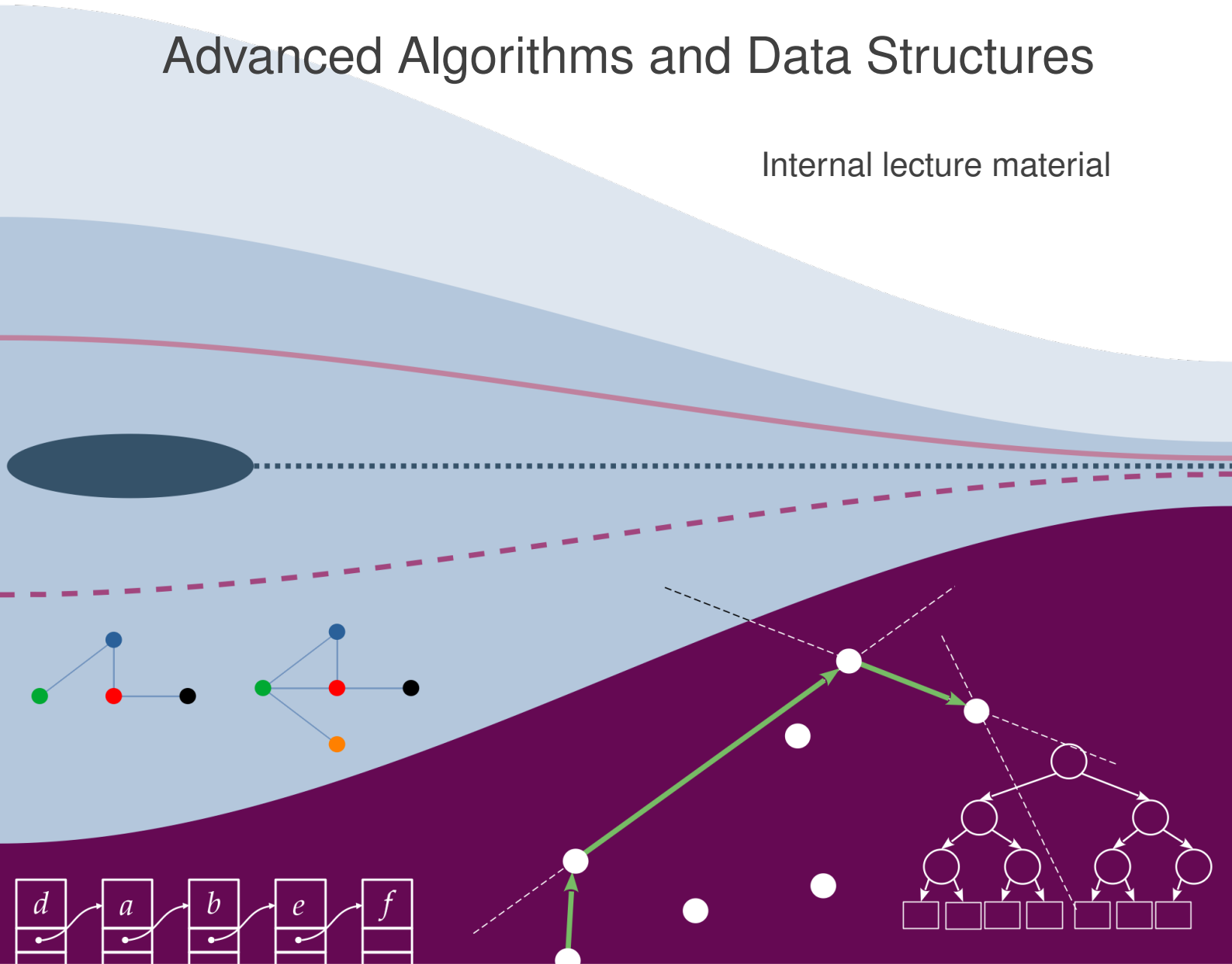
;
Faculty of Electrical Engineering and Computing, Department of Applied Computing

D. Krleža, M. Brčić

Advanced Algorithms and Data Structures

Internal lecture material



PUBLISHED BY FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING, DEPARTMENT OF APPLIED COMPUTING

First printing, August 2022

Contents

Introduction

Studying algorithms is a multidisciplinary endeavor. It contains three intertwined layers of abstraction. First, we need to understand the problem we are trying to solve. Many authors define algorithms merely as a set of tools, without giving the usage context. All algorithms were first introduced to solve real-world problems, and we find the problem to be the key part of understanding the algorithm. After understanding the problem, we need to model it mathematically, which makes the problem formally defined. Based on the mathematical model of the problem we can draw many conclusions that can help us automate the solution of the problem. The final solution to the problem is automation that can cover the problem fully or partially and can be written down as algorithms.

When we talk about algorithms, we think about a set of steps that need to be undertaken to solve the problem. In time, many algorithms were optimized to be more efficient, taking less time to execute by improving their complexity. In this lecture book, we give an overview of this progressing, by comparing complexities between algorithms intended to solve similar problems. Beyond seeing algorithms just as a step of steps that need to be undertaken, we understand that an algorithm has no real value if it cannot be executed on some apparatus, for example, computers. Programming algorithms is equally important, as not all languages are the same, and do not offer the same capabilities to a developer, which consequently reflects in the algorithm implementation. We all heard the saying "if you want it to be fast, program it in the C programming language". While this might not stand nowadays anymore, it still holds certain wisdom when we implement an algorithm. For example, is the C programming language a good choice when we need to implement Red-Black tree rules? Beyond only discussing the efficiency of algorithms through their complexity, we need to be aware that they also consume a significant amount of memory too. While the memory consumption goes in the algorithm implementation details, rearranging algorithm steps can influence memory consumption as well. All this creates an understanding that the algorithm's abstract set of steps is only a prerequisite for the implementation. Only when we implement the algorithm in some programming language we can assess its true efficiency. For that reason, this lecture book gives python codes for all algorithms. Some might argue about the usage of python for such mathematically advanced algorithms. We are sure that these algorithms can be implemented to be more efficient in some programming languages.

In our lectures, we also encountered that many of the lecture books we used for the course were too formal for our students, giving too many details around the mathematical model and apparatus that was used to create these algorithms. By writing this lecture book, we wanted our students to focus on understanding problems that are solved by introduced algorithms, and understand each algorithm in-depth, which is supported by visual examples, algorithm pseudo-codes, and real python code that can be downloaded from the *github*. Of course, many algorithms are mathematically complex, which requires basic explanation and understanding that cannot be avoided. With that in our minds, we hope that this lecture book will satisfy both Computer Science students and IT professionals, giving the right level of details, explanations, and examples.

1 Advanced data structures

By affecting the complexity, speed, and memory usage of algorithms,advancements in accompanying data structures is important part of the Computer Science. The way we structure and organize data is directly impeding the retrieval process, from the speed and complexity point of view. For example, we could store a simple set of numbers $S = \{1, 6, 2, 20, -5, 11, 4\}$ into a sequential list. $20 \in S$ and $21 \notin S$ is a simple mathematical membership checking task. When developing an algorithm that will perform this membership checking in our sequential list, we must be aware of the sequential list limitations. Knowing that we need to traverse the whole sequential list in a worst-case scenario, or $O(n)$, to check a number membership, we must start wondering whether there is a better structure that would support faster membership checking? Besides efficient and optimal data searching and retrieval, our data structure must be able to store new data equally efficiently. Too complex data structures usually require significant effort to insert new data, which destroys data structure usability, since algorithms need to spend more time inserting new data into it. There must be a reasonable trade-off between retrieval and storing of data into data structures we use.

The other problem we face with data is the memory consumption. Some very efficient data structures, retrieval and storing wise, can take significant amount of memory. For example in-memory multidimensional hypercubes (*OLAP databases*), which are used to store unnormalized data for analytical purposes. Although very efficient in data retrieval and storing, these hypercubes require significant memory, which turns out to be quite inefficiently used. Is it possible to improve the hypercube memory consumption? It certainly is, but it mostly means some trade-off with data retrieval and storing efficiency.

Data structures are usually studied and selected based on the related use case. When benchmarking a data structure, we usually express the following three complexities:

- **The search (query) complexity**. It expresses complexity when searching for a specific data element in the data structure. If we take a sequential linked list, the search complexity for it is $O(n)$, meaning that we need to iterate the whole linked list in the worst-case scenario. In the database context, this is called a *sequential scan*, and is considered for the slowest data search.

- **The insertion (storing) complexity**. Used for data insertion benchmarking, or how complex is to store a new data element in the data structure. For the sequential linked list this complexity is $O(1)$, as we do not need to find the new data element position in the list, we just add it to the end of the list.

- **The space complexity**. Points the amount of storage needed to store the data in the data structure. While the linked list has the space complexity $O(n)$, not all data structures are so efficient. In some structures there are additional pointers and data elements. There could be some data redundancy in some data structures. All this must be summed up in the space complexity indicator.

In this chapter we give a selected list of advanced data structures that can be used in generic use cases [Cormen et al., 2009, Drozdek, 2012]. Further in the script we additionally use and extend these data structures, to show how to adapt them for specific use cases, for example *geometric algorithms*.

1.1 Skip lists

A basic single linked list is a structure where nodes having values are connected by pointers. We begin with a starting pointer, called a *head*, which points to a first node having a value and a pointer to a second node. The second node has its own value and a pointer to the third node. This continues until the last node, whose pointer does not point anywhere, it is a *NIL* pointer. An example of a single linked list can be seen in Figure 1.1.

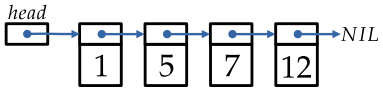


Figure 1.1: A linked list.

