

CSC373 Algorithms

Jonah Chen

1 DIVIDE AND CONQUER

- Divide and Conquer algorithm:
 1. Divide problem of size n into a smaller subproblems of size n/b each
 2. Recursively solve each subproblem
 3. Combine the subproblem solutions into the solution of the original problem
- Runtime: $T(1) = c$ and $T(n) = aT(n/b) + cn^d$ for $n > 1$
- Master Theorem: $T(n)$ depends on relation between a and b^d .

$$\begin{cases} a < b^d : T(n) = \Theta(n^d) \\ a = b^d : T(n) = \Theta(n^d \log n) \\ a > b^d : T(n) = \Theta(n^{\log_b a}) \end{cases} \quad (1)$$

- Note that the running time does not depend on the constant c
- In many algorithms $d = 1$ (combining take linear time)
- Examples:
 - Merge sort — sorting array of size n ($a = 2, b = 2, d = 1 \rightarrow a = b^d$) so $T(n) = \Theta(n \log n)$
 - Binary search — searching sorted array of size n ($a = 1, b = 2, d = 0 \rightarrow a = b^d$) so $T(n) = \Theta(\log n)$

1.1 KARATSUBA MULTIPLICATION

- **Add** two binary n -bit numbers naively takes $\Theta(n)$ time
- **Multiply** two binary n -bit numbers naively takes $\Theta(n^2)$ time
- Divide and Conquer approaches

1. Multiply x and y . We can divide them into two parts

$$x = x_1 \cdot 2^{n/2} + x_0 \quad (2)$$

$$y = y_1 \cdot 2^{n/2} + y_0 \quad (3)$$

$$x \cdot y = x_1 \cdot y_1 \cdot 2^n + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^{n/2} + x_0 \cdot y_0 \quad (4)$$

- $T(n) = 4T(n/2) + cn; T(1) = c$
- $a = 4, b = 2, d = 1$ Master Theorem case 3, $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$.
- This is the same complexity of the naive approach, making this approach useless.

2. Reconsider (4), we may rewrite $(x_1 \cdot y_0 + x_0 \cdot y_1)$ as $(x_1 + x_0) \cdot (y_1 + y_0) - x_1 \cdot y_1 - x_0 \cdot y_0$

$$x \cdot y = x_1 \cdot y_1 \cdot 2^n + ((x_1 + x_0) \cdot (y_1 + y_0) - x_1 \cdot y_1 - x_0 \cdot y_0) \cdot 2^{n/2} + x_0 \cdot y_0 \quad (5)$$

- $T(n) = 3T(n/2) + cn; T(1) = c$
 - $a = 3, b = 2, d = 1$, Master Theorem case 3, $T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$
 - Minor issue: a carry may increase $x_1 + x_0$ and $y_1 + y_0$ to $\frac{n}{2} + 1$. We can easily prove this by isolating the carry bit and reevaluating the complexity.
- To deal with integers which doesn't have a power of 2 number of bits, we can pad the numbers with 0s to make them have a power of 2 number of bits. This may at most increase the complexity by 3x.
 - 1971: $\Theta(n \cdot \log n \cdot \log \log n)$
 - 2019: Harvey and van der Hoeven $\Theta(n \log n)$. We do not know if this is optimal.

1.2 STRASSEN'S MATMUL ALGORITHM

- Let A and B be two $n \times n$ matrices (for simplicity n is a power of 2), we want to compute $C = AB$.
- The naive approach takes $\Theta(n^3)$ time.

1. Divide A and B into 4 submatrices of size $\frac{n}{2} \times \frac{n}{2}$ each

$$A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix}. \quad (6)$$

Then, C can be calculated with

$$C_1 = A_1 B_1 + A_2 B_3 \quad (7)$$

$$C_2 = A_1 B_2 + A_2 B_4 \quad (8)$$

$$C_3 = A_3 B_1 + A_4 B_3 \quad (9)$$

$$C_4 = A_3 B_2 + A_4 B_4 \quad (10)$$

- $T(n) = 8T(n/2) + cn^2; T(1) = c$

– $a = 8, b = 2, d = 2$, case 3 $T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$

2. **Idea:** Compute C_1, C_2, C_3, C_4 with only 7 multiplications, not 8.

