Algorithms - Summary

Sebastien Speierer

17 janvier 2014

Table des matières

1 Sorting algorithms

1.1 Insertion Sort

We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left. The running time of this algorithm is $\Theta(n^2)$

Algorithm 1: Insertion sort algorithm

```
Data: A the array to sort
Result: The sorted array
for j = 2 to A.length do
    key = A[j];
    i = j-1;
    while i > 0 and A[i] > key do
        A[i+1] = A[i];
        i = i -1;
    end
    A[i+1] = key;
end
```

1.2 Merge Sort

The merge sort is a divide-and-conquer algorithm and is running time is $\Theta(n \log n)$.

Algorithm 2: Merge sort algorithm

```
Data: A the array to sort,p,r
Result: The sorted array
if p < r then
    q = ⌊(p + r)/2⌋;
    MERGE-SORT(A,p,q) //subproblem with the first half of the array ;
    MERGE-SORT(A,q +1,r) //subproblem with the second half of the array ;
    MERGE(A,p,q,r) //merge the two subproblems ;
end
```

The merge function use the fact that the two subproblem are already sorted. So it can be done in $\Theta(n)$ (Iterate on the two subarrays and at each step take the smaller element).

1.3 Heap Sort

1.3.1 Heap

The (binary) **heap** data structure is an array object that we can view as a nearly complete binary tree. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. Each node of the tree corresponds to an element of the array. Here are the tree basic function of the heap :

– PARENT(i) = return $\lfloor 1/2 \rfloor$
– LEFT(i) = return 2i
– RIGHT(i) = return 2i + 1

There are the max-heap and the min-heap. In the max-heap, the max-heap property is that for every node i other than the root, A[PARENT(i)] $\geq$ A[i].

The height of a node in a heap is the number of edge on the longest simple downward path from the node to a leaf. The height of a heap of n element is $\Theta(\log n)$. Here are the basic function on the heap used in the heap sort and to do a priority queue data structure :

- MAX-HEAPIFY(A,i) : Runs in $O(\log n)$ and is used to maintain the max-heap property. In this procedure, we suppose that the LEFT(i) and RIGHT(i) are already max-heaps, and we switch at each step with the highest children (it A[i] is lower than its children). If a switch is done, we call recursively MAX-HEAPIFY on the node that have been switched (the new location of A[i]).
- BUILD-MAX-HEAP : run in $\Theta(n)$ and produces a max-heap from an unordered input array. This function call iteratively MAX-HEAPIFY from the node in the middle (A.length/2) up to the root.
- HEAPSORT : runs in $O(n \log n)$ and sorts an array in place. We know that in a max-heap, the maximum element is always stored at the root. The algorithm goes like this :

Algorithm 3: Heap sort algorithm

```
Data: A the array to sort
Result: The sorted array
BUILD-MAX-HEAP(A);
for i = A.length downto 2 do
    exchange A[1] with A[i];
    A.heap-size = A.heap-size -1;
    MAX-HEAPIFY(A,1)
end
```

1.3.2 Priority Queue

The priority queue (max-priority queue) is a data-structure for maintaining a set S of elements, each with an associated value called a key. It has the following operations :

- HEAP-MAXIMUM : return the element of S with the largest key that is A[1] in $\Theta(1)$.
- HEAP-EXTRACT-MAX : removes and return the maximum in $O(\log n)$. It returns A[1], put the A[heap-size] in A[1] and call MAX-HEAPIFY(A,1). (also decrements the heap-size)
- HEAP-INCREASE-KEY(A,i,key) : it change the value of A[i] with key (only if key is higher) in $O(\log n)$. It update the value and then with a while loop, up the node while the key is higher than its parent's key.
- MAX-HEAP-INSERT(A,key) : insert the element x into the set S in $O(\log n)$. It increment the heap-size, set A[heap-size(+1)] to $-\infty$, then call HEAP-INCREASE-KEY(A,heap-size,key)

1.4 Quick Sort

The quick sort algorithm has a worst case running time of $\Theta(n^2)$ the it is often the best practical choice for sorting because it is remarkably efficient on the average-case running time of $\Theta(n \log n)$ with a small constant factor.

It is a divide-and-conquer algorithm that at each step, take a pivot and partition the array into two subarrays such that each elements of the first are less or equal to the pivot and each elements of the second are greater than the pivot. Then to combine the subproblems, there is nothing to do because everything are already sorted.