Listing 1.1: A linked list python code.

```
class List:
  head=Node(None)
  def search(self,v):
    return self.head.search(v)
  def insert(self,v):
    self.head.insert(v)
```

```
class Node:
  v,p=None,None
  def __init__(self,v):
    self.v=v
  def search(self,v):
    if (self.v is None or v>self.v) and self.p is not None:
      return self.p.search(v)
    elif self.v is not None and self.v==v: return self
    else: return None
  def insert(self,v):
    nv=None
    if self.p is not None: nv=self.p.v
    if (self.v is None or v>self.v) and (nv is None or v<nv):
      tp=self.p
      self.p=Node(v)
      self.p.p=tp
    elif self.v is not None and v==self.v: return
    else: self.p.insert(v)
```

The search on the linked list has $O(n)$ complexity, which means we need to traverse the whole list in the worst-case scenario. If we keep our linked list unsorted, we can just add new values at the end of the list, which is $O(1)$. In case of a sorted linked list, first we need to find the position for the new value, and then to insert the value by rearranging pointers in the list, which takes $O(n)$ in the worst-case scenario.

How do we improve the search on a linked list? Any improvement must improve the inserting for a sorted linked list as well, since we need to find belonging position for our value before inserting it. One idea is to skip some of the values, or to jump over several nodes at once by introducing another layer of pointers. Such a structure is called a *skip list*[1]. Each layer of additional pointers allows us skipping more and more values, which is related to the skip list length. Skipping values in the skip list is of a probabilistic nature.

[1] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6): 668–676, 1990

The ideal skip list structure

To describe the ideal structure of the skip list, we need to start from basic definitions, for which we can look Figure 1.2. We can see 3 layers of pointers, where the bottom layer (node layer) has as many pointers as nodes (values), and the top layer has only one pointer. In the ideal skip list structure, the number of pointers is 2^0 for the top layer, 2^1 for the layer below, rising as we descend down to the node layer. So, for Figure 1.2, the top layer has $2^0 = 1$ pointers, the middle layer $2^1 = 2$ pointers, and the node layer has as many pointers as nodes.

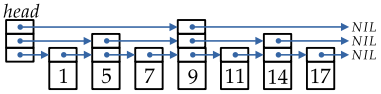


Figure 1.2: An ideally structured skip list.

It becomes immediately clear that the depth of the ideal skip list structure follows the number of nodes, so we need to have at least $k = \lceil \log n \rceil$ layers of pointers, which comes from the number of pointers needed for each layer. Looking deeper into the ideal skip list structure, we can see similarities to binary trees. In Figure 1.2, the node 9 can be taken as the *root node*. All nodes before 9 can be taken as the left subtree, and all nodes after 9 can be taken as the right subtree. Descending one layer below, we see the same structure, which sets the obvious resemblance between an ideally structured skip list and a binary tree. The main disadvantage of such ideal structure is the need for restructuring the whole skip list for each inserted value, similarly as in binary trees. This rises our inserting complexity quite significantly, and makes any practical implementation of a skip list unusable.

Definition 1.1 (Skip list node degree (level))**.** A node degree is the number of points above the node, written as $deg(n_i)$, which is always greater than 1.

In Figure 1.2, we have $deg(9) = 3, deg(5) = 2, deg(1) = 1$, and so on. For a skip list comprising values

$$SL = \{v_1, v_2, ..., v_n\} \tag{1.1}$$

we have a head

$$|head| = k = \max_{v_i \in L} deg(v_i) \tag{1.2}$$

Searching values in skip lists

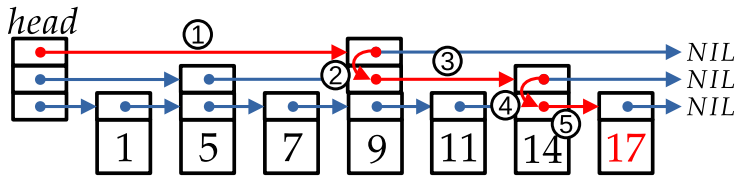


Figure 1.3: Searching 17 in the skip list.

Let us say that we need to search for 17 in the skip list from Figure 1.2. We start finding a value in the skip list from the *head* pointers. When traversing the skip list, we descend from the top layer ($l = k - 1$) down to the node layer ($l = 0$). If our current pointer points to a node having value that is bigger than the searched value or NIL, we descent down to the layer below. If the current pointer points to a value that is less or equal to the searched value, we jump to the node of our current pointer, and continue searching from there. Searching 17 in the previously defined skip lisp can be seen in Figure 1.3, and is done with following search steps:

1. We start from the *head* and the top layer ($l = 2$). Our current pointer points to 9, which is $9 \leq 17$. We jump to the node 9.
2. The current pointer in the top layer ($l = 2$) for the node 9 points to NIL. We descend one layer down, or to $l = 1$.

3. The current pointer in the middle layer ($l = 1$) for the node 9 points to 14, which is $14 \leq 17$. We jump to the node 14.

4. The current pointer in the middle layer ($l = 1$) for the node 14 points to NIL. We descend one layer down, or to $l = 0$.

5. The current pointer in the bottom layer ($l = 0$) for the node 14 points to 17, which is our searched value. The search stops.

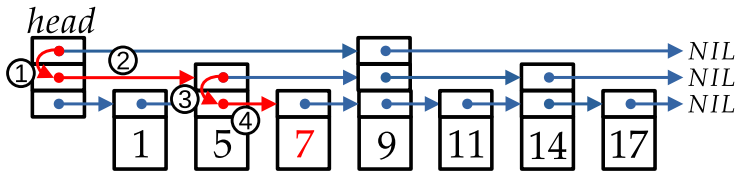


Figure 1.4: Finding 7 in the skip list.

Searching 7 in the previously defined skip lisp can be seen in Figure 1.4, and is done with following search steps:

1. We start from the *head* and the top layer ($l = 2$). Our current pointer points to 9, which is $9 > 7$. We descend one layer down, or to $l = 1$.

2. The current pointer in the middle layer ($l = 1$) for the *head* points to 5, which is $5 \leq 7$. We jump to the node 5.

3. The current pointer in the middle layer ($l = 1$) for the node 5 points to 9, which is $9 > 7$. We descend one layer down, or to $l = 0$.

4. The current pointer in the bottom layer ($l = 0$) for the node 5 to 7, which is our searched value. The search stops.

Any attempt to descend further than the bottom layer means that the skip list does not contain the searched value. We can write search steps into the following algorithm

```
class SkipNode:
  ptrs,v,pred=None,None,None
  def __init__(self,v,p=None,s=None):
    self.v=v
    self.ptrs=[None]
    if p is not None:
      p.ptrs[0]=self
      self.pred=p
    if s is not None:
      self.ptrs[0]=s
      s.pred=self
```

Listing 1.2: Skip list node python class. p is a predecessor, and s is a successor for the newly created skip list node. v is the value of the node, and *pred* is the direct pointer to the predecessor in the layer 0.

function SEARCHSKIPLIST($head, s_v$)
  $n \leftarrow head$
  $l \leftarrow |pointers(n)| - 1$
  while $l \geq 0$ **do**
    $n_{next} \leftarrow pointers(n)[l]$
    if $value(n_{next}) > s_v$ or n_{next} is NIL **then**
      $l \leftarrow l - 1$
    else
      $n \leftarrow n_{next}$
  return n

Algorithm 1.1: Skip list searching algorithm.

Inserting new values in skip lists

We already concluded that the ideal skip list structure is impractical for the real use. For this reason, we need to relax the structure

constrains, to avoid reworking the whole skip list for each value insertion. To keep the probabilistic nature of the skip list, instead of focusing on creating highly regular structure of pointers, we use the geometric distribution when inserting a new node. Each inserted node has to have the bottom layer, or at least one pointer to the next element in the list. All the higher layers of pointers, allowing skipping values, are created according to the geometric distribution. We have a probability p (usually $1/2$ or $1/4$) to create an additional layer of pointers. If we take $p = 1/2$, after creating the bottom layer, we flip a coin looking for the head. If the head appears, we add another layer of pointers on the newly inserted node. First time we get the coin tail, we stop adding new pointer layers. Total probability that the newly inserted node n_{new} attains $deg(n_{new}) = k$ is

$$P(k) = P(\text{adding a layer})^{k-1} P(\text{stop adding layers}) = p^{k-1}(1-p) \tag{1.3}$$

which means $k-1$ heads (having probability p) and one tail (having $1-p$ probability). For an infinite skip list, we can calculate the expected mean degree of nodes as

$$\begin{aligned} E(k) = \sum_{k=1}^{\infty} kP(k) = (1-p)\sum_{k=1}^{\infty} kp^{k-1} \\ \sum_{k=1}^{\infty} kp^{k-1} = \frac{1}{(1-p)^2} \\ E(k) = \frac{1}{1-p} \end{aligned} \tag{1.4}$$

When inserting a new value, first we need to find the appropriate place for it. Remember, skip lists are sorted. Once we find the spot between two nodes that is appropriate for the new value, as in Figure 1.5, we can create the node for it. The newly created node must

```
class SkipList:
  head=SkipNode(None)
  p=0.5
  def __init__(self,p=0.5):
    self.p=p
  def search(self,v):
    n=self._search(v)
    if n.v==v: return n
    else: return None
  def _search(self,v):
    n=self.head
    l=len(n.ptrs)-1
    while l>=0:
      nn=n.ptrs[l]
      if nn is None or nn.v>v:
        l=l-1
      else:
        n=nn
    return n
```

Listing 1.3: Skip list search python code.

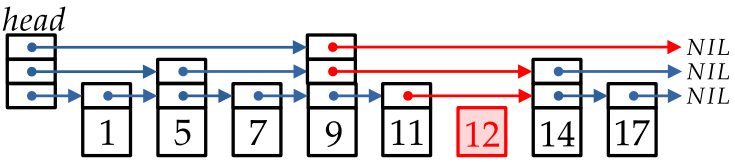


Figure 1.5: Inserting 12 in the skip list.

