UiO : Department of Informatics
University of Oslo

INF4130

Algoritmer: Design og effektivitet

Joakim Myrvoll

2014

Contents

Part I
Theory

1 Turing machine

- Problems $\rightarrow$ Natural problems $\rightarrow$ Formal languages
- Solutions $\rightarrow$ Algorithms $\rightarrow$ Turing machines
- Efficiency $\rightarrow$ Complexity $\rightarrow$ Complexity classes
- Alphabet $\rightarrow$ Finite set of symbols. E.g. $\Sigma = \{0, 1\}$, $\Sigma = \{A, \ldots, Z\}$
- Σ^*: All finite strings over Σ in lexicographic order;$\Sigma^* = \{\in, 0, 1, 00, 01, \ldots\}$
- A formal language L over Σ is the subset of all YES-instances in Σ^*.
- Turing machine:

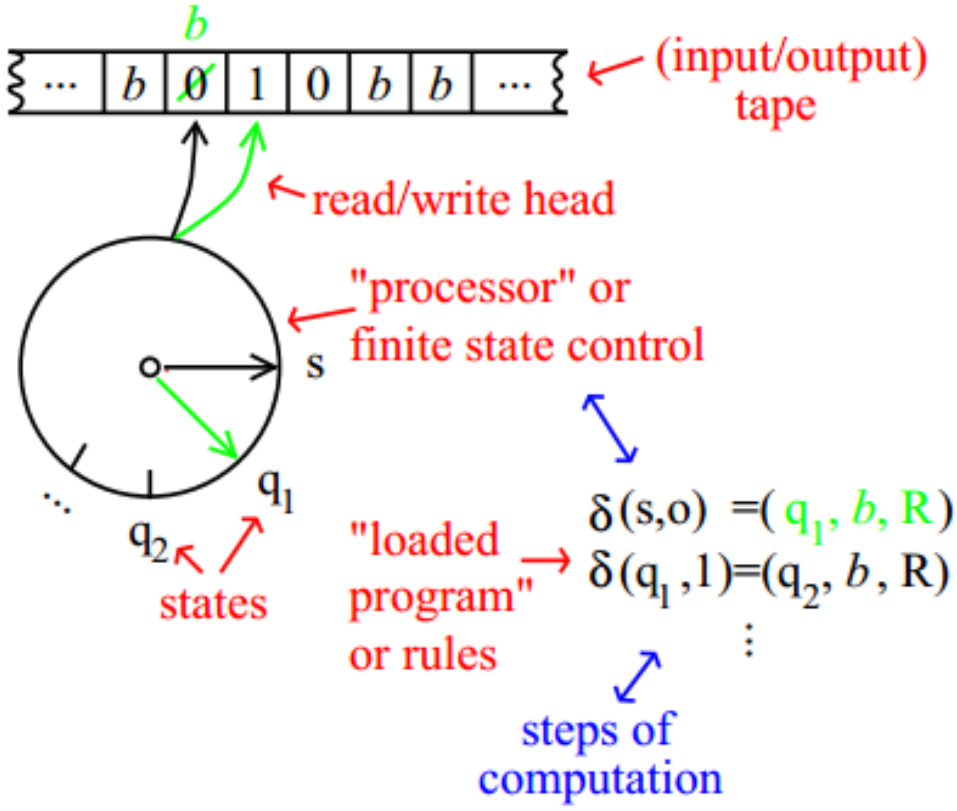


Figure 1: Turing machine - intuitive description

- Turing machine M decides language L iff M computes the function

$$f(x) : \Sigma^* \rightarrow \begin{cases} Y & \text{for } x \in L \\ N & \text{otherwise} \end{cases}$$

- Language L is Turing decidable iff there is a Turing machine which can decide it.
- Turing machine M accepts language L if M halts iff its input is a string in L.
- Language L is Turing acceptable iff there is a Turing machine which accepts it.
- A Turing machine is "a mathematician executing some algorithm".
- There are two special states;
  - s : start state
  - h : halt state
- Formally a Turing machine is a quadruple;
  - sigma (Σ): input alphabet.
  - gamma (Γ): I/O tape - input alphabet, blank symbol and possible others.
  - Q: finite set of states (including start and halt).
  - delta (δ): transition function. L/R representing left/right for the read/write head.
- Configuration of a Turing machine: (q,w_l,w_r):
  q : current state
  w_l : portion of tape to the left of read/write head
  w_r : portion of tape to the right of read/write head

1.1 Church's thesis (Church–Turing thesis)

The Church-Turing thesis states that a function is computable (a problem is solvable) in an informal sense (i.e., computable by a human being using a pencil-and-paper method, ignoring resource limitations) if and only if it is computable (solvable) by a Turing machine. This is a 'thesis' because it combines real-world concepts ('computable'-by any algorithm or machine) with theoretical/abstract/mathematical ones (Turing machine). Only statements about mathematical concepts are provable.

- 'Turing machine' $\cong$ 'algorithm'
  Turing machines can compute every function that can be computed by some algorithm, program or computer.
- 'Expressive power' of programming languages
  Turing complete programming languages. Prove that a Turing machine can be written in a programming language.
- 'Universiality' of computer models
  Neural networks are Turing complete.
- Uncomputability
  If a Turing machine can't compute f, no computer can.

1.2 Exercises

- Construct a Turing machine which accepts the alphabet {0,1} and checks wheter a string consists of 0's only.
  - The complexity is O(n) where n is the length of the input

State	0	1	b
s	(s_1," ",R)	(h,"N",-)	(h,"Y",-)
s_1	(s_1," ",R)	(h,"N",-)	(h,"Y",-)

- Contruct a Turing machine which accepts the alphabet {0,1} and recognizez the language L={1^k0^k | k=0,1,2,...}
  - The machine will traverse half the tape for each character on the tape. Resulting in O(n $\cdot \frac{1}{2}$n) = O($\frac{1}{2}n^2$) = O(n^2)

State	0	1	x	b
s	(s_1,"x",R)	(h,"N",-)	-	(h,"Y",-)
s_1	(s_1,"0",R)	(s_2,"x",L)	(s_2,"x",R)	(h,"N",-)
s_2	(s_1,"x",R)	(s_2,"1",L)	(s_2,"x",L)	(h,"Y",-)

2 String search

2.1 The naive algorithm

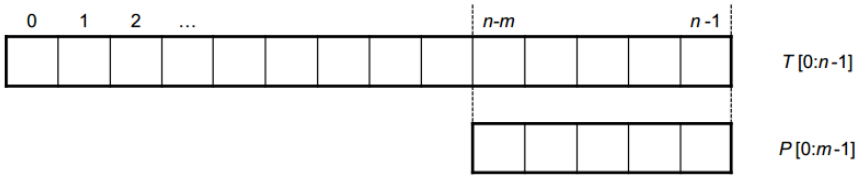


Figure 2: Naive search

```
function NaiveStringMatcher (P [0:m−1], T [0:n−1])
  for s = 0 to n−m do
    if T [s:s m−1] = P then
      return(s)
    endif
  endfor
  return(−1)
end
```

The for-loop is executed n-m+1 times. Each string test has up to m symbol comparisons. O(nm) execution time (worst case).

2.2 Knuth-Morris-Pratt (prefix-based search)

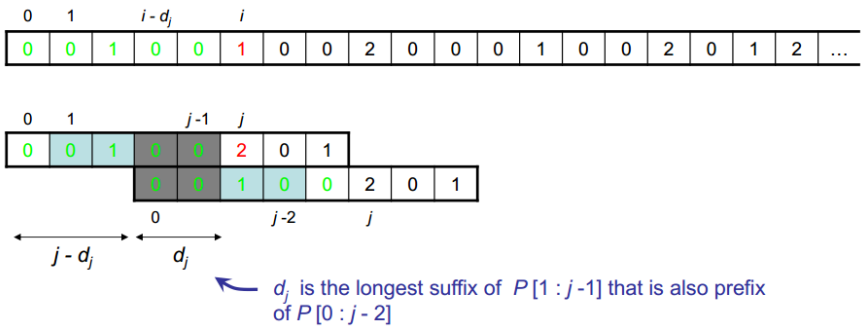


Figure 3: Prefix search

We know that if we move P less than j-d_j steps, there can be no (full) match. And we know that, after this move, P[0:d_j-1] will match the corresponding part of T. Thus we can start the comparison at d_j in P and compare P[d_j:m-1] with the symbols from index i in T.

2.2.1 Next[ ]

We will produce a table Next[0:m-1] that shows how far we can move P when we get a (first) mismatch at index j in P, j = 0,1,2, . . . , m-1. But the array Next will not give this number directly. Instead, Next[j] will contain the new (and smaller value) that j should have when we resume the search after a mismatch at j in P. That is: Next [j] = j–<number of steps that P should be moved>, or: Next[j] is the value that is named d_j. After P is moved, we know that the first d_j symbols of P are equal to the corresponding symbols in T (that's how we chose d_j). So, the search can continue from index i in T and Next[j] in P.

2.2.2 Pseudocode

```
1  function KMPStringMatcher (P[0:m-1], T[0:n-1])
2    i = 0 // index in T
3    j = 0 // index in P
4    CreateNext(P[0:m-1], Next[n-1])
5    while i < n do
6      if P[j] = T[i] then
7        if j=m-1 then // check full match
8          return(i-m+1)
9        endif
10       i++
11       j++
12     else
13       j = Next[j]
14       if j = 0 then
15         if T[i] != P[0] then
16           i++
17         endif
18       endif
19     endif
20   endwhile
21   return(-1)
22 end
```

O(n)

2.2.3 Example

The array Next for the string P={0 0 1 0 0 2 0 1}:

j	P[0:j-1]	prefix	suffix	Next[j] (longest match)
0	-	-	-	0
1	0	{}	{}	0
2	00	{0}	{0}	1
3	001	{0,00}	{1,01}	0
4	0010	{0,00,001}	{0,10,010}	1
5	00100	{0,00,001,0010}	{0,00,100,0100}	2
6	001002	{0,00,001,0010,00100}	{2,02,002,1002,01002}	0
7	0010020	{0,00,001,0010,00100,001002}	{0,20,020,0020,10020,010020}	1

j	= 0 1 2 3 4 5 6 7
Next[j]	= 0 0 1 0 1 2 0 1

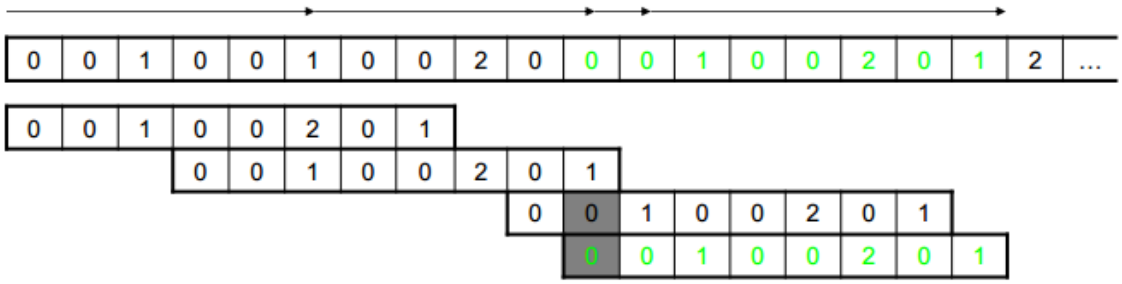


Figure 4: Example

2.3 Boyer-Moore-Horspool (suffix-based search)

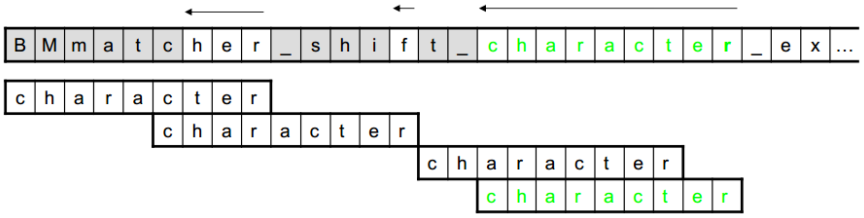


Figure 5: *Comparing from the end*

Worst case execution time O(mn), same as for the naive algorithm. However: Sub-linear (<= n), as the average execution time is O(n ($\log_{|A|}$ m) / m)

2.3.1 Shift[ ]

We must preprocess P to find the array Shift. The length of Shift[ ] is the number of symbols in the alphabet. We search from the end of P (minus the last symbol), and calculate the distance from the end for every first occurence of a symbol. For the symbols not occuring in P, we know: Shift [t] = <the length of P> (m) This will give a "full shift".

2.3.2 Pseudocode

```
function HorspoolStringMatcher (P [0:m-1], T [0:n-1])
  i = 0
  CreateShift(P [0:m-1], Shift [0:|A|-1])
  while i < n-m do
    j = m-1
    while j >= 0 and T[i+j] = P[j] do
      j--
    endwhile
    if j = 0 then
      return(i)
    endif
    i = i + Shift[T[i+m-1]]
  endwhile
  return(-1)
end
```

2.4 Karp-Rabin (hash-based search)

2.4.1 Wikipedia

For text of length n and p patterns of combined length m, its average and best case running time is O(n+m) in space O(p), but its worst-case time is O(nm). In contrast.

Rather than pursuing more sophisticated skipping, the Rabin–Karp algorithm seeks to speed up the testing of equality of the pattern to the substrings in the text by using a hash function. A hash function is a function which converts every string into a numeric value, called its hash value; for example, we might have hash("hello")=5. The algorithm exploits the fact that if two strings are equal, their hash values are also equal. Thus, it would seem all we have to do is compute the hash value of the substring we're searching for, and then look for a substring with the same hash value.

However, there are two problems with this. First, because there are so many different strings, to keep the hash values small we have to assign some strings the same number. This means that if the hash values match, the strings might not match; we have to verify that they do, which can take a long time for long substrings. Luckily, a good hash function promises us that on most reasonable inputs, this won't happen too often, which keeps the average search time within an acceptable range.

The algorithm is as shown:

```
function RabinKarp(string s[1..n], string pattern[1..m])
  hpattern := hash(pattern[1..m]);  hs := hash(s[1..m])
  for i from 1 to n−m+1
    if hs = hpattern
      if s[i..i+m−1] = pattern[1..m]
        return i
    hs := hash(s[i+1..i+m])
  return not found
```

2.4.1.1 Hash function

T = "abracadabra"
|P| = 3

First substring is "abr" (ASCII a = 97, b = 98, r = 114)
hash("abr") = $(97 \times 101^2) + (98 \times 101^1) + (114 \times 101^0) = 999{,}509$

We can then compute the hash of the next substring, "bra", from the hash of "abr";

	base		old hash		old 'a'		new 'a'	new hash
hash("bra") = [	101	$\times$ (	999,509	-	(97×101^2)	)] +	(97×101^0)	= 1,011,309

If the substrings in question are long, this algorithm achieves great savings compared with many other hashing schemes.

Theoretically, there exist other algorithms that could provide convenient recomputation, e.g. multiplying together ASCII values of all characters so that shifting substring would only entail dividing by the first character and multiplying by the last. The limitation, however, is the limited size of the integer data type and the necessity of using modular arithmetic to scale down the hash results. Meanwhile, naive hash functions do not produce large numbers quickly, but, just like adding ASCII values, are likely to cause many hash collisions and hence slow down the algorithm. Hence the described hash function is typically the preferred one in the Rabin–Karp algorithm.

2.4.1.2 Multiple pattern search

The Rabin–Karp algorithm is inferior for single pattern searching to Knuth–Morris–Pratt algorithm, Boyer–Moore string search algorithm and other faster single pattern string searching algorithms because of its slow worst case behavior. However, it is an algorithm of choice for multiple pattern search.

That is, if we want to find any of a large number, say k, fixed length patterns in a text, we can create a simple variant of the Rabin–Karp algorithm that uses a Bloom filter or a set data structure to check whether the hash of a given string belongs to a set of hash values of patterns we are looking for:

```
function RabinKarpSet(string s[1..n], set of string subs, m):
  set hsubs := emptySet
  foreach sub in subs
      insert hash(sub[1..m]) into hsubs
  hs := hash(s[1..m])
  for i from 1 to n-m+1
      if hs in hsubs and s[i..i+m-1] in subs
          return i
      hs := hash(s[i+1..i+m])
  return not found
```

We assume all the substrings have a fixed length m.

A naive way to search for k patterns is to repeat a single-pattern search taking O(n) time, totalling in O(nk) time. In contrast, the variant algorithm above can find all k patterns in O(n+k) time in expectation, because a hash table checks whether a substring hash equals any of the pattern hashes in O(1) time.

2.4.2 Lecture

We assume that the alphabet for our strings is A = {0, 1, 2, . . . , k-1}. Each symbol in A can be seen as a digit in a number system with base k. Thus each string in A^* can be seen as number in this system (and we assume that the most significant digit comes first)

Example: k = 10, and A = {0,1, 2, . . . , 9} we get the traditional decimal number system. The string "6832355" can then be seen as the number 6 832 355. Given a string P[0:m-1].

We can then get the corresponding number P' using m multiplications and m additions (Horners rule, computed from the innermost right expression and outwards):

$$P' = P[m-1] + k(P[m-2] + \ldots + k(P[1] + k(P[0]) \ldots))$$

Example:

$$1234 = 4 + 10 * (3 + 10 * (2 + 10 * 1))$$

Given a string T[0:n-1], and an integer s (start-index), and a pattern of length m. We then refer to the substring T[s:s+m-1] as T_s, and its value is referred to as T'_s.

The algorithm:

- We first compute the value P' for the pattern P
- Based on Horners rule we compute T'_0, T'_1, T'_2, . . . and successively compares these numbers to P'

This is very much like the naive algorithm. However given T'_{s-1} and k_{m-1}, we can compute T'_s in constant time.

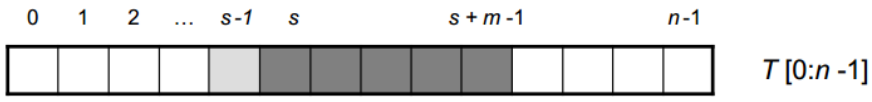


Figure 6: Karp-Rabin

This constant time computation can be done as follows (where T'$_{s-1}$ is defined as shown earlier, and k$_{m-1}$ is pre-computed):

$$T'_s = k * (T'_{s-1} - k_{m-1} * T[s]) + T[s+m] \text{ where } s = 1, \ldots, n-m$$

Example:
k = 10, A = {0,1, 2, . . . , 9} (the usual decimal number system) and m = 7.
T'$_{s-1}$ = 7937245
T'$_s$ = 9372458
T'$_s$ = 10 * (7937245 - (1000000 * 7)) + 8 = 9372458

We can compute T'$_s$ in constant time when we know T'$_{s-1}$ and k$_{m-1}$. We can therefore compute P' and the n-m+1 numbers: T'$_s$, s = 0, 1, . . . , n-m in time O(n). Thus, we can "theoretically" implement the search algorithm in time O(n). However the numbers T'$_s$ and P' will be so large that storing and comparing them will take too long time (in fact O(m) time). The Karp-Rabin trick is to instead use modular arithmetic: We do all computations modulo a value q. The value q should be chosen as a prime, so that kq just fits in a register (of e.g. 32/64 bits). A prime number is chosen as this will distribute the values well.

2.4.2.1 Spurious match

We compute T'$_s^{(q)}$ and P'$^{(q)}$, where T'$_s^{(q)}$ = T'$_s$ mod q, P'$^{(q)}$ = P' mod q, (only once) and compare. We can get T'$^{(q)}{}_s$ = P'$^{(q)}$ even if T'$_s \neq$ P'. This is called a spurious match. So, if we have T'$^{(q)}{}_s$ = P'$^{(q)}$, we have to fully check whether T$_s$ = P. With large enough q, the probability for getting spurious matches is low.

2.4.2.2 Pseudocode

```
function KarpRabinStringMatcher (P[0:m-1], T[0:n-1], k, q)
        c = k^(m-1) mod q
        P'^(q) = 0
        T'_s^(q) = 0
        for i = 1 to m do
                P'^(q) = (k*P'^(q) + P[i ]) mod q
                T'_0^(q) = (k*T'_0^(q) + T[i ]) mod q
        endfor
        for s = 0 to n-m do
                if s > 0 then
                        T'_s^(q) = (k*(T'_(s-1)^(q) - T[s] * c) + T[s+m]) mod q
                endif
                if T'_s^(q) = P'^(q) then
                        if T_s = P then
                                return(s)
                        endif
                endif
        endfor
        return(-1)
end
```

The worst case running time occurs when the pattern P is found at the end of the string T. If we assume that the strings are distributed uniformally, the probability that T'$_s^{(q)}$ is equal to P' (which is in the interval {0, 1, . . . , q-1}) is 1/q. Thus T'$_s^{(q)}$, for s = 0, 1, . . . , n-m-1 will for each s lead to a spurious match with probability 1/q. With the real match at the end of T, we will on average get (n-m)/q spurious matches during the search. Each of these will lead to m symbol comparisons. In addition, we have to check whether T'$_{n-m}^{(q)}$ equals P when we finally find the correct match at the end. Thus the number of comparisons of single symbols and computations of new values T'$_s^{(q)}$ will be:

$$\left(\frac{n-m}{q} - 1\right) m + (n - m + 1)$$

We can choose values so that q » m. Thus the runing time will be O(n).

2.5 Multiple searches in a fixed string T (structure)

It is then usually smart to preprocess T, so that later searches in T for different patterns P will be fast. We often refer to this as indexing the text (or data set), and this can be done in a number of ways. E.g. suffix trees, which relies on "Tries" trees. T may also gradually change over time. We then have to update the index for each such change.

2.5.1 Trie tree

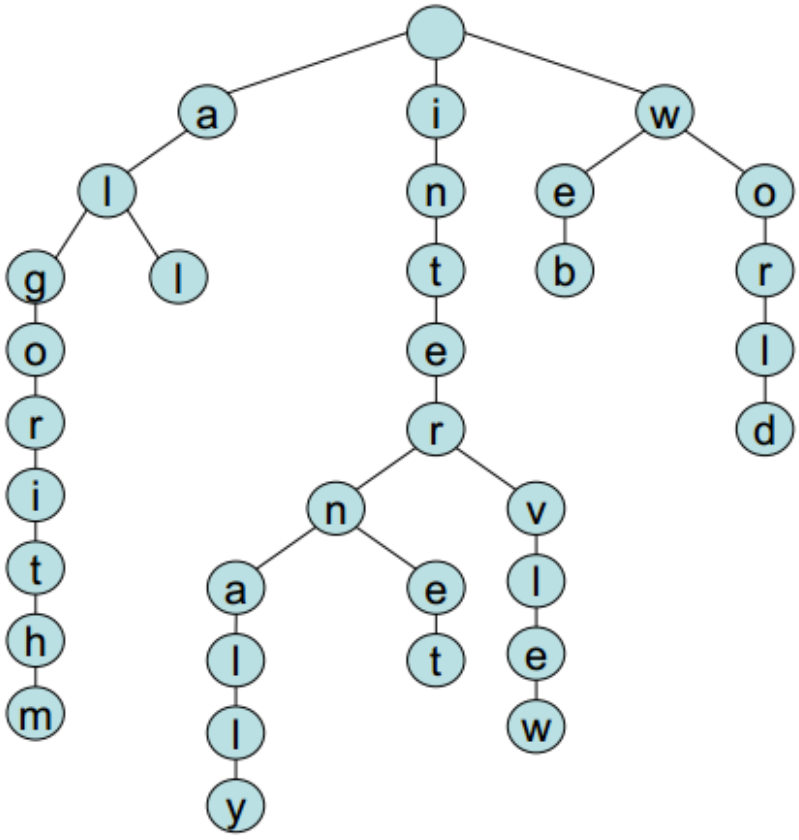


Figure 7: Trie

2.5.2 Compressed trie tree

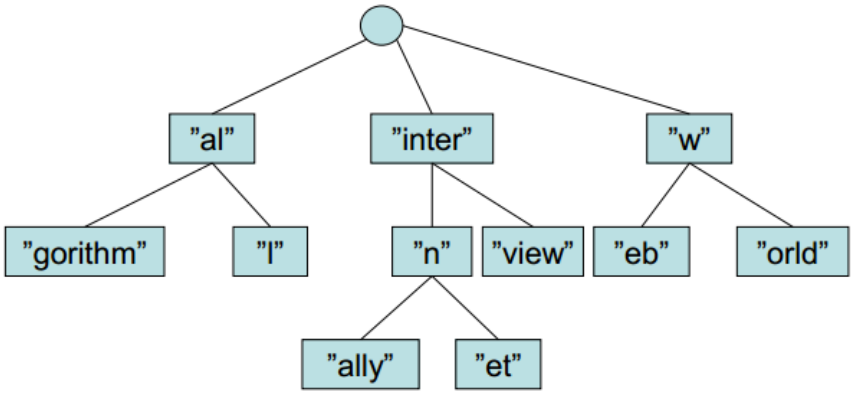


Figure 8: Compressed trie

2.5.3 Suffix tree (compressed)

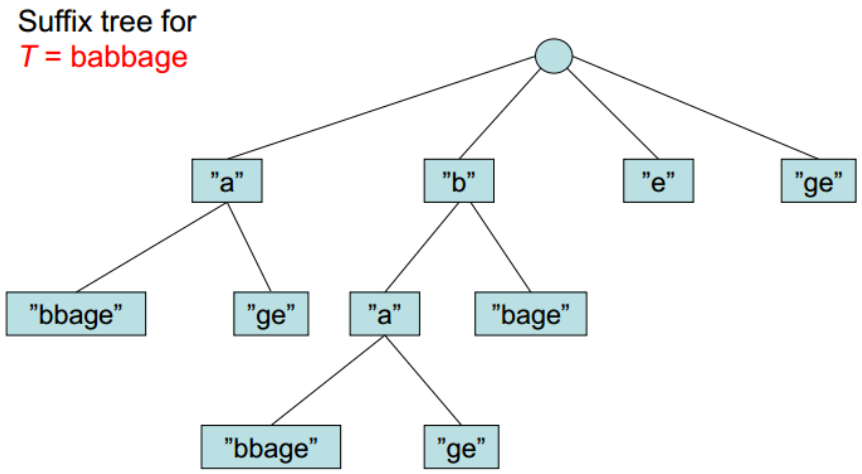


Figure 9: Suffix tree

Looking for P in this Trie will decide whether P occurs in T. There are a lot of optimizations that can be done for such trees.

