortest path distance to the source vertex s

- initially set to ∞ for all vertices except start
- step: figure out whether there is a shorter path from s to v by using an edge (u, v) and thus extending the shortest path of s to u

Formally:

```
function Relax(u,v)
    if v.dist > u.dist + w(u, v)
      v.dist = u.dist + w(u,v)
      v.$\pi$ = u
```

Also useful:

```
function InitializeSingleSource(G,s)
    for all $v \in V$
      v.dist = $\infty$ # current distance estimate
      v.$\pi$ = NIL
    s.dist = 0
```

10.15 Bellman-Ford algorithm

SSSP algorithm for general weighted graphs (including negative edges).

```
function BellmanFord(G,s)
    InitializeSingleSource(G,s)
    for i = 1, ..., |V| - 1
      for all edges $(u,v) \in E$
        Relax(u,v)
    for all edges $(u,v) \in E$
      if v.dist > u.dist + w(u,v)
        return false # cycle detected
    return true
```

Running time: $\mathcal{O}(|V| \cdot |E|)$

Note

Originally designed for directed graphs. Edges need to be relaxed in both directions in an undirected graph. Negative weights in an undirected graph result in an undefined shortest path.

10.16 Decentralized Bellman-Ford

Idea: "push-based" version of the algorithm: Whenever a value `v.dist` changes, the vertex `v` communicates this to its neighbors.

Synchronous algorithm:

```
function SynchronousBellmanFord(G,w,s)
    InitializeSingleSource(G,s)
    for i = 1, ..., |V| - 1
      for all $u \in V$
        if u.dist has been updated in previous iteration
          for all edges $(u,v) \in E$
            v.dist = min{v.dist , u.dist + w(u, v)}
      if no v.dist changed
```

```
        terminate algorithm
```

Asynchronous algorithm for static graphs with non-negative weights:

```
function AsynchronousBellmanFord(G,w,s)
    InitializeSingleSource(G,s)
    set $s$ as active, other nodes as inactive
    while an active node exists:
      u = active node
      for all edges $(u,v) \in E$
        v.dist = min{v.dist, u.dist + w(u, v)}
        if last operation changed v.dist
          set v active
      set u inactive
```

10.17 Dijkstra's algorithm

Works on any weighted, (un)directed graph in which all edge weights $w(u,v)$ are non-negative.

Greedy algorithm: At each point in time it does the "locally best" action resulting in the "globally optimal" solution.

10.17.1 Naive algorithm

Idea:

- maintain a set S of vertices for which we already know the shortest path distances from s
- look at neighbors u of S and assign a guess for the shortest path by using a path through S and adding one edge

```
function Dijkstra(G,s)
    InitializeSingleSource(G,s)
    S = {s}
    while S $\neq$ V
      U = {u $\notin$ S | u neighbor of vertex $\in$ S}
      for all u $\in$ U
        for all pre(u) $\in$ S that are predecessors of u
          d'(u, pre(u)) = pre(u).dist + w(pre(u), u)
      d* = min{d'(u,pre(u)) | u $\in$ U, pre(U) $\in$ S}
      u* = argmin{d'(u,pre(u)) | u $\in$ U, pre(U) $\in$ S}
      u*.dist = d*
      S = S $\cup$ {u*}
```

Running time: $\mathcal{O}(|V| \cdot |E|)$

10.17.2 Using min-priority queues

Algorithm:

```
function Dijkstra(G,w,s)
    InitializeSingleSource(G,s)
    Q = (V, V.dist)
    while Q $\neq \emptyset$
      u = Extract(Q)
      for all v adjacent to u
        Relax(u,v) and update keys in Q
```

It follows that $Q = V \setminus S$.

Running time: $\mathcal{O}((|V| + |E|) \log |V|)$

10.18 All pairs shortest paths

Naive approach:

- run Bellman-Ford or Dijkstra with all possible start vertices
- running time of $\approx \mathcal{O}(|V|^2 \cdot |E|)$
- doesn't reuse already calculated results

Better: Floyd-Warshall

10.19 Floyd-Warshall algorithm

Idea:

- assume all vertices are numbered from 1 to n.
- fix two vertices s and t
- consider all paths from s to t that only use vertices $1, ..., k$ as intermediate vertices. Let $\pi_k(s,t)$ be a shortest path *from this set* and denotee its length by $d_k(s,t)$
- recursive relation between π_k and π_{k-1} to construct the solution bottom-up

Algorithm:

```
function FloydWarshall(W)
    n = number of vertices
    $D^{(0)}$ = W
    for k = 1,...,n
      let $D^{(k)}$ be a new n $\times$ n matrix
      for s = 1,...,n
        for t = 1,...,n
          $d_k$(s,t) = min{$d_{k-1}$(s,t), $d_{k-1}$(s,k) + $d_{k-1}$(k,t)}
    return $D^{(n)}$
```

Running time: $\mathcal{O}(|V|^3) \implies$ not that much better than naive approach but easier to implement

Note

Negative-weight cycles can be detected by looking at the values of the diagonal of the distance matrix. If it contains negative entries, the graph contains a negative cycle.