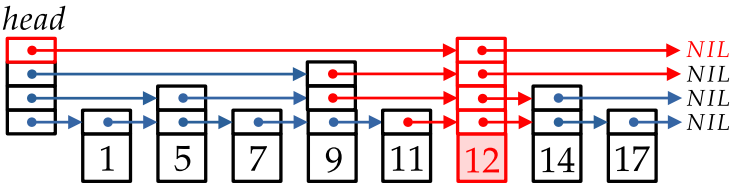


Figure 1.6: Reworking pointers after Inserting 12 in the skip list.

have its bottom layer (layer 0) of pointers which connects it to the predecessor and successor nodes. Then we start flipping a coin. If we get the coin head, we add another layer of pointers. The newly added node must have its own predecessor and successor in this layer, which are also present in this layer. For determining the predecessor in the newly added layer, we need to return back in the list

to the first predecessor that has a pointer in this layer. In Figure 1.6, the first node that has pointer in the layer 1 is node having value 9. We redirect this pointer to the node 12, and pointer of the node 12 to the previous successor of the node 9. This coin flipping, returning back, and redirecting pointers continues until we get first tail.

If the newly added node has more layers than any of the already present nodes, which mean that is higher than the *head*, we need to extend the *head*, as in Figure 1.6.

procedure INSERTSKIPLIST($head, v_{new}$)
  $pred \leftarrow$ SEARCHSKIPLIST($head, v_{new}$)
  $n_{new} \leftarrow$ new node having value v_{new}
  $succ \leftarrow pointers(pred)[0]$
  $pointers(n_{new})[0] \leftarrow succ$
  $pointers(pred)[0] \leftarrow n_{new}$
  $predecessor(succ) \leftarrow n_{new}, predecessor(n_{new}) \leftarrow pred$
  $x \leftarrow 1, k \leftarrow |pointers(head)| - 1$
  repeat
    $c_r \leftarrow$ flip a coin
    if c_r is head **then**
      while $(|pointers(pred)| - 1) < x$ or $pred$ is $head$ **do**
        $pred \leftarrow predecessor(pred)$
      if $pred$ is head and $x > k$ **then**
        extend head to layer x
        $pointers(head)[x] \leftarrow NIL$
      $pointers(n_{new})[x] \leftarrow pointers(pred)[x]$
      $pointers(pred)[x] \leftarrow n_{new}$
      $x \leftarrow x + 1$
  until c_r is tail

Algorithm 1.2: Algorithm for inserting new values in skip lists.

```
class SkipList:
  ...
  def insert(self,v):
    pred=self._search(v)
    s=pred.ptrs[0]
    nn=SkipNode(v,p=pred,s=s)
    x,k=1,len(self.head.ptrs)-1
    while True:
      ctoss=random.uniform(0,1)
      if ctoss<=self.p:
        nn.ptrs.append(None)
        while (len(pred.ptrs)-1)<x
            and pred is not self.head:
          pred=pred.pred
        if pred==self.head and x>k:
          self.head.ptrs.append(None)
        nn.ptrs[x]=pred.ptrs[x]
        pred.ptrs[x]=nn
        x=x+1
      else: break
  def remove(self,v):
    n=self.search(v)
    while n is not None:
      x,pred=0,n.pred
      while (len(n.ptrs)-1)>=x:
        while (len(pred.ptrs)-1)<x
            and pred is not self.head:
          pred=pred.pred
        pred.ptrs[x]=n.ptrs[x]
        x=x+1
      n=self.search(v)
```

Listing 1.4: Skip list insertion and removal python code.

Flipping one coin supports $p = 1/2$. For all other p values, we need to update the previous algorithm accordingly. For $p = 1/4$ this can be two heads for two consecutive coin tosses.

Removing values in skip lists

Removing values in skip lists works similarly to inserting values. First, we find the node that needs to be removed. To remove this node from the skip list, we need to rework all pointers on all layers. We have direct pointer to the predecessor only in the layer 0. For all other layers, we need to return back to determine predecessors. Reworking pointers means redirecting pointer from the predecessor of the removed node to its successor.

Complexity assessment

For the ideal skip list structure, being equal to the binary tree structure, we have searching complexity $O(\log n)$. Since we relaxed this ideal structure, to avoid reworking the whole structure each time we insert or remove a value, the skip list complexity is somewhere between binary trees and linked lists, or $O(\log n) \leq t(\text{skip list}) \leq$

$O(n)$. Obviously, when our coin tossing gets close to the ideal structure, we get the lowest possible complexity. The worst-case scenario is to get tails for each coin tossing, which makes our skip list structure not so *skippable*, or equivalent to the linked list.

When inserting a new value (or removing one), the worst case scenario is to add a new node to the end of the skip list, and coin tossing making the number of layers above the newly added node higher than the head. For this scenario, we get the complexity $O(2 \log n)$ on the low end, and $O(2n)$ on the high end. Nevertheless, skip list basic operations can be taken to have complexity between $O(\log n)$ and $O(n)$, which is better than linked lists.

1.2 *Sparse data structures (tables)*

Regular multidimensional data stores, such as in-memory cubes can be sometimes so large it can take a significant amount of memory to store all data objects. For example, a 2-dimensional memory table having 8000×10000 64-bit integers takes approximately 611 MB of memory. Then, we can use very little of the allocated memory. An example would be to store course grades for 8000 students that averagely take 6 courses out of 300 offered courses. In a table 8000×300 we have populated only 2% of values. Again, if we take that each grade is an 8-bit integer, we can quickly calculate that we wasted 2297 kB of memory out of 2344 kB totally allocated memory.

We can take the following structure for each data object in the students $\times$ courses table: student identifier (2 bytes), course identifier (2 bytes), grade (1 byte), pointer to the next student in the list (8 bytes), pointer to the next course in the list (8 bytes), which totally takes 21 bytes.

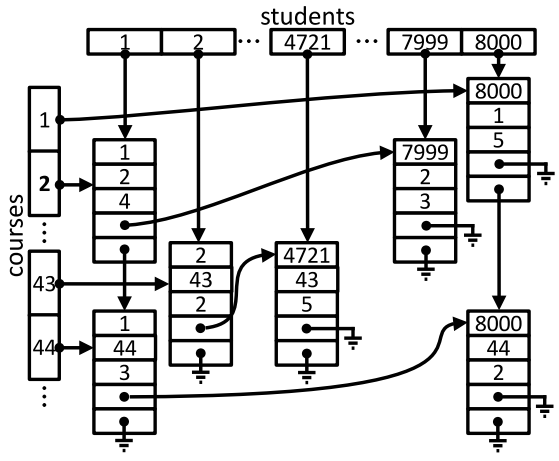


Figure 1.7: 2-dimensional linked list as the sparse table structure.

The 2-dimensional linked list in Figure 1.7 is one of the elegant solutions how to implement sparse tables. This 2-dimensional linked list can be extended to support any n-dimensional case. We have 2 *head* structures, used to store dimensional indices, which can be of any data type (or even structures). Head structures can be anything

that helps us organize and quickly access dimensional index values. In Figure 1.7 we used a simple array of values. However, this can also be a binary tree. Each dimensional index value has its own linked list, which is intertwined with other dimensions linked lists. In the 2-dimensional case, we have 2 pointers in each data structure, each supporting one dimension. In Figure 1.7, horizontal linked lists represent courses while vertical linked lists represent students. If we need all courses of a specific student, we start by finding the student in the *student* head structure, which gives us the pointer to the first data structure in the student's linked list. Traversing the linked list, we access all courses for that student. If we need all students that passed a specific course, we start by finding the course in the *course* head structure. Then we proceed by traversing the related linked list for the course, which contains all students that passed the course.

Such data structure fully uses the allocated memory. Although very efficient memory-wise, it takes longer to access required data. For an $n \times n$ sparse table, having heads organized as binary trees, finding our dimensional index will take $O(\log_2 n)$. Traversing through the linked list takes $O(n)$. The same complexity is for the inserting as well.

1.3 Binary trees

Graph theory defines trees as directed acyclic graphs. In such graphs, each vertex, except the root vertex, has exactly one parent node, and there are no cycles in the tree. Binary trees are a specific kind of trees, where each vertex can have at most two distinct child vertices, or that the degree of a vertex is at most 2. For a binary tree $G = (V, E)$, we say that

$$\forall v \in V : deg(v) \leq 2 \tag{1.5}$$

For $deg(v) = 0$, v is considered for a leaf vertex. Although binary trees originate from the graph theory, we usually relax and adjust binary tree definitions and terminology.

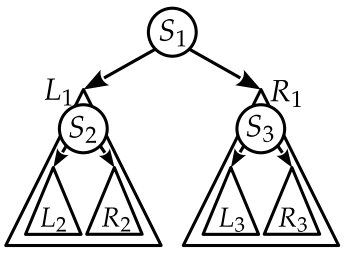


Figure 1.8: Binary tree triplet structure.

A binary tree is an ordered triplet

$$B = (L, S, R) \tag{1.6}$$

where S is a *singleton set* (a set that can have at most 1 element) containing a binary tree *root node* (a vertex in the graph theory terminology), L is the left binary subtree (again B), and R is the right binary subtree (again B). Both subtrees are **recursively** defined. Left and right subtrees can be left undefined ($L = nil$, $R = nil$), which means that the *root node* in the set S can miss one or both children. The set S has exactly one element $|S| = 1$, which represents the binary tree (or subtree) root node. Some authors allow S to be empty, which is not necessary, as there is no binary tree without nodes, so

we need to have at least one node. The binary tree triplet structure can be seen in Figure 1.8.