The key to the algorithm is the PARTITION precedure, which rearranges the subarray in place :

Algorithm 4: Partition algorithm for quicksort

```
Data: A,p,r
x = A[r] ;
i = p − 1 ;
for j = p to r − 1 do
    if A[j] ≤ x then
        i = i + 1 ;
        exchange A[i] whit A[j];
    end
end
exchange A[i+1] with A[r] // put the pivot in the good place;
return i + 1 // return the pivot's index ;
```

In order to avoid the worst-case, we would use a randomised version of the quick sort algorithm. This algorithm take a random pivot instead of taking the last element of the array (just exchange a random element with the last element each time we need to partition).

1.5 Counting sort

The counting sort is an linear-time sorting algorithm which assumes that each of the n input elements is an integer in the range 0 to k. It first create an array C[0..k] that contain, for each different possible value the number of element that are less or equal to this value. Then it can directly put the elements in the rights place in a new array B[1..n].

Algorithm 5: Counting sort algorithm

```
Data: A, B, k
let C[0..k] a new array;
for j = 1 to A.length do
    C[A[j]] = C[A[j]]+1 ;
end
// now C[i] contains the number of elements equal to i;
for j = 1 to k do
    C[i] = C[i] + C[i-1] ;
end
// now C[i] contains the number of elements less than or equal to i ;
for j = A.length downto 1 do
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]]-1 ;
end
```

This algorithm's running time is $\Theta(n + k)$.

2 Divide and conquer

Some algorithms follow a **divide-and-conquer** approach : they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem. The running time of a divide-and-conquer algorithms can be describe by :

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

where $D(n)$ is the cost of the division, $C(n)$ is the combination time. Our division of the problem yields a subproblems, each of which is $1/b$ the size of the original.

2.1 Substitution method

The **substitution method** comprises two steps :

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

2.2 Recursion-tree method

In a recursion tree, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. We sum the cost within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level cost to determine the total cost of all levels of the recursion. It is best used to generate a good guess.

2.3 Master method

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function. and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n)$$

Then $T(n)$ has the following asymptotic bounds :
– If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
– If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log(n))$.
– if $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

3 Probabilistic Analysis and Randomized Algorithms

We analyse our algorithm, computing an **average-case running time**, where we take the average over the distribution of the possible inputs. We call an algorithm **randomized** if its behavior is determined not only by its input, but also by values produced by a random-number generator. (the choose of the pivot in quicksort, or the mix of the input in the hiring problem)

3.0.1 The Birthday paradox

The Birthday paradox show that if at least 23 people are in a room, the probability that two of them have the same birthday is at least 50%. It is used to show that in a hash table, in order to have good probability of no collision, you should have a table length of $m > K^2$, where K is number of different keys.

4 Dynamic Programming

The idea behind the dynamic programming is to breaking the problem into simpler subproblems and combine them to find a solution. Moreover, the dynamic programming approach seeks to solve each subproblems only once, thus reducing the number of computations. There are two different approaches in the dynamic programming :

- The first approach is **top-down with memoization**. In this apporach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblems (usually in a array). The procedure now first checks to see whether it has preciously solved this subproblem. If so it returns the saved value, saving further computation at this level ; if not, the procedure computes the value in the usual manner and save it in the array. We say that the recursive procedure has been **memoized** ; it remembers what results it has computed previously.
- The second approach is the **bottom-up method**. We sort the subproblems by size and solve them in size order, smallest first. When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions. We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

Algorithm 6: Top-down Fibonacci algorithm : Fib(n,r)

```
Data: n, r the array with the saved values (init to −∞)
if n == 0 then
    return 0;
end
if n == 1 then
    return 1;
end
if r[n] == −∞ then
    r[n] = Fib(n-1) + Fib(n-2);
end
return r[n];
```

Algorithm 7: Bottom-up Fibonacci algorithm : Fib(n)

```
Data: n
let r[0..n] be a new array (with values initialize to −∞);
r[0] = 0 ;
r[1] = 1 ;
for j = 2 to n do
    r[j] = r[j-1] + r[j-2];
end
return r[n];
```

Rod cutting

For each step of the rod cutting, we choose the best of the optimal solution for the length i adding the price of the piece of length $n - i$:

$$r_n = \max_{1 \leq i \leq n} (p_1 + r_{n-i})$$

Matrix-chain multiplication

For each step of the matrix-chain multiplication, we take the best of the multiplication of the first i matrix, the multiplication of the $j - k$ last matrix plus the multiplication of these two matrix :

