

Algorithms and Data Structures Compendium

Olav Landmark Pedersen

2018



NTNU – Trondheim
Norwegian University of
Science and Technology

Department of Engineering Cybernetics

Innhold

1	Course Content	1
2	The Role of Algorithms	2
2.1	Learning goals part 1	2
3	Getting Started	3
4	Growth of Functions	4
4.1	4
5	Divide-and-Conquer	5
5.1	What is Divide and Conquer	5
5.2	Reccurence methods	6
5.2.1	Substitution	6
5.2.2	Recursion-tree method	7
5.2.3	Master's Method	8
5.3	Insertion Sort	8
6	Dynamic Programming	9
6.1	0/1 Knapsack Problem	9
6.2	Rod cutting	10
7	Greedy Programming	11
7.1	Activity Selection	11
7.2	Huffman Coding Algorithm	12
8	Graphs	14
8.1	Binary search trees	14
8.2	Heaps	14
I	Graph Traversal	15
9	General Graph Traversal	15
9.1	Breadth First Search (BFS)	15
9.2	Depth First Search (DFS)	15
9.3	Topological sorting	16
10	Minimal Spanning Trees	16
10.1	Kruskal	16
10.2	Prim's Algorithm	17

11 Shortest Path Algorithms	17
11.1 Single Source Shortest Path (SSSP)	17
11.1.1 Bellman Ford	17
11.2 Dijkstra	18
11.3 DAG-Shortest-Path	18

1 Course Content

See the course document to see the learning goals for each group of chapters as these are what can be tested.

- ☐ Kap. 1. The role of algorithms in computing
- ☐ Kap. 2. Getting started
- ☐ Kap. 3. Growth of functions: Innledning og 3.1
- ☐ Kap. 4. Divide-and-conquer: Innledning, 4.1 og 4.3–4.5
- ☐ Kap. 6. Heapsort
- ☐ Kap. 7. Quicksort
- ☐ Kap. 8. Sorting in linear time
- ☐ Kap. 9. Medians and order statistics
- ☐ Kap. 10. Elementary data structures
- ☐ Kap. 11. Hash tables: s. 253–264
- ☐ Kap. 12. Binary search trees: Innledning og 12.1–12.3
- ☐ Kap. 15. Dynamic programming: Innledning og 15.1, 15.3–15.4
- ☐ Kap. 16. Greedy algorithms: Innledning og 16.1–16.3
- ☐ Kap. 17. Amortized analysis: Innledning og s. 463–465 (tom. «at most 3»)
- ☐ Kap. 21. Data structures for disjoint sets: Innledning, 21.1 og 21.3
- ☐ Kap. 22. Elementary graph algorithms: Innledning og 22.1–22.4
- ☐ Kap. 23. Minimum spanning trees
- ☐ Kap. 24. Single-source shortest paths: Innledning og 24.1–24.3
- ☐ Kap. 25. All-pairs shortest paths: Innledning og 25.2
- ☐ Kap. 26. Maximum flow: Innledning og 26.1–26.3
- ☐ Kap. 34. NP-completeness
- ☐ Oppgaver 2.3-5, 4.5-3, 16.2-2 og 34.1-4, med løsning
- ☐ Appendiks A–E i dette heftet

2 The Role of Algorithms

2.1 Learning goals part 1

- Chapters
 - ☐ Chapter 1: The role of algorithms in computing
 - ☐ Chapter 2: Getting Started:
 - ☐ Introduction
 - ☐ 2.1
 - ☐ 2.2
 - ☐ Chapter 3: Growth of Functions:
 - ☐ Introduction
 - ☐ 3.1
- Goals
 - ☐ Forstå bokas pseudokode-konvensjoner
 - ☐ Kjenne egenskapene til random-access machine-modellen (RAM)
 - ☐ Kunne definere problem, instans og problemstørrelse
 - ☐ Kunne definere asymptotisk notasjon
 - ☐ Kunne definere best-case, average-case og worst-case
 - ☐ Forstå løkkeinvarianter og induksjon
 - ☐ Forstå rekursiv dekomponering og induksjon over delproblemer
 - ☐ Forstå Insertion-Sort