To describe relations between nodes in the binary tree, we use genealogy terms. We already defined the parent-child relationship, where each node can have up to two children, and each child (except the root) has only one parent. Children of the same parent node are siblings. Also, nodes can have a grandfather, an uncle, and so on. Some of these terms are used in binary tree algorithms.

Definition 1.2 (Full binary tree). A tree whose inner nodes have exactly two children is called a *full binary tree*.

Definition 1.3 (Complete binary tree). A tree having all levels, except the last one, completely filled with the maximal number of nodes in this level, is called a *complete binary tree* or a *proper binary tree*. The last level is filled with leaf nodes from the left side.

Definition 1.4 (Perfect binary tree). A tree having all levels completely filled with nodes is called a *perfect binary tree*.

A perfect binary tree can be seen in Figure 1.9. Each generation of nodes (genealogically speaking) in the binary tree are separated into several levels. The root node is in the 1st level. All of its children are in the 2nd level. All of the root node grandchildren are in the 3rd level, and so on.

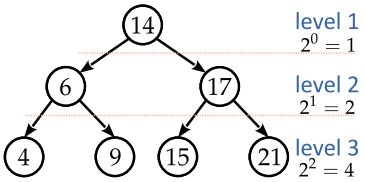


Figure 1.9: A perfect binary tree with related levels.

Definition 1.5 (Binary tree depth). The number of levels that are in a binary tree is called *binary tree depth* (sometimes *height*). A binary tree B having h levels is said to be of depth h, or $h(B) = h$.

If we observe Figure 1.9, we can see a correlation between levels of the binary tree and a maximal number of nodes in this level, which is 2^i for $i \in [0, h)$. The total maximal number of nodes in a binary tree having h levels is $n = 2^h - 1$. For the case in Figure 1.9 this is $2^3 - 1 = 8 - 1 = 7$. Using this principle, we get

$$\begin{aligned} n &= 2^h - 1 \\ 2^h &= n + 1 \\ h &= \log_2(n+1) = \log_2 2^h \end{aligned} \tag{1.7}$$

as the total number of levels in the perfect binary tree. When $n < 2^h - 1$ and the tree is complete, it means that only the bottom level of the binary tree is missing some nodes on the right side. In such case $h = \lceil \log_2(n+1) \rceil$.

A perfect binary tree properties:

- Total nodes: $n = 2^h - 1$
- Total leafs: $l = 2^{h-1}$
- Internal nodes: $i = 2^{h-1} - 1$
- Height: $h = \log_2(n+1)$

A complete binary tree properties:

- Total nodes: $n \leq 2^h - 1$
- Total leafs: $l \leq 2^{h-1}$
- Internal nodes: $i \leq 2^{h-1} - 1$
- Height: $h = \lceil \log_2(n+1) \rceil$
- $n = l + i \leq 2^h - 1$

Binary tree searching

In most cases, binary trees are used to organize data for improving search complexity. As previously mentioned, linked list search complexity is $O(n)$ and skip list search complexity is somewhere between $O(\log_2 n)$ and $O(n)$. Is it possible to have better search complexity than this?

We can organize nodes in binary trees to adhere the following ordering

$$v(S(L)) < v(S) < v(S(R)) \tag{1.8}$$

where $v(S)$ is the value of the root node, $v(S(L))$ is the value of the root node of the left subtree, and $v(S(R))$ is the value of the root node of the right subtree. This means that all nodes in the left subtree L have values that are smaller or equal than the root node value. All nodes in the right subtree R are greater than the root node value. Operators *smaller* ($<$) and *greater* ($>$) are being defined as the node values set relationship that defines the set ordering. For example in a set of characters $C = \{a, b, c, d, e, f, g\}$, the operator $\leq$ is the binary relation over the set C that defines ordering of its elements, resulting in the ordered set $(C, \leq)$. For naturally ordered sets, such as $\mathbb{N}$, we take their natural orderings. Such a binary tree can be seen in Figure 1.9. It is intuitively clear that searching any element in a complete binary tree takes checking at most $h \approx \log_2 n$ elements, which is also the searching complexity of the binary tree $O(\log_2 n)$, similar to the ideally structured skip list.

It is a common sense rule that we want to adhere to the nicely structured binary tree, as close to the perfect binary tree as possible. On the other side of this is a degenerate binary tree.

Definition 1.6 (Degenerate binary tree). A tree where most of the nodes have at most one child is a *degenerate binary tree*.

The most extreme case of a degenerate binary tree can be seen in Figure 1.10, which is equal to the linked list structure, and has the searching complexity $O(n)$. However, such a binary tree is also being *unbalanced*.

Definition 1.7 (Balanced binary tree). A binary tree B is called *balanced* when depths of left and right subtrees conform the following depth difference

$$|h(R) - h(L)| \leq 1 \tag{1.9}$$

which means that the depth difference between left and right subtrees can be at most one level.

Definition 1.8 (Perfectly balanced binary tree). A binary tree B is considered to be *perfectly balanced* when balanced and all levels except the last one are completely filled with nodes.

An example of a balanced binary tree can be seen in Figure 1.11. We can see depths for all nodes in the binary tree. The depth difference in the root node is $|3 - 2| = 1$. A perfectly balanced binary tree can be seen in Figure 1.12, which has the search complexity $O(\lceil \log_2(n+1) \rceil)$, becoming $O(\log_2(n+1)$ for a perfect tree.

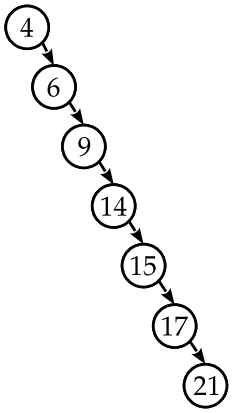


Figure 1.10: A degenerate binary tree.

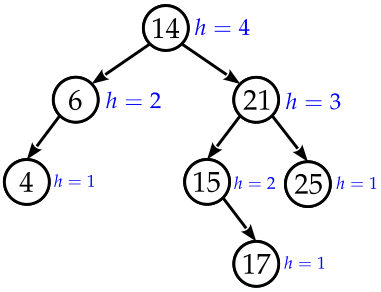


Figure 1.11: A balanced binary tree.

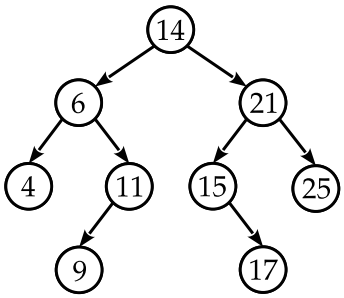


Figure 1.12: A perfectly balanced binary tree.

Notice that the depth difference of a perfectly balanced binary tree calculated in the root node does not need to be 0. A complete binary tree is perfectly balanced, yet due to the filling of the last level from the left side, the root node left subtree can be deeper than the right subtree.

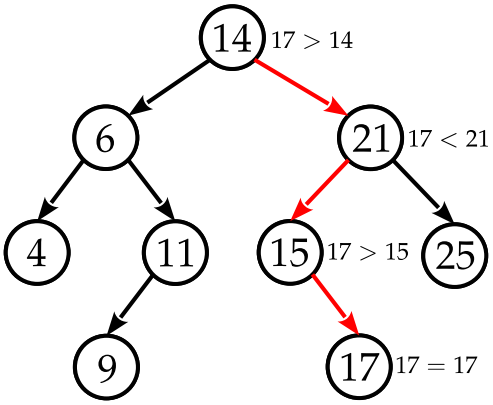


Figure 1.13: Searching value in a binary tree.

Searching in a binary tree is a recursive procedure. If a search value v_s is lesser than the root node value $v_s < v(S)$, then we recursively continue to search in the left subtree, if there is one. If the search value v_s is greater than the root node value $v_s > v(S)$, then we recursively continue to search in the right subtree, if there is one. Such a procedure can be seen in Figure 1.13 for a search value $v_s s = 17$.

Inserting new values in a binary tree

New values are always inserted as leaves in the binary tree structure. Before inserting a new value v_n, we need to find the closest node n having $deg(n) < 2$, which involves searching through the binary tree as we described previously. $deg(n) < 2$ means that the closest node n does not have both children, and the new value v_n can be added as a new child of the node n. This takes $O(\lceil \log_2(n+1) \rceil)$.

What happens if we find the value that we want to insert already present in the binary tree? Do we insert the same value again? This is a matter of allowing a binary tree to contain duplicate values. If we allow inserting duplicate values, we must redefine ordering of the binary tree structure from (1.8) to

$$v(S(L)) \leq v(S) < v(S(R)) \tag{1.10}$$

or

$$v(S(L)) < v(S) \leq v(S(R)) \tag{1.11}$$

If we **do not allow** inserting duplicate values, once we find that v_n already exists in the binary tree, we stop the insertion procedure.

Once we found the closest node to the new value v_n, we insert a new node using the default ordering (1.8).

```
class Node:
  S=None
  L,R,P=None,None,None
  h=1
  def __init__(self,v):
    self.S=v
  def setLeftChild(self,n):
    self.L=n
    if n is not None: n.P=self
  def setRightChild(self,n):
    self.R=n
    if n is not None: n.P=self
  def toRoot(self):
    self.P=None
    return self
  def query(self,v):
    if v<self.S
        and self.L is not None:
      return self.L.query(v)
    elif v>self.S
        and self.R is not None:
      return self.R.query(v)
    else:
      return self
```

Listing 1.5: Binary tree node (triplet) python code.

```
class SimpleBinaryTree:
  root=None
  def query(self,v):
    if self.root is None: return None
    else:
      n=self.root.query(v)
      if v==n.S: return n
      else: return None
```

Listing 1.6: Binary tree search python code.