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j \end{cases}$$

Longest common subsequence

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_i \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_i \end{cases}$$

Optimal binary search trees

For each step of the optimal binary search trees computation, we take the best of the expected cost of the left subtree, plus the expected cost of the right subtree, plus the sum of all the probability :

$$e[i,j] = \begin{cases} 0 & \text{if } i = j + 1 \\ \min_{i \leq r \leq j}\{e[i, r-1] + e[r+1, j] + \sum_{l=i}^{j} p_l\} & \text{if } i \leq j \end{cases}$$

5 Data Structures

5.1 Stack

A stack is a dynamic sets in which the element removed from the set by the DELETE operation is prespecified. In a stack, the element deleted from the set is the one most recently inserted : the stack implements a **last-in, first-out** policy.
The INSERT operation is often called PUSH, and the DELETE operation is often called POP. We can implement a stack of at most n elements with an array S[1..n]. The array has an attribute S.top that indexes the most recently inserted element. When S.top = 0, the stack is empty. The PUSH(S,x) function increment S.top and do S[S.top] = newElement. The POP(S) function first checks if the stack is empty, decrements S.top and return S[S.top+1].

5.2 Queue

A queue is a dynamic sets in which the element removed from the set by the DELETE operation is prespecified. In a queue, the element deleted is always the one that has been in the set for the longest time : the queue implements a **first-in, first-out** policy.
We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE. The queue has two attribute, the **head** and the **tail**. On way to implement a queue of at most $n-1$ elements is to use an array Q[1..n]. The ENQUEUE(Q,x) add the element in the array at the position Q.tail and increment (modulo n) the Q.tail. The DEQUEUE(Q) increment (modulo n) the Q.head and then return Q[(Q.head$-1)\%n$].

5.3 Linked List

A linked list is a data structure in which the objects are arranged in a linear order. In a **doubly linked list**, each element is an object with an attribute key and two other pointer attributes : prev and next. If x.prev = NIL, it means that x is the head of the list and if x.tail = NIL, it means that x is the tail of the list.
The **LIST-SEARCH(L,k)** function finds the first element with key k in list L by a simple iterative search (with a while loop), returning the pointer to this element. It is done in $\Theta(n)$.
The **LIST-INSERT(L,x)** function simply put the element in front of the list (just also set the prev of the previous head to the new one). This is obviously done in O(1).
The **LIST-DELETE(L,x)** is done in O(1) if with receive a pointer to the element to delete. Otherwise, it is done in O(n) because we first have to find the element (with LIST-SEARCH). Then we just have to set the pointers (next and prev).
A **sentinel** is a object that allows us to simplify boundary conditions. It is implement by L.nil and it contains the pointer to the tail and the pointer to the head.

$$L.nil.prev.next = L.nil /// L.nil.next = "head" /// ...$$

5.4 Hash Table

A hash table is a data structure where the elements are saved in an array T[0..m-1] of linked lists. A hash table uses a hash function to compute an index i such that the element must be found in the list T[i]. Two elements hashes to the same slot, we call this situation a collision. One solution is to place all the elements that hash to the same slot into the same linked list.
With this data structure, the INSERT running time is O(1), the DELETE running time is O(1) (if we use double linked list in the array). On average, in a hash table, the searching function takes constant time too, O(1) (complex analysis).

5.5 Binary Search Tree

A binary search tree can be represent such a tree by a linked data structure in which each node is an object that has, in addition of its key, attributes left, right and p corresponding to its parent. If a child or the parent is missing, the appropriate attribute contains the value NIL. The root node is the only node in the tree whose parent is NIL. The keys in a binary search tree are always stored in such a way as to satisfy that :

$$x.key > x.left.key \text{ and } x.key < x.right.key$$

We can iterate on the binary search tree by three different processes :
- **Inorder tree walk :** prints the root of a subtree between printing the values in its left subtree and printing those in its right subtree.
- **Preorder tree walk :** prints the root before the values in either subtree.
- **Postorder tree walk :** prints the root after the values in its subtrees.

The **TREE-SEARCH(x,k)** procedure is done recursively. If the current node contains the key, it return it. Else it compare the searched key with its one and then call recursively TREE-SEARCH(x.left,k) or TREE-SEARCH(x.right,k) (depends if x.key $>$ or $<$ k). The running time of this function is $O(h)$ where h is the height of the tree.