3 Getting Started

- Goals
 - ☐ Forstå bokas pseudokode-konvensjoner
 - ☐ Kjenne egenskapene til random-access machine-modellen (RAM)
 - ☐ Kunne definere problem, instans og problemstørrelse
 - ☐ Kunne definere asymptotisk notasjon
 - ☐ Kunne definere best-case, average-case og worst-case
 - ☐ Forstå løkkeinvarianter og induksjon
 - ☐ Forstå rekursiv dekomponering og induksjon over delproblemer
 - ☐ Forstå Insertion-Sort

4 Growth of Functions

$$\begin{aligned} f(n) = O(g(n)) &\text{ is like } a \leq b \\ f(n) = \Omega(g(n)) &\text{ is like } a \geq b \\ f(n) = \Theta(g(n)) &\text{ is like } a = b \\ f(n) = o(g(n)) &\text{ is like } a < b \\ f(n) = \omega(g(n)) &\text{ is like } a > b \end{aligned} \tag{1}$$

4.1

5 Divide-and-Conquer

Pensum:

Læringsmål:

- Chapters

- ☐ Kap. 2. Getting started: 2.3
- ☐ Kap. 4. Divide-and-conquer
 - ☐ Introduction
 - ☐ 4.1
 - ☐ 4.3
 - ☐ 4.4
 - ☐ 4.5
- ☐ Kap. 7. Quicksort
- ☐ Oppgaver 2.3-5 og 4.5-3 med løsning (binærsøk)
 - ☐ 2.3
 - ☐ 2.4
 - ☐ 2.5
 - ☐ 4.5-3 (binary search)
 - ☐ Appendix B and C in the syllabus
- ☐ Appendiks B og C i dette heftet

- Goals

- ☐ Forstå designmetoden divide-and-conquer (splitt og hersk)
- ☐ Forstå maximum-subarray-problemet med løsninger
- ☐ Forstå Bisect og Bisect 0 (se appendiks C i dette heftet)
- ☐ Forstå Merge-Sort
- ☐ Forstå Quicksort og Randomized-Quicksort
- ☐ Kunne løse rekurrenser med substitusjon, rekursjonstrær og masterteoremet
- ☐ Kunne løse rekurrenser med iterasjonsmetoden (se appendiks B i dette heftet)
- ☐ Forstå hvordan variabelskifte fungerer

5.1 What is Divide and Conquer

Just dividing the problem into subproblems. These subproblems should be the same as the main problem. They cannot be different problems as the main problem, but have to be broken down into smaller steps.

The implementation of these algorithms are recursive.


```

DAC(P) {
    if (small (P))
        S(P)
    else
        divide P into P1, P2, P3, ..., Pk
        Apply DAC(P1), DAC(P2), DAC(P2)
        Combine(DAC(P1), DAC(P2), DAC(P2))
    return output
}

```

Functions that use Divide and Conquer

- Binary Search
- Finding Maximum and Minimum
- MergeSort
- QuickSort

5.2 Recurrence methods

5.2.1 Substitution

We guess a bound and then use mathematical induction to prove our guess correct. There are two steps:

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

After making a good guess of the run time $T()$ for the function. Write it out it out mathematically, and set it into the recurrence. Feks:

- With a recurrence such as $T(n) = 2T([n/2]) + n$ and you guess that the function has a run time of $O(n)$ then you would "convert" your run time guess into $cn \cdot \log(n)$ and sub it into in the recurrence and iterate through it.
- For the recurrence below the run time iteration that would be smart to start with would be:

$$T([n/2]) \leq c[n/2] \cdot \log([n/2]) \quad (2)$$

$$T(n) \leq 2(c[n/2] \cdot \log([n/2])) + n \quad (3)$$

$$T(n) \leq cn \cdot \lg(n/2) + n \quad (4)$$

$$\cdot \quad (5)$$

$$\cdot \quad (6)$$

$$\cdot \quad (7)$$

$$(8)$$

Eventually you will get back to the runtime that you proposed and have therefore proven that the recurrence has your runtime.

Changing Variables: Sometimes it might be beneficial to make a substitution of a variable f.e. $m = \lg(n)$ and then for the runtime as well: $S(m) = 2S(m/2) + m$. Making the induction easier.

5.2.2 Recursion-tree method

Converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.