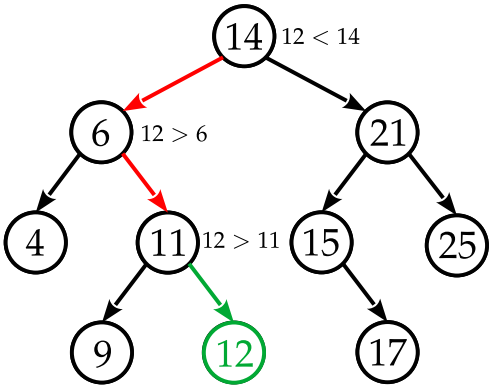


Figure 1.14: Inserting new value in a binary tree.

Inserting a new value $v_n = 12$ in the binary tree in Figure 1.12 starts with searching the closest node n to the value $v_n = 12$, having $deg(n) < 2$. Searching starts from the root node 14 and finishes on the node $n = 11$. Since $v_n = 12 > n = 11$, and there is no right subtree on the node 11, we add a new node $v_n = 12$ as the right child of the node $n = 11$. The final result, seen in Figure 1.14, is still perfectly balanced. However, this does not need to be the case with further insertions, such as in the example in Figure 1.15. Additional inserting $v_n = 16$ makes the binary tree locally unbalanced for nodes 15 and 21. The resulting binary tree is still considered to be overall balanced. However, it ceases to be perfectly balanced, since we now have 2 bottom levels that are not fully populated.

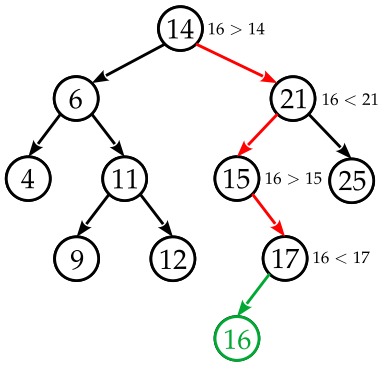


Figure 1.15: Inserting new value makes binary tree locally unbalanced.

```
class Node:
  ...
  def insert(self,v):
    n=self.query(v)
    if v<n.S:
      n.setLeftChild(Node(v))
    elif v>n.S:
      n.setRightChild(Node(v))
    else:
      raise Exception('cannot insert')
    return n

class SimpleBinaryTree:
  ...
  def insert(self,v):
    if self.root is None:
      self.root=Node(v)
      return None
    else:
      return self.root.insert(v)
```

Listing 1.7: Binary tree insertion python code.

Removing values from a binary tree

First we need to find the node we need to remove. Based on the node position in the binary tree structure, we can have multiple cases.

The removed node is a leaf

In such case we simply remove the node without making any structural rearrangement of the binary tree. For the example, if we want to remove 16 from the binary tree in Figure 1.15, we simply remove the node 16.

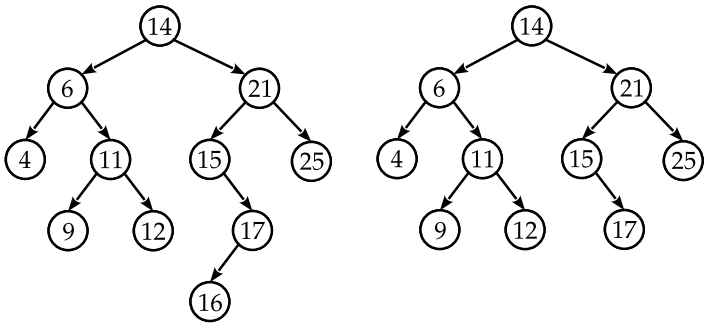


Figure 1.16: Removing leaf values from the binary tree.

The removed node has one child

If the removed node has one child, this child if the root of the single subtree that is left after the removal. This only child becomes the root

```
class Node:
  ...
  def children(self):
    c=0
    if self.L is not None: c=c+1
    if self.R is not None: c=c+1
    return c
  def rightmost(self):
    if self.R is not None:
      return self.R.rightmost()
    else: return self
  def leftmost(self):
    if self.L is not None:
      return self.L.leftmost()
    else: return self
```

Listing 1.8: Binary tree removal helping python code.

instead of the removed node. We connect the parent of the removed node with its only child. The new connection must be on the same side as the removed node. If we remove 17 from the binary tree in Figure 1.12, we get the case as in Figure 1.17. 16 is the root of the subtree that is left after we remove 17, and is also the only child of 17. We connect the parent 15 and the only child 16, where 16 must be the root of the right subtree (the same side as the removed 17).

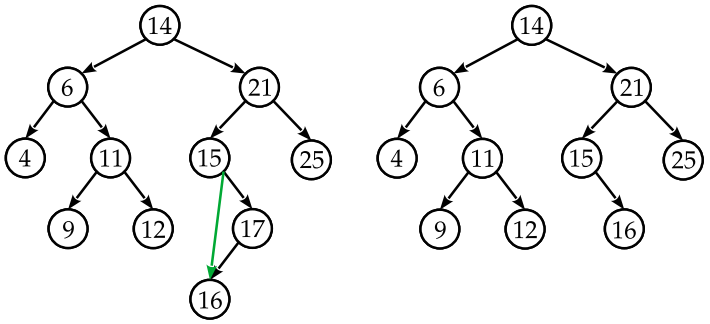


Figure 1.17: Removing values from the binary tree when the removed node has one child.

The removed node has both children

When we remove a node that has both left and right subtrees, we get two loose subtrees that need to be reworked to fit into the binary tree structure. This is more complex than previous two cases. We have two distinct approaches how to remove nodes having both children.

1. **Remove by merging**. First, we need to determine on which side is the removed node. If the removed node is the left child, we merge the right subtree of the removed node. If the removed node is the right child, we merge the left subtree of the removed node. In the merged subtree, we focus on the root node and the leftmost node (having the smallest value) when we merge the right subtree. Similarly, if we merge the left subtree, we focus on the root node and the rightmost node. The merged subtree is inserted between the parent of the removed node, and the sibling of the root node of the merged subtree.

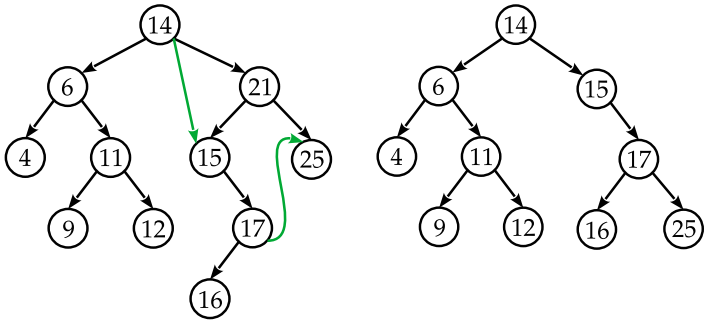


Figure 1.18: Removing values from the binary tree by merging the left subtree of the removed node.

In Figure 1.18 we are removing the node 21. The removed node 21 is the right child of its parent node 14, which means that the merging subtree is the left subtree of the removed node 21, having the root node 15 and the rightmost node 17. We merge this subtree into the rest of the binary tree, so that the merged subtree root node 15 becomes the right child instead of the removed node

21, and the sibling node 25 of the merging subtree root node 15 becomes the right child of the merging subtree rightmost node 17.

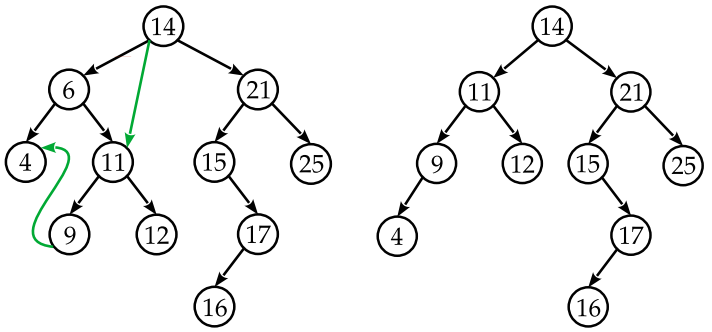


Figure 1.19: Removing values from the binary tree by merging the right subtree of the removed node.

In Figure 1.19 we are removing the node 6. The removed node 6 is the left child of its parent node 14, which means that the merging subtree is the right subtree of the removed node 6, having the root node 11 and the leftmost node 9. We merge this subtree into the rest of the binary tree, so that the merged subtree root node 11 becomes the left child instead of the removed node 6, and the sibling node 4 of the merging subtree root node 11 becomes the left child of the merging subtree leftmost node 9.

2. **Remove by copying**. Instead of removing the node that has the value which needs to be removed, we actually remove a replacement node instead. The replacement node is usually a predecessor or a successor of the node that has the value that needs to be removed. When we remove the replacement node, this becomes one of the simple removal cases where the removed node has no children or has only one child. Remove by copying is minimally changing the structure of the binary tree. We can find a predecessor by searching for the rightmost node in the left subtree, and a successor by searching for the leftmost node in the right subtree.

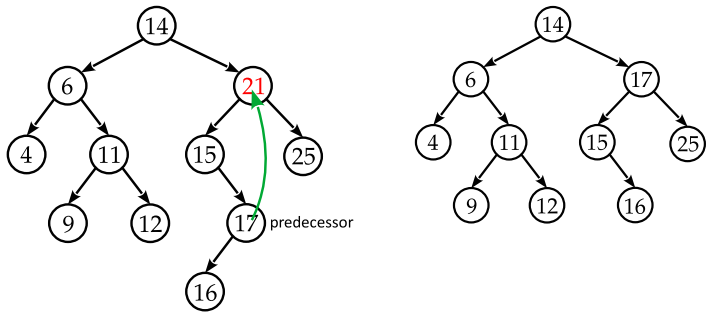


Figure 1.20: Removing values from the binary tree by copying.