The **MAXIMUM(t)** and **MINIMUM(t)** functions are very simple. They just return the node the most on the left (min) or the most on the right (max) of t using a while loop. These function are also in $O(h)$

The **SUCCESSOR(x)** function and the **PREDECESSOR(x)** work the same. For SUCCESSOR(x), first, if x.right $\neq$ NIL, then we simply return the MINIMUM(x.right). Otherwise, with a while loop, we look for the first parent that is one the right side of this node. The running time of this function is also $O(h)$

INSERT(T,z) : In order to insert a element in the tree, we first have to find the position of this new element in the tree. We do that with a while loop, going down in the tree. When we finally reach a leaf, we just have to set all the corresponding pointers (check if the tree wasn't empty).

DELETE(T,z) : From the deletion, there are many cases to take into account :
- z has no children : we simply remove it (setting the corresponding pointers).
- z has just one child : we set the pointers such that the child replace z.
- z has two children :
  1. find the z's successor y
  2. y takes z's position : y.right takes y position, y.right = z.right and y.left = z.left (lots of other pointers to set !) $\Leftarrow$ a bit tricky !!!

5.6 Disjoint Sets

A disjoint-set data structure maintains a collection $S = \{S_1, S_2, ..., S_k\}$ of disjoint dynamic set. Each set is identified by a **representative**, which is some member of the set.

MAKE-SET(x) creates a new set whose only member is x. We require that x is not already in some other set.

UNION(x,y) unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets and its representative is any member of $S_x \bigcup S_y$. Since we require the sets in the collection to be disjoint, we destroy sets S_x and S_y, removing them from the collection S.
FIND-SET(x) returns a pointer to the representative of the unique set containing x.

A simple way to implement a disjoint-set data structure is to do as each set is represented by its own linked list. Each set has the attributes head and tail and each member of the set contains a pointer to the next element and a pointer back to the set object.
To carry out **MAKE-SET(x)**, we simply create a new linked list whose only object is x and this is done in O(1).

For **FIND-SET(x)**, we just follow the pointer from x back to its set object and then return the member in the object that had point to (also in O(1)).

We perform **UNION(x,y)** by appending y's list onto the end of x's list. The representative of x's becomes the representative of the resulting set. We use the tail pointer for x's list to quickly find where to appends y's list. We also must update the pointer to the set object for each object originally on y's list. This operation is in O(m) where m is the size of y.

To avoid to union the biggest set, we can simple suppose that each list also includes its length. Then, with this simple weighted-union heuristic, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, takes O(m + n log n) time.

6 Graph Algorithms

A graph is a combination of a set of vertices V and a set of edges E. So we have the graph $G = (V, E)$.

6.1 Representations of graphs

We can represent a graph $G = (V, E)$ as a collection of adjacency lists or as an adjacency matrix.
– The **adjacency-list representation** consists of an array Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list Adj[u] contains all the vertices v such that there is an edge $(u, v) \in E$. The amount of memory required for this representation is $\Theta(V + E)$.
– The **adjacency-matrix representation** consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

This representation requires $\Theta(V^2)$ memory, independent of the number of edges in the graph. For this reason, the list representation is better for the sparse graphs.

6.2 Breadth-first search

Breadth-first search is one of the simplest algorithms for searching a graph and works on both, directed and undirected graphs. Given a graph G and a source vertex s, it explores the edges of G to discover every vertex that is reachable from s. It computes the distance from s to each reachable vertex. Then it iteratively take the nearest non-used vertex and discovers all its adjacent vertices. Moreover, BFS keeps track of the predecessor of each vertices and colors them :
– white ⇒ non-discovered vertex
– gray ⇒ discovered vertex

– black ⇒ finished vertex, its adjacency list has been examined completely
Here is an implementation of BFS using a queue :

Algorithm 8: Breadth-first search

```
Data: G,s)
initialize all the vertices to WHITE, set predecessor (π) to NIL and set the distance to ∞ ;
set s GRAY and distance 0;
initialize a queue Q and enqueue s in Q
while Q ≠ ∅ do
    u = DEQUEUE(Q);
    for each v ∈ G.Adj[u] do
        if v.color == WHITE then
            v.color = GRAY ;
            v.π = u;
            v.distance = u.distance + 1;
            ENQUEUE(Q,v);
        end
    end
    u.color = BLACK ;
end
```

The running time of BFS is $O(V + E)$.