$$M_1 = (A_2 - A_4)(B_3 + B_4) \quad (11)$$

$$M_2 = (A_1 + A_4)(B_1 + B_4) \quad (12)$$

$$M_3 = (A_1 - A_3)(B_1 + B_2) \quad (13)$$

$$M_4 = (A_1 + A_2)B_4 \quad (14)$$

$$M_5 = A_1(B_2 - B_4) \quad (15)$$

$$M_6 = A_4(B_3 - B_1) \quad (16)$$

$$M_7 = (A_3 + A_4)B_1 \quad (17)$$

With these we can compute C_1, C_2, C_3, C_4 with only additions of the M matrices.

$$C_1 = M_1 + M_2 - M_4 + M_6 \quad (18)$$

$$C_2 = M_4 + M_5 \quad (19)$$

$$C_3 = M_6 + M_7 \quad (20)$$

$$C_4 = M_2 - M_3 + M_5 + M_7 \quad (21)$$

– $T(n) = 7T(n/2) + cn^2; T(1) = c$

– $a = 7, b = 2, d = 2$, case 3 $T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$

- If n is not a power of 2, we zero-pad the matrices to have n as a power of two. This may increase the complexity by at most a factor of 7.

1.3 MEDIAN OF UNSORTED ARRAYS

- For an unsorted array A , we can find the average, max, min, sum, etc. in linear time.
- The trivial algorithm is to sort A then get the median. That takes $O(n \log n)$ time.
- We will solve a more general problem: Find the k^{th} smallest element in A . (e.g. $A = 5, 2, 6, 7, 4$, $\text{Select}(A, 1) = 2$, $\text{Select}(A, 4) = 6$)
- if $|A| = 1$, then return $A[1]$. Otherwise find a splitter s in arbitrary element of A . Partition A into A^+ and A^- , then divide
- $T(n) = T(\max(|A^-|, |A^+|)) + cn = T(\max(i - 1, n - i)) + cn$.
- Worst case: $T(n) = T(n - 1) + cn = \Theta(n^2)$
- Best case: $T(n) = T(n/2) + cn = \Theta(n)$. Suppose $b > 1$, by the Master Theorem $T(n) = T(n/b) + cn = \Theta(n)$.

We define s is a good splitter if s is greater than $1/4$ of the elements of A and less than $1/4$ of the elements of A . We can make the following observation:

1. With this splitter, we will reduce the size to at most $3n/4$.

- 2. Half the elements are good splitters.

We should select splitter s uniformly at random.

- $P(\text{splitter is good}) = \frac{1}{2}$
- $P(\text{splitter is bad}) = \frac{1}{2}$
- We can show that the expected number of trials (splitter selections) until obtaining a good splitter is 2.

1.3.1 EXPECTED RUNTIME

$$\underbrace{n_0 \rightarrow n_1 \rightarrow n_2}_{\text{Phase 0, size} \leq n} \rightarrow \underbrace{n_3 \rightarrow n_4}_{\text{Phase 1, size} \leq \frac{3}{4}n} \rightarrow \underbrace{n_5 \rightarrow n_6}_{\text{Phase 2, size} \leq \frac{3^2}{4^2}n} \rightarrow \dots \quad (22)$$

- Phase j : input size $\leq (\frac{3}{4})^j n$
- Random variable $y_j = \#$ of recursive calls in phase j . Note that $E[y_j] = 2$.
- Random variable $x_j = \#$ of steps to all the recursive calls in phase j .
- Total number of steps is $x = x_0 + x_1 + x_2 + \dots$
- We can compute $E[x] = E[x_0] + E[x_1] + E[x_2] + \dots$

$$x_j \leq cy_j \frac{3^j}{4} n \quad (23)$$

$$E[x_j] \leq cE[y_j] \frac{3^j}{4} n \leq 2c \frac{3^j}{4} n \quad (24)$$

$$E[x] = \sum_j E[x_j] \leq \sum_{j=1}^{\infty} 2c \frac{3^j}{4} n = \frac{2c}{1 - \frac{3}{4}} n = 8cn = \Theta(n) \quad (25)$$

1.3.2 DETERMINISTIC ALGORITHM

- If $|A| \leq 5$ then we sort A and return the k^{th} smallest.
- Otherwise, partition A into $n/5$ groups of size 5 each, then find the median of each group (constant time) and store in list M . This takes linear time.
- Select the median of M with the Select algorithm, this is a good splitter.
- the worst case running time is $T(n) = T(\lceil \frac{n}{5} \rceil) + T(\lfloor \frac{3n}{4} \rfloor) + cn$.
- This recursive relation cannot be solved by the Master Theorem. We can prove using induction that $T(n) < 20cn$.