Start by ignoring the recursive call and put what is left over at the sender start of the tree. F.e. if your recurrence is

$$T(n) = 2T([n/2]) + 4n \quad (9)$$

Then the root of the tree would be $4n$. The number multiplied with $T(n)$, 2, indicates how many iterations must be taken in the step below. In the next iteration (step below) at each node, the part of the recurrence that is not a recursive call, $4n$, is then subbed into the node. However instead of n the level or the call/iteration is used instead. In this case it would be $n/2$.

For each row make note of:

- Number of nodes
- Sum of node values
- Call or current iteration.

The total number of iterations of iterations would be the final form of the calls. In the example above it would be: $T(n/2^i)$. To find out the number of iterations i , simply rearrange algebraically.

$$i = \log_2(n)$$

Number of iteration are used to calculate the total sum of all the row sum.

A good example: here

5.2.3 Master's Method

Provides bounds for currences of the form

$$T(n) = aT(n/b) + f(n) \quad \text{where } a \geq 1, b > 1 \quad (10)$$

Provides bounds for currences of the form

$$T(n) = aT(n/b) + f(n) \quad (11)$$

where $a \geq 1$, and $b > 1$. Then $T(n)$ has the asymptotic bounds:

1. If $f(n) = O(n^{\log_b(a-\epsilon)})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b(a)})$
2. If $f(n) = \Theta(n^{\log_b(a)})$, then $T(n) = \Theta(n^{\log_b(a)} \cdot \log(n))$
3. 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 $c < 1$ then $T(n) = \Theta(f(n))$

5.3 Insertion Sort

This algorithm is one of the most straight forward and slowest sorting algorithms. It starts on the left of the algorithm

6 Dynamic Programming

Two things that are needed for Dynamic Programming:

Optimal Substructure: If an optimal solution to the problem can be constructed from optimal solutions to subproblems.

Overlapping Subproblems: When a recursive algorithm revisits the same problem repeatedly we say it has overlapping subproblems.

Divide and Conquer is not suited for problems that occur several times as this means it solves those *overlapping subproblems* each time and dynamic programming is a strategy to not have to recompute these problems.

There are two main strategies of Dynamic programming.

Memoisation: This strategy is a top-down idea that saves the solution to a subproblem in a data structure, and then checks that value when you come across the same subproblem again.

Iterative: This is a bottom-up strategy

6.1 0/1 Knapsack Problem

The 0/1 Knapsack problem is if a single object of weight and price should be included in the knapsack that can only carry a certain weight. The goal is to maximize the profit.

Given a price vector $p = 1, 2, 5, 6$ and a weight vector $w = 2, 3, 4, 5$. The solution to the problem is a vector that is binary indicating whether the object is included or not, $x = 0, 1, 0, 1$. Since every item is either included or not, and all solutions have to be checked the problem is $\mathcal{O}(2^n)$. This can be shortened by dynamic programming.

You set up a table where the columns are every weight that can be had in the bag. For example, 1 - 8 for a bag that can take 8 kg. The row values are the objects in addition to having nothing in the bag. Since we are solving the problem for each step, there is a smell of optimal substructure and overlapping subproblems.

The algorithm for this might be slightly confusing, but just think logically about the table. Fill the weight for each of these and put in the next object for the next weight.

0/1 Knapsack Problem

$M=8$
 $n=4$

$P=\{1, 2, 5, 6\}$
 $W=\{2, 3, 4, 5\}$

W

		V	0	1	2	3	4	5	6	7	8
P_i	W_i	0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5				

$V[i, W] = \max \{V[i-1, W], V[i-1, W-W[i]] + P[i]\}$
 $V[4, 1] = \max \{V[3, 1], V[3, 1-5] + 6\}$
 $V[3, -4]$

6.2 Rod cutting

Rod cutting problem is determining the optimal number of cuts to be done a rod. There are different prices for different lengths of the rod. You would have a length of the rod and a corresponding price $p = 1, 5, 8, 9, 10, 17, 17, 20$. The idea is to solve the problem for each lengths i of the rod and a corresponding *optimal price* vector. The algorithm uses the following to solve the problem.

$$C(i) = \max_{1 \leq k \leq i} \{V_k + C(i - k)\} \quad (12)$$

Where $C(i)$ is the optimal cost at index i , and k is the number of cuts. This essentially means taking the maximum previous ($C(i - k)$) and adding on the price, based on the length left over based on the price (5). This uses the table below and previously calculated problems to find the optimal solution.