6.3 Depth-first search

The strategy followed by depth-first search is to search "deeper" in the graph whenever possible. Depth-first search explores edges out of the most recently discovered vertey 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 we have discovered all the vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex.
As in BFS, DFS colors vertices during the search to indicate their state. DFS also **timestamps** each vertex. Each vertex v has two timestamps : the first v.d records when v is first discovered (and grayed), and the second, v.f, records when the search finishes examining v's adjacency list (and blackens v).

Algorithm 9: Depth-first search : DFS(G)

```
Data: G
initialize all vertices with color WHITE and predecessor to NIL;
time = 0 ;
for each vertex u ∈ G.V do
    if u.color == WHITE then
        DFS-VISIT(G,u);
    end
end
```

Algorithm 10: DFS-VISIT(G,u)

```
Data: G,u
time = time + 1;
u.d = time;
u.color = GRAY ;
for each v∈ G.Adj[u] do
    if v.color == WHITE then
        v.π = u ;
        DFS-VISIT(G,v);
    end
end
u.color = BLACK;
time = time +1;
u.f = time ;
```

The running time of DFS is therefore $\Theta(V + E)$.

6.4 Topological sort

We can use depth-first search to perform a topological sort of a directed acyclic graph. We can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right. (Example with the garments and the topological sort gives an order for getting dressed) Topological sort can be done just using the DFS and by adding a vertex to a linked list when it is finished. It has so a running time of $\Theta(V + E)$.

6.5 Minimum Spanning Tree

Given a undirected graph G and for each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost to connect u and v. We then wish to find an acyclic subset T$\subseteq$ E that connects all of the vertices and whose total weight is minimized.

6.5.1 Kruskal's algorithm

Kruskal's algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u, v) of least weight. Here is an algorithm that uses the disjoint set data structure and has a running time of O(E log V) :

Algorithm 11: Kruskal's algorithm

```
Data: G,w the function that gives the weight
A = ∅;
for each vertex v∈ G.V do
    MAKE-SET(v);
end
sort the edges of G.E into nondecreasing order by weight;
for each edge (u,v)∈ G.E, taken in nondecreasing order by weight do
    if FIND-SET(u) ≠ FIND-SET(v) then
        A = A ⋃{(u, v)} ;
        UNION(u,v);
    end
end
return A;
```

6.5.2 Prim's algorithm

Prim's algorithm has the property that the edges in the set A always form a single tree. The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V. Each step adds to the tree A a light edge that connects A to an isolated vertex, one on which no edge of A is incident. We need a fast way to select a new edge to add to the tree formed by the edges in A. During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue Q based on a key attribute. For each vertex v, the attribute v.key is the minimum weight of any edge connecting v to a vertex in the tree.

Algorithm 12: Prim's algorithm

```
Data: G,w,r the root
for each u ∈ G.V do
    u.key = ∞;
    u.π = NIL;
end
r.key = 0;
Q = G.V // the min-priority queue;
while Q ≠ ∅ do
    u = EXTRACT-MIN(Q);
    for each v ∈ G.Adj[u] do
        if v ∈ Q and w(u,v) < v.key then
            v.π = u;
            v.key = w(u,v);
        end
    end
end
```

This implementation of the Prim's algorithm has a running time of O(E + V logV).

6.6 Bellman-Ford algorithm

The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative. The algorithm relaxes edges, progressively decreasing an estimate v.d on the weight of a shortest path from the source s to each vertex v ∈ V until it achieves the actual shortest-path weight.

Algorithm 13: Bellman-Ford algorithm

```
Data: G,w,s
initialize all the vertices but s to distance ∞, set their predecessor to NIL and set s.d =0;
for i = 1 to |G.V| − 1 do
    for each edge (u,v) ∈ G.E do
        RELAX(u,v,w) // check on all its "parent" for the shortest path
    end
end
for each edge (u,v) ∈ G.E do
    // check if the graph contain negative-weight cycles if v.d > u.d + w(u,v) then
        return FALSE;
    end
end
return TRUE;
```

The Bellman-Ford algorithm runs in time $O(VE)$.

6.7 Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph for the case in which all edge weights are nonnegative. It maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.

Algorithm 14: Dijkstra's algorithm

```
Data: G,w,s
initialize all the vertices but s to distance ∞, set their predecessor to NIL and set s.d =0;
S = ∅;
Q = G.V;
while Q ≠ ∅ do
    u = EXTRACT-MIN(Q);
    S = S⋃{u} ;
    for each vertex v ∈ G.Adj[u] do
        RELAX(u,v,w);
    end
end
```

The running time of the Dijkstra's algorithm is O(VlogV + E).