3 Dynamic programming

Dynamic Programming is useful if the solution to a certain instance is used in the solution of many instances of larger size. In the problem C below, an instance is given by some data (e.g two strings) and by two intergers i and j. The corresponding instance is written C(i, j). Thus the solutions to the instances can be stored in a two-dimentional table with dimensions i and j. The size of an instanstance C(i, j) is j-i. Below, the children of a node N indicate the instances of which we need the solution for computing the solution to N. Note that the solution to many instances, e.g. C(3,4), is used multiple times. This is required to make DP a preferable alternative! Also note that all terminal nodes have size 0.

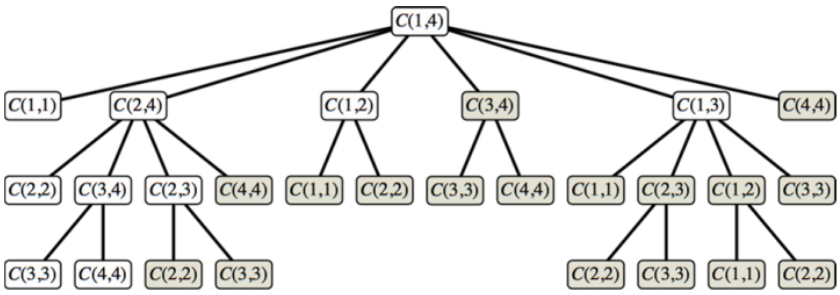


Figure 10: Problem C

3.1 Edit distance

The Edit Distance, ED(P,T), between two strings T and P is the smallest number of such operations needed to convert T to P (or P to T!).

Example.

T		P
logarithm	→ alogarithm → algarithm →	algorithm
	(Steps: +a, -o, a→o)	

Thus ED("logarithm", "algorithm") = 3 (as there are no shorter way)

3.1.1 Finding the edit distance

The problem is for two given strings P[1:m] and T[1:n] to find ED(P,T). We shall refer to this instance as I[P,m,T,n]. The size of an instance is defined as m+n. It turns out that the Combine function that solves this problem is defined by the following recurrence relation:

$$D[i,j] = \begin{cases} D[i-1,j-1] & \text{if } P[i] = T[j] \\ min\{\underbrace{D[i-1,j-1]+1}_{\text{substitution}}, \underbrace{D[i-1,j]+1}_{\text{extra in T, deletion in P}}, \underbrace{D[i,j-1]+1}_{\text{deletion in T}}\} & \text{else} \end{cases}$$

$$D[0,0] = 0, D[i,0] = D[0,i] = i \qquad \text{initialization}$$

This Combine function satesfies the DP-requirement for our problem. We here observe that k will be 3 (except for the initialization in the last line), and that B(I[P,n,T,m]) will be the set:

$$\{I[P,m-1,T,n-1], I[P,m-1,T,n], I[P,m,T,n-1]\}$$

Our instance is I[P,m,T,n], and it has size = m+n. Also, B(I[P,n,T,m]) = {I[P,m-1,T,n-1], I[P,m-1,T,n], I[P,m,T,n-1]}. We observe (by using the above B-function backwards, repeatedly) that solving all instances I[P,j,T,i] with i = 0, . . . , m and j = 0, . . . , n are at least enough to solve instance I[P,n,T,m]. These instances will fit in a twodimensional table D[0:n, 0:m] as shown below. We want to fill in the table so that table D[0:n, 0:m] so that D[i, j] is the solution to the instance I[P,i,T,j]. Thus the solution to our problem can be found in D[m, n].

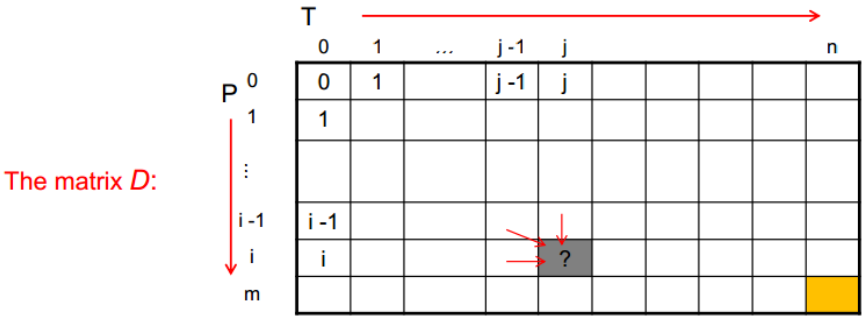


Figure 11: Edit distance matrix

Our intent is that we shall systematically fill this table from smaller to larger instances, with the solution to these instances, by using the Combine function, which is the recurrence formula:

$$D[i,j] = \begin{cases} D[i-1,j-1] & \text{if } P[i] = T[j] \\ min\{\underbrace{D[i-1,j-1]+1}_{\text{substitution}}, \underbrace{D[i-1,j]+1}_{\text{extra in T, deletion in P}}, \underbrace{D[i,j-1]+1}_{\text{deletion in T}}\} & \text{else} \end{cases}$$

$$D[0,0] = 0, D[i,0] = D[0,i] = i \qquad \text{initialization}$$

The entries corresponding to instances of equal size will form diagonals, so filling in one diagonal after the other from upper left is a legal order. When entry D[m, n] is filled, it will contain the solution to I[P, m, T, n], and we are finished. For small instances the last line in the recurrence above can be used to initialize D before filling in the rest.

3.1.2 Verifying the DP-requirement

As the function Combine we use the same function. A proof is we assume that a minimal sequence of edit operations can always be written in the following form:

```
  l o g a r i t h m
a l   g o r i t h m
+   -   *
```

Here '+' = insertion, '-' = deletion, and '*' = substitution.

3.1.3 The intuition behind the general recurrence relation

It turns out that to find the value D[i, j] we only need to look at the entries D[i-1, j-1], D[i , j-1], and D[i-1, j] which all have smaller sizes than D[i, j]. We look at two cases:

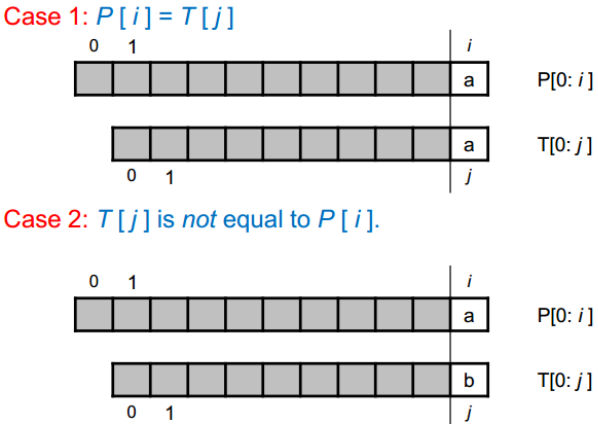


Figure 12: Edit distance example

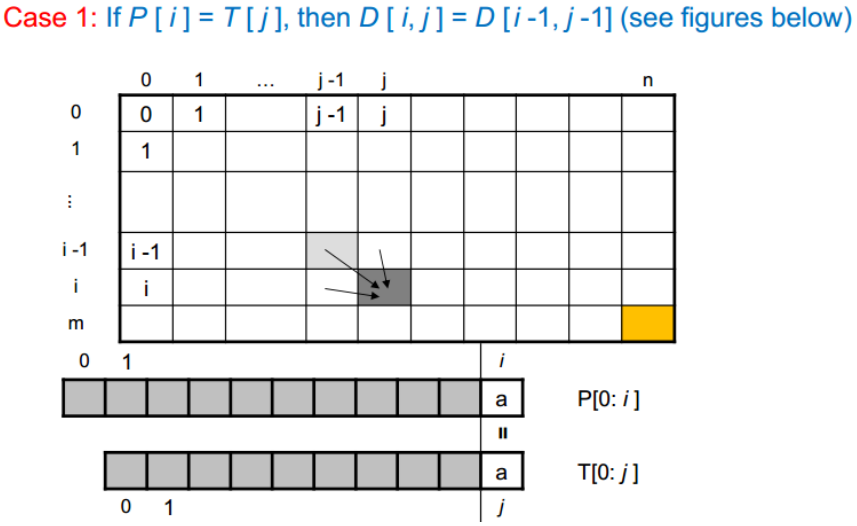


Figure 13: Case 1

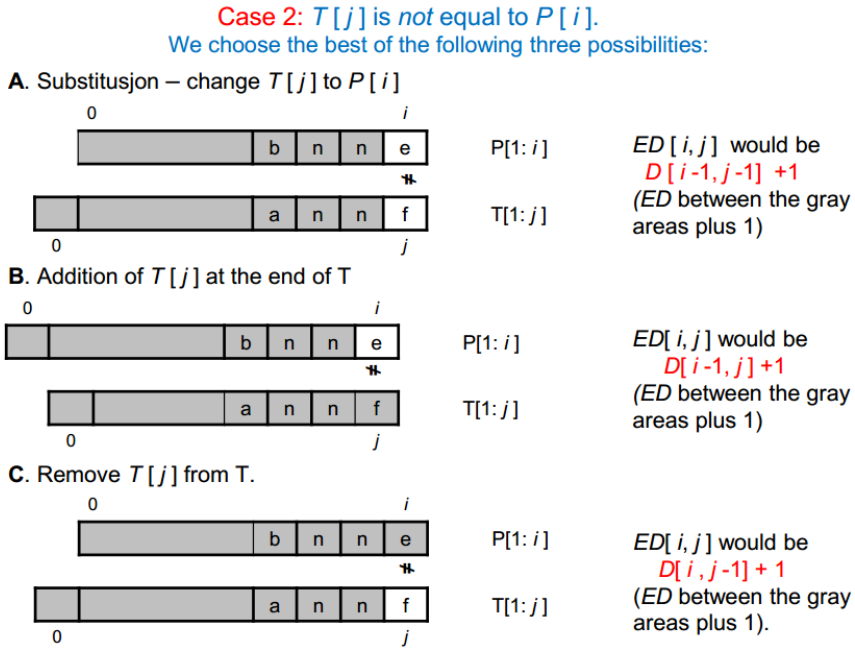


Figure 14: Case 2

3.1.4 Pseudocode

```
function EditDistance ( P [1:n ], T [1:m ] )
  for i = 0 to n do D[ i, 0 ] = i endfor
  for j = 1 to m do D[ 0, j ] = j endfor
  for i = 1 to n do
    for j = 1 to m do
      if P [ i ] = T [ j ] then
        D[i, j] = D[i−1, j−1]
      else
        D[i, j] = min{D[i−1, j−1]+1, D[i−1, j]+1, D[i, j−1]+1}
      endif
    endfor
  endfor
  return(D[n, m])
end
```

Note that this algorithm does not go through the instances strictly in the order from smaller to larger ones. In fact, after the initialization we use the following order of the pairs (i, j): (1,1) (1,2) ...(1,m) (2,1) (2,2) ...(2,m) (3,1) (3,2) ...(n,m). This is OK as also this order ensures that the smaller instances are solved before they are needed to solve a larger instance. An order strictly following increasing size would also work fine, but is slightly more complex to program (following diagonals).

3.1.5 Example

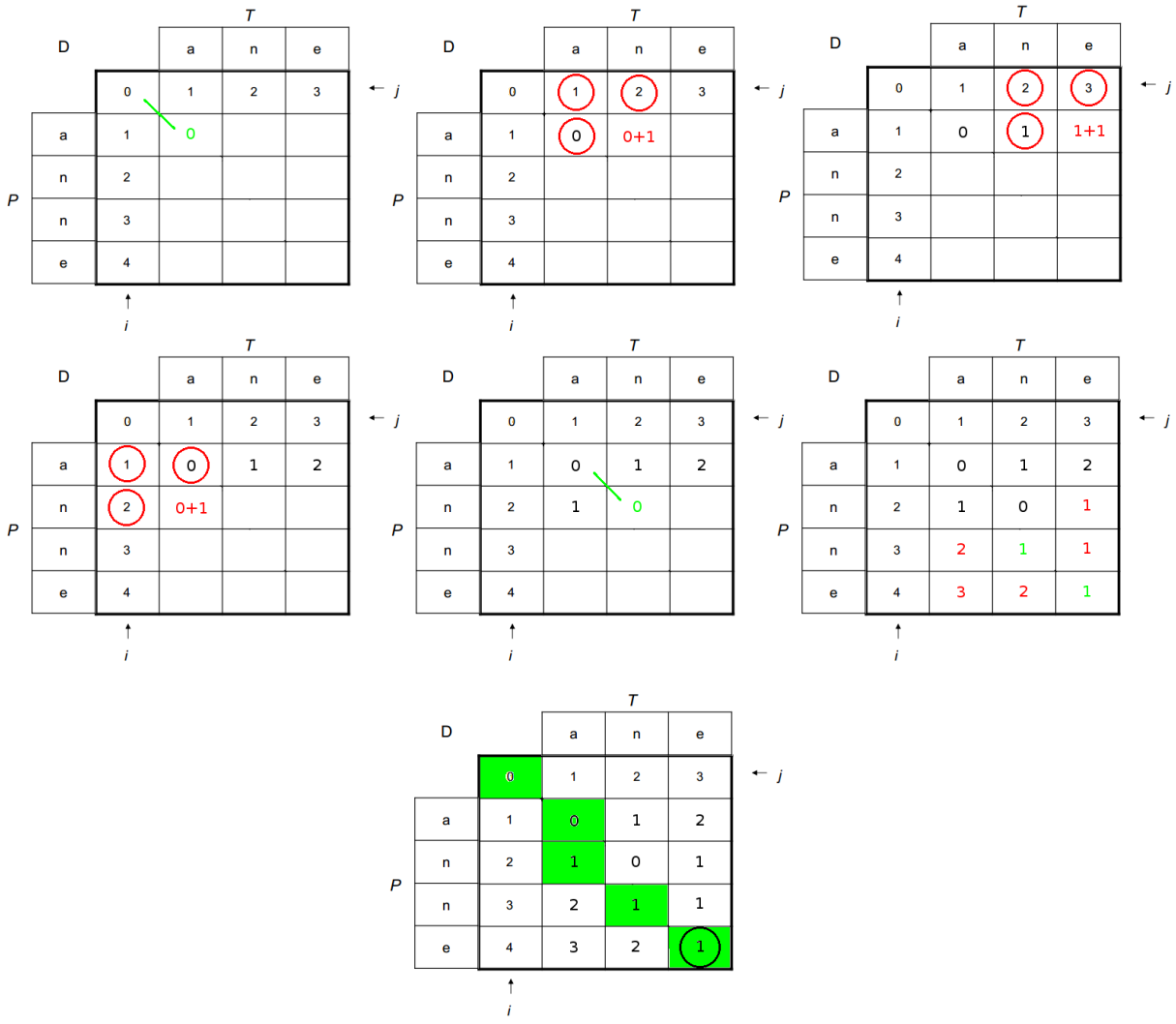


Figure 15: Edit distance = 1

3.2 Bottom-up

Traditionally performed bottom-up. All relevant smaller instances are solved first, and the solutions are stored in a table. Works best when the answers to smaller instances are needed by many larger instances.

3.3 Top-down

A drawback with (traditional) dynamic programming is that one usually solve a number of smaller instances that turns out not to be neededed for the actual (larger) instance you originally wanted to solve. We can instead start at this actual instance we want to solve, and do the computation top-down (usually recursively), and put all computed solutions into the same table as above (see later slides). The table entries then need a special marker "not computed", which also should be the initial value of the entries.

3.3.1 Example: Optimal Matrix Multiplication

Given the sequence M_0, M_1,..., M_{n-1} of matrices. We want to compute the product: $M_0 \cdot M_1 \cdot \ldots \cdot M_{n-1}$. Note that for this multiplication to be meaningful the length of the rows in M_i must be equal to the length of the columns M_{i+1} for i = 0, 1, ..., n-2. Matrix multiplication is associative: (A · B) · C = A · (B · C) (but not symmetric, since A · B generally is different from B · A) Thus, one can do the multiplications in different orders. E.g., with four matrices it can be done in the following five ways:
($M_0 \cdot (M_1 \cdot (M_2 \cdot M_3))$)
($M_0 \cdot ((M_1 \cdot M_2) \cdot M_3)$)
($(M_0 \cdot M_1) \cdot (M_2 \cdot M_3)$)
($((M_0 \cdot (M_1 \cdot M_2)) \cdot M_3)$
($(((M_0 \cdot M_1) \cdot M_2) \cdot M_3)$

The cost (the number of simple (scalar) multiplications) of these will usually vary a lot for the differnt alternatives. We want to find the one with as few scalar multiplications as possible.

Given two matrices A and B with dimentions:

- A is a p $\times$ q matrix
- B is a q $\times$ r matrix

The cost of computing A · B is p · q · r, and the result is a p $\times$ r matrix.

Example
Compute A · B · C, where A is a 10 $\times$ 100 matrix, B is a 100 $\times$ 5 matrix, and C is a 5 $\times$ 50 matrix.

Computing D = (A · B) costs 5,000 and gives a 10 $\times$ 5 matrix.
Computing D · C costs 2,500.
Total cost for (A · B) · C is thus 7,500.

Computing E = (B · C) costs 25,000 and gives a 100 $\times$ 50 matrix.
Computing A · E costs 50,000.
Total cost for A · (B · C) is thus 75,000.

We would indeed prefer to do it the first way!

Given a sequence of matrices M_0, M_1, ..., M_{n-1}. We want to find the cheapest way to do this multiplication (that is, an optimal paranthesization). From the outermost level, the first step in a parenthesizaton is a partition into two parts: ($M_0 \cdot M_1 \cdot \ldots \cdot M_k$) · ($M_{k+1} \cdot M_{k+2} \cdot \ldots \cdot M_{n-1}$)

If we know the best parenthesizaton of the two parts, we can sum their cost and get the cost of the best parenthesizaton given that we have to use this outermost partition. Thus, to find the best parenthesizaton of M_0 , M_1, ..., M_{n-1}, we can simply look at all the n-1 possible outermost partitions (k = 0, 1, n-2), and choose the best. But we will then need the cost of the optimal parenthesizaton of all (or a lot of) instances of smaller sizes.

We shall say that the size of the instance M_i, M_{i+1}, ..., M_j is j-i.

We therefore generally have to look at the best parenthesizaton of all intervals M_i, M_{i+1}, ..., M_j, in the order of growing sizes.

We will refer to the lowest possible cost for M_i, M_{i+1}, ..., M_j as $m_{i,j}$.

Let d_0, d_1, ..., d_n be the dimensiones of the matrices M_0, M_1, ...,M_{n-1}, so that matrix M_i has dimension $d_i \times d_{i+1}$.

As on the previously: Let $m_{i,j}$ be the cost of an optimal parenthesizaton of M_i, M_{i+1}, ..., M_j. Thus the value we are interested in is $m_{0,n-1}$.

The recurrent relation for $m_{i,j}$ will be:

$$m_{i,j} = \begin{cases} min_{i \leq k < j}\{m_{i,k} + m_{k+1,j} + d_i d_{k+1} d_{j+1}\} & \text{when } 0 \leq i < j \leq n-1 \\ 0 & \text{when } 0 \leq i \leq n-1 \end{cases}$$

Note that the values $m_{k,l}$ that we need for computing $m_{i,j}$ are all for smaller instances. That is: l - k < j - i.

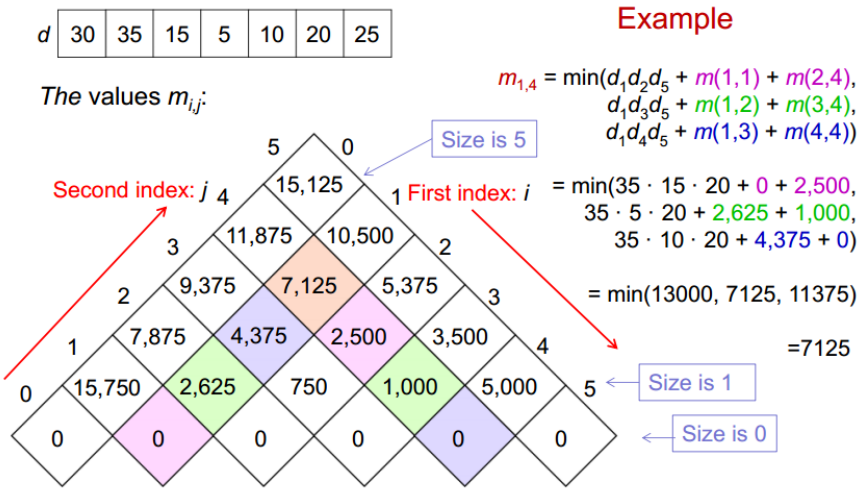


Figure 16:

3.3.1.1 Pseudocode

```
function OptimalParens(d[0:n-1])
  for i = 0 to n-1 do
    m[i,i] = 0
  for diag = 1 to n-1 do
    for i = 0 to n-1-diag do
      j = i + diag
      m[i,j] = infinite // Relative to the scalar values that can occur
      for k = i to j-1 do
        q = m[i,k] + m[k+1,j] + d[i] d[k+1] d[j+1]
        if q < m[i,j] then
          m[i,j] = q
          c[i,j] = k
        endif
  return m[0,n-1]
end
```

3.4 Memoization

A drawback with bottom up dynamic programming is that you might solve a lot of smaller instances whose answers are never used. We can instead do the computation recursively from the top, and store the (really needed) answers of the smaller instances in the same table as before. Then we can later find the answers in this table if we need the answer to the same instance once more. The reason we do not always use this technique is that recursion in itself can take a lot of time, so that a simple bottom up may be faster. For the recursive method to work, we need a flag "NotYetComputed" in each entry, and if this flag is set when we need that value, we compute it, and save the result (and turn off the flag, so the recursion from here will only be done once). The "NotYetComputed" flag must be set in all entries at the start of the algorithm. It is always safe to solve all the smaller instances before any larger ones, using the defined size of the instances. However, if we know what smaller instances are needed to solve a larger instance, we can deviate from that. The important thing is that the smaller instances needed to solve a certain instance J is computed before we start solving J. Thus, if we know the "dependency graph" of the problem (which must be cycle-free, see examples below), the important thing is to look at the instances in an order that conforms with this dependency. This freedom is often utilized to get a simple computation.

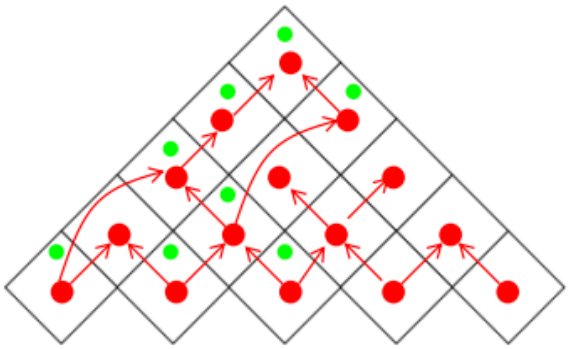


Figure 17: At most the entries with a green dot will have to be computed

4 Priority queues

Priority queues are data structures that hold elements with some kind of priority (key) in a queue-like structure, implementing the following operations:

- insert() – Inserting an element into the queue.
- deleteMin() – Removing the element with the highest priority.

And maybe also:

- buildHeap() – Build a queue from a set (>1) of elements.
- increaseKey()/DecreaseKey() – Change priority.
- delete() – Removing an element from the queue.
- merge() – Merge two queues.

An unsorted linked list can be used. insert() inserts an element at the head of the list (O(1)), and deleteMin() searches the list for the element with the highest priority and removes it (O(n)). A sorted list can also be used (reversed running times). Not very efficient implementations. To make an efficient priority queue, it is enough to keeps the elements "almost sorted".

4.1 Binary heaps

A binary heap is organized as a complete binary tree. (All levels are full, except possibly the last.) In a binary heap the element in the root must have a key less than or equal to the key of its children, in addition each sub-tree must be a binary heap.