In Figure 1.20 we are removing the value 21. We are not removing the node that has the value 21. First, we need to find a direct predecessor to 21, which is the rightmost node of the 21 left subtree. This is the replacement node, having the value 17. We copy 17 into the node with 21, and remove the replacement node. It must be noticed that the replacement node has only one child, having the value 16, which is a simple removal case.

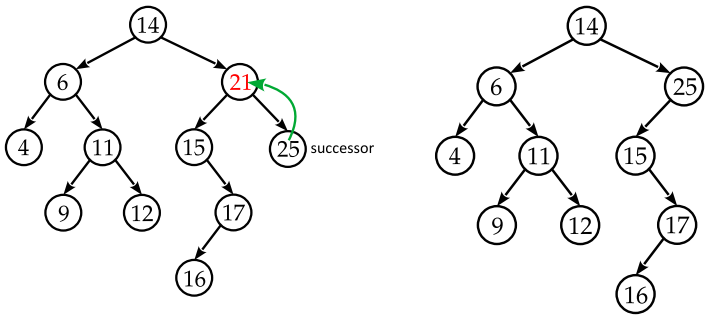


Figure 1.21: Removing values from the binary tree by copying.

In Figure 1.21 we are again removing the value 21. This time we want to find a direct successor to 21, which is the leftmost node of the 21 right subtree. This is the replacement node, having the value 25. We copy 25 into the node with 21, and remove the replacement node, which has no children. Notice that the right subtree of the root node becomes degenerate.

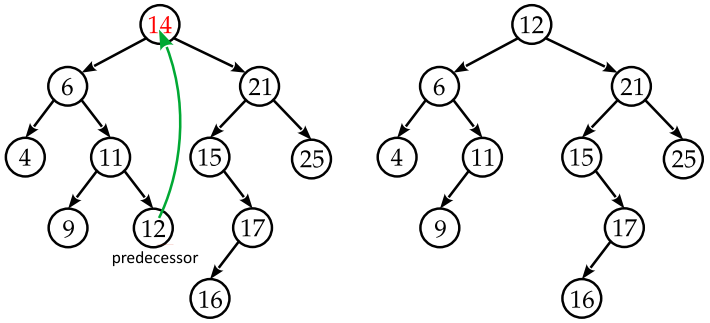


Figure 1.22: Removing values from the binary tree by copying.

In Figure 1.22 we are removing the value 14. A direct predecessor to 14 is the rightmost node of the 14 left subtree. This is the replacement node, having the value 12. We copy 12 into the node with 14, and remove the replacement node. The replacement node has no children, so the removal is simple.

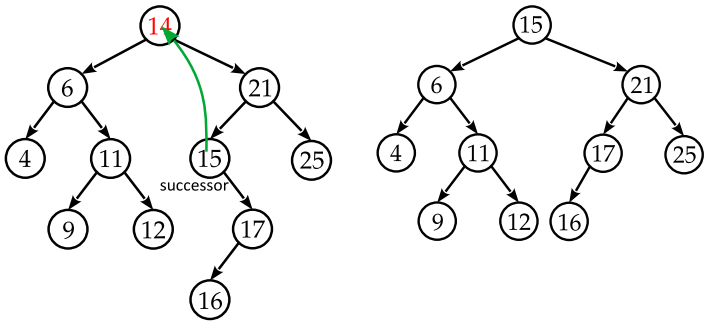
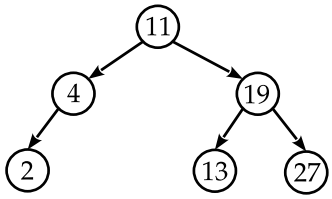


Figure 1.23: Removing values from the binary tree by copying.

In Figure 1.23 we are again removing the value 14. A direct successor to 14 is the leftmost node of the 14 right subtree. This is the replacement node, having the value 15. We copy 15 into the node with 14, and remove the replacement node. The replacement node has one child, which is the root of its subtree.

Exercise 1.3.1. For the following binary tree



perform the following operations

insert 9, insert 15, insert 6, remove 11

When removing elements, do it *by copying*.

Using binary trees to determine algorithm complexity

In the generic binary tree we use ordering (1.8) to determine the binary tree structure and by that searching through it. Whenever the search value v_s is $\leq$ than the node value, we proceed to the left subtree. Otherwise, we proceed to the right subtree. However, this ordering can be something else. For example, if we wrap logical formulas in nodes of the binary tree, then we can construct the ordering in which we move to the left subtree if the result of the node logical formula is *true*, and to the right subtree if the result of the node logical formula is $false$. Such binary tree is called *a decision tree*.

Decision trees can be used to model various logical formulas. Figure 1.25 shows an example of a decision tree that models the logical formula $\overline{A} + \overline{B} \cdot C$. We have three logical variables A, B, and C. The internal nodes are elementary logical expressions. The ordering is according to the result of the logical expression in each node. For example, if the result of the logical expression A is *true*, we move to the left subtree. Otherwise, if the result of the logical expression A is $false$, we move to the right subtree. This is continued until we reach one of the leafs, where we know the value of all logical variables participating in the decision tree logical formula. We can say that leaves comprise all permutations of logical variables participating in the logical formula. Since our *boolean* logical variables can have two states, *true* (1) and $false$ (0), for three variables this gives the total number of permutations as $2^3 = 8$.

Determining the complexity of sorting algorithms

Let us define a set of values we need to sort, for example alphabet letters

$$s = \{c, a, b\} \tag{1.12}$$

```
class SimpleBinaryTree:
  ...
  def remove(self,v):
    if self.root is None: return
    n=self.root.query(v)
    fp=n.P
    c=n.children()
    if c==0 and n.P is None:
      self.root=None
    elif c==0 and n.P is not None:
      if n.P.L is n:
        n.P.setLeftChild(None)
      elif n.P.R is n:
        n.P.setRightChild(None)
    elif c==1 and n.P is None:
      if n.L is not None:
        self.root=n.L.toRoot()
      elif n.R is not None:
        self.root=n.R.toRoot()
    elif c==1 and n.P is not None:
      if n.P.L is n:
        if n.L is not None:
          n.P.setLeftChild(n.L)
        elif n.R is not None:
          n.P.setLeftChild(n.R)
      elif n.P.R is n:
        if n.L is not None:
          n.P.setRightChild(n.L)
        elif n.R is not None:
          n.P.setRightChild(n.R)
    elif c==2: # by copy
      pred=n.L.rightmost()
      val=pred.S
      fp=self.remove(pred.S)
      n.S=val
    return fp
```

Listing 1.9: Binary tree node remove python code, containing all cases and remove by copy for nodes that have both children.

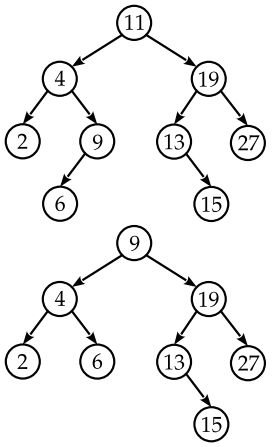


Figure 1.24: Solution of the Exercise 1.3.1. The upper binary tree is after all inserts. The bottom binary tree is after the removal.

We begin by knowing that this set of values having $|s|$ elements, has $\Pi_{i=0}^{n} n^i = 1 * 2 * 3 = n! = 3! = 6$ permutations

$$\mathcal{P} = \{(a,b,c),(a,c,b),(c,a,b),(b,a,c),(b,c,a),(c,b,a)\} \tag{1.13}$$

Sorting is defined as an ordered *sequence*

$$sort(s) = (s_i : s_i \in s) \tag{1.14}$$

satisfying

$$\forall 1 \leq i,j \leq n : i < j, s_i \leq s_j \tag{1.15}$$

The sorted sequence is one of the sequences in the set of all permutations of the input set of values

$$sort(s) \in \mathcal{P} \tag{1.16}$$

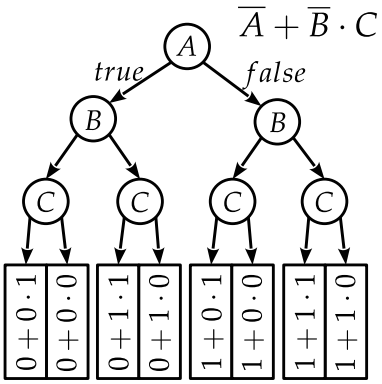


Figure 1.25: Decision tree for the logical formula $\overline{A} + \overline{B} \cdot C$.

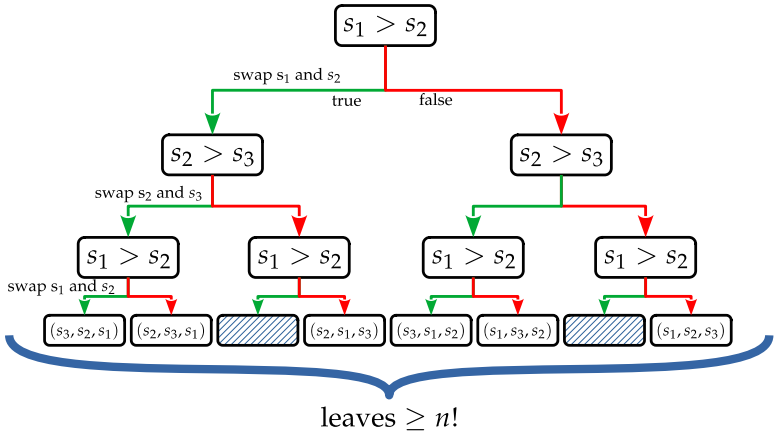


Figure 1.26: Bubble sort decision tree.

For example, we can model *bubble sort* algorithm into a decision tree, which can be seen in Figure 1.26. We deliberately create logical expressions as $s_i > s_{i+1}$, since we want to swap values for s_i and s_{i+1} when any of these logical expressions shows to be true. This represents the swapping action in the bubble sort algorithm. No matter how we populate the input sequence (s_1, s_2, s_3), we get a sorted output sequence after we end in one of the decision tree leaves. We can end up in any of the leaves.