Question: Why groups of 5?

- With groups of 5, the total size of subproblems: $\frac{n}{5} + \frac{3n}{4} = \frac{19n}{20} < n$
- With groups of 3, the total size of subproblems: $\frac{n}{3} + \frac{3n}{4} = \frac{13n}{12} > n$, not sufficient.
- So group size of 5, 7, 9, 11, ... would also work.

2 CLOSEST PAIR OF POINTS

- Problem: Given a set of n points, find the pair of points that are the closest in $O(n \log n)$.

2.1 CLOSEST PAIR IN 2D

- Divide: points roughly in half by drawing vertical line on midpoint
- Conquer: Find closest pair on each half, recursively.
- Combine: Find the closest pair (p, q) , $p \in L$, $q \in R$. However, there may be $\Theta(n^2)$ pairs.
- Claim: Let $p = (x_p, y_p) \in B_L, q = (x_q, y_q) \in B_R$ with $y_p \leq y_q$. If $d(p, q) < \delta$ then there are at most **six** other points (x, y) in B such that $y_p \leq y \leq y_q$.
- Proof:
- $S_L = \{p' = (x, y) : p' \neq p \in B_L \wedge y_p \leq y \leq y_q\}$ (other points on the left of the middle)
- $S_R = \{p' = (x, y) : p' \neq q \in B_R \wedge y_p \leq y \leq y_q\}$ (other points on the right of the middle)
- Assume by contradiction that $|S_L \cup S_R| \geq 7$. WLOG assume $|S_L| \geq 4$.
- In a $\delta \times \delta$ square there are at least $4 + 1 = 5$ points. Divide the square into 4 smaller squares, by Pigeonhole Principle, there is a square with at least 2 points, whose distance is at most $\delta/\sqrt{2}$. This contradicts the assumption that the closest pair on the left is at most δ .
- Then, we can sort everything in the y axis, and check the next seven points by the y coordinate for the minimum distance. This takes linear time.
- We only need to modify the combine step in the algorithm so it's $\Theta(n)$ runtime.
- So $T(n) = 2T(\frac{n}{2}) + cn$, which is $O(n \log n)$.

Algorithm 1 Closest Pair in 2D

```

1: procedure CLOSESTPAIR( $P$ )
2:    $P_x :=$  the list of points in  $P$  sorted by x-coordinate
3:    $P_y :=$  the list of points in  $P$  sorted by y-coordinate
4: procedure RCP( $P_x, P_y$ )
5:   if  $|P_x| \leq 3$  then return brute force( $P_x$ )
6:    $L_x :=$  the first half of  $P_x$ ;  $R_x :=$  the second half of  $P_x$ 
7:    $m := (\max \text{ x-coordinate of } L_x + \min \text{ x-coordinate of } R_x)/2$ 
8:    $L_y :=$  sublist of  $P_y$  with points in  $L_x$ 
9:    $R_y :=$  sublist of  $P_y$  with points in  $R_x$ 
10:   $(p_L, q_L) := \text{RCP}(L_x, L_y)$ ;  $(p_R, q_R) := \text{RCP}(R_x, R_y)$ 
11:   $\delta := \min\{d(p_L, q_L), d(p_R, q_R)\}$ 
12:  if  $\delta = d(p_L, q_L)$  then
13:     $p := p_L$ ;  $q := q_L$ 
14:  else
15:     $p := p_R$ ;  $q := q_R$ 
16:   $B :=$  sublist of  $P_y$  with points in  $[m - \delta, m + \delta]$ 
17:  for each  $p$  in  $B$  do
18:    for each next seven  $q$  after  $p$  in  $B$  do
19:      if  $d(p, q) < d(p^*, q^*)$  then  $(p^*, q^*) := (p, q)$ 

```

3 GREEDY ALGORITHMS

- There is an optimization problem: given an input, find a solution that minimize/maximize an objective function f under some constraint.
- Build the solution incrementally in stages
- At each stage, extend the solution greedily and irrevocably.
- For some problems this gives optimal solutions (i.e. MST), but for other problems it does not.
- The order of the stages is very important.