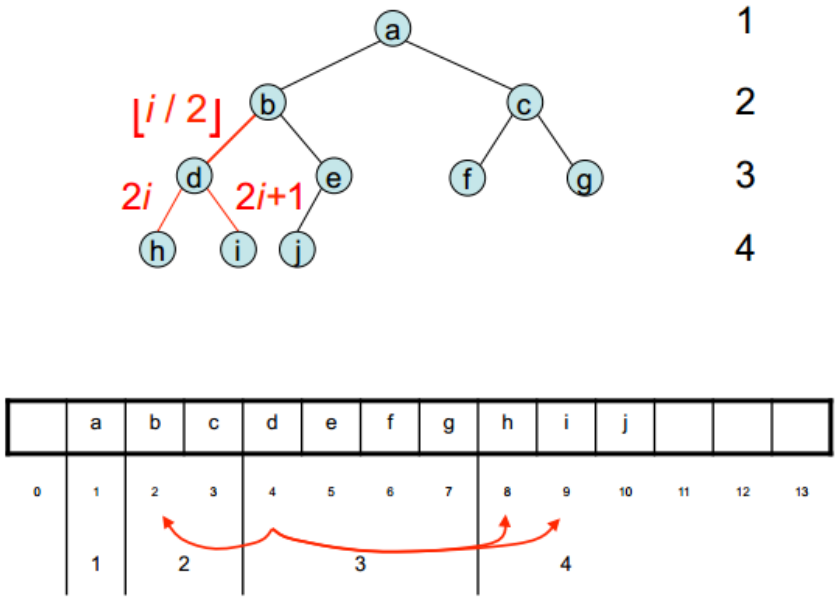


Figure 18: Binary heap

4.1.1 insert()

insert(14)

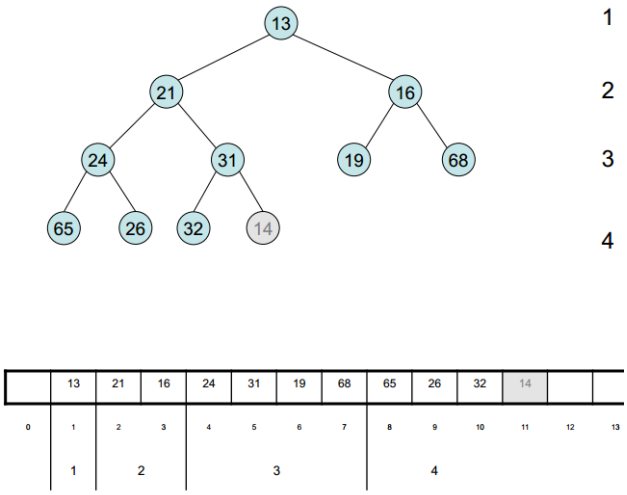


Figure 19: Insert last

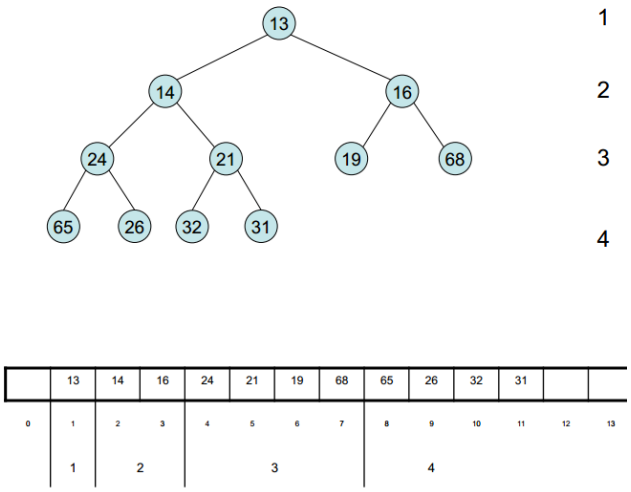


Figure 20: Percolate/swap upward

4.1.2 deleteMin()

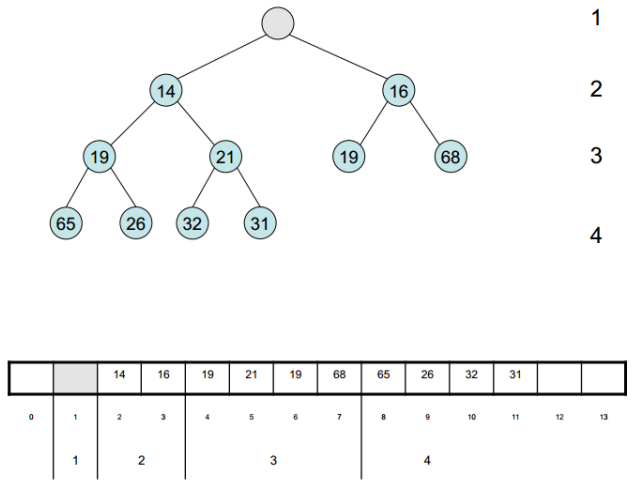


Figure 21: Remove smalest

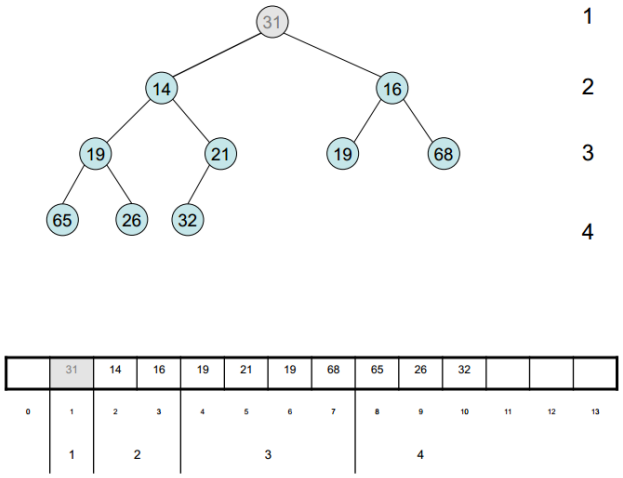


Figure 22: Set last as root

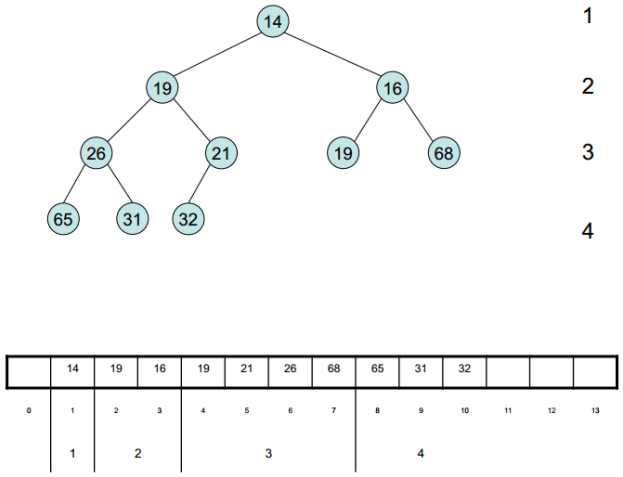


Figure 23: Percolate/swap down

4.1.3 Complexity

	worst case	average
insert()	O(log N)	O(1)
deleteMin()	O(log N)	O(log N)
buildHeap()	O(N)	

(Insert elements into the array unsorted, and run percolateDown() on each root in the resulting heap (the tree), bottom up)
(The sum of the heights of a binary tree with N nodes is O(N).)

merge()	O(N)

(N = number of elements)

4.2 Leftist heaps

To implement an efficient merge(), we move away from arrays, and implement so-called leftist heaps as pure trees. The idea is to make the heap (the tree) as skewed as possible, and do all the work on a short (right)

branch, leaving the long (left) branch untouched. A leftist heap is still a binary tree with the heap structure (key in root is lower than key in children), but with an extra skewness requirement. For all nodes X in our tree, we define the null-path-length(X) as the distance from X to a node without two children (i.e. 0 or 1). The skewness requirement is that for every node the null path length of its left child be at least as large as the null path length of the right child.

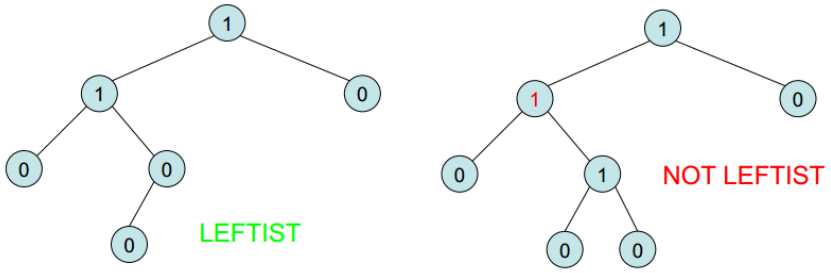


Figure 24: Leftist heap

4.2.1 merge()

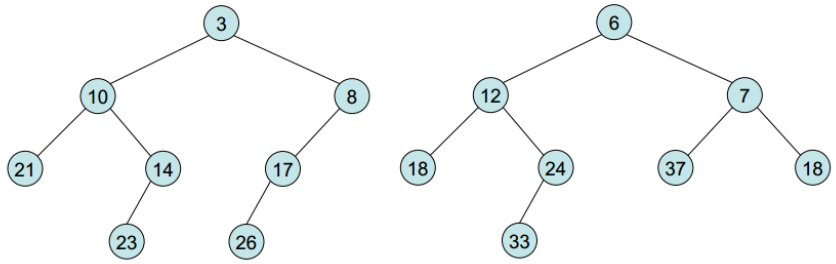


Figure 25: merge()

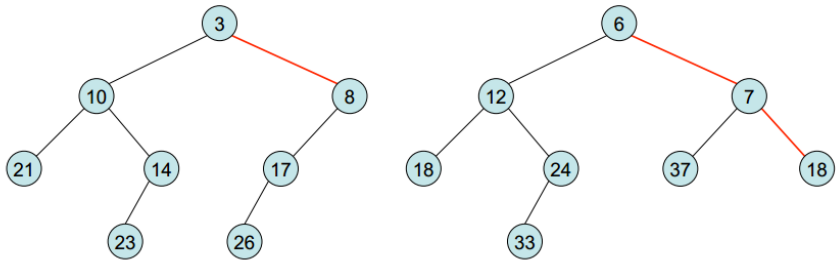


Figure 26: check right side

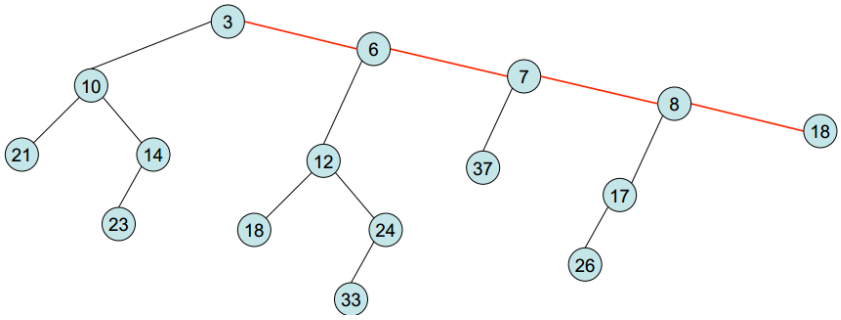


Figure 27: sorted connect

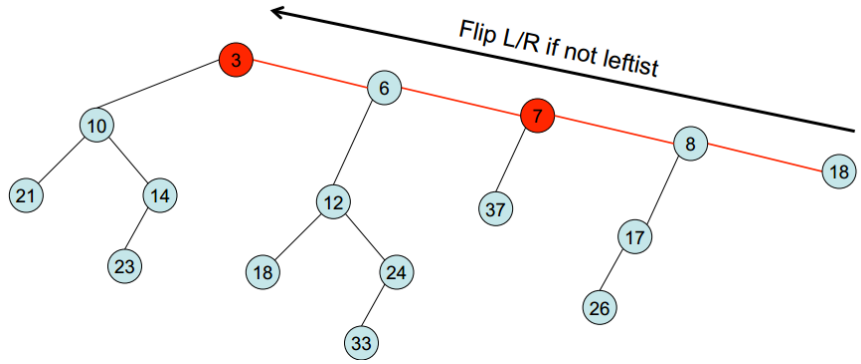


Figure 28: flip if not leftist

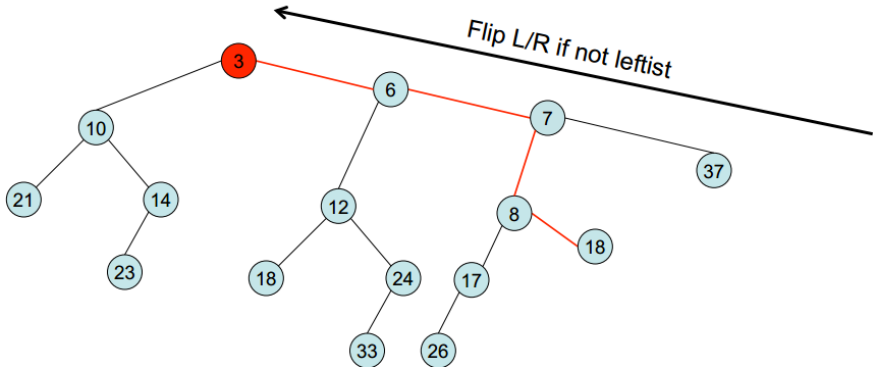


Figure 29: flip 7

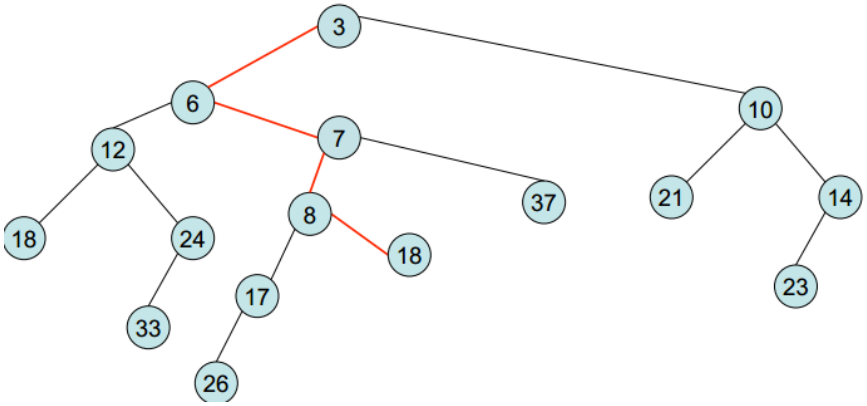


Figure 30: flip 3

4.2.2 insert()

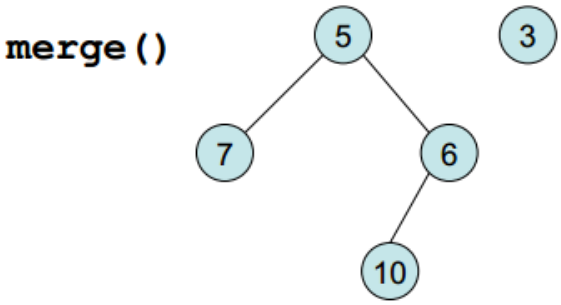


Figure 31: insert(3)

4.2.3 deleteMin()

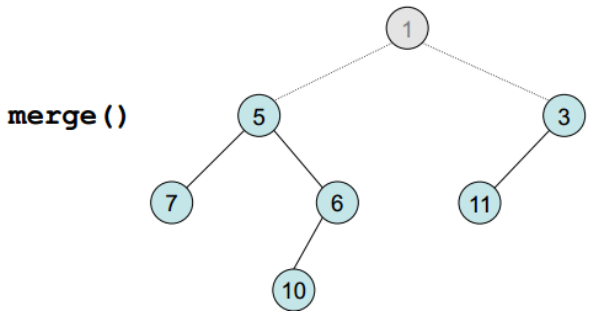


Figure 32: delete min and merge children

4.2.4 Complexity

	worst case
merge()	O(log N)
insert()	O(log N)
deleteMin()	O(log N)
buildHeap()	O(N)

(N = number of elements)

In a leftist heap with N nodes, the right path is at most log(N+1) long.

4.3 Binomial heaps

Binomial heaps are collections of trees (sometimes called a forest), each tree a heap.

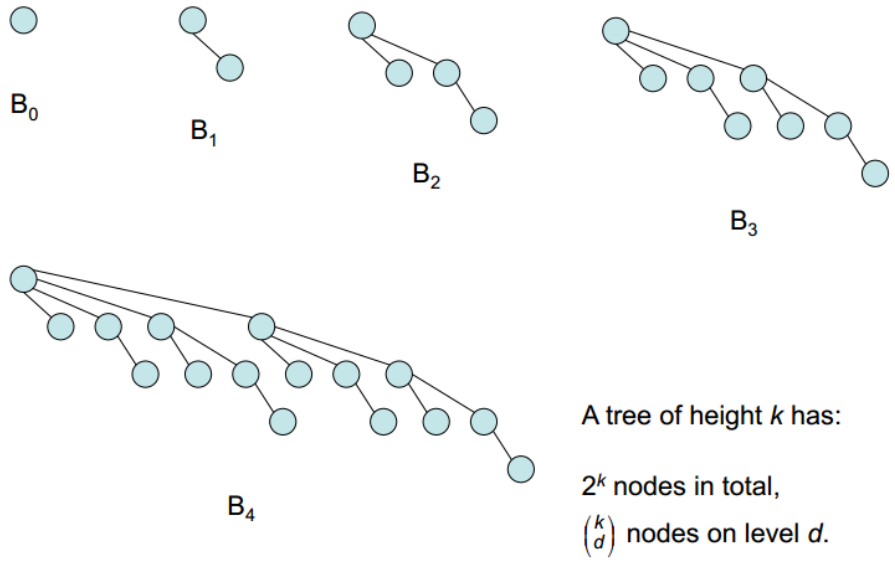


Figure 33: Binomial trees

$$\binom{k}{d} = \frac{k!}{d!(k-d)!} \text{ for } 0 \leq k \leq n$$

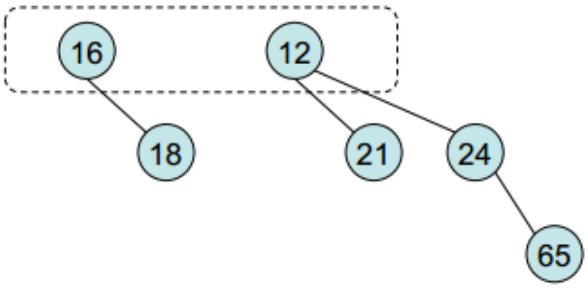


Figure 34: Binomial heap

Maximum one tree of each size:
6 elements: 6 binary = 110 (4+2+0) $\rightarrow$ B_2 B_1 ~~B_0~~

The length of the root list in a heap of N elements is O(log N). Doubly linked, circular list.

4.3.1 merge()

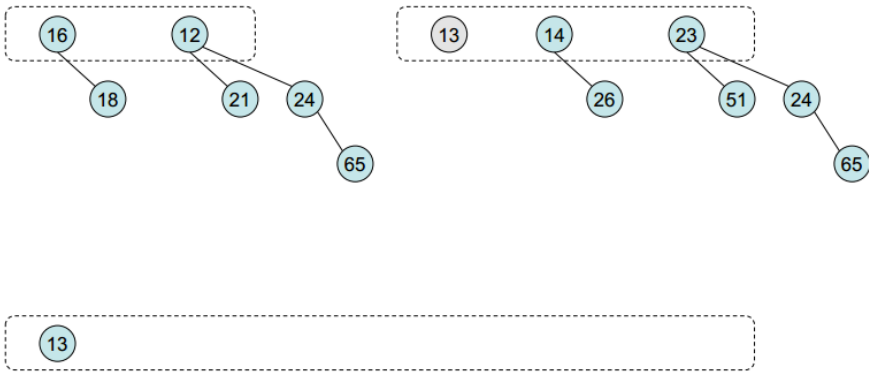


Figure 35: Take smalest tree

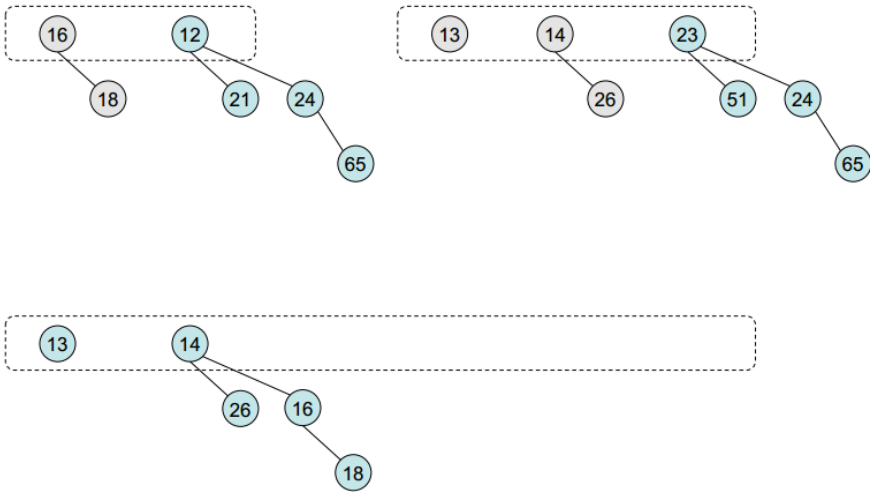


Figure 36: Take next tree (more than one $\rightarrow$ merge with smallest as root)

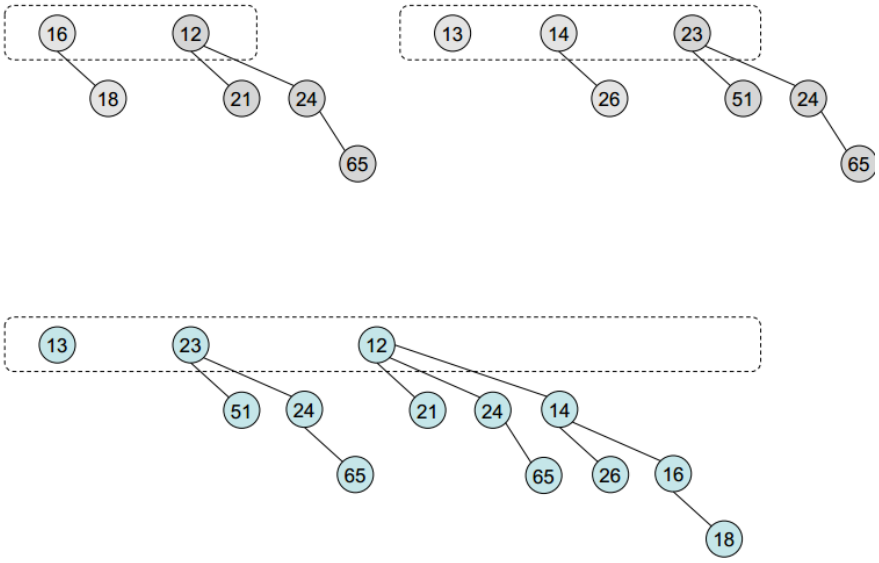


Figure 37: Next → merge smallest

The trees (the root list) is kept sorted on height.

4.3.2 deleteMin()

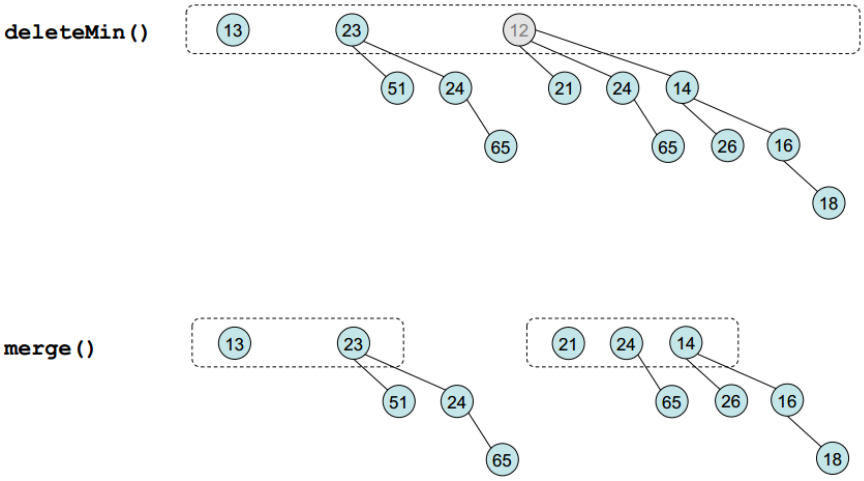


Figure 38: Delete min → merge()

4.3.3 Complexity

	worst case	average
merge()	O(log N)	O(log N)
insert()	O(log N)	O(1)
deleteMin()	O(log N)	O(log N)
buildHeap()	O(N)	O(N)

(Run N insert() on an initially empty heap.)
(N = number of elements)

4.3.4 Implementation

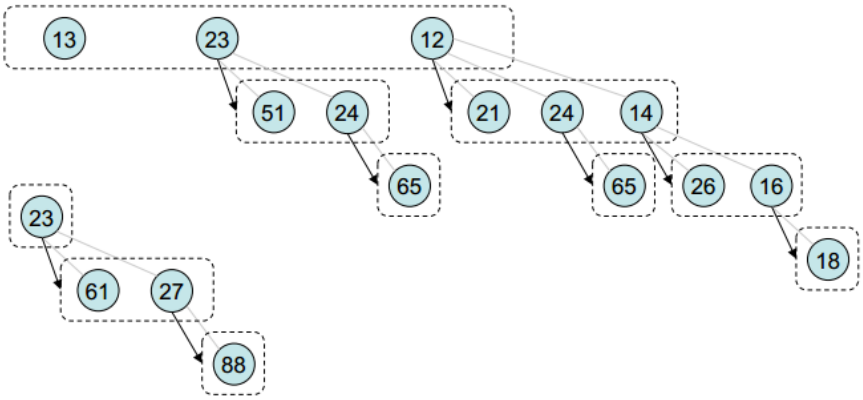


Figure 39: Doubly linked, circular lists

4.4 Fibonacci heaps

Very elegant, and in theory efficient, way to implement heaps: Most operations have O(1) amortized running time (Fredman & Tarjan '87).

Combines elements from leftist heaps and binomial heaps. A bit complicated to implement, and certain hidden constants are a bit high. Best suited when there are few deleteMin() compared to the other operations. The data structure was developed for a shortest path algorithm (with many decreaseKey() operations), also used in spanning tree algorithms.

4.4.1 decreaseKey()

We include a smart decreaseKey() method from leftist heaps.