Let us take $(s_1, s_2, s_3) = (c, a, b)$ for the input sequence. In the first node we check $s_1 > s_2$, or $c > a$, which is true. For this reason we swap s_1 and s_2, resulting in (a, c, b), and move to the left subtree. Next, we check $s_2 > s_3$, or $c > b$, which is true. We swap s_2 and s_3, resulting in (a, b, c), and move to the right subtree. Finally, we check $s_1 > s_2$, or $a > b$, which is false. The output sequence is (s_2, s_3, s_1), or (a, b, c) if we correctly reorder elements of the input sequence.

No matter how we form the input sequence, we need to have as many actions as many logical expressions are in the decision tree, which is the same as the tree depth. This means that the complexity of the bubble sort algorithm is

$$O(bubble) = \log_2 n! \tag{1.17}$$

Using Stirling's approximation

$$\log_2 n! = n \log_2 n - n \log_2 e + O(\log_2 n) \tag{1.18}$$

we get

$$O(bubble) = O(n \log_2 n) \tag{1.19}$$

which is the theoretical complexity of sorting algorithms.

1.4 Balancing binary trees

When using binary trees, there is a structural trade-off that we need to be prepared for. This means that a degenerate binary tree, such as in Figure 1.10, can give the search complexity $O(n)$, making the binary tree practically useless. On the other side, there is a perfectly balanced binary tree, which gives the ideal binary tree search complexity $O(\log_2 n)$. Since binary trees are constantly changing, as we insert and remove data, we need to keep the binary tree structure balanced. There are two approaches to the binary tree balancing:

1. Creating a new balanced tree

2. Restructuring an existing binary tree after inserting or removing values to keep it balanced

Creating balanced binary tree from a sorted sequence

One of the solutions is to create a new balanced binary tree from a sorted sequence. We define a set of values that the newly created binary tree must comprise as

$$V = \{3, 7, 1, 15, 11, 21, 24, 14, 9\} \tag{1.20}$$

First, we want to sort the set of values into an ordered sequence, adhering to the ordering (1.8). As we revealed previously, the maximal theoretical sorting complexity is $O(\log_2 n)$. The resulting sorted sequence is

$$V_s = (1, 3, 7, 9, 11, 14, 15, 21, 24) \tag{1.21}$$

which we use to start creating a new balanced binary tree. We take the middle element of the sorted sequence, which is 11 in our case, and create the root node from it. The subsequence left of 11 is then used to create the left subtree, and the subsequence right of 11 is used to create the right subtree. We create subtrees recursively, using the same principle, repeating it until we can create binary tree nodes. This procedure of creating a balanced tree from a sorted sequence (1.21) can be seen in Figure 1.27.

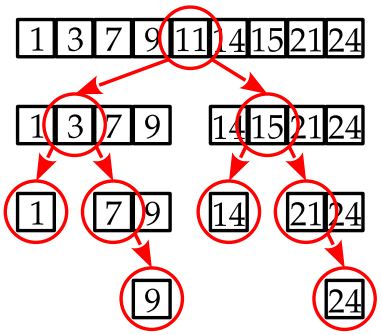


Figure 1.27: Creating balanced binary tree from a sorted sequence

The middle element in a sorted sequence of length n has index $(n \div 2) + (n \,\%\, 2)$, using modular arithmetic. If the length n is *odd*, the left and the right subsequences are of the same length. If the length n is *even*, the left subsequence has one element less then the right subsequence.

Algorithm 1.3: Creating a balanced binary tree from a sorted sequence.

function CREATEBALANCEDTREE(V_s)
  $n \leftarrow |S|$
  if $n > 0$ **then**
    $i \leftarrow (n \div 2) + (n \,\%\, 2)$
    $root \leftarrow$ create node having value $V_s[i-1]$
    $leftChild(root) \leftarrow$CREATEBALANCEDTREE($V_s[0, i-1]$)
    $rightChild(root) \leftarrow$CREATEBALANCEDTREE($V_s[i, n]$)
    return *root*
  else
    return nil

```
def CreateBalBT(Vs):
  n=len(Vs)
  if n>0:
    i=(n // 2)+(n % 2)
    root=Node(Vs[i-1])
    root.setLeftChild(
        CreateBalBT(Vs[0:i-1]))
    root.setRightChild(
        CreateBalBT(Vs[i:n]))
    return root
  else: return None
```

Listing 1.10: Python code for creating binary tree from sorted sequence.

The final creation complexity comprises the sorting step $O(\log_2 n)$ and the creation recursion which has the complexity $O(n)$. Such creation of a balanced binary tree from a sorted sequence cannot be used always. Sometimes we are not having all the input data at glance, which implies subsequent addition (insertion) and removal of values. While adding (inserting) and removing values are not complex operations, they can lead to a degeneration of the underlying binary tree structure. Once degenerated, to retain the binary tree efficiency we need either to create the binary tree from scratch or to rebalance it. This is the reason why this creation procedure is usually used once, to create an initially balanced binary tree. After that, it makes only sense to rebalance the binary tree as we add or remove values.

Day-Stout-Warren (DSW) algorithm

The Day-Stout-Warren (DSW) algorithm is one of the main binary tree balancing algorithms. It is used to globally balance any binary tree, by degenerating it into a right or a left backbone, and then balancing it using elementary restructuring actions on the binary tree structure in form of node rotations.

Right rotation

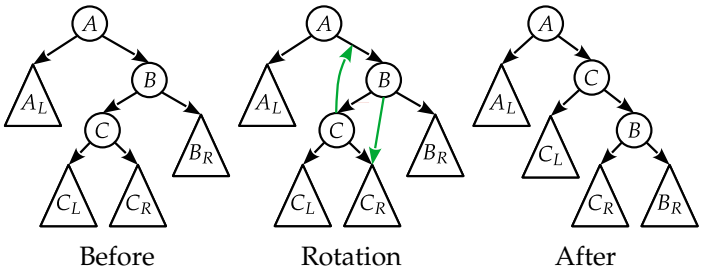


Figure 1.28: The right rotation of the node C around the node B.

Let us define a small portion of a binary tree, as on the left side of Figure 1.28. We have 3 nodes A, B, and C in the specific ordering. All 3 nodes have both subtrees, which means that they all have both children. We want to perform the right rotation of the node C around the node B. First, we want to observe the ordering of their subtrees. Using the ordering (1.8), we can conclude

$$v(A_L) < v(A) < v(C_L) < v(C) < v(C_R) < v(B) < v(B_R) \qquad (1.22)$$

This ordering of elements must be preserved after the rotation action. The rotation action can be seen in the center of Figure 1.28. We move (rotate) the node C between nodes A and B. C becomes the right child of the node A, and B becomes the right child of the node C. Before the rotation action, we had C_R as the right subtree of the node C. To retain the same ordering as in (1.22), the subtree C_R must become the left subtree of the node B. The final result of the rotation action can be seen on the right side of Figure 1.28.

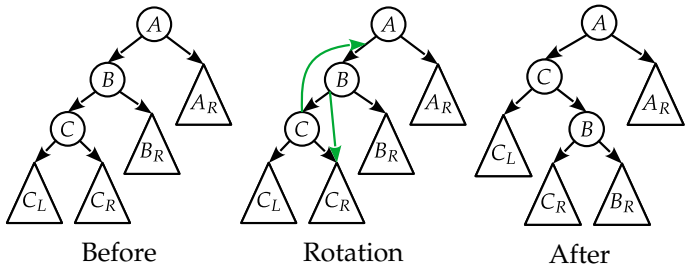


Figure 1.29: The right rotation of the node C around the node B.

If the node B becomes the left child of the node A, the right rotation of the node C around the node B remains the same, which can be seen in Figure 1.29. The rule is that the rotated pair of nodes remains on the same side of the parent node. It means, if the node B was the right child of the node A, then the node C must also be the right child after the rotation action, and *vice versa*.

If the node B has no parent, which could happen when B is the root node for the whole binary tree, then C becomes the new parent node of the node B after the right rotation around the node B, and the new root node for the binary tree. This is the most generic rotation description, and can be seen in Figure 1.30.

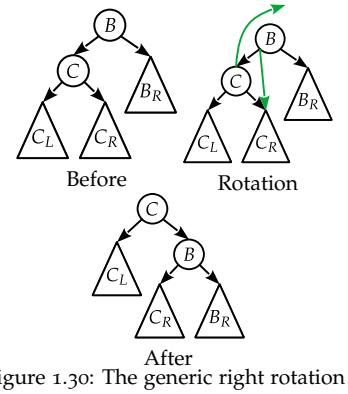


Figure 1.30: The generic right rotation of the node C around the node B.

It must be noticed that we **right rotate the left child around the node**. The node C in all previous cases is the left child of the node B.

procedure RIGHTROTATION(A, B)
  $C \leftarrow leftChild(B)$
  if $C = nil$ **then**
    return
  if $A \neq nil$ **then**
    if $B = leftChild(A)$ **then**
      $leftChild(A) \leftarrow C$
    if $B = rightChild(A)$ **then**
      $rightChild(A) \leftarrow C$
  $t \leftarrow rightChild(C)$
  $rightChild(C) \leftarrow B$
  $leftChild(B) \leftarrow t$

Algorithm 1.4: Binary tree right rotation.

The right rotation action is summarized in Algorithm 1.4. One of input parameters is the current parent A of the node B, which can be *nil* in case when B is the overall root node for the binary tree.

```
class Node:
  ...
  def rightrotate(self,tree):
    C=self.L
    if C is None: return
    A=self.P
    if A is not None:
      if A.L is self:
        A.setLeftChild(C)
      if A.R is self:
        A.setRightChild(C)
    else: tree.root=C.toRoot()
    t=C.R
    C.setRightChild(self)
    self.setLeftChild(t)
```

Listing 1.11: Python code for binary tree right rotation.