3.1 INTERVAL SCHEDULING

- Input: n intervals, with interval j starts at time s_j and finishes at time f_j .
- Output: maximum-size set of intervals that do not overlap
- Naive algorithm is to try each subset of n intervals by brute force $O(2^n)$, way too slow.
- Greedy algorithm sorts interval in some order, then if it doesn't overlap then add it to the solution.
- **What order gives the biggest feasible set?**

1. Increasing start time: s_j
 2. Increasing finish time: e_j
 3. Shortest interval: $e_j - s_j$
 4. Fewest conflicts
- The question is which one is optimal? Option 1 has a clear counterexample with one very long interval that overlaps each other interval. Option 3 is also not optimal using a short interval between two long interval. Option 4 is not optimal (not very clear counterexample).
 - Option 2 is optimal. The intuition is that choosing these intervals first will leave the most time for the rest of the intervals.
 - To find if the interval is compatible, we just need to check if the start time of the new interval is greater than the finish time of the latest scheduled interval.

3.1.1 PROOF OF OPTIMALITY

- Suppose for contradiction that this greedy algorithm is not optimal
- Say greedy selects interval i_1, \dots, i_k sorted by increasing finish time.
- Suppose the optimal schedule j_1, \dots, j_m has $m > k$ intervals, and sort by increasing finish time. Consider an optimal schedule that can match the greedy $j_1 = i_1, \dots, j_r = i_r$ for the greatest possible r .
- By the nature of the greedy algorithm, then i_{r+1} finishes the earliest amongst the compatible intervals remaining. So consider the schedule $S : i_1, \dots, i_r, i_{r+1}, j_{r+2}, \dots, j_m$.
- This is also optimal (contains m intervals) but matches the greedy algorithm by one more position. This is a contradiction.

3.2 INTERVAL PARTITIONING

- Motivation: given a set of lecture time intervals, schedule them into as few classrooms as possible.
- Input: n intervals, interval j starts at s_j and finishes at f_j .
- Output: group interval into fewest partitions such that intervals in each partition are compatible.
- An idea is to find the maximum set of compatible intervals using the previous algorithm. This doesn't work.
- We can try the same orders as the previous problem, but only earliest start time is optimal in this case.
- To implement it efficiently, we will use a heap with increase-key operation.

3.2.1 PROOF OF OPTIMALITY

- We define the **depth** at time t as the number of intervals that contain time t . The **maximum depth** d_{max} is the maximum depth over all times.
- Clearly, the number of partitions needed is at least d_{max} . We will show that this greedy algorithm create only d_{max} partitions.
- Let d be the number of partition the greedy algorithm opened.
- Partition d was created because there was an interval j that overlaps with some previously scheduled interval in each of the $d - 1$ other partitions.
- This means that for $d - 1$ intervals, their start times are all before s_j and their finish times are all after f_j (otherwise j must be compatible). Hence, the depth at s_j is exactly d .
- Thus, $d_{max} \geq d$ so the greedy algorithm is optimal.
- Warning: this proof rely on the fact that the start time

3.3 MINIMUM LATENESS SCHEDULING

- n intervals: $1, 2, \dots, n$, with each interval j requires t_j units of time and has deadline d_j .
- The lateness of an interval is $l_j = \max\{0, f_j - d_j\}$
- Output the schedule that minimizes the maximum lateness $L = \max_j l_j$.
- Fact 1: there is an optimal schedule with no gaps.
- The naive algorithm is to try all possible schedules, which is $O(n!)$.
- For greedy algorithm, we will sort intervals in this order
 1. Shortest processing time, t_j
 2. Smallest slack first $d_j - t_j$
 3. Earliest deadline first d_j
- 1 and 2 are not optimal, and they have simple counterexamples with $n = 2$.
- an **inversion** is two intervals i, j such that $d_i > d_j$ but i is scheduled before j .
- **Note that in general:** Define inversion as a violation of what your order is, then prove that inversion is slightly worse or bad.