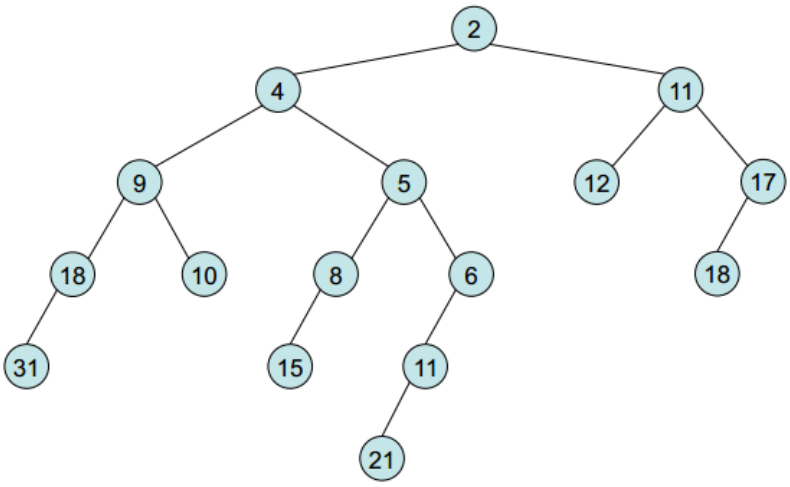


Figure 40: Leftist heap

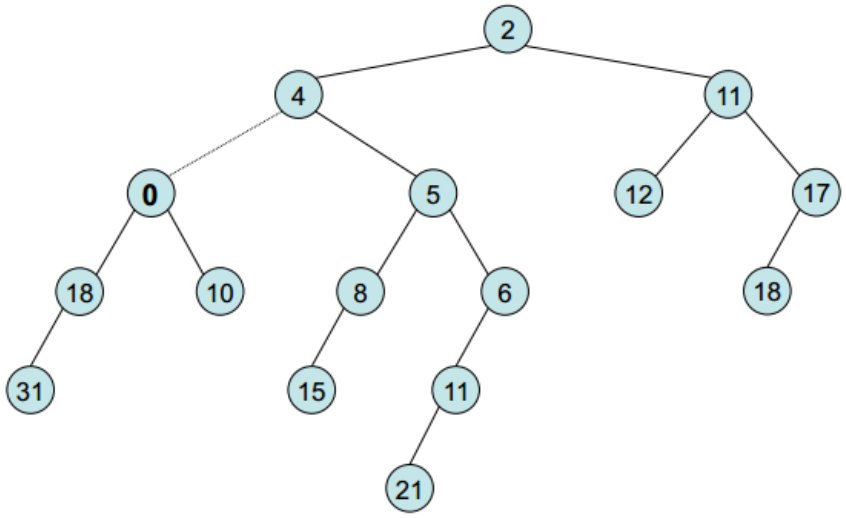


Figure 41: Decrease key

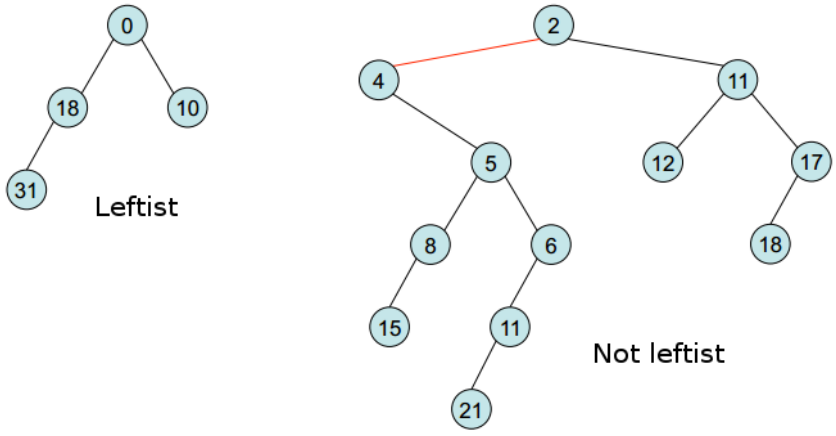


Figure 42: Detach key and subtree

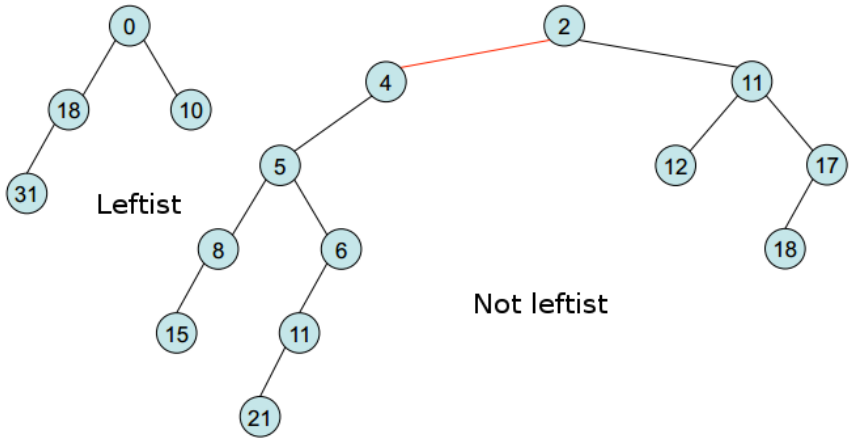


Figure 43: Make both leftist

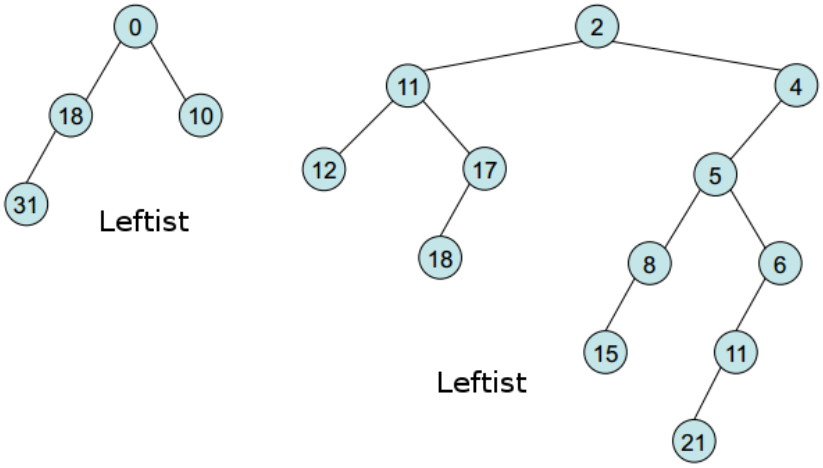


Figure 44: Both leftist

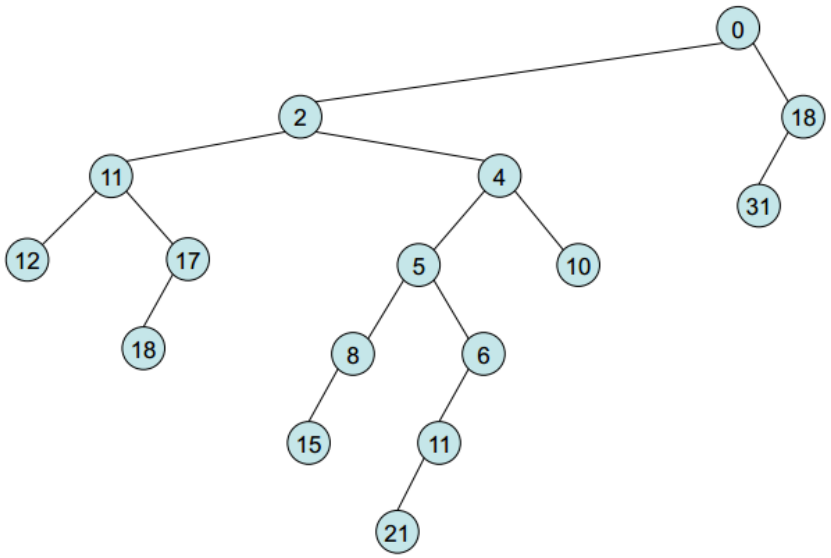


Figure 45: Attach

The method must be modified a bit, as we wish to use trees that are binomial trees, or partial binomial trees.

- Nodes are marked the first time a child is removed.
- The second time a node gets a child removed, it is cut off, and becomes the root of a separate tree.

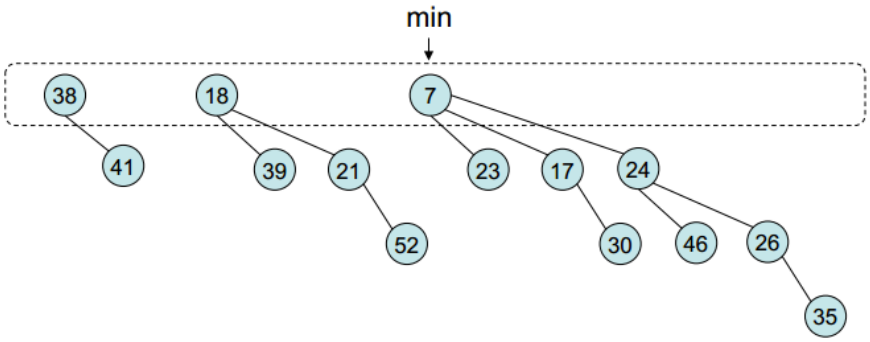


Figure 46: Attach

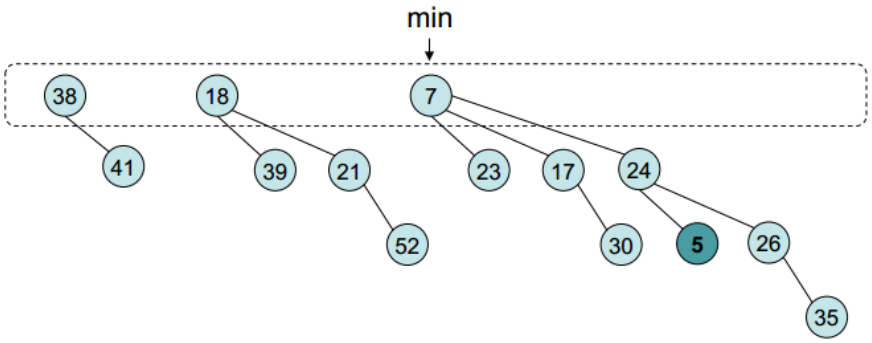


Figure 47: Attach

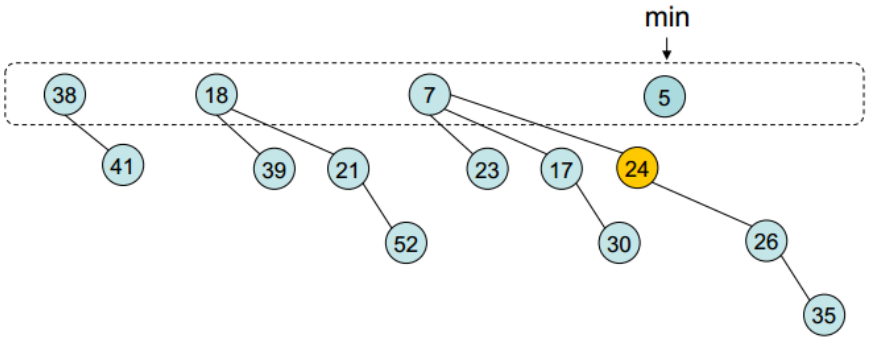


Figure 48: Attach

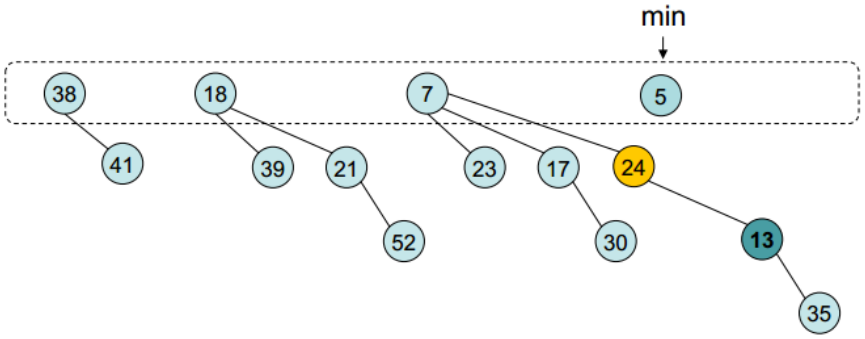


Figure 49: Attach

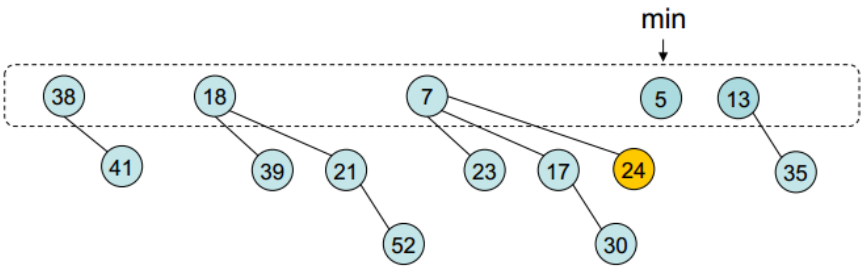


Figure 50: Attach

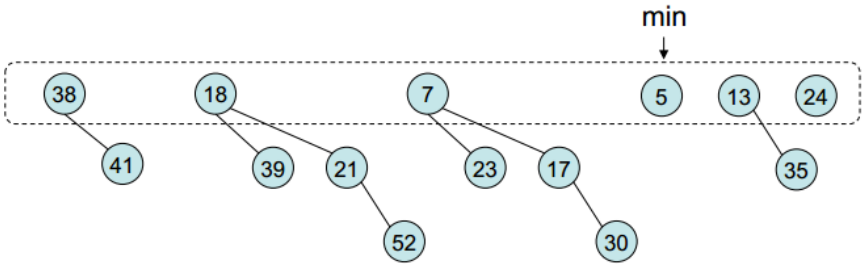


Figure 51: Attach

4.4.2 merge()

We use lazy merging/lazy binomial queue.

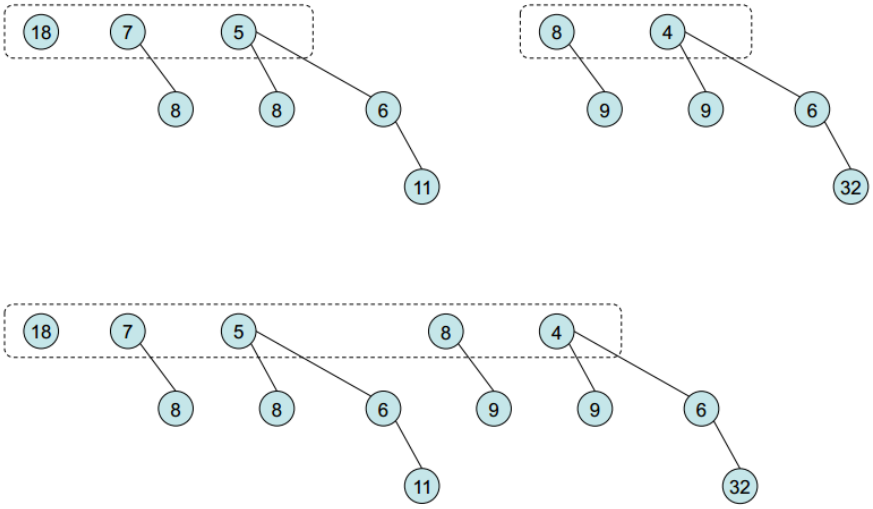


Figure 52: merge()

4.4.3 deleteMin()

The problem with our decreaseKey()-method and lazy merging is that we have to clean up afterwards. This is done by the deleteMin()-method which becomes expensive (O(log N) amortized time): All trees are examined, we start with the smallest, and merge two and two, so that we get at most one tree of each size. Each root has

a number of children - this is used as the size of the tree. (Recall how we construct binomial trees, and that they may be partial as a result of deleteMin() operations) The trees are put in lists, one per size, and we begin merging, starting with the smallest.

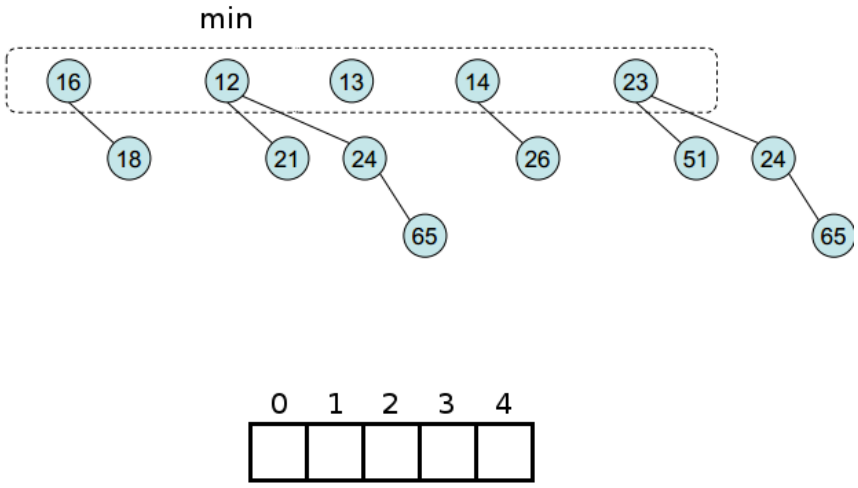


Figure 53: delete min and add children to heap

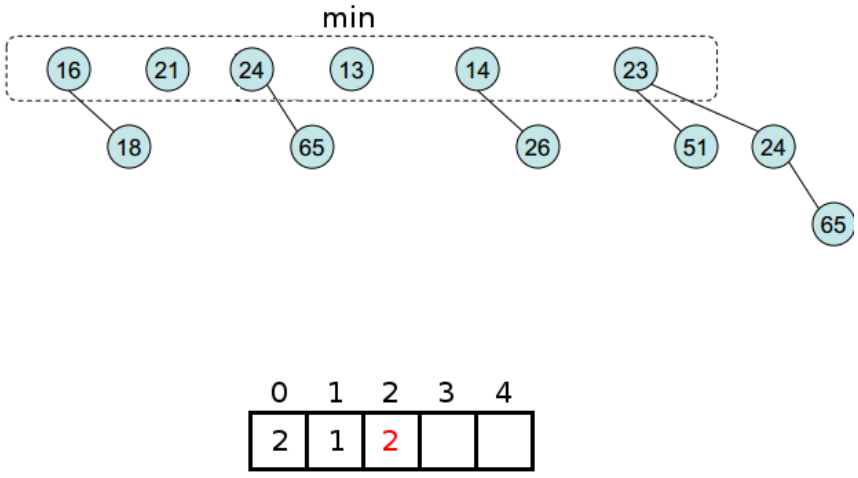


Figure 54: two trees with same rank $\rightarrow$ merge

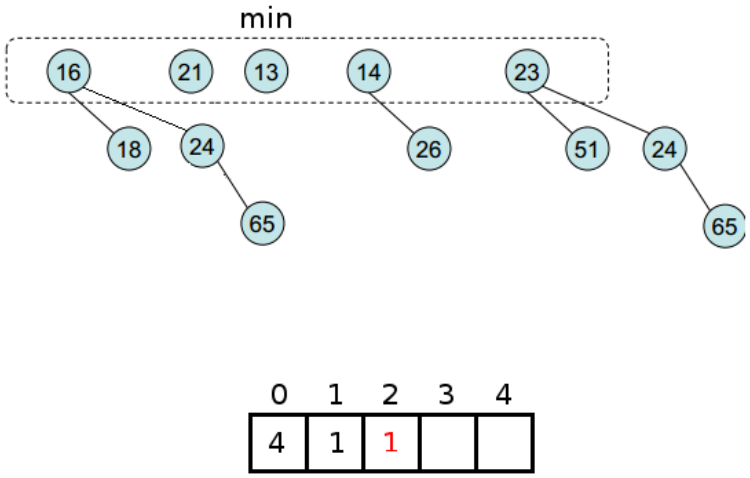


Figure 55: two trees with same rank → merge

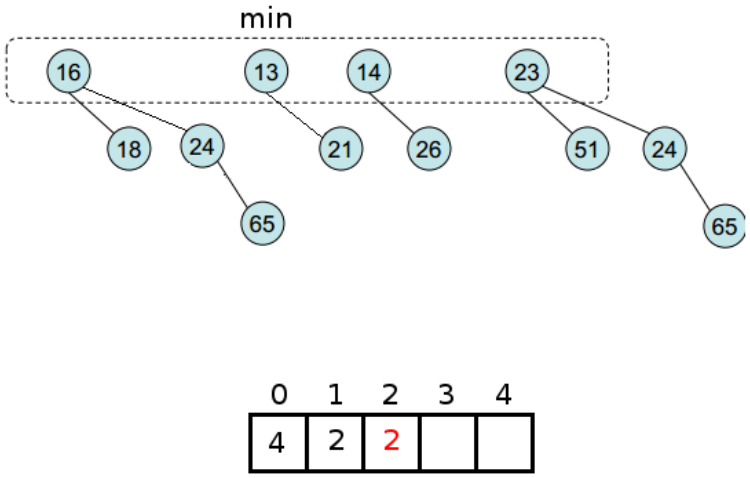


Figure 56: two trees with same rank → merge

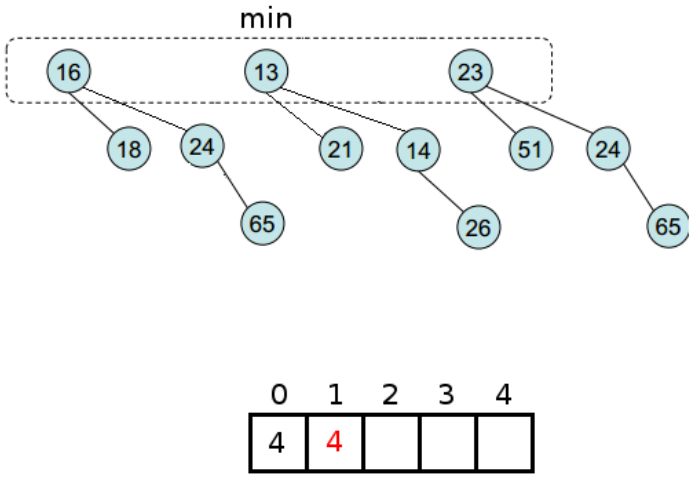


Figure 57: two trees with same rank → merge

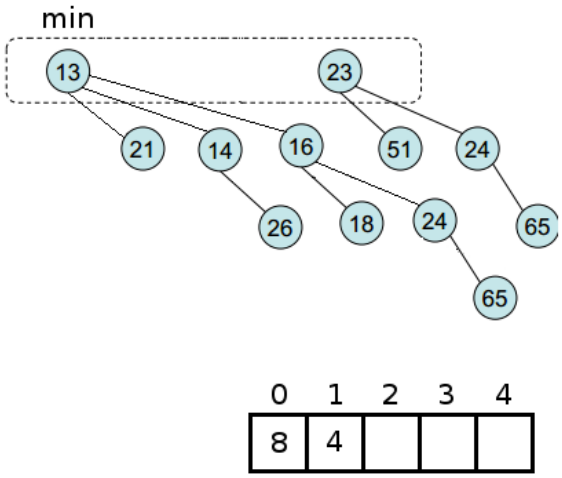


Figure 58: no trees with same rank

4.4.4 Complexity

	amortized time
insert()	O(1)
decreaseKey()	O(1)
merge()	O(1)
deleteMin()	O(log N)
buildHeap()	O(N)

(Run N insert() on an initially empty heap.) (N = number of elements)

5 Search strategies in State-Spaces

The state-space of a system is the set of states in which the system can exist. Some states are called goal states. That's where we want to end up. Each search algorithm will have a way of traversing the states, and these are usually indicated by directed edges, as is seen on the figure below. Such an algorithm will usually have a number of decision poins: "Where to seach next?". The full tree with all choices is the state space tree for that algorithm. Thus, different algorithms will have different state space trees. Main problem: The state space is usually very large.

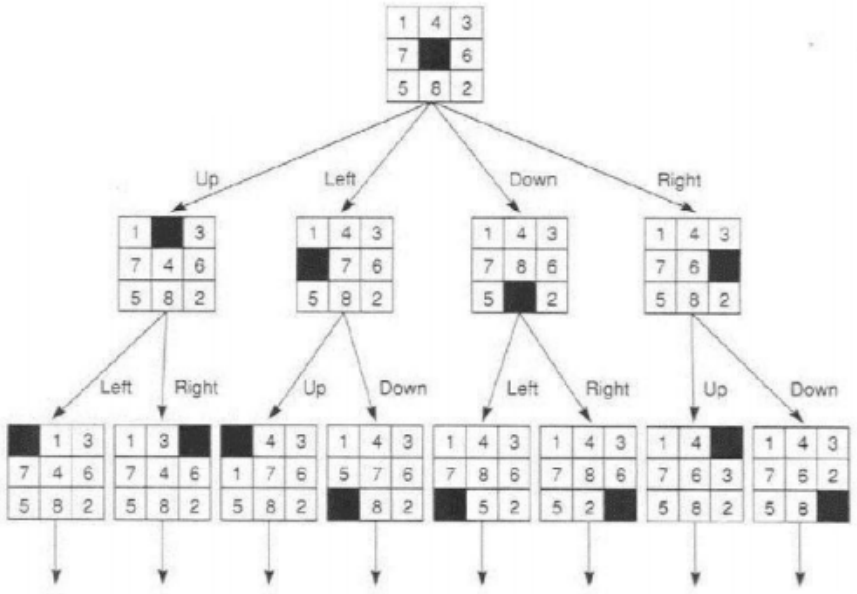


Figure 59: Each constellation of an 8-puzzel is a state.

5.1 "Models" for decision sequences

here are usually more than one decision sequence for a given problem, and they may lead to different state space trees. Example: Find, if possible, a Hamiltonian Cycle. There are (at least) two natural decision sequences:

1. Start at any node, and try to grow paths from this node in all possible ways.
2. Start with one edge, and add edges as long as they don't make a cycle with already chosen edges.