Without dynamic programming this problem has a solution of $\mathcal{O}(2^n)$, but with dynamic programming it reduces to $\mathcal{O}(n^2)$,



Rod Cutting DP


$$C(i) = \max_{1 \leq k \leq i} \{V_k + C(i-k)\}$$

Length	1	2	3	4	5	6	7	8
Price \$	1	5	8	9	10	17	17	20

Len (i)	1	2	3	4	5	6	7	8
Opt	1	5	8	10	13	17	18	

$C(8) = \max$

- $V_1 + C(7) = 1 + 18 = 19$
- $V_2 + C(6) = 5 + 17 = 22$ 
- $V_3 + C(5) = 8 + 13 = 21$
- $V_4 + C(4) = 9 + 10 = 19$
- $V_5 + C(3) = 10 + 8 = 18$
- $V_6 + C(2) = 17 + 5 = 22$ 
- $V_7 + C(1) = 17 + 1 = 18$
- $V_8 = 20$



7 Greedy Programming

There are two requirements for greedy programming:

The Greedy Choice Property is that a global optimal solution can be arrived at by selecting a local optimum.

Optimal Substructure: If an optimal solution to the problem can be constructed from optimal solutions to subproblems.

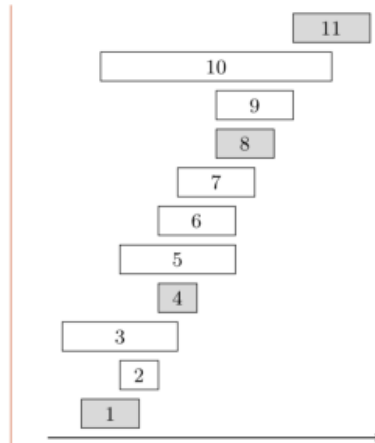
7.1 Activity Selection

This is the problem is to choose as many activity as possible that do not overlap. In this type of problem the optimal solution is the always take the activity that finishes first relative to the ending time of the last activity. Therefore the problem is an example of a use of a greedy algorithm.

```

REC-ACT-SEL( $s, f, k, n$ )
1   $m = k + 1$ 
2  while  $m \leq n$  and  $s[m] < f[k]$ 
3       $m = m + 1$ 
4  if  $m \leq n$ 
5       $S = \text{REC-ACT-SEL}(s, f, m, n)$ 
6      return  $\{a_m\} \cup S$ 
7  else return  $\emptyset$ 

```



```

GREEDY-ACTIVITY-SELECTOR( $s, f$ )
1   $n = s.length$ 
2   $A = \{a_1\}$ 
3   $k = 1$ 
4  for  $m = 2$  to  $n$ 
5      if  $s[m] \geq f[k]$ 
6           $A = A \cup \{a_m\}$ 
7           $k = m$ 

```

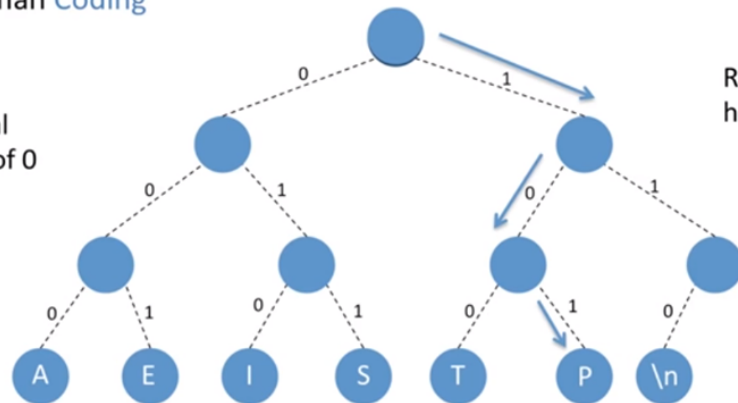
7.2 Huffman Coding Algorithm

This is a compression algorithm that uses greedy programming. Remember information flow from TilpDat. The idea is that you try to minimize the code, by assigning less bits to the symbol that occurs the most and more bits for those that occur less often.