3.3.1 PROOF OF OPTIMALITY

- Observation 1: The greedy algorithm has no gaps.
- Observation 2: The greedy algorithm has no inversions.
- We will prove
- **Claim 1:** If a schedule S with no gaps has an inversion, then S has a pair of inverted intervals that are adjacent.
- Suppose for contradiction that there is S with an inversion but does not have adjacent intervals that are inverted. Then, $d_j < d_i \leq d_{i+1} \leq d_{i+2} \leq \dots \leq d_{j-1} \leq d_j$, so $d_j < d_j$ which is a contradiction.
- **Claim 2:** All schedules with no gaps and no inversions have the same lateness
- Let S and S' be two distinct schedules with no gaps and no inversions
- Note S and S' differ only by the schedule of intervals with the same deadline
- Consider the intervals with the same deadline, they must be adjacent in both S and S' . As a group, the maximum lateness of these intervals is the same because the group will finish at the same time.
- **Claim 3:** Swapping adjacent inverted interval does not increase lateness and reduces the total number of inversions by one.
- Let i and j denote two adjacent inverted intervals in the schedule S . By swapping i and j , we get a schedule S' with one fewer inversion.
- Let l and l' be the lateness before/after swap. $\forall k \neq i, j, l_k = l'_k$.
- We know $d_i > d_j$ because it was an inversion. Thus, $l_j \geq l'_j$ and $l_j \geq l'_i$
- $l_j = f_j - d_j \geq f'_i - d_i = l'_i$
- Suppose by contradiction that the greedy schedule S is not optimal.
- Let S^* is the schedule with no gaps and fewest inversion.
- **Case 1:** S^* has no inversion. By claim 2, this is contradiction.
- **Case 2:** S^* has at least one inversion. By claim 1, it must have two inverted intervals that are adjacent. By claim 3, we can swap them to get a schedule with no greater lateness with one fewer inversion so it must be optimal. Then S^* does not have the fewest inversions among all optimal schedules. This is also a contradiction.

3.4 HUFFMAN CODE

- Given an alphabet Γ : a set of n symbols
- You need to encode symbols using binary code.
- Fixed length code requires $\lceil \log n \rceil$ bits per symbol.
- This is easy to decode, but this is not optimal
 - “e”: 12.7% of letters in english
 - “z”: 0.07% of letters in english
 - We should give a shorter code to “e” than “z”.
- The goal is to find a code that minimize the length of text coding
- Variable length code can save space, but is harder to decode. Suppose $\Gamma = \{a, b, c\}$ and $a = 1, b = 01, c = 010$. Then 0101 is ambiguous as bb or ca .
- A **prefix code** is a code where no codeword is a prefix of any other. Scan from left to right until you see a codeword.
- Prefix code can be represented as a binary tree (edges are leaf), with the leaves as the symbols.
- The efficiency of a code is the weighted height of this tree, where the weight of a leaf is the probability of that symbol.

3.5 PREFIX CODE PROBLEM

- Input: A set of Γ with their frequencies $f : \Gamma \rightarrow \mathbb{R}$ where $\sum_{x \in \Gamma} f(x) = 1$
- Output: Binary tree T representing the optimal prefix code for Γ, f
- Optimal Solution is a tree with the weighted average height $AD(T)$ of the tree is minimum.
- **Fact 1:** An optimal tree is a full binary tree, where each internal has two children.
- **Fact 2:** In an optimal tree T , $\forall x, y; f(x) < f(y) \implies \text{depth}_T(x) \geq \text{depth}_T(y)$.
- **Fact 3:** If x, y have minimum frequency, then there is an optimal tree T such that x, y are siblings and are at max depth.
- **Huffman’s Algorithm:** Combine the two smallest frequencies into a new symbol z with frequency $f(z) = f(x) + f(y)$, and repeat recursively until there are only two symbol left, in which case we assign them 0 and 1.

3.5.1 PROOF OF OPTIMALITY

- We will prove by induction. The base case for $n = 2$ is trivial.
- Induction hypothesis: for all Γ, f for $n - 1$ symbols, the algorithm produces an optimal tree.
- Let Γ, f be an alphabet with frequencies with n symbols. Let H be the tree produced by the algorithm.
- The algorithm constructed by H by: (1) replacing two symbols $x, y \in \Gamma$ of minimum frequency with a new symbol z , and

$$\Gamma' = (\Gamma \setminus \{x, y\}) \cup \{z\} \quad (26)$$

$$f'(\alpha) = \begin{cases} f(x) + f(y), & \alpha = z \\ f(\alpha), & \text{otherwise} \end{cases} \quad (27)$$

- We know that x, y are siblings (with parent z) in H
- Since H' has $n - 1$ symbols, H' is optimal for Γ', f'
- We know $AD(H) = AD(H') + f(x) + f(y)$
- Now take an optimal tree T for (Γ, f) where x, y are siblings. We know T exists by fact 3.
- Let T' be the tree constructed from T by removing x, y and replacing their parent with symbol z (and $f(z) = f(x) + f(y)$)
- Clearly, T' is a prefix code for (Γ', f') with $AD(T) = AD(T') + f(x) + f(y)$
- We know $AD(T') \geq AD(H')$ as H' is optimal for (Γ', f') Thus, $AD(T) - f(x) - f(y) = AD(T') \geq AD(H') = AD(H) - f(x) - f(y) \implies AD(T) \geq AD(H)$. but T is optimal so H is optimal.