10.20 Point to Point Shortest Paths

Given a graph G and two vertices s and t we want to compute the shortest path between s and t only.

Idea:

- run `Dijkstra(G,s)` and stop the algorithm when we reached t
- has the same worst case running time as Dijkstra
  - often faster in practice

10.21 Bidirectional Dijkstra

Idea:

- instead of starting Dijkstra at s and waitung until we hit t, we start copies of the Dijkstra algorithm from s as well as t
- alternate between the two algorithms, stop when they meet

Algorithm:

- $\mu = \infty$ (best path length currently known)
- alternately run steps of `Dijkstra(G,s)` and `Dijkstra(G',t)`
  - when an edge (v, w) is scanned by the forward search and w has already been visited by the backward search:
    - found a path between s and t: $s ... v\ w ... t$
    - length of path is $l = d(s, v) + w(v, w) + d(w, t)$
    - if $\mu > l$, set $\mu = l$
  - analogously for the backward search
- terminate when the search in one direction selects a vertex v that has already been selected in other direction
- return μ

Note

It is not always true that if the algorithm stops at v, that then the shortest path between s and t has to go through v.

10.22 Generic labeling method

A convenient generalization of Dijkstra and Bellman-Ford:

- for each vertex, maintain a status variable $S(v) \in \{\text{unreached}, \text{labelchanged}, \text{settled}\}$
- repeatedly relax edges
- repeat until nothing changes

```
function GenericLabelingMethod(G,s)
    for all v ∈ V
      v.dist = ∞
      v.parent = NIL
      v.status = unreached
    s.dist = 0
    s.status = labelchanged
    while a vertex exists with status labelchanged
      pick such vertex v
      for all neighbors u of v
        Relax(v,u)
      if relaxation changed value u.dist
        u.status = labelchanged
    v.status = settled
```

10.23 A* search

Idea:

- assume that we know a lower bound $\pi(v)$ on the distance $d(v,t)$ for all vertices v:

$$\forall v : \pi(v) \leq d(v,t)$$

- run the *Generic Labeling Method* with start in s
- while Dijkstra selects by $d(s,u) + w(u,v)$, A* selects by $d(s,u) + w(u,v) + \pi(v)$

Algorithm:

```
function AstarSearch(G,s,t)
    for all v ∈ V
      v.dist = ∞
      v.status = unreached
    s.dist = 0
    s.status = labelchanged
    while a vertex exists with status labelchanged
      select u = argmin(u.dist + $\pi$(u))
      if u == t
        terminate, found correct distance
      for all neighbors v of u
        Relax(u,v)
        if relaxation changed value v.dist
          v.status = labelchanged
      u.status = settled
```

Running time: If the lower bounds are feasible, A*-search has the same running time as Dijkstra. Can often work fast but in rare cases very slow.

10.24 Union-find data structure

TODO.

10.25 Operation O1

TODO.

10.26 Minimal spanning trees

Idea:

Given an undirected graph $G = (V,E)$ with real-valued edge values $(w_e)_{e \in E}$ find a tree $T = (V', A)$ with $V = V', A \subset E$ that minimizes

$$\text{weight}(T) = \sum_{e \in A} w_e.$$

Minimal spanning trees are not unique and most graphs have many minimal spanning trees.

10.26.1 Safe edges

Given a subset A of the edges of an MST, a new edge $e \in E \setminus A$ is called **safe** with respect to A if there exists a MST with edge set $A \cup \{e\}$.

- start with MST $T = (V, E')$
- take some of its edges $A \subset E'$
- new edge e is *safe* if $A \cup \{e\}$ can be completed to an MST T'

10.26.2 Cut property to find safe edges

A cut $(S, V \setminus S)$ is a partition of the vertex set of a graph in two disjoint subsets.

TODO.

10.26.3 Kruskal's algorithm

Idea:

- start with an empty tree
- repeatedly add the lightest remaining edge that does not produce a cycle
- stop when the resulting tree connects the whole graph

Naive algorithm using cut property:

```
function KruskalNaiveMST(V,E,W)
    sort all edges according to their weight
    A = {}
    for all e ∈ E, in increasing order of weight
      if A ∪ {e} does not contain a cycle
        A = A ∪ {e}
        if |A| = n - 1
          return A
```

Running time:

- sorting $\mathcal{O}(|E| \log |E|)$
- check for cycle: $\mathcal{O}(|E| \cdot ?)$
- total: $\mathcal{O}(|E| \log |E| + |E| \cdot ?)$