1. The first step of the algorithm is to take the two characters with the lowest frequency.
2. Then make a subtree from them and write the sum of the frequencies of both letters in the node above them.
3. Now take the next lowest occurring symbol and set it beside the previous parent node and create a new parent node from the two.
4. Label the parent node with the total number of occurrences in the new tree.
5. Continue iterating through this method until the current parent node is greater than the frequency of all other nodes.
6. When this occurs, you need to start a new subtree with the remaining lowest frequency letters and create a parent node like we initially did.
7. If there are remaining letters you want to take the lowest value subtree and lowest value letter and attach them together.
8. Finally, attach the subtrees together.

Huffman Coding

Code Tree
Left Traversal
has a Value of 0



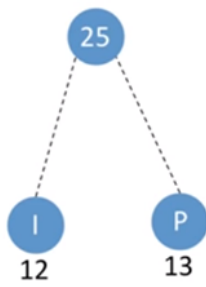
Right Traversal
has a Value of 1

computer science

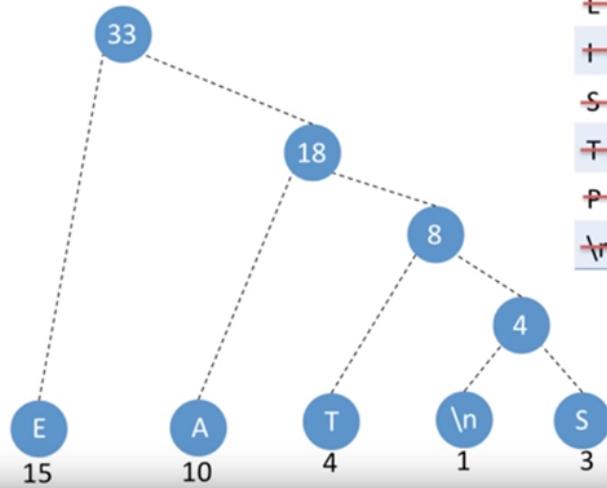
A 000

P 101

Huffman Coding



computer science



Char	Freq
A	10
E	15
I	12
S	3
T	4
P	13
\n	1

8 Graphs

Graphs are made up of

- **Vertices:** Nodes or the round dots
- **Edges:** the lines between vertices

A graph can either be **directed** or **undirected**. When making neighboring matrices or hashtable, treat the row index as the source that the vertex point to. In undirected graphs the different just treat them like bidirectional graphs. This means that the neighbor matrices become symmetrical graphs.

8.1 Binary search trees

a *Successor* is the node with the smallest key, greater than the current node. In a graph this will be the neighboring smaller value. In a normal setting this will be the right most leaf node of the LEFT branch. a *Predecessor* is the node with the greatest key, smaller than the current node. In a graph this will be the neighboring larger value. In a normal setting this will be the left most leaf node of the node on the RIGHT branch.

8.2 Heaps

The way a heap is indexed in an array is that for a node i

$$\text{Left child} = 2i \tag{13}$$

$$\text{Right child} = 2i + 1 \tag{14}$$

The Heap Property: The heap must have all children nodes filled, except the highest vertices.

Del I

Graph Traversal

Notation!

Term	Definition
G	Graph
V	Vertex
E	Edge
s	source (The node we are starting with)
u	parent or predecessor vertex
v	child or successor vertex
$u.\text{color}$	This is the color attribute of the node
$u.d$	This is the distance of the current node from the source
$u.\pi$	This is predecessor node of the node u
$u.f$	This records the <i>time</i> in a DFS when the node has been blackend.
Q	The queue that nodes are loaded onto.

Color scheme

- All **black** and **grey** nodes have been discovered.
- All **black** nodes have had all of their children nodes discovered. Note that all parent nodes, u , are always black.
- All **grey** nodes might potentially have undiscovered adjacent children nodes.

9 General Graph Traversal

These graphs and functions are to be used on directed or undirected (or bidirectional) and non-weighted edges and are general principles for searching through these structures.

9.1 Breadth First Search (BFS)

Shortly put, it discovers all vertices at distance k from s before discovering vertices at distance $k + 1$. This algorithm uses a queue to discover new nodes and unloads them as it goes.