3.6 DIJKSTRA'S SINGLE-SOURCE SHORTEST PATH ALGORITHM

- Input: A weighted directed graph $G = (V, E)$ with weights $w : E \rightarrow \mathbb{R}_+$ and a source vertex $s \in V$
- Output: Length of the shortest path from s to every other node $t \in V$
- Suppose R is a subset of nodes of G that includes s . A path $s \rightarrow v$ is a **R-path** from $s \rightarrow v$ is restricted to contain only in R before reaching v .
- At a high level, the algorithm maintains a set $R \subseteq V$, (at the beginning s , and at the end V)
- The algorithm also contains $d(v)$ where

1. if $v \in R$, then $d(v)$ is the length of the shortest path from $s \rightarrow v$. This is exactly what we want.
 2. if $v \in V \setminus R$, then $d(v)$ is the length of the shortest $s \rightarrow v$ R -path.
- The goal of the algorithm is to have R contain all the nodes. In each iteration, the algorithm adds one more node from R and maintains its properties.
 - The algorithm will greedily choose the node $u \in V \setminus R$ with the minimum $d(u)$.
 - We know that $d(u)$ is the length of the shortest $s \rightarrow u$ R -path. We will show that this is also the length of the shortest $s \rightarrow u$ path. If the claim is true, we can
 - a) move u to R and still maintain property 1.
 - b) update $d(v)$ for nodes in $V \setminus R$ to satisfy property 2.
 - The proof of the claim is simple, suppose by contradiction that there is a shorter path P with $\text{len}(P) < d(u)$. P must cross from R to $V \setminus R$ at least once. Let v be the first node this P reaches in $V \setminus R$, so first section of P , we denote P_1 from $s \rightarrow v$ is an R -path. As all the weights are non-negative, $\text{len}(P) \geq \text{len}(P_1) \geq d(v) \geq d(u)$, so $\text{len}(P)$ cannot be less than $d(u)$, which is a contradiction.
 - Upon adding u to R , we gain more freedom for R -paths. For points in $V \setminus R$, we can choose to retain the previous R -path or to use u for the new R -path. Thus, for some $v \in V \setminus R$,
 - if the shortest R -path does not contain u , do nothing.
 - if the shortest R -path contains u , then the new $d(v) = d(u) + w(u, v)$.
 - * Note that u must be the last node in the shortest R -path. Suppose there is another node $r \in R$ where $s \rightarrow u \rightarrow r \rightarrow v$ is shorter than $s \rightarrow u \rightarrow v$.
 - * Since $r \in R$ when u was still in $V \setminus R$, $d(r)$ must be the length of the shortest path of $R \setminus \{u\}$. So in this case, the shortest path to $v : s \rightarrow r \rightarrow v$ doesn't require u .
 - So for every $v \in V \setminus R$, we update $d(v) \leftarrow \min\{d(v), d(u) + w(u, v)\}$.
 - **Note** that the algorithm does not work with negative weights.

Algorithm 2 Dijkstra's Algorithm

```

1: procedure DIJKSTRA
2:    $R \leftarrow \{s\}$ 
3:    $d(s) \leftarrow 0$  for each nodes  $v \neq s$  do if  $(s, v)$  is an edge then  $d(v) \leftarrow w(s, v), p(v) \leftarrow s$ 
   else  $d(v) \leftarrow \infty, p(v) \leftarrow Nil$ 
4:   while  $R \neq V$  do
5:      $u \leftarrow$  node not in  $R$  with the minimum  $d(u)$ 
6:      $R \leftarrow R \cup \{u\}$ 
7:     for each  $v$  s.t.  $(u, v)$  is an edge do
8:       if  $d(v) > d(u) + w(u, v)$  then
9:          $d(v) \leftarrow d(u) + w(u, v)$ 
10:         $p(v) \leftarrow u$ 

```