These lead to different state space trees;

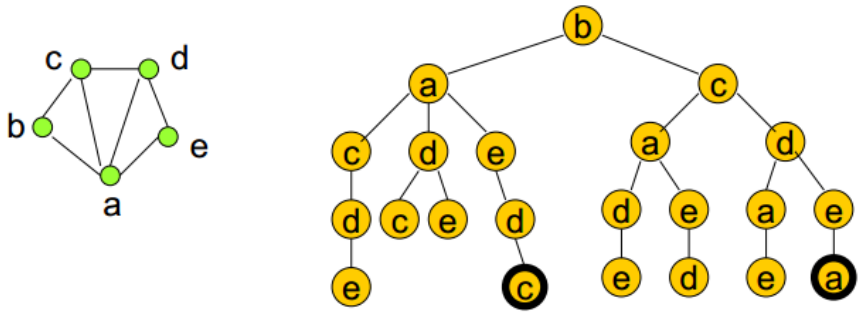


Figure 60: 1. decision sequence

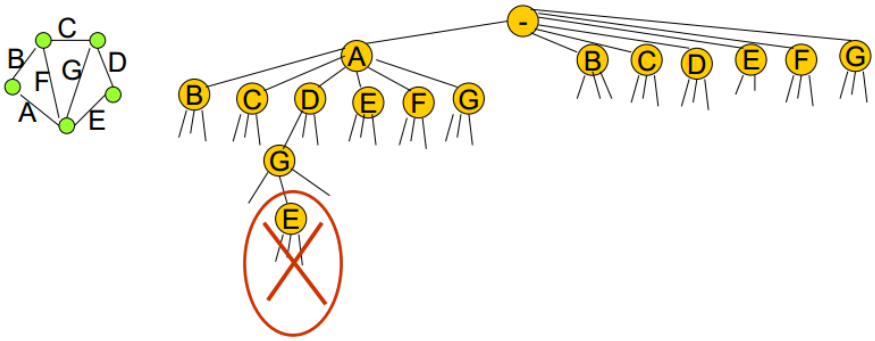


Figure 61: 2. decision sequence

Different decision sequences can give better or worse possibilities for pruning.

Some times the path leading to the goal node is an important part of the solution. E.g. eight-puzzle: Here the path leading to the goal node is the sequence of moves we should perform to solve the puzzle. But e.g. adding Hamiltonian cycle edges: Here the order in which we added the edges is usually of no significance for the Hamiltonian circuit we end up with.

5.2 Backtracking (DFS)

Searches the state space tree depth first with backtracking, until it reaches a goal state (or has visited all states). The easiest implementation is usually to use a recursive procedure. Needs little memory (only O(<the depth of the tree>)). If the edges have lengths and we e.g. want a shortest possible Hamiltonian cycle, we can use heuristics to choose the most promising direction first (e.g. choose the shortest legal edge from where you are now). One has to use "pruning" (or "bounding") as often as possible. This is important, as a search usually is time-exponential, and we must use all tricks we can find to limit the execution time. Main pruning principle: Don't enter subtrees that cannot contain a goal node. But the difficulty is to find where this is true.

5.2.1 Pseudocode

A template for implementing depth-first-search may look like this:

```
1  procdure DFS(v){
2    If <v is a goal node> then return "..."
3    v.visited = TRUE;
4    for <each neighbour w of v> do
5      if not w.visited then DFS(w)
6  }
```

It can not only be used for trees, but also for graphs, because of the "visited" feature.

5.3 Branch-and-bound

It needs a lot of space: Must store all nodes that have been seen but not explored further. We can also indicate for each node how "promising" it is (heuristric), and always proceed from the currently most promising one. Natural to use a priority queue to choose next node.
Uses some form of breadth-first-search. We have three sets of nodes:

1. The "finished nodes" (dark blue). Often do not need to be stored
2. The "live nodes" (green) seen but not explored further. Large set, that must be stored.

3. The "unseen nodes" (light blue). We often don't have to look at all of them.

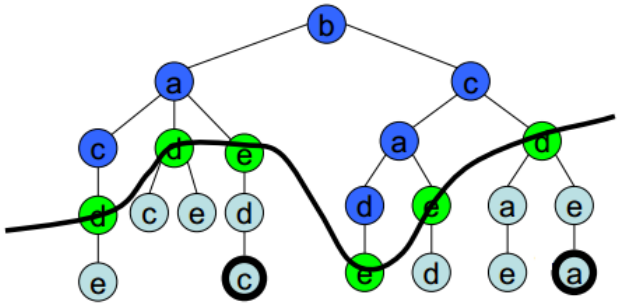


Figure 62: DFS

The live nodes (green) will always be a cut through the state-space tree (or likewise if it is a graph). The main step:

- Choose a node N from the set of Live Nodes.
- If N is a goal node, then we are finished, else:
- Take N out of the Live-Node set and insert it into the finished nodes.
- Insert all children of N into the Live-Node set.
- BUT: if we are searching a graph, only insert unseen ones.

5.3.1 Strategies

Three strategies:

- The Live-Node set is a FIFO-queue: We get traditional breadth first
- The Live-Node set is a LIFO-queue: The order will be similar to depth-first, but not exactly
- The LiveNode queue is a priority queue: We can call this priority search. If the priority stems from a certain kind of heuristics, then this will be A*-search.

5.4 Iterative deepening

A drawback with DFS is that you can end up going very deep in one branch without finding anything, even if there is a shallow goal node close by. We can avoid this by first doing DFS to level one, then to level two, etc. With a reasonable branching facor, this will not give too much extra work, and we never have to use much space. We only "test for goal nodes" at the levels we have not been on before.

Modifies depth-limited search by iteratively trying all possible depths as the cutoff limit. Combines benefits of depth-first and breadth-first. Time complexity is $O(b^d)$, space complexity $O(bd)$. Iterative deepening is the preferred (uninformed) search strategy when there is a large search space and the solution depth is unknown.

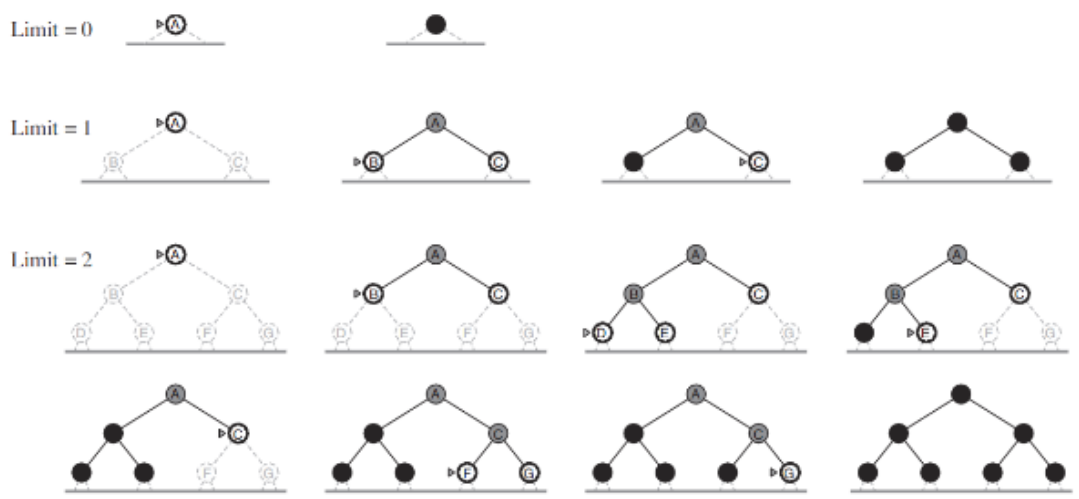


Figure 63: Iterative deepening search illustrated

5.4.1 Pseduocode

```
function ITERATIVE_DEEPENING_SEARCH(problem) returns a solution or failure
  for depth <= 0 to infinity do
    result <= DEPTH_LIMITED_SEARCH(problem, depth)
    if result not equal to cutoff then return result
```

5.5 Dijkstra's

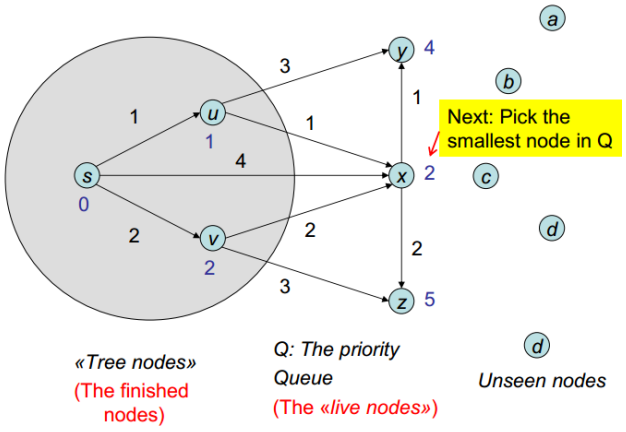


Figure 64: Dijkstra's for directed graph

5.5.1 Pseudocode

```
procedure Dijkstra(graph G, node source)
  for each each node v in G do  // Initialisation
    v.dist := infinte           // Marks as unseen nodes
    v.previous := NIL           // Pointer to remeber the path back to source
  source.dist := 0              // Distance from source to itself
```

```
6     Q := { source }                 // The initial priority queue only contains source
7     while Q is not empty do
8       u := extract_min(Q)           // Node in Q closest to source. Is removed from Q
9       for each neighbor v of u do // Key in prio-queue is distance from source
10        x = length(u, v) + u.dist
11        if x < v.dist then          // Nodes in the "tree" will never pass this test
12          v.dist := x
13          v.previous := u           // Shortest path "back towards the source"
14  end
```

5.6 A*-search

Backtracking/depth-first, LIFO/FIFO, branch-and-bound, breadthfirst and Dijkstra use only local information when choosing the next step.

A*-search is simular to Dijkstra, but it uses some global heuristics ("qualified guesses") to make better choices from Q at each step. Much used in AI and knowlegde based systems. A*-search (as Dijkstra) is useful for problems where we have. An explicit or implicit graph of <states>. There is a start state and a number of goal states. The (directed) edges represent legal state transitions, and they all have a cost And (like for Dijkstra) the aim is to find the cheapest (shortest) path from the start node to a goal node. A*-search: If we for each node in Q can "guess" how far it is to a goal node, then we can often speed up the algorithm considerably!

5.6.1 Heuristic

The strategy is a sort of breadth-first search, like in Dikstra. However, we now use an estimate h(v) for the shortest path from the node v to some goal node (h for heuristic). The value we use for choosing the best next node is now: f(v) = g(v) + h(v) (while Dijkstra uses only g(v)). Thus, we get a priority-first search with this value as priority, but few guarantees can be given about the result.

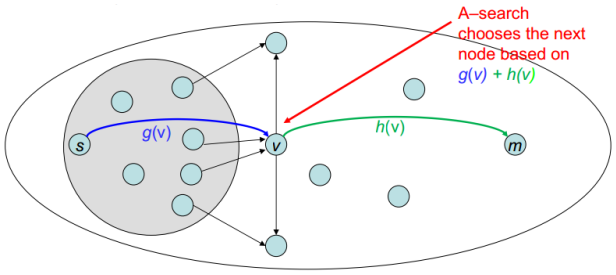


Figure 65: A*

If we know that h(v) is never larger than the really shortest path to the closest goal node, and we use the extended Dijkstra algorithm, we will finally get the correct result (this is not proven here). We shall call this algorithm "weak" A*-search. Here, we will often have to go back to nodes that we "thought we were finished with" (nodes in the tree will have to go back to the priority queue, as we may later find an even shorter path to them). This usually gives a lot of extra work. Dijkstra never move a tree-node back into the queue, and still get the correct answer. We will put requirements on h(v) so that this will also be true for the extended algorithm. Using such a heuristics, we'll talk about "strong" A*-search. "weak" and "strong" are not used in this way in the textbook.

5.6.2 Monotone heuristics

The function h(v) should be an estaimate of the distance from v to the closest goal node. If h(v) is never larger than the shortest distance to a goal, we know that the search will terminate, but maybe slowly. However, we can restrict h(v) further, and get more efficient algorithms. This gives the "strong" A*-algorithm. The function h(v) must then have the follwing properties:

- (As before) The function h(v) is never larger that the real distance to the closest goal-node.
- h(g) = 0 for all goal nodes g
- And a sort of "triangule inequality" must hold:
  If there is a transition from node u to node v with cost c(u,v), then the following should hold:

$$h(u) \leq c(u,v) + h(v)$$

  In this case, h(v) is said to be monotone.

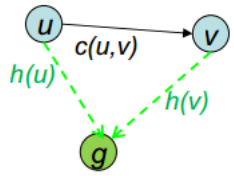


Figure 66:

The criteria for monotonicity on h:

- If there is an edge from v to w with weight c(v,w), then we should always have: h(v) $\leq$ c(v,w) + h(w)
- Every goal node m should have h(m) = 0

A nice thing here is that if these two requirements are fulfilled. Then the first requirement (that h(v) is never larger than the real shortest distance to a goal) is automatically fulfilled.

Sketch of a proof: We assume that u $\rightarrow$v $\rightarrow$ w $\rightarrow$ m is a shortest path from u to a goal state m. We set up the inequalities we know:

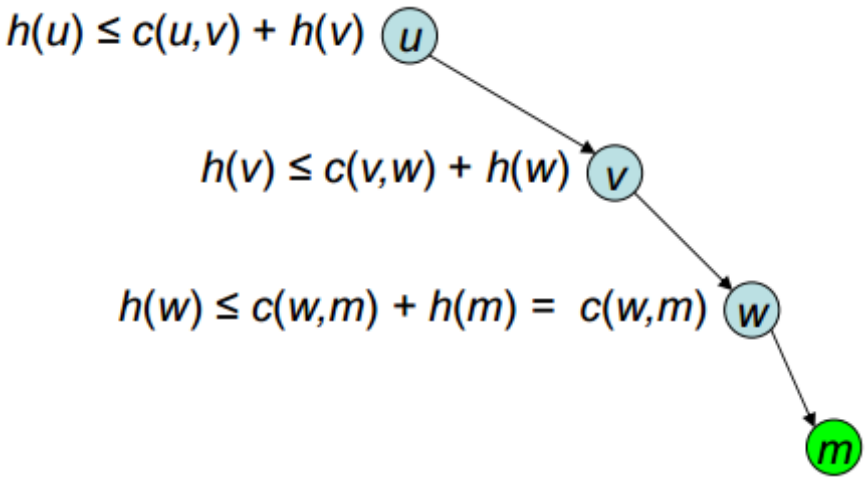


Figure 67: if we combine, we get: h(u) $\leq$ c(u,v) + c(v,w) + c(w,m)

5.6.3 Relation to Dijkstra's algorithm

If we use h(v) = 0 for all nodes, this will be Dijkstra's algorithm. By using a better heuristic we hope to work less with paths that can not give solutions, so that the algorithm will run faster. However, we will then remove the nodes from Q in a differnt order than in Dijkstra. Thus, we can no longer know that when v is moved from Q to the tree, it has the correct shortest path length v(g). BUT luckily, we can prove this:

- If h is monotone, then values of g(v) and parent(v) will always be correct when v is removed from Q to become a tree node.
- Therefore, we never need to go back to tree nodes, move them back into Q, and update their variables once more.

5.6.4 Data for the algorithm

We have a directed graph G with edge weights c(u,v), a start node s, a number of goal-nodes, and finally a monotone heuristic function h(u), (often set in all nodes during initialisation, and never changed). In addition, each node v has the following variables:

- g: This variable will normally change many times during the execution of the algorithm, but its final value (after being moved to the tree) will be the length of the shortest path from the start node to v.
- parent: This variable will end up pointing to the parent in a tree of shortest paths back to the start node
- f: This variable will all the time have the value g(v) + h(v), that is, an estimate of the path length from the start node to a goal node, through the node v

We keep a priority queue Q of nodes, where the value of f is used as priority. This queue is initialized to contain only the start node s, with g(s) = 0. The value of f will change during the algorithm, and the node must then be "moved" in the priority queue. The nodes that is not in Q at the moment are partitioned into two types:

- Tree nodes: In these the parent pointer is part of a tree with the start node as root. These nodes have all been in Q earlier.
- Unseen nodes (those that we have not touched up until now).

5.6.5 The algorithm

Q is initialized with the start node s, with g(s) = 0 (all other nodes are unseen). The main step, that are repeted until Q is empty: Find the node v in Q with the smallest f-value. We then have two alternatives:

1. If v is a goal node the algorithm should terminate, and g(u) and parent(u) indicates the shortest path (backwards) from the start node to v.
2. Otherwise, remove v from Q, and let it become a tree node (it now has its parent pointer and g(v) set to correct values)
   - Look at all the unseen neighbours w of v, and for each do the following:
     - Set g(w) = c(v,w) + g(v)
     - Set f(w) = g(w) + h(w)
     - Set parent(w) = v
     - Put w into PQ
   - Look at all neighbours w' of v that are in Q, and for each of them do:
     - If g(w') > g(v) + c(v, w') then;
       - set g(w) = c(v,w) + g(v)
       - set parent(w)= v

Note that we (as in "Dijkstra") do not look at neighbours of v that are tree nodes.

5.6.6 Proof that strong A*-search works

Proof of proposition 23.3.2: We must use induction

- Induction hypothesis: Proposition 23.3.2 is true for all nodes w that are moved from Q into the tree before v. That is, g(w) and parent(w) is correct for all such w.
- Induction step: We have to show that after v is moved to the tree, this is also true for the node v (and none of the old tree nodes are cahanged)

Def: Let generally g*(u) be the length of the shortest path from the start node to a node u. We want to show that g(v) = g*(v) after v is moved from Q to the tree. We look at the situation exactly when v is moved from Q to the tree, and look at the node sequence P from the start node s to v, following the parent pointers from v.

$$s = v0, v1, v2, \ldots, vj = v$$

We now assume that v_0, v_1,..., $v_{k,,}$ but not v_{k+1}, where k+1 < j, has become tree nodes when v is removed from Q, so that v_{k+1} is in Q when v is removed from Q. We shall show that this cannot be the case.

5.6.6.1 Illustrating the proof for A*-search

The value we use for choosing the best next node is now: f(v) = g(v) + h(v). NB: We assume that the heuristic function h is monotone. The important point is to show that there cannot be a shorter path to v that pass through another node outside the tree, e.g. trought u.

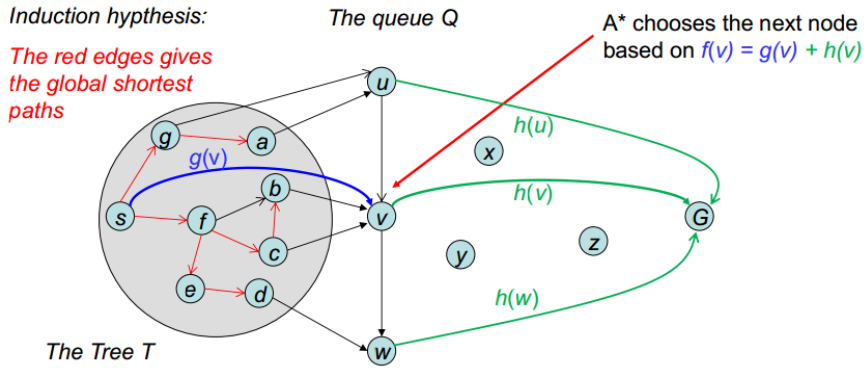


Figure 68:

From the monotonicity we know that (for i = 0, 1, . . . , j-1)

$$g*(v_i) + h(v_i) \leq g*(v_i) + c(v_i, v_{i+1}) + h(v_{i+1})$$

Since the edge (v_i, v_{i+1}) is part of the shortest path to v_{i+1}, we have:

$$g*(v_{i+1}) = c(v_i, v_{i+1}) + g*(v_i)$$

From these two together, we get:

$$g*(v_i) + h(v_i) \leq g*(v_{i+1}) + h(v_{i+1})$$

By letting i be k+1, k+2, . . . , j-1 respectively, we get

$$g*(v_{k+1}) + h(v_{k+1}) \leq g*(v_j) + h(v_j) = g*(v) + h(v)$$

From the induction hypotheses we know that g(v_k) = g*(v_k), (by looking at the actions done when v_k was taken out of Q) g(v_k+1) = g*(v_k+1), even if it occurs in Q. We thus know:

$$f(v_{k+1}) = g(v_{k+1}) + h(v_{k+1}) = g*(v_{k+1}) + h(v_{k+1}) \leq g*(v) + h(v) \leq g(v) + h(v) = f(v)$$

Here, all '$\leq$' must be equalities, otherwise f(v_{k+1}) < f(v), and then v would not have been taken out of Q before v_{k+1}. Therefore g*(v)+h(v) = g(v)+h(v) and thus g*(v) = g(v).

American highways. Shortest path Cincinatti-Houston.

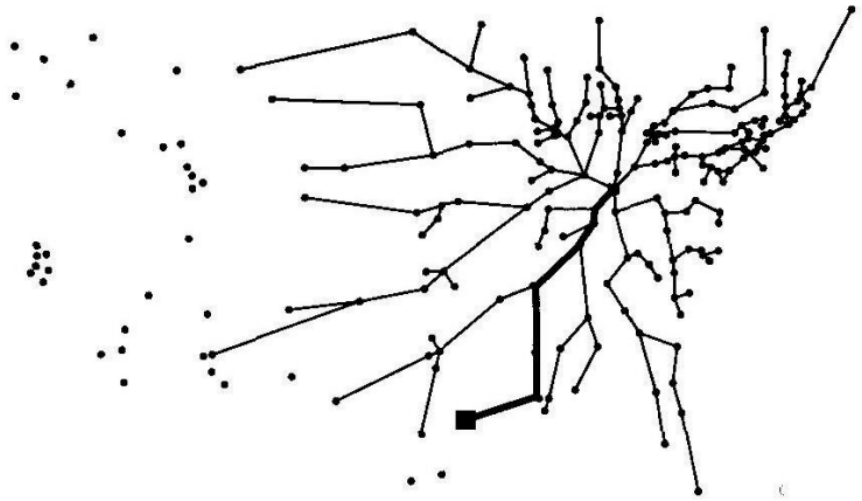


Figure 69: The tree generated by Dijkstra's algorithm (stops in Houston)

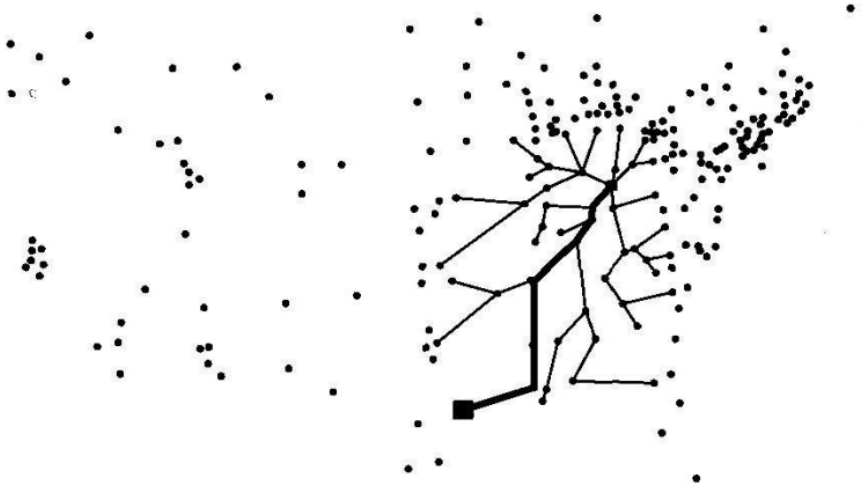


Figure 70: The tree generated by the strong A*-algorithm with the monotone h(v): h(v) = the "geographical" distance from v to the goal-node

6 Game trees and strategies for two-player games

When two players are playing against each other, things get very different. What is good for one player is bad for the other. The trees of possible plays are often enormous. For chess it is estimated to have 10100 nodes (and can therefore never (?) be searched exhaustively!). We only look at zero-sum games:

- The quality of a situation is represented by numbers.
- The sum of A's evaluation and B's evaluation of a situation is always zero.
- Then: What one player gains in a move is lost by the other.

6.1 Example: Tic-tac-toe and game trees

The board has 3x3 squares.

The game: Alternately do the moves:

- A chooses an unused square and writes 'x' in it,
- B does the same, but writes 'o'.

When a player has three-in-arow, he/she has won.

Player A (always) starts

- And we will here do all our considerations from A's point of view.
- We use numbers for node quality.
- High numbers are good for A and small numbers are good for B.

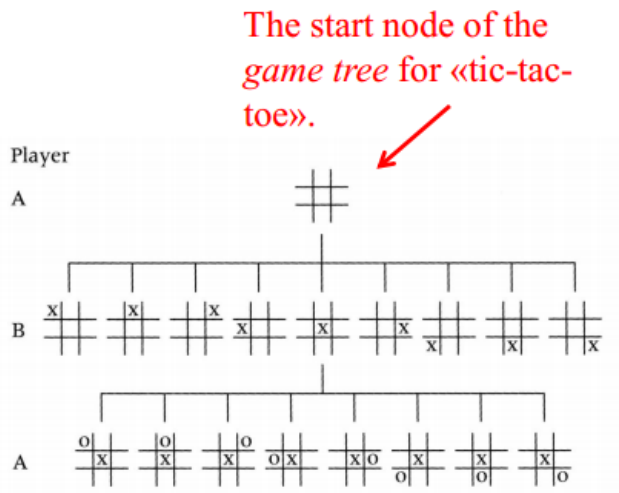


Figure 71:

9 nodes 9*8 = 72 nodes 9*8*7 = 504 nodes 9*8*7*6 = 3024 nodes 9*8*7*6*5 = 15120 nodes . . . 9*8*7*6*5*4*3*2*1 = 362 880 nodes	By searching depth-first in this tree, you never need to store more than 9

nodes, but it will take some time to go through all 362 880 nodes (and for "interesting" games there are usually a lot more!). In some games you can gain a lot by reconizing equal nodes, and not repeat the analysis for these (this is somewhat like dynamic programming). This usually requires a lot of memory (but above simple game we never need more than 1680 nodes).