9.2 Depth First Search (DFS)

This is complementary to the BFS as this algorithm adds children vertices to the stack and continues up the tree until it reaches a leaf vertex. When it cannot go further it backtracks and tries again, until it ultimately has searched the entire tree.

It keeps iterating down and turning vertices GRAY as it does. Each time the node is tuned gray the 'time' of discovery is set by

$$u.d = time \tag{15}$$

When it it reaches the point of no return it turns the node BLACK and then records the time it was blackend by setting:

$$u.f = time \tag{16}$$

As this was called recursively each node that was visited will be blackend and $u.f$ recorded recursively.

9.3 Topological sorting

Sorting a graph simply means that you want to ensure that a parent, u , comes before its child, v . There are two requirments for this is a DAG (Directed Acyclic Graph):

- The vertices are directed
- The graph is *acyclic*. A graph is acyclic if it cannot get stuck in a cycle between parents and children.

DAG-shortest-path() Uses DFS to traverse through the nodes and then uses the $u.f$ blackening times of all the nodes to sort the graph into a list.

10 Minimal Spanning Trees

These algorithms are performed on undirected (or bidirectional) graphs that can be cyclic, and that are weighted. They need to be weighted for the algorithms to work.

10.1 Kruskal

Kruskal creates a minimum spanning tree of undirected graphs with weighted edges. The algorithm uses the edges to find a minimum spanning tree. It will go through all of the edges of nodes that are not connected. The run time for this algorithm is that of sorting the edges. It typically uses **merge sort** and therefore the compilation time will be the the time of merge sort on the edges $\mathcal{O}(E \log E)$.

The algorithm sorts the edges E by using Merge-sort. It will add the smallest weighted vertex to the minimum spanning tree. It will keep adding the smallest edge in the and add it to the minimum spanning tree, unless the two nodes the edges connect are already in the set of connected vertices. It continues this loop until all the nodes are in the set of connected nodes.

Kruskal is a greedy algorithm.

10.2 Prim's Algorithm

This algorithm basically uses BFS from a random node. It finds the smallest weighted edge that is connected to the set of connected nodes. It finds the smallest one that does not add a node that is already in the set, and adds it to the set.

11 Shortest Path Algorithms

The name is self explanatory, but these algorithms find the shortest path between nodes. These functions all use the *RELAX()* function. This function will examine a parental node, child node, and a weight on the edge inbetween them. The single purpose the function is to update the $v.d$ if the current $v.d$ is higher than another parent $u.d$ and weight $w(u, v)$.

```
Relax(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

11.1 Single Source Shortest Path (SSSP)

These types of problems exhibit **optimal substructure**.

11.1.1 Bellman Ford

This function updates an array of node and the total weight needed to travel to the vertex.

1. The array is initialized by setting all the weights to ∞ .
2. It starts by iterating through this array. For each node it checks if the vertex has nodes that it can travel to. For each of those neighboring vertices it updates the weight using *RELAX()*.
3. It iterates through the index and updates/ *RELAX()* at each step.
4. It continues looping through this array until it cannot does not change any values through the array.

It takes at most $V - 1$ iterations. The compilation time takes $\mathcal{O}(|V| * |E|)$. Note that this algorithm can deal with negative weights as the final array can be offset by the lowest weight in the array and then sorted.

11.2 Dijkstra

This function is based on a BFS approach. It is important to note that for this algorithm there cannot be negative weights on the edges.

1. It starts at the source node and relaxes all the adjacent vertices.
2. After they have all been relaxed it goes to its set of visited nodes and finds adjacent nodes with the smallest weight and visits that node.
3. It relaxes all of that nodes neighboring weights and will then look at the node in the structure that is not in the set and will visit that node. (remember all unvisited vertices are ∞).
4. This continues until the nodes have ultimately finds all the vertices.

Dijkstra's shortest path can be $\mathcal{O}(V^2)$. However, if a min-priority queue with a Fibonacci heap is used the algorithm can take: $\mathcal{O}(V \log V + E)$

11.3 DAG-Shortest-Path

If it can be used it is the most efficient algorithm. The name indicates the requirements DAG = Directed Acyclic Graph. The algorithm topologically sorts the graph and then for each node in topological order, each of its children are relaxed.

This algorithm has a shortest run time of $\mathcal{O}(|V| + |E|)$