6.8 Flow networks

A flow network G=(V,E) is a directed graph in which each edge (u,v) ∈ E has a nonnegative capacity c(u,v)≥ 0. We further require that if E contains an edge (u,v), then there is no edge (v,u) in reverse direction. We distinguish two vertices in a flow network : a **source** s and a **sink** t. A **flow** in G is a real-valued function $f : V \times V \to \mathbb{R}$ that satisfies the following two properties :

Capacity contraint : For all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$

Flow conservation : For all $u \in V - \{s, t\}$, we require

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

In a **maximum-flow problem**, we are given a flow network G with source s and sink t, and we wish to find a flow of maximum value.

6.9 Ford-Fulkerson method

The Ford-Fulkerson method iteratively increases the value of the flow. We start with $f(u, v) = 0$ for all $u, v \in V$, giving an initial flow of value 0. At each iteration, we increase the flow value in G by finding an "augmenting path" in an associated "residual network" G_f. Once we know the edges of an augmenting path in G_f, we can easily identify specific edges in G for which we can change the flow so that we increase the value of the flow. We repeatedly augment the flow until the residual network has no more augmenting paths.

The **residual network** G_f consists of edges with capacities that represent how we can change the flow on edges of G. More formally, suppose that we have a flow network $G = (V, E)$ with source s and sink t. Let f be a flow in F, and consider a pair of vertices $u, v \in V$. We define the residual capacity $c_f(u, v)$:

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & if (u, v) \in E \\ f(v, u) & if (v, u) \in E \\ 0 & \text{otherwise} \end{cases}$$

So the residual network of G induced by f is $G_f = (V, E_f)$, where

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$$

Given a flow network G and a flow f, an **augmenting path** p is a simple path from s to t in the residual network G_f. By the definition of the residual network, we may increase the flow on an edge (u,v) of an augmenting path by up to $c_f(u, v)$ without violating the capacity constraint on whichever of (u,v) and (v,u) is in the original flow network G.

A **cut** (S,T) of flow network G is a partition of V into S and $T0V - S$ such that $s \in S$ an $t \in T$. If f is a flow, hen the **net flow** f(S,T) across the cute (S,T) is defined to be

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u)$$

The **capacity** of the cut is :

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

A **minimum cut** of a network is a cut whose capacity is minimum over all cuts of the network.

The Max-flow min-cut theorem : If f is a flow in a flow network G, with source s and sink t, then the following conditions are equivalent :

1. f is a maximum flow in G
2. The residual network G_f contains no augmenting paths
3. $|f| = c(S, T)$ for some cut (S,T) of G with

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

Algorithm 15: Ford-Fulkerson algorithm

```
Data: G,w,t
for each edge (u,v) ∈ G.E do
    //initialize all flows to 0 (u,v).f = 0;
end
while there exists a path p from s to t in the residual network G_f do
    c_f(p) = min{c_f(u,v) : (u,v)in p};
    for each edge (u,v) in p do
        if (u,v) ∈ E then
            (u,v).f = (u,v).f + c_f(p);
        end
        else
            (v,u).f = (v,u.f − c_f(p));
        end
    end
end
```

The total running time of this Ford-Fulkerson algorithm is $O(E|f^*|)$, where $|f^*|$ is the maximum flow in the transformed network.

6.10 Maximum bipartite matching

Given an undirected graph G, a **matching** is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v. We say that a vertex $v \in V$ is matched by the matching M if some edge in M is incident on v; otherwise, v is **unmatched**. A **maximum matching** is is

a matching of maximum cardinality, that is, a matching M such that for any matching M', we have $|M| \geq |M'|$. We can use the Ford-Fulkerson method to find a maximum matching in an undirected bipartite graph G in time polynomial in $|V|$ and $|E|$. The trick is to construct a flow network in which flows correspond to matchings. We add two new vertices s and t where s is connected to the first partition of G and t to the second. Now we just have to apply the Ford-Fulkerson method.

7 Other

In place algorithm : is an algorithm which transforms input using a data structure with a small, constant amount of extra storage space. The input is usually overwritten by the output as the algorithm executes.

We use loop invariants to help us understand why an algorithm is correct :

1. **Initialization :** It is true prior to the first iteration of the loop.
2. **Maintenance :** If it is true before an iteration of the loop, it remains true before the next iteration.
3. **Termination :** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.