6.1.1 Representing symmetric solutions by one node

(also usable for "one player games")

One can also gain a lot by looking at symmetries:

- Represent positions that are symmetries of each other only once.
- "Symmetry" means: All further positions will also be symmetric.
- Tic-tac-toe: Symmetric solutions will always be at the same depth, but this is not generally the case!

Using this will often reduce the memory needs quite a lot! But in e.g. chess there are not so many symmetries to utilize.

6.2 Zero-sum games

We will look for a strategy (if one exists) so that A is sure to win. A strategy for A is a rule telling A what to do in all possible A-situations. Fully analyzable games: Three possibilities for each A-situation S:

1. A has a winning strategy from S (+1).
2. Whatever A does from S, B has a winning strategy from the new situation (-1).
3. If A and B both play perfectly, it will end in a tie.

This is represented by 0, and can occur only for some games. The game tic-tac-toe ends in a tie if both players play perfectly.

6.3 Example: The game Nim

The game Nim:

- We start with two (or more) piles of sticks.
- Number of sticks: m and n.
- One player can take any number of sticks from one pile, but at least 1.
- The player taking the last stick has lost.

Nim will never end in a tie. With m = 3 and n = 2, the full game tree (utilizing some symmetries) is shown to below. The value seen from A is indicated for the final situations (leaf nodes).

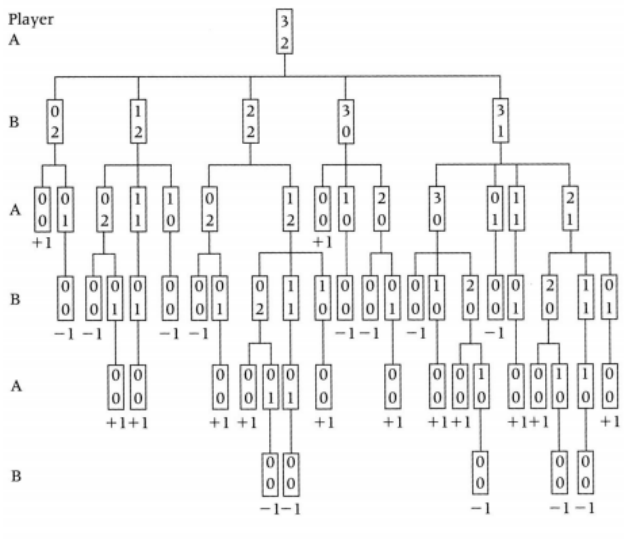


Figure 72:

How can we find a strategy so that A wins?
Or prove that no such strategy exists.

A wants to find an optimal move. We must assume that also B will do optimal moves seen from its point of view. Since the values are as seen from A, B will move to the subnode with smallest value.

Min-Max Strategy:
To compute the value of a node, we have to know the values of all the subnodes. This can be done by a depth first search, computing node values during the withdrawal (postfix).

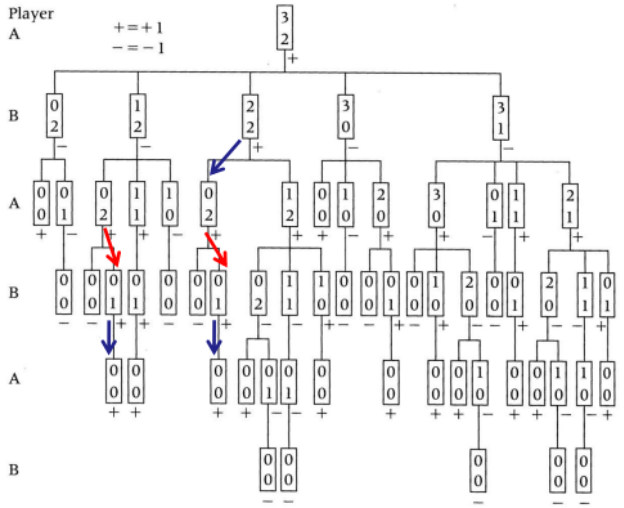


Figure 73:

Strategy for A: If possible, move to a node with value +1. Otherwise make a random move.
Strategy for B: If possible, move to a node with value -1. Otherwise make a random move.

6.3.1 The Min-Max-Algorithm in action

With simple alpha-beta cutoff (pruning)

This is done by a deph first traversal of the game tree, computing values on withdrawal (Min-Max Strategy). The result of this is given in the figure below. Possible optimalization:
- From the start-position S, assume that A has looked at three of its subtrees (from the left). A has then found a winning node U (marked +1), and the value of V and W then does not matter.
- This is a simple version of alfa-beta cutoff (pruning).

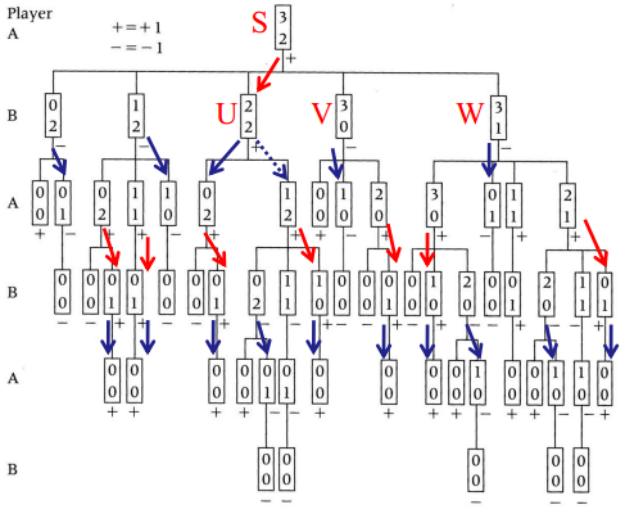


Figure 74:

Usually, the game tree is too large to traverse.

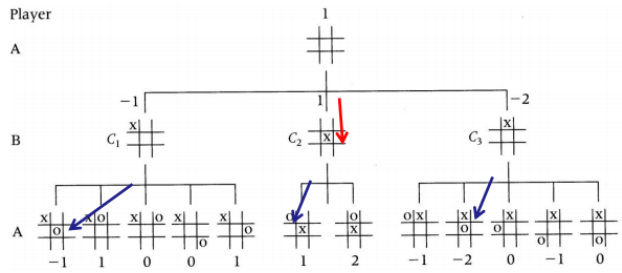


Figure 75:

One then usually searches to a certain depth, and then estimate (with some heuristic function) how good the situation is for A at all nodes at that depth (and we then don't use only -1, 0 and +1). In the figure above we go to depth 3 (or 2, depending on how you define depth). The heuristic function above is: Number of "winning lines for A" minus the same for B (this is given below each leaf nodes). The best move for A from the start position is therefore (according to this heuristic) to C_2.

However, this heuristic is not good later in the game. It does not take into account that winning is better than any heuristic. We therefore, in addition, give winning nodes the value $+\infty$ = 9. This will give quite a good strategy. But as said above: tic-tac-toe will end in a tie if both players play perfectly. We have to add that the tie-situation (e.g. the one to the right) gets the value 0. Thus, if we fully analyze the game, the value of the root node will be 0. NB: The difficult choice is between searching very deep or using a good but time consuming heuristic.

6.4 Alfa-beta cutoff (pruning)

This technique is only usable for two-player games!

But we should also use alpha-beta cutoff. Which is specific to two-player games. Intuitively this goes as follows (assuming it is A's move):

- A will consider all the possible moves from the current situation, one after the other
- A has already seen a move in which it can obtain the value u (after C_1 and C_2, u = 1)
- A looks at the next potantial move, which leads to situation C_3
- However, A then observes that from C_3, B has a very good move (= bad for A) that seen from A has the value v = -1. Then the value of C_3 cannot be better than -1.
- As v < u, player A has no interest in looking for even better moves for B from situation C_3. A already knows that it has a better move than to C_3, which is to C_2.

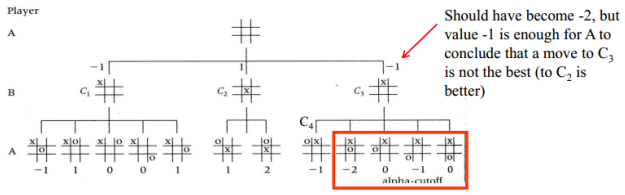


Figure 76:

6.4.1 Examples

When A considers the next move:
- Cutoffs from A-situations is called alpha-cutoffs.
- Corresponding cutoffs from B-situations are called beta-cutoffs.

The figure below shows alpha- and beta-cutoffs at different stages of a DF-search of a game tree. When implementing alpha-beta-cutoffs during a DF-search, it is usual to switch viewpoints between the levels. Then we can always maximize the value. But we have to negate all values for each new level.

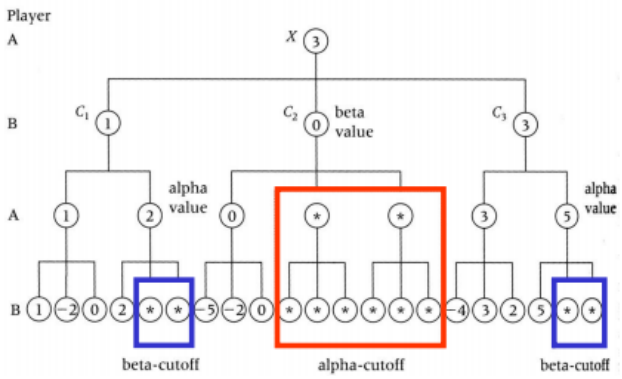


Figure 77:

6.4.2 Alpha-beta-search (negating the values for each level)

```
real function ABNodeValue (
    X,         // The node we compute alpha/beta value for. Children: C[1],C[2]… C[k]
    numLev,    // Number of levels left
    parentVal,// The alpha-beta-value from the parent node (-LB from the parent)
               // Returned value: The final alpha/beta-value for the node X
{
    real LB;   //  Current Lower Bound for the alpha/beta value of this node (X)

    if <X is a terminal node> or numLev = 0 then {
        return <An estimate of the quality of the situation (the heuristic)>;
    } else {
        LB := -ABNodeValue(C[1], NumLev-1, ∞);
        for i :=  2 to k do {
            if  LB >= parentValue  then {
                return LB;                    // Cutoff, no further calculation
            }
            else {
                LB := max(LB, -ABNodeValue(C[i], Numlev-1, -LB) );
            }
        }
    }
    return LB;
}
```

Start the recursive call to calculate value for the (current) rootnode (down to depth 10) by calling
`ABNodeValue(rootnode, 10, -∞)`

Figure 78:

7 Matching

7.1 Matching in undirected bipartite graphs

Bipartite graph = The set of nodes can be partitioned into two sets X and Y, so that each edge has one end in X and the other in Y. It is the same as a two-colorable graph or a graph without odd loops:

- The node set X, e.g. workers in a workshop

- The node set Y, e.g, the jobs of the day

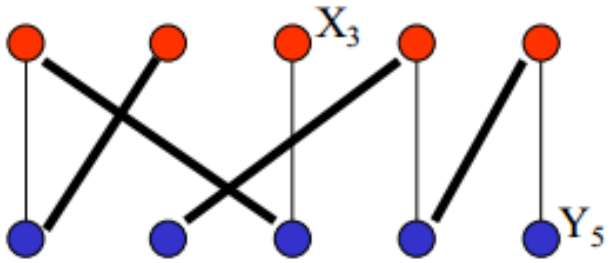


Figure 79: not able to find a "perfect matching", and thus all jobs cannot be done that day.

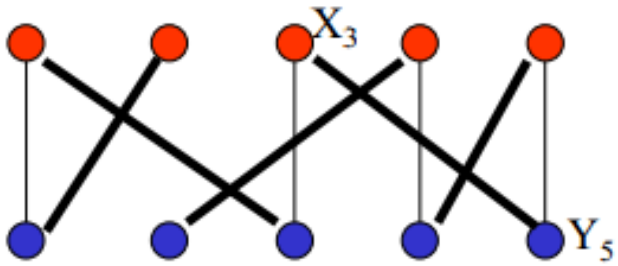


Figure 80: if we add the edge X_3-Y_5 we are suddenly able to find a "perfect matching", so that all jobs can be done.

Some variations over the same theme:

- We might have |X| $\neq$ |Y|, and then we can obviously have no perfect matching
- Even if there is no perfect matching, we are often interested in finding a match that is as large as possible.
- We can have "weights" on the edges, and ask for the matching with max. sum of weights

7.1.1 Hall's Theorem

When can we find a perfect matching?

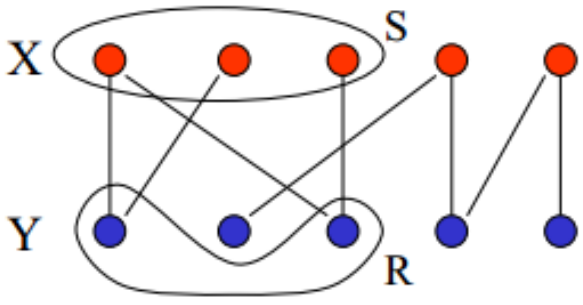


Figure 81: A subset S of X is only connected to R in Y, and R has fewer nodes than S

Hall's Theorem: There is a perfect matching if and only if there is no subset S so that R = Γ(S) has fewer nodes than S. If there is such an S so that R = Γ(S) is smaller than S, there is obviously no perfect matching. We are not able to match each node in S with separate nodes in R.

7.1.2 Naive "greedy algorithm"

Instace: Given a bipartite graph.
Question: Find, if possible, a perfect matching.

We could try a simple greedy approach, which could go as follows: Look repeatedly at the edges of the graph, and include an edge in the matching if it has no node in common with an already included edge. The greedy strategy is not working here.

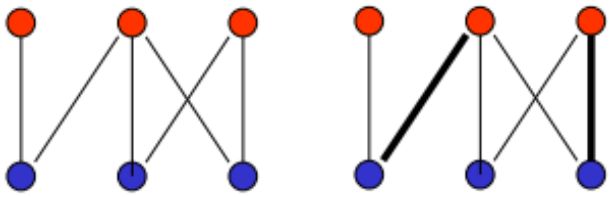


Figure 82:

Example: Given the bipartite graph to the left. A greedy approach may, after two steps, give the matching to the right. However, there exists a matching with three edges (lower right), but we cannot use a simple greedy scheme to extend the left matching to the one with three edges.

A problem where the greedy algorithm works: In connection with greedy algoritms the elements often also have a weight, A place where such a greedy algorithm really works is if we want the heaviest span tree in an undirected graph, where the edges have weights. Algorithm: "Look at the edges in order of decreasing weight, and include those that do not make a loop with those already included (Kruskal's algorithm)."

7.1.3 Hungarien algorithm

We assume here that |X| = |Y|

With the simple greedy strategy we only looked for "fully independant" edges, when we wanted to increase the size of the current matching M. This was obviously too simple, but it turns out that if we instead look for "M-augumenting paths", and each time pick the best, the algorithm will work. This can be shown directly, but we'll do it so that we also show Halls Theorem. An M-augumenting path must first of all be an "M-alternating path", which is a (simple) path where alternating edges are in M and not in M. In addition both end-nodes of the path must be "unmatched" (and then one endnode will be in X and the other in Y).

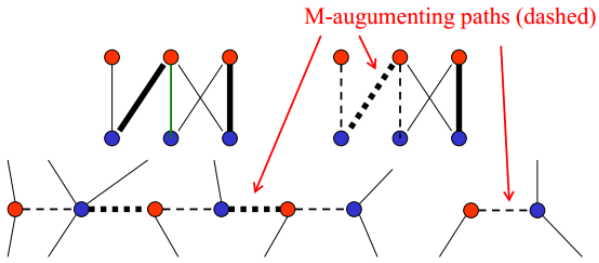


Figure 83:

We can "use" an augumenting path to obtain a larger matching. If we have found an augumenting path, we can obviously "use this" to find a matching which is one larger (written M $\otimes$ P). The results are shown for the dashed M-augumenting paths below:

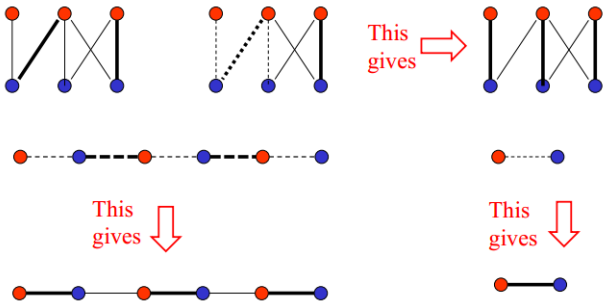


Figure 84:

7.1.3.1 Finding a possible augumenting path

The Hungarien algorithm goes as follows:

- Start with an empty matching
- Search for an augumenting path
- If you find one, use this to find a matching with one more edge

epeat this until either you have a perfect matching or you cannot find an augumenting path relative to the current M. In the last case, the sitiation will show us a subset S in X where R = Γ(S) (those in Y connected to S) is smaller than S. Thus, when the algorithm stops, we can show that there is no perfect matching.

The search for an augumenting path is done as follows:

- Choose an unmatched node 'r' in X. This node will be the root in a tree where all paths out from the root are alternating paths.
- We then grow the tree by adding two and two edges until we we have found an augumenting path, or we cannot grow the tree any further by using legal steps.

Growing a tree to find an augumenting path

We assume that we have a matching M that is not perfect, and we will search for an augumenting path. To try to find a larger M we will build an alternating tree T, At the start the tree will consist only of a root node 'r' in X which must be unmatched (and such a node can always be found when M is not perfect and |X| = |Y|). Building the alternating tree is done by repeating the following step. We search for an unmatched edge U outside T, between a red node x in T and another node y (which then must be blue). If we find such an edge, there are three cases:

1. The node y is already in T: Then do nothing
2. The node y is unmatched. We have then found an augumenting path, and we can use that to find a larger M
3. The node y is a matched node in Y. We then include in T the chosen edge (x,y) from T, and the edge adjacent to y in the matching. The tree T will then be extended by two edges/nodes.

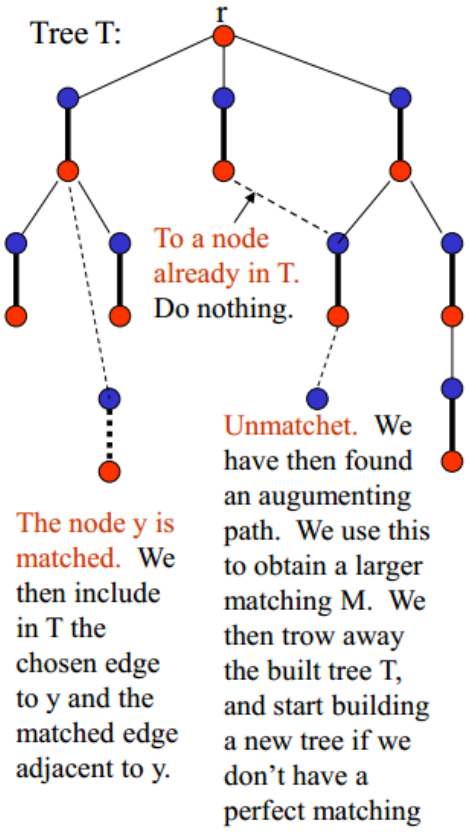


Figure 85:

The tree looks nice and clean. Note that only the the edges of the tree, and a few potential new ones, are drawn. There may be a number of other nodes. But the tree can obviously also be drawn inside the bipartite graph. Then it will look as shown below (where all nodes, but not all edges, is drawn), but it is easier to get an overview in a tree structure.

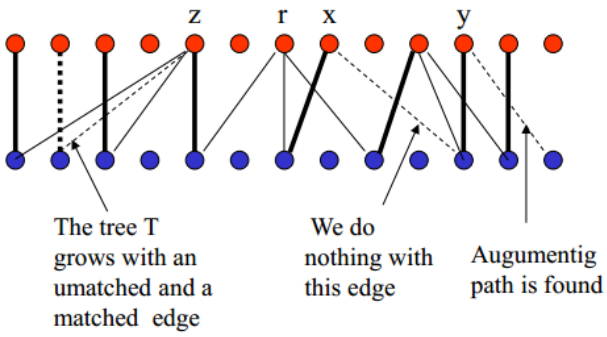


Figure 86:

7.1.4 Example

7.1.4.1 No perfect matching

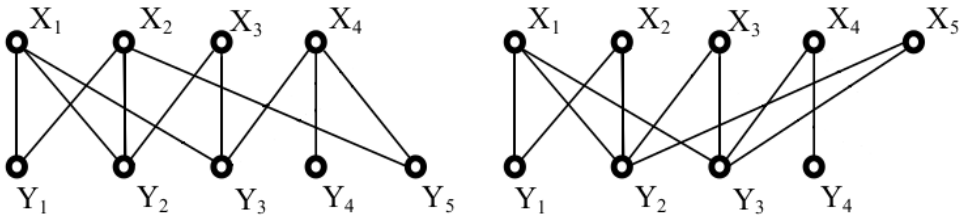


Figure 87: |X| $\neq$ |Y|

7.1.4.2 No perfect matching (Hall's theorem)

S = $\{Y_2, Y_3\}$
R = Γ(S) (*X-nodes that only have edges to S*)

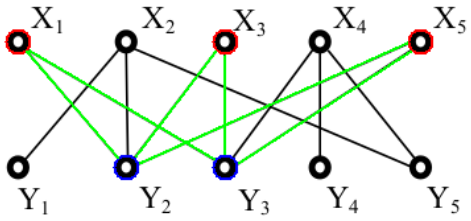


Figure 88: |R| < |S| $\rightarrow$ no perfect matching

7.1.4.3 Perfect matching

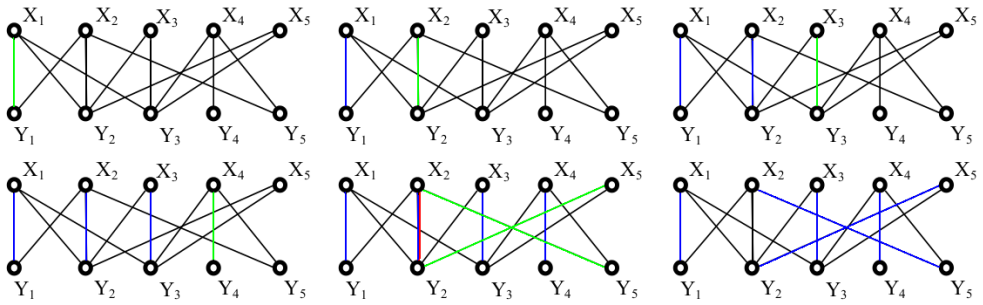


Figure 89: Perfect matching

7.1.5 Variants of the matching problems

- Find a matching with as many edges as possible (and then X and Y don't have to be of the same size)
- Given "weights" on the edges: Find a perfect matching with as high weight as possible.
- Flow in "networks", where the matching for bipartite graphs occurs as a special case.
- Generalizing to graphs that are not bipartite • Will be discussed at next slides

7.2 Matching in graphs that are not bipartite

Figure 90:

Generalization of matchings beyond bipartite graphs. Pose the same questions for general graphs:

- Find a perfect matching (or show that no one can be found)
- Find a maching with as many edges as possible
- With weights on the edges: Find a (perfect?) matching with the largest possible weight

All these can be solved in polynomical time. The algorithm for largest matching in general graphs is sligthly more complicated to describe, but it is considerably vorse to prove that it is correct. The algorithm is a generalization of that for the bipartite case.

7.2.1 Extended Hungarien Algorithm

New elements in the algorithm:

- There should be no node colors at the outset
- Each tree building starts with unmatched node. We color it red, and it will be the root of the new tree
- When the graph is not bipartite, there can be edges in the tree from from red to red nodes, like the edge (u,v) in the figure. This will form an odd loop with the rest of the tree. This loop is treated by simply collapsing it (including its internal edges) to one red node. If it stops without finding an augumenting path, start with another unmatched node as root.

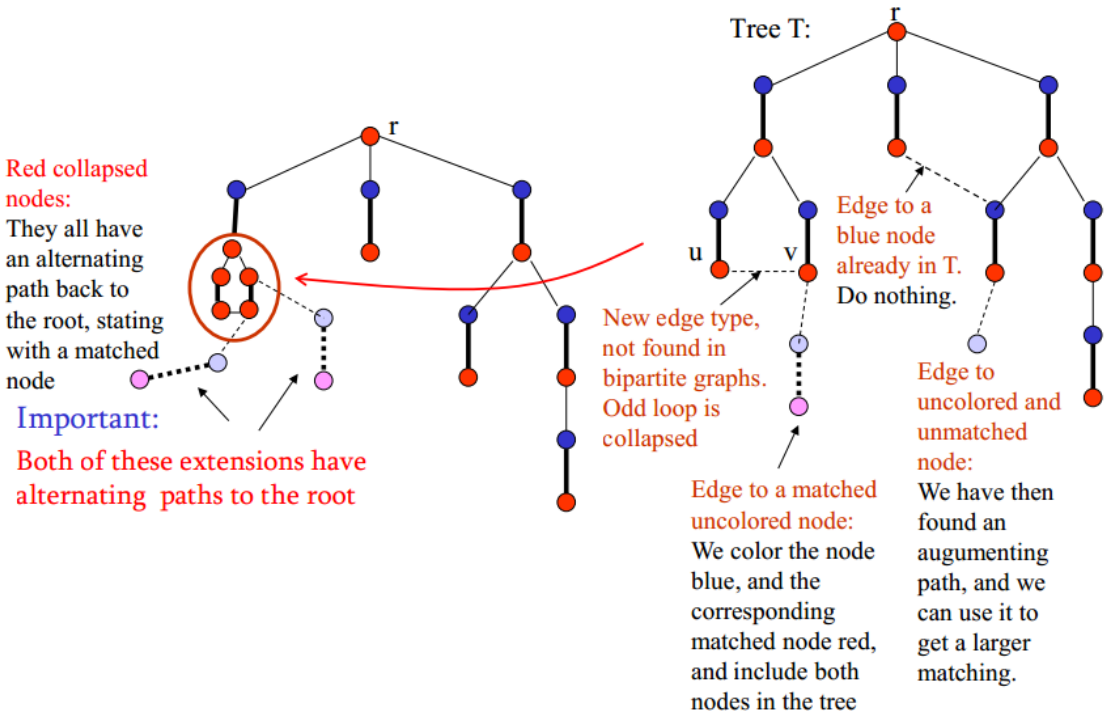


Figure 91:

If you find an augementing path, we then go backwards along the alternating path, and along the way we unpack the collapsed nodes, and find the alternating path througt them. We thereby get an alternating path in the original graph back to the root. We can use this to find a matching that is one edge larger than the one we have.

Otherwise the treebuilding stops because there are no more unmatched nodes, and no edge from a red node to a node that is uncolored and unmatched. Then no larger matching exist.

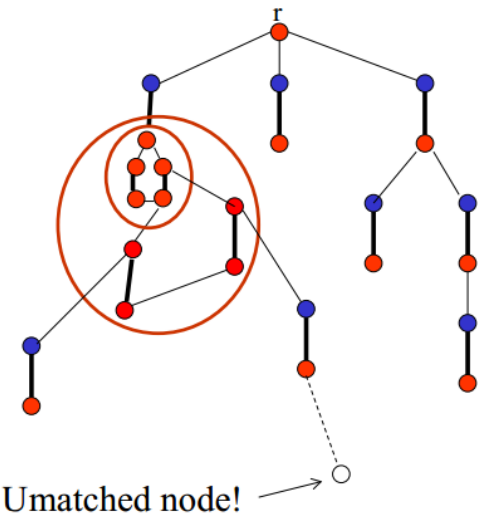


Figure 92:

8 Flow in Networks

The use of the word "Network" is simply a tradition in this area. It is the same as directed graphs, usually with some weight, capacity etc. for each edge. There are a lot of practical problems that can be seen as flow problems in networks.

- Data nets, where there is a flow of data packages through the edges.
- Different types of pipe-networks where fluid or gas can flow, and where each pipe has a capacity.
- Networks of roads with different capacities, where cars are «flowing» on the roads.

The networks we shall study here have:

- A capacity on each of the edges
- One source node s og one sink node t
- And the goal is usually to find a largest possible flow from s to t

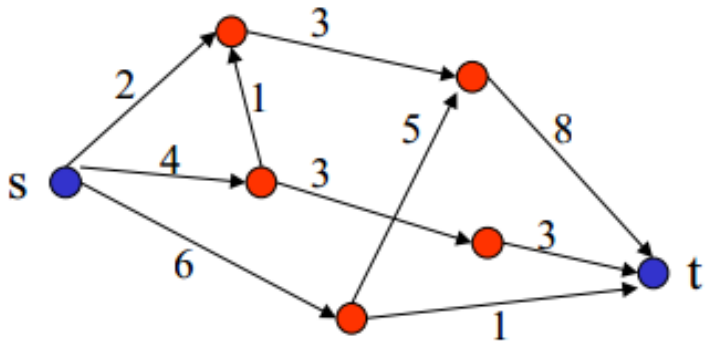


Figure 93:

A flow f in such a network is composed of a flow f(e) $\geq$ 0 for each edge e, with the follow properties:

- **Flow conservation**: For each node, except for s and t, the sum of flow into the node is equal to the sum of the flow out of the node (where into and out of is defined according to the directions of the edges).
- **In networks with capacities**: Each edge has a capacity c(e) $\geq$ 0, and the flow f(e) must be between 0 and c(e).

We assume there are no edges leading into s or out of t. val(f) is by definition the sum of the flow out of s. Lemma: The sum of the flow into t is the same as val(f). Can be proved by summation of the flow out of all nodes.

Our goal:
Given a network with capacities, we want to find edgeflows f(e) that

- Satesfy the capacity requirement 0 $\leq$ f(e) $\leq$ c(e)
- Forms a maximum flow (there are no legal flow with larger val(f))

The example below is a network with given capacities. We can easily see: Maksimum flow is 7.

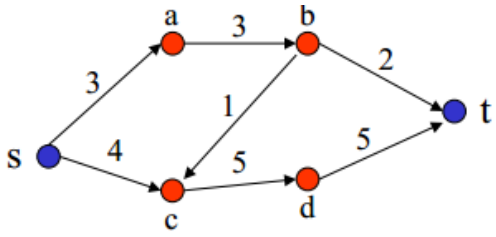


Figure 94:

8.1 Naive (greedy) algorithm

The naive greedy algorithm (that in fact don't work!) would be as follows:

Repeat this step until no paths can be found:

- Find a directed, simple path from s to t where all the current f(e) are positive and smaller than c(e)
- Increase the flow along this path as much as possible (dictated by the edge that has the smallest c(e) – f(e) along the path)

In the figures below the capacity is given above the edges (all c(e) =1) and the current flow is given below the edge (initially zero everywhere). We first find a simple flowincreasing path, e.g. s-a-b-c-d-t. We can increase each edgeflow along this path with 1, and get the situation to the right. Now val(f)=1. But this is not a maximum flow, as we can easily find a flow with val(f)=2. BUT, there is no flowincreasing path in the right network that can bring us to a flow with value 2. Thus this simple scheme won't bring us to a maximum flow.

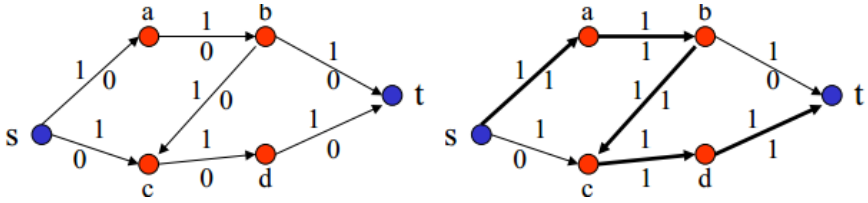
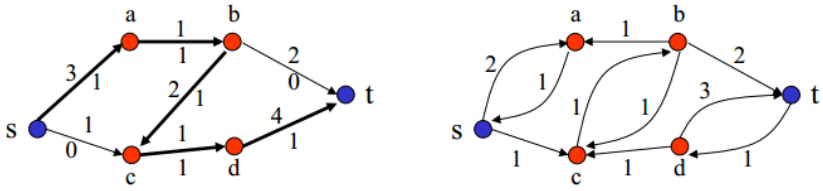


Figure 95:

8.2 The f-derived network N(f)

What we haven't taken into account earlier, is that we, while searching for a larger matching, also can decrease the flow for edges with nonzero flow. And by utilizing this, we in fact get a working algorithm! To get an overview of the ways we can change the current flow on each edge we can set up the f-derived network referred to as N_f, Nf, or N(f). We will here use N(f).



A network with capacities (above the edge) and a flow (under)

The *f*-derived network, of the situation to the left

Figure 96:

8.2.1 f-improvement of paths

We the search for paths from s to t in the f-derived network N(f)

- Such paths are called f-augumenting paths
- The search can be done e.g. bredth-first or depth-first in N(f) from s.
- We can e.g choose the path F = s-c-b-t. The maximal flow-increase along this path is h, which here is 1 (minimum change possible over the path edges)

We then perform the corresponding flow change, by increasing the flow with h for the edges where their direction is the same as in F. Reduce the flow with h where edge direction is opposite to that of F. This gives the new flow;

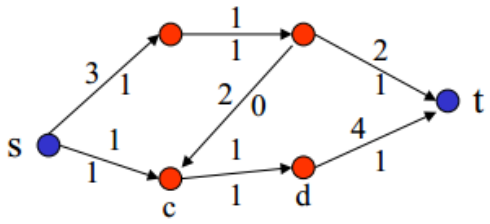


Figure 97:

We then forget the old f-derived network, and build a new one relative to the new flow.

8.3 Cuts in networks

A cut in a network is simply a devision of the set of nodes into two sets X and Y, where s is in X and t is in Y. The capacity of a cut K=(X,Y), written cap(K), is the sum of the capacities of all edges leading from a node in X to a node in Y. In the figure, the capacity of the cut is 3 + 7 = 10. Thus, the edges from Y to X do not influence the capacity of the cut.

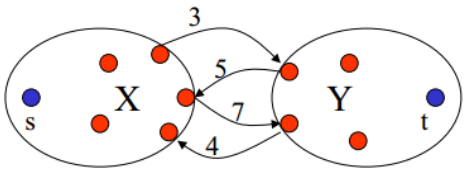


Figure 98:

Lemma: Given a legal flow f and a cut K = (X,Y) $\rightarrow$ val(f) $\leq$ cap(K).

This can be shown as follows:

- By adding together the flow in/out of all nodes in X' = X-s, we find that
  (flow out of s) + (flow backwards over K) = (flow forwards over K)
- This means (as (flow out of s) = val(f)):
  val(f) = (flow forward over K) - (flow backwards over K)
- The right hand side of the above equality is called the flow over K, and (as all flows are positve) we know that it will not exceed cap(K)
- Thus, we know, for any K: val(f) $\leq$ cap(K).
- In the figure below: 5 = 2 + 6 - 0 - 3 $\leq$ 3 + 7

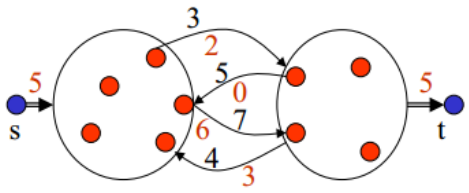


Figure 99:

This gives us a way to decide whether a given flow is optimal. If we have a flow f and a cut K so that val(f) = cap(K). then we have a maximum flow, and there is no cut with smaller capacity!

8.4 Ford-Fulkerson

The FordFulkerson-algorithm goes as follows:

- Start with zero flow (which is always a legal flow)
- The main step (and at the start of this we generally have any legal flow):
  - Find the f-derived network N(f) (that shows all possible changes for the edgesflows)
  - Find, if possible, an f-augumenting path from s to t, and find the maximum increase it allows (before any of the edgeflows exceed the capacity or will go under zero).
  - Do the changes that this f-augumenting path indicate
  - Repeat this step until we can no longer find an f-augumenting path in N(f)
- The algorithm stops when there are no directed path from s to t in N(f).
- A proof showing that we now have a maximum flow, is that we can now show a cut with capasity equal to the current flow. Thus, there can be no larger flow!

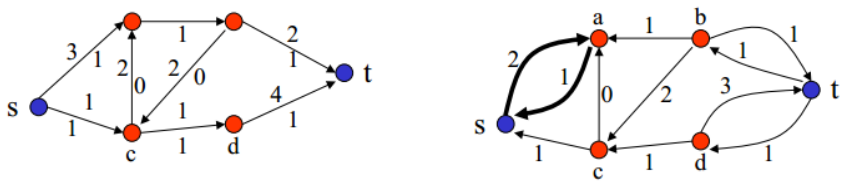


Figure 100:

8.4.1 Termiation

It stops when there is no connection from s to t in N(f). As indicated: To show that we now have a maximum flow, we will show that we can construct a cut K with capacity equal to the current flow. That is: cap(K)=val(f). It turns out that such a cut is easy to find: Let X be the set of nodes reachable from s in N(f), and let Y be the rest of the nodes (including t). As no edges in N(f) is leading from X to Y, we know by the def. of N(f):

- All edges in N (the original network) from X to Y are used to its full capacity.
- All edges in N leading from Y to X have flow f = 0

From the definition of cap(K) we see that the current flow over K is val(f). Thus, we know we have a maximum flow, and we have proven the following Theorem:

Theorem (Max-flow, min-cut): In a network with capacities we can find a flow f and a cut K so that val(f)=cap(K). Then we know that we have a maximum flow, and that no cut has lower capacity.

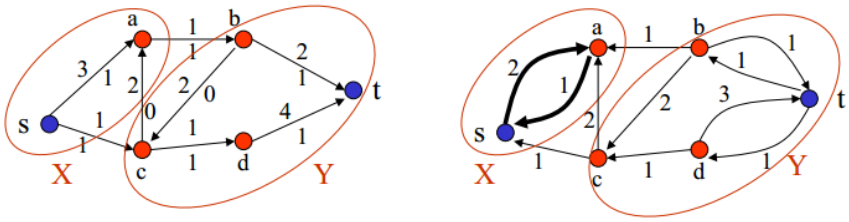


Figure 101:

8.4.2 Variations of Ford-Fulkerson

The Ford-Fulkerson algorithm says nothing about which f-augumenting path should be chosen in each step, if there are more than one. If we do not decide anything about the choice of f-augumenting paths, we know:

- If all capacities are (positive) intergers, then the number of steps can be at least at large as the size of the largest capacity. Example:

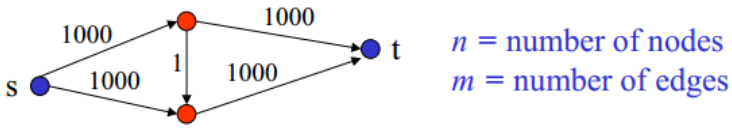


Figure 102:

- If the capacities are real numbers, the algorithm can in theory loop forever.

Proposal 1: All the time, choose the f-augumenting path that gives that largest possible incremet in the flow. This one can be found by an algorithm similar to a shortest path algorithm). This gives a worst-case-time: O(m log(n) log(max-flow))

Proposal 2 (Edmonds-Karp): All the time, choose the f-augumenting path that has the smallest number of edges (can be found by a bredth-first search). This gives a worst-case-time: O(n m^2) (and this independant of the max. flow, which is very convenient).

8.5 Variations of the problem of max. flow

First of all, there are alternatives to the Ford-Fulkerson algorithm

- Dinac has designed an algorithm
- Goldberg and Tarjan (preflow push algorithm)

We may also have a minimal flow for each edge

- Then it is an interesting problem just to find a possible flow
- But after that you can proceed as in Ford-Fulkerson

We may also have a price on each edge, saying, how much a flow of one will cost over this edge.

- For this problem there is a well known algorithm: The Out-of-kilter algorithm, which is also polynomial.

We can also have multiple sources and/or multiple sinks, with different requirements to the flow in and out of these.

We may also have different "commodities" that should flow in the network (cars, busses, trucks, ... in a street network), and the edges may have a different capacity an cost for each commodity.

- This is a field of active research, in connection with e.g. traffic planning, routing in communication networks, etc.

9 A connection between flow in networks and matching

A simple but important lemma, which is obvious from the Ford-Fulkerson algorihtm:

1. If we have interger will always find an interger max. flow
2. When all the capacities are 1, we can find a max. flow where all edgeflows are either 0 or 1. Such a flow can be seen as pointing out a subset of the edges (those with flow 1)

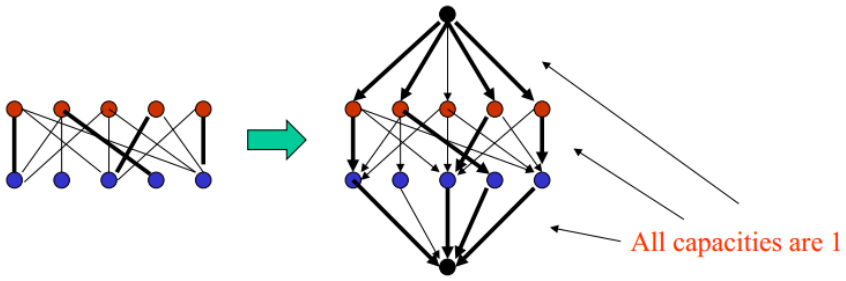


Figure 103:

10 Convex hull

Let P be a set of points in the plane (R^2) (Can also be defined for R^k, k > 2)

Definition: A set Q $\in R^2$ is convex if: for all q_1, $q_2 \in$ Q the line q_1q_2 is fully within Q.

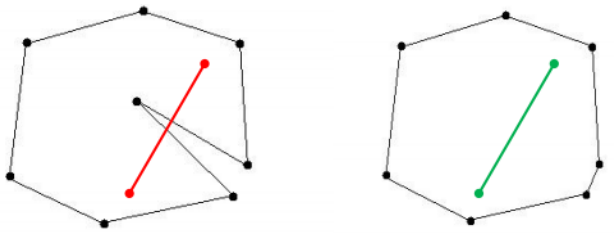


Figure 104:

Definition: The convex hull of a set of points P $\in$ R^2 is the smallest convex set Q that contains all the points of P.

10.1 Jarvis' March

Idea: One way to find the convex hull is to use a thin rope and wrap it around the points, step by step. To get a staring point, choose e.g. the point with the smallest x-value. This is always a corner of the convex hull.

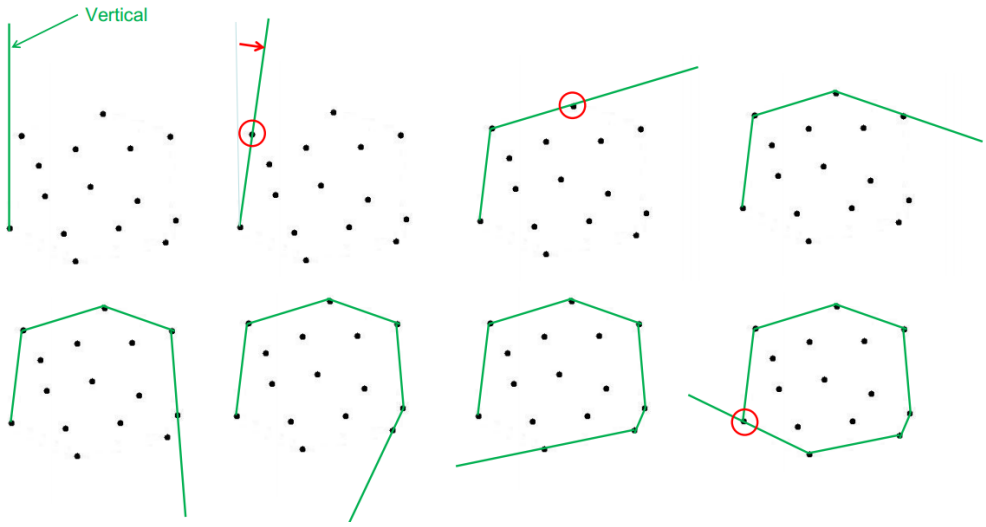


Figure 105:

In two dimensions worst case running time is: O(n^2).
In d dimensions the running time is: O($n^{(d/2)+1}$).

10.2 Divide-and-conquer

A faster method for finding the convex hull

The first step is to divide the set into two sets with the same number of nodes (+1 or -1 for odd numbers), using a vertical line. Divide the set at the median of the x-values. Repeat this for each of the sets until you have 1, 2, or 3 points in each set. Then divide each of the smaller sets in the same way. The depth of the "division tree" will be no larger than $\log_2 n$. Solve lowest level: Find the convex hull for 2 or 3 points (easy). We merge together two and two convex hulls, following the structure of the division tree. We merge two hulls into one by finding the upper and lower "bridge" between them.

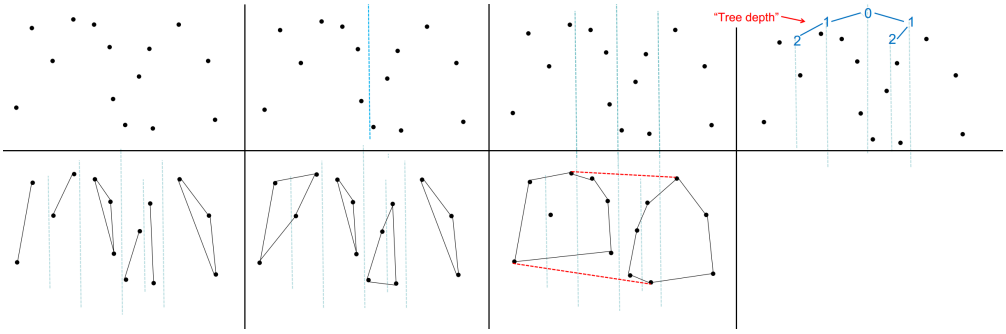


Figure 106:

10.2.1 Finding the upper bridge

We know that all x-values in the left set are smaller than those in the right set. Let p_1 be the rightmost corner of the left hull and q_1 the leftmost of the right. Number the corners of the left hull counterclockwise p_1, p_2, ..., and the corners of the right hull clockwise q_1, q_2, ...

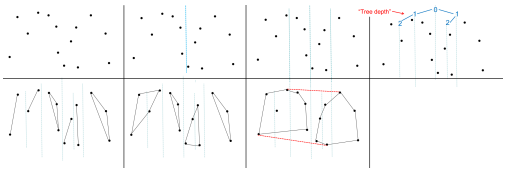


Figure 107:

We start by "crossing" the edge p_1-q_1, and moving to the next corner, q_2, of the right convex hull. To check if this was the upper bridge, we have to be able to decide whether three consecutive points represent a turn to the left or to the right. We can find whether three consecutive points represent a turn to the left or to the right by assuming that the three points are p_1=(x_1,y_1), p_2=(x_2,y_2), p_3=(x_3,y_3).

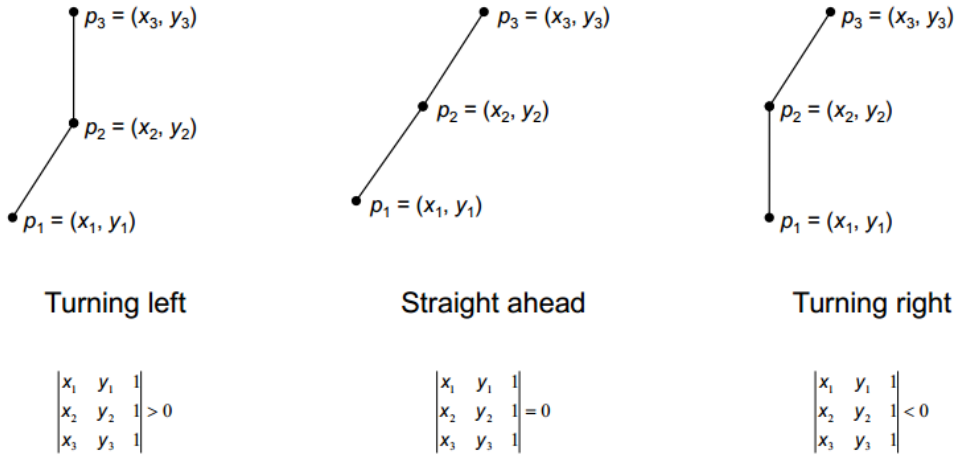


Figure 108: Determinant

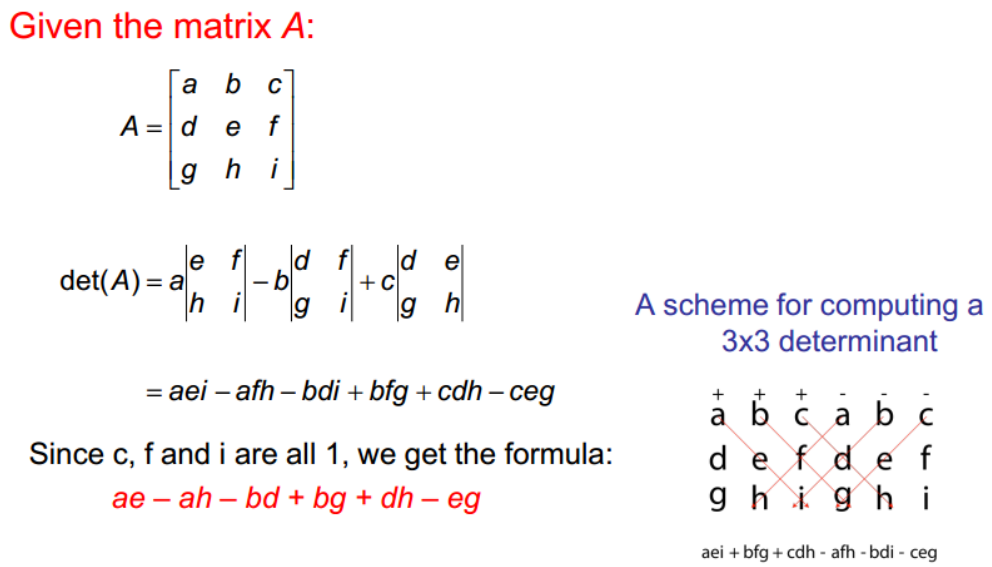


Figure 109: Compute 3x3 determinants

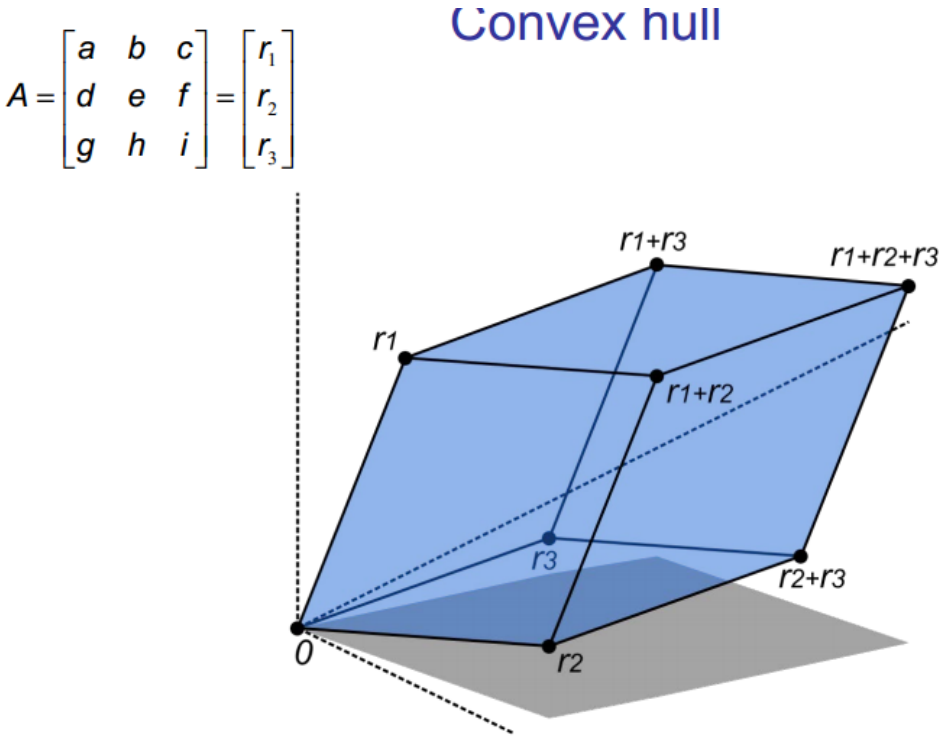


Figure 110:

The geometric interpretation of the determinant of matrix A (det(A), or simply |A|) is the volume of the parallelepiped spanned by the row vectors of A (Or the column vectors, it is the same volume).

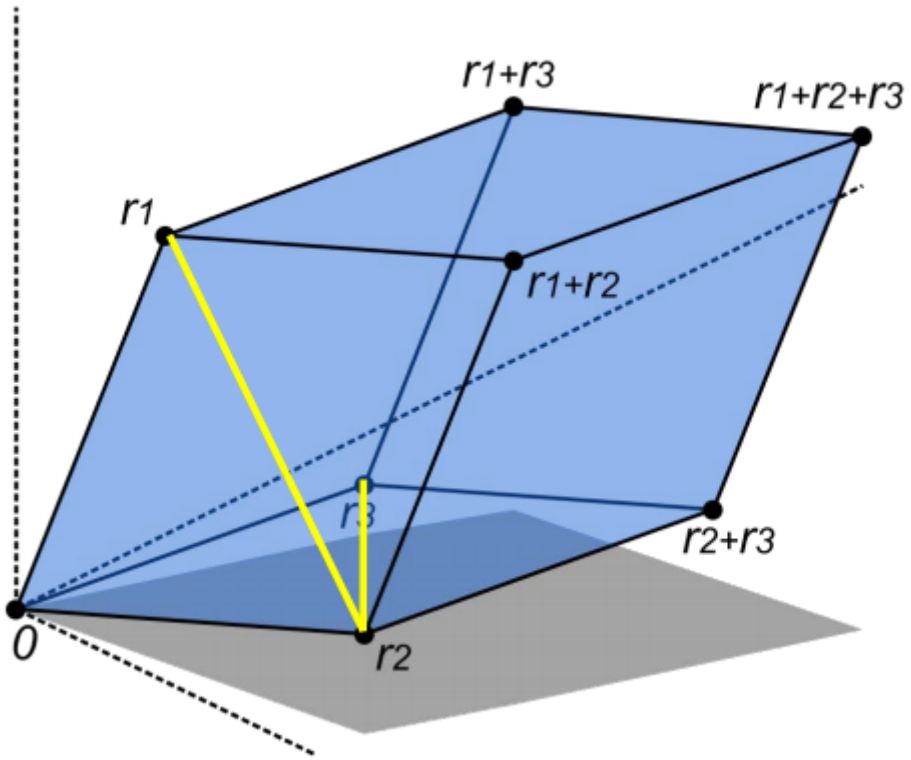


Figure 111:

10.2.2 Finding the lower bridge

The lower bridge is found the same way (upside down). Afterwards we need to remove some "old" corners (now interior points), and renumber the current corners. All points visited during the search for the bridges, except the endpoints of the bridges are no longer corners of the merged hull. We re-number remaining corners so that we get a continuous numbering of the corners around the merged hull.

10.2.3 Time complexity

We first sort the points according to their x-value. This takes time O(n log n), where n is the total number of points. We can then easily do the partitioning. Each time we merge two convex hulls, we may have to move the endpoints of the potential bridges m times, where m is the total number of nodes in the two merged sets. Thus, each merge takes time O(m). All merging at one tree-depth will therefore take time O(m_1) + O(m_2) + O(m_3) +. . . + O(m_k) which becomes O(n), since $m_1 + m_2 + m_3 + \ldots + m_k = n$. The number of tree-levels do not exceed $\log_2 n$, so the total running time is therefore: O(n log n).

11 Triangulation

The general problem is finding a triangulation of a given set of points in the plane. Below, a set of points are given (left), and a arbitrarily chosen triangulation of this set is drawn (right).

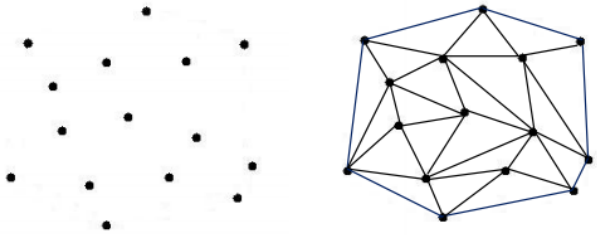


Figure 112:

When finding a triangulation, a polygon should also be given, where the corners are are points of the given point set, with the rest of the points inside the polygon (to define a boundary). We will assume that this polygon always is the convex hull of the set of points.

11.1 Constructing a triangulation

Assume that the outer (convex) polygon has n corners, and that there are m points inside (so that the total number of points is n+m).

- We can find a triangulation of the corners by choosing one of them, p, and draw an edge to the n-3 corners that do not already have an edge to p.
- Together with the outer edges this gives n+(n-3) = 2n-3 edges, and n-2 triangles.
- We then take each of the inner points, q, and do as follows: We find the triangle that q resides in, and draw edges from q to the three corners of this triangle. This gives three extra edges and two extra triangles for each of the m inner points
- We then get:
  Number of edges: 2n-3+3m = 2n+3m-3
  Number of triangles: n-2+2m = n+2m-2

Even if this is a special way to construct a triangulation, the answer (number of edges and triangles) holds for any triangulation, even if the outer polygon is not convex.

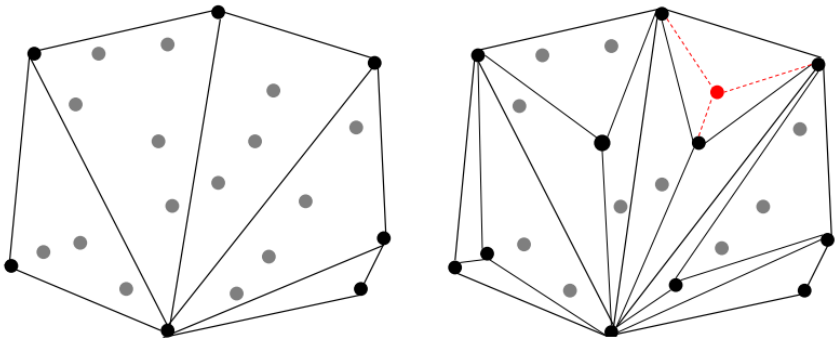


Figure 113: The new red node gives: three extra edges, two extra triangles

11.2 Different triangulations of the same set of points

We assume that the following special cases do not occur:

1. Four points lie on one circle (along the circumference),
2. all the points occurs on the same straight line.

A "good" triangulation? is usually one where each triangle is as close to an equilateral triangle (with all angles equal to 60° as possible. Two reasonable goals could be:
- To minimize the maximal angle: The largest angle is as small as possible (And notice: The largest angle in a triangle is always at least 60°)
- To maximize the minimal angle: The smallest angle (which is never larger than 60°) is as large as possible.

It turns out that max-of-min is easier to handle than min-of-max, so it's most commonly used.

11.3 Delaunay-triangulation (max-of-min)

Roughly speaking, a triangulation of a set of points is, a Delaunay triangulation if, over all triangulations, the smallest angle is as large as possible. Or more precisely: When the angles of a triangulation are sorted from smallest to largest, the Delaunay triangulation is the sequence with the highest lexicographic order.

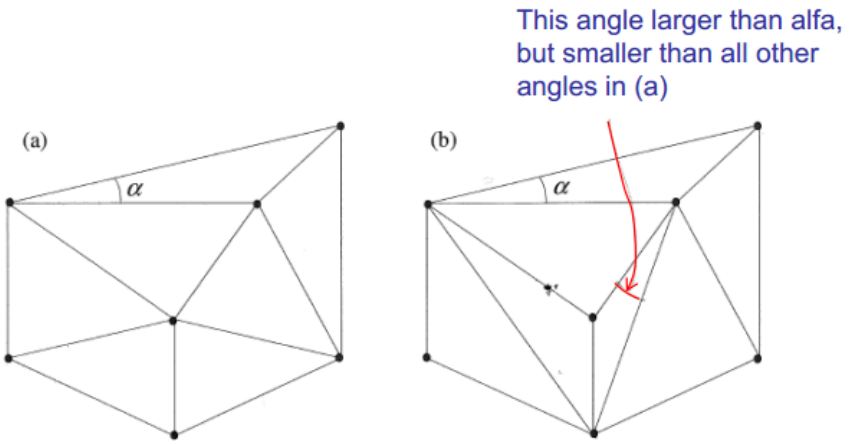


Figure 114:

Figure (a) is a Delaunay triangulation, but not figure (b). They have he same smallest angle, but the next to smallest is largest in (a).

11.3.1 The Voronoi diagram

Another definition of Delaunay triangulation

The diagram to the left is the Voronoi Diagram of the given points. It is constructed as if the points are islands, and so that that the sea area closer to island x than to any other island belongs to island x ("closest shore").

Figure 115:

The Delaunay-triangulation is then obtained by drawing an edge between those points/islands that have a common border, you will get a Delaunay triangulation (and this edge will be orthogonal to the common border). Note that these edges will not always pass through the common border (this is the case for the top two edges, and one at the bottom left).

A triangulation is a Delaunay triangulation if and only if for all triangles, the circle through the three corners does not contain (in its inside) any of the other points. It is at the outset not clear that such a triangulation will always exist. However, it in fact does, and there is only one such triangulation (if we don't have any of the special cases).

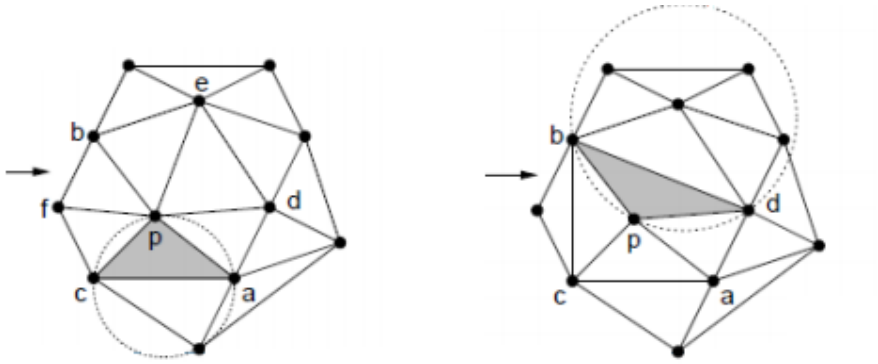


Figure 116:

The triangulation to the left is a Delaunay triangulation, but not the one to the right. Note that we here have one of the special cases (four points on a circle), and then the Delaunay triangulation is not unique. We could have replaced the edge c-a with the one from p and downwards. This is the definition we used for the Delaunay triangulation.

11.4 The Delaunay trick

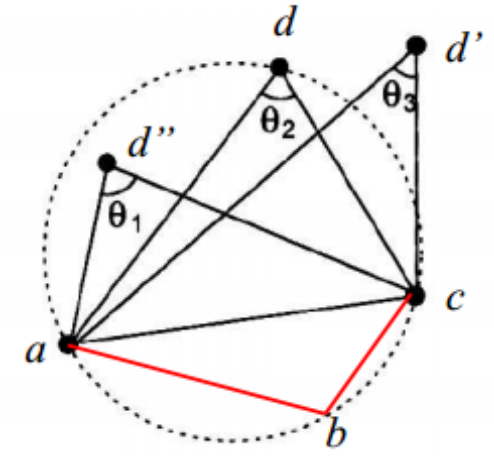


Figure 117:

We have: $\Theta_1 > \Theta_2 > \Theta_3$. To decide whether the point d is inside, on, or outside the circle through a, b and c, we have to assure that a, b, c and d are taken in counter clockwise order, and then compute the value of the determinant:

$$inCircle(a,b,c,d) = det \begin{pmatrix} a_x & a_y & a_X^2 + a_y^2 & 1 \\ b_x & b_y & b_X^2 + b_y^2 & 1 \\ c_x & c_y & c_X^2 + c_y^2 & 1 \\ d_x & d_y & d_X^2 + d_y^2 & 1 \end{pmatrix} > 0$$

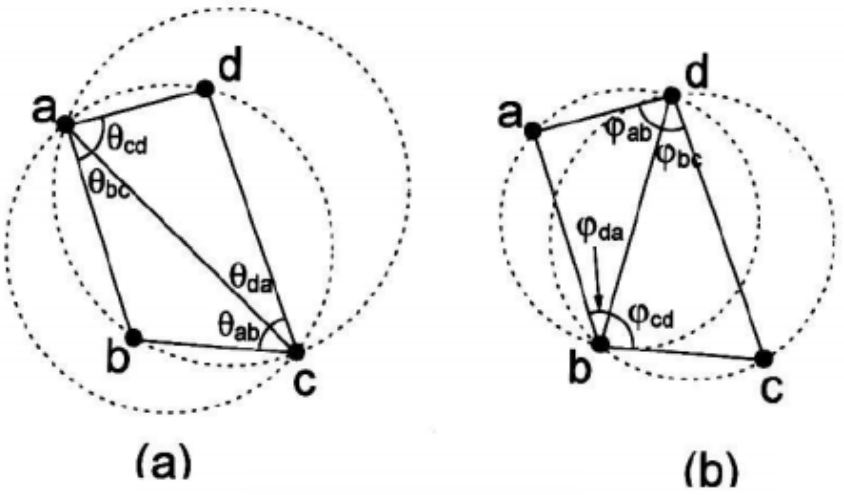


Figure 118: Assume the quadrangle a-b-c-d is convex

We can observe that if d is inside the circle through a, b, and c, then the circle through a, c, and d will contain b. The Delaunay requirement is not fulfilled. If we in (a) remove the edge a-c and instead insert b-d (see (b)), then point a will be outside circle through b, c and d, and c will be outside the one through a, b, and d. Now the Delaunay requirement is fulfilled, at least locally.

11.5 Delaunay triangulation algorithm

We are given a set of points in the plane.

- Start: To have a triangulation to begin with, we add three points, so that the triangle defined by these points contains all the given points.
  - And we let this triangle be the initial triangle (see below).
  - This single triangle is a Delaunay triangulation of the three points.
- We then do as we did earlier when we constructed a triangulation to count triangles and edges:
  - That is, we add one node at a time, and for each node we also add the three edges to the corners of the triangle where the node resides.
  - This may locally destroy the Delaunay property, and before we add the next node we will restore the Delaunay property, when necessary.

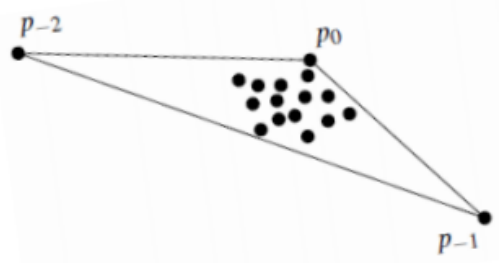


Figure 119:

It is important here to choose two of the three starting nodes far away from the node set. Then it will be easier to remove them afterwards. The third point can in fact be one of the original points.

11.5.1 Restoring the Delaunay property

Adding a point:

- Before the addition we have a Delaunay triangulation (invariant).
- We add a point p in one of the triangles, and draw edges from p to each of the three corners of this triangle.
- This may destroy the Delaunay property, and we want to restore it by using the Delaunay trick.

To restore the Delaunay property we look at all triangles that have a corner in p. We look at each of these triangles together with the neighboring triangle opposite to p, and check whether the Delaunay property holds for these two triangles together. If not, we use the Delaunay trick on those two triangles. While this checking and repair goes on, the set of triangles with a corner in p can increase, and we have to go on testing until we have gone a full round without any Delaunay property being broken. This process will always stop, as the number of edges to p will increase each time we use the Delaunay trick, and there is only a finite number of points that p can have edges to.

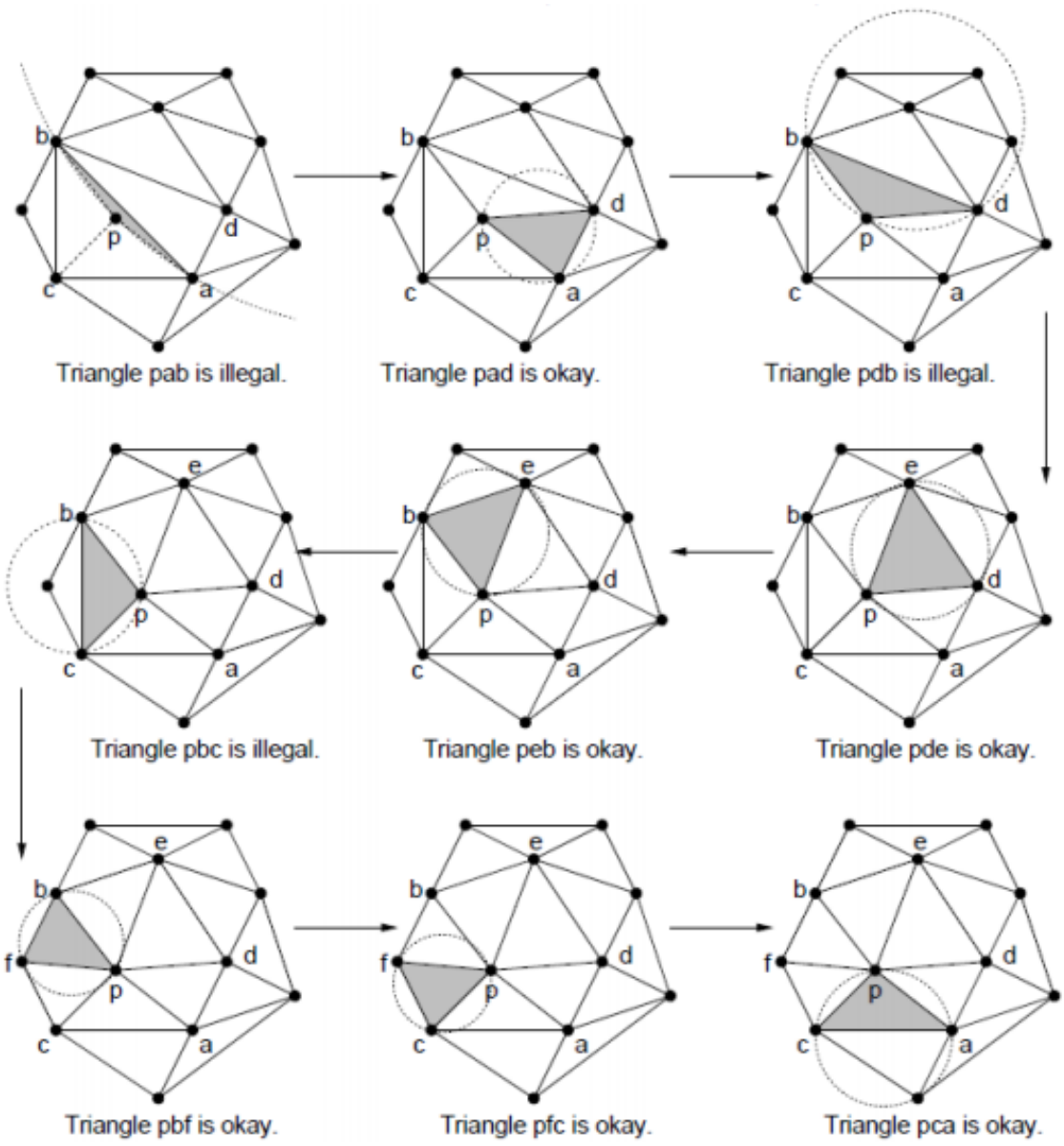


Figure 120:

11.5.2 Starting and ending the algorithm

As we have described the algorithm, we start the algorithm by adding three extra points, so that the triangle spanned by these contains all the given ("real") points. This triangle makes up our initial Delaunay triangulation. We then perform the algorithm. Thus, the last problem is to "get rid of" the extra points we started with. The trick here is to place at least two of the extra nodes far away from the given node set. Then we can simply remove the extra points, and the edges to them. This works because the convex hull of the original points will all be edges in the triangulation.

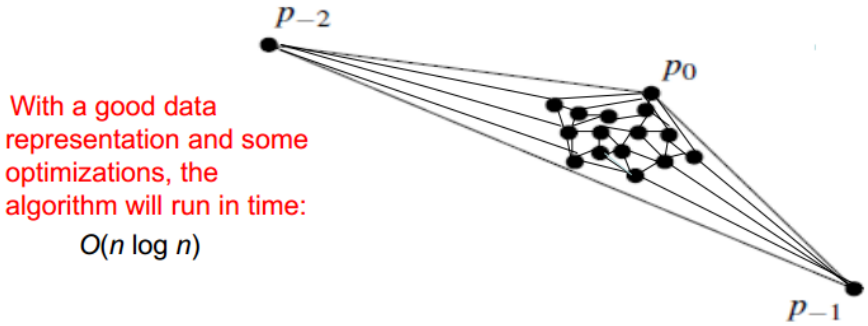


Figure 121:

11.6 Large angles

The figure shows heights at different points. One usually assumes that the edges of the triangulation are straight lines in the terrain. That means that the height of a point on an edge can be found by interpolation of the height of the endpoints of the edge. We can see that the left triangulation below gives an intuitively better height for q than the triangulation to the right.

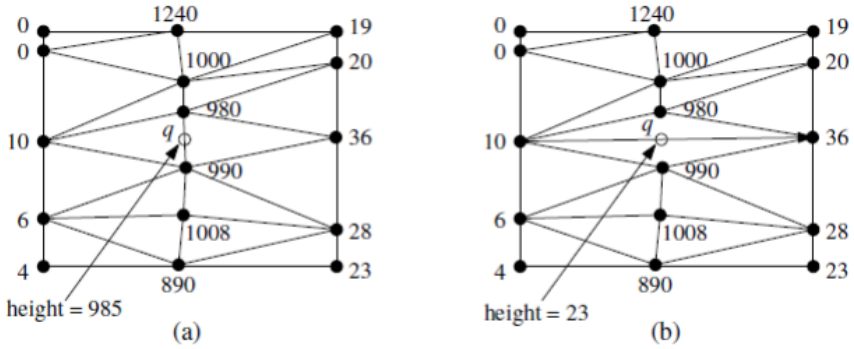


Figure 122:

12 Undecidability

13 NP-completeness

14 Proving NP-completeness

14.1 SAT $\propto$ 3SAT

14.2 3SAT $\propto$ 3DM

14.3 3DM $\propto$ Subset sum

15 Coping with NP Completeness

Part II
Appendix

16 Notations

16.1 Big-O (upper limit)

- Notation: f(n) = O(g(n))
- Intuiton: f is smaller than g
- When n $\to \infty$: $f(n) \leq c \cdot g(n)$
- Definition:
  $\exists(c > 0), n_0 : \forall(n > n_0)$
  $f(n) \leq c \cdot g(n)$

16.2 Big Omega (lower limit)

- Notation: f(n) = Ω(g(n))
- Intuiton: f is larger than g
- When n $\to \infty$: $f(n) \geq c \cdot g(n)$
- Definition:
  $\exists(c > 0), n_0 : \forall(n > n_0)$
  $\to \infty$: $f(n) \geq c \cdot g(n)$

16.3 Big Theta ("As")

- Notation: f(n) = Θ(g(n))
- Intuiton: f grows like g

- When n $\to \infty$: $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$
- Definition:
  $\exists(c_1, c_2 > 0), n_0 : \forall(n > n_0)$
  $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

16.4 Little-o

- Notation: f(n) = o(g(n))
- Intuiton: f is a lot smaller than g
- When n $\to \infty$: $f(n) << g(n)$
- Definition: $lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

16.5 Example

- Show whether n+1 is O(n).
  We know that 2n is O(n) and n is O(n). So to show that n+1 is O(n), we need to prove that; $n \leq n+1 \leq 2n$.
  $2n = n + n \geq n + 1$ for $n \geq 1$
  n+1 is trivially larger than n
- Show whether n log n is O(n^2).
  O(n^2) allows functions on the form An^2 where A is som sort of factor. This can be written as $Bn \cdot Cn$ where $A = B \cdot C$. The logarithmic function is a function which fullfill the requirements for B and C.
- Show whether 2^{n+1} = O(2^n).
  For something to be O(2^n) it has to fullfill the requirements; $c \cdot f(g) \leq O(g(n))$. There exists no constant c such that $c \cdot 2^{n+1} \leq 2^n$.
- Show whether $\frac{10n+16n^3}{2}$ = O(n^2).
  If $\frac{10n+16n^3}{2}$ = O(n^2), $c \cdot \frac{10n+16n^3}{2}$ has to be smaller than or equal to n^2. As n^3 grows faster than n^2, we can see that there is no such constant when $n \to \infty$.

Index