Left rotation

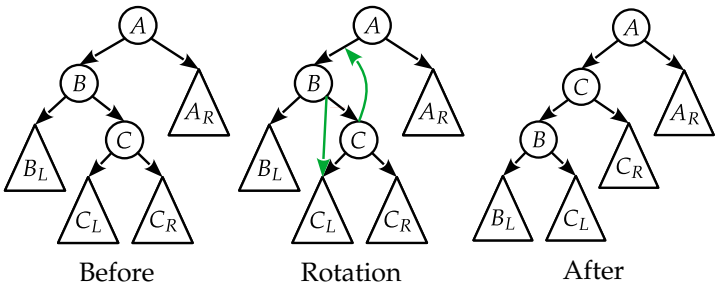


Figure 1.31: The left rotation of the node C around the node B.

The left rotation is the inverse operation of the right rotation. We again have 3 nodes A,B, and C. Looking on the left side of Figure 1.31, we can see the same pattern as the final result of the right rotation in Figure 1.29. The node C is rotated left around the node B, and placed between nodes A and B. After that, the node C becomes the new child node of the node A, and the node B becomes the left child of the node C. The previous left child of the node C is moved to be the new right child of the node B. The left rotation action is explained in the center of Figure 1.31. The final result of the left rotation is seen on the right side of Figure 1.31, which is the same as the starting binary tree on the left side of Figure 1.29. This also proves that the left rotation is the inverse of the right rotation, and *vice versa*. We need to pay attention to the ordering of elements

$$v(B_L) < v(B) < v(C_L) < v(C) < v(C_R) < v(A) < v(A_R) \qquad (1.23)$$

which needs to be preserved by the left rotation action.

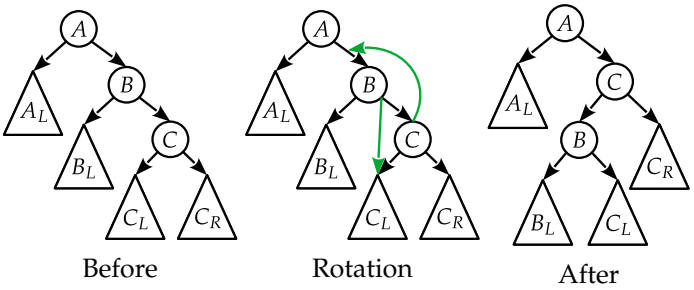


Figure 1.32: The left rotation of the node C around the node B.

If the node B becomes the right child of the node A, the left rotation of the node C around the node B remains the same, which can be seen in Figure 1.32. The rule is that the rotated pair of nodes remains on the same side of the parent node. It means, if the node B was the right child of the node A, then the node C must also be the right child after the rotation action, and *vice versa*. The left rotation action in Figure 1.32 is the inverse of the right rotation action in Figure 1.28.

If the node B has no parent, which could happen when B is the root node for the whole binary tree, then C becomes the new parent node of the node B after the left rotation around the node B , and the new root node for the binary tree. This is the most generic rotation description, and can be seen in Figure 1.33.

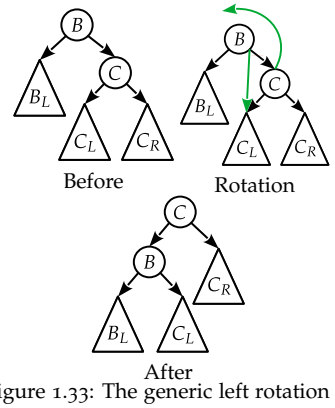


Figure 1.33: The generic left rotation of the node C around the node B.

It must be noticed that we **left rotate the right child around the node**. The node C in all previous cases is the right child of the node B.

```
procedure LeftRotation(A, B)
    C ← rightChild(B)
    if C = nil then
        return
    if A ≠ nil then                    ▷ The same as in the right rotation
        if B = leftChild(A) then
            leftChild(A) ← C
        if B = rightChild(A) then
            rightChild(A) ← C
    t ← leftChild(C)
    leftChild(C) ← B
    rightChild(B) ← t
```

Algorithm 1.5: Binary tree left rotation.

```
class Node:
  ...
  def leftrotate(self,tree):
    C=self.R
    if C is None: return
    A=self.P
    if A is not None:
      if A.L is self:
        A.setLeftChild(C)
      if A.R is self:
        A.setRightChild(C)
    else: tree.root=C.toRoot()
    t=C.L
    C.setLeftChild(self)
    self.setRightChild(t)
```

Listing 1.12: Python code for binary tree left rotation.

The left rotation action is summarized in Algorithm 1.5. One of input parameters is the current parent A of the node B, which can be *nil* in case when B is the overall root node for the binary tree.

Generating backbones

Definition 1.9 (Binary tree backbone)**.** Binary tree backbone is a fully degenerated binary tree, whose nodes have only left or right children. A backbone having only left children is called a *left backbone*, while the opposite one is called a *right backbone*

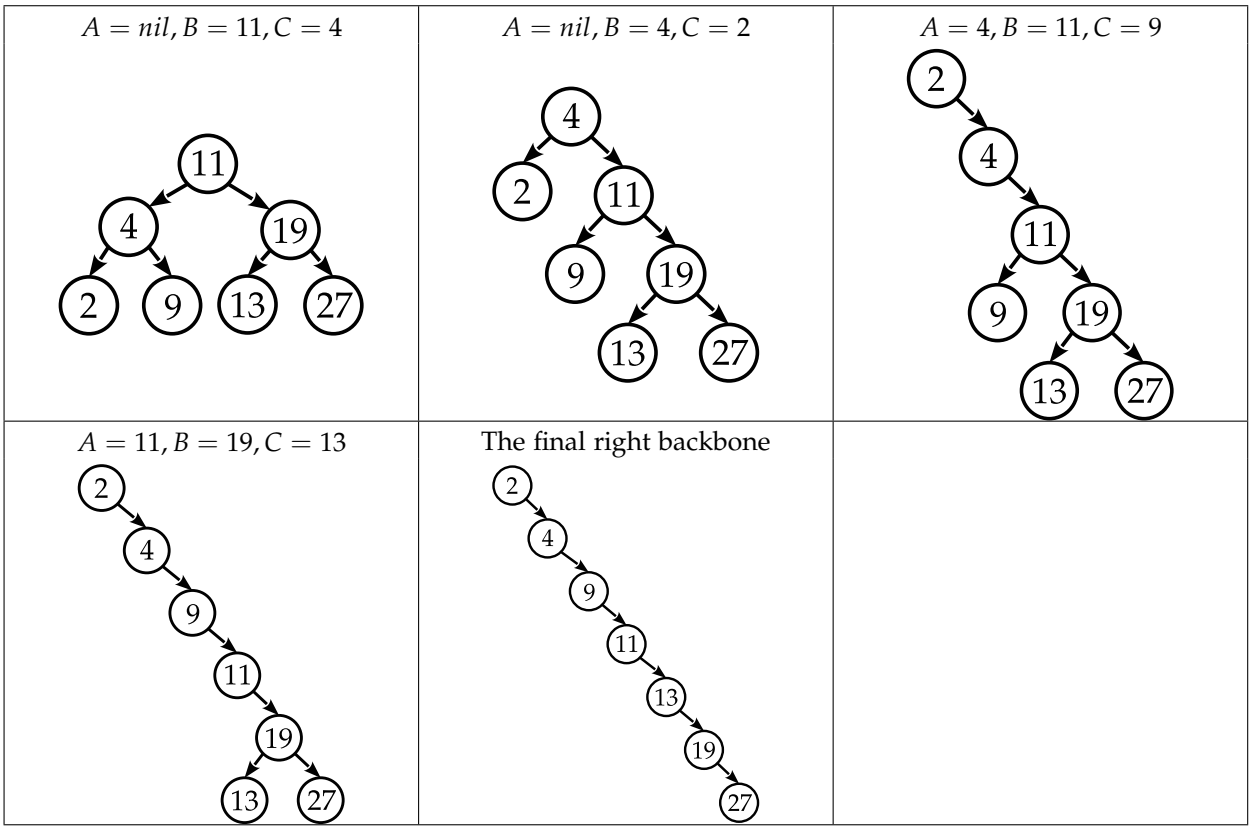
An example of the right backbone can be seen in Figure 1.10. A backbone can be generated from any binary tree by using left or right rotations. We traverse through the binary tree and right rotate all left children, which creates a right backbone.

```
procedure RightBackbone(root)
    B ← root
    A ← nil
    while B ≠ nil do
        C ← leftChild(B)
        if C ≠ nil then
            RightRotate(A, B)
            if A = nil then
                root ← C
            B ← C
        else
            ▷ Descending right to the first node that has the left child
            A ← B
            B ← rightChild(B)
```

Algorithm 1.6: Algorithm for creating a right backbone.

```
class SimpleBinaryTree:
  ...
  def rightbackbone(self):
    B=self.root
    while B is not None:
      C=B.L
      if C is not None:
        B.rightrotate(self)
        B=C
      else:
        B=B.R
```

Listing 1.13: Python code for creating a right backbone.



Creating a backbone requires $O(n)$. However, we need to notice several details about it. Creating a right backbone from a left backbone is not necessary, as DSW can work on either of them. Also, creating a backbone from a backbone is the worst case scenario, since it would require to rotate $n-1$ nodes. The middle case scenario would be creating a backbone from a perfectly balanced binary tree, which makes no sense. Binary tree balancing would be happening for cases where we have a unbalanced binary tree that is somewhere between a perfectly balanced tree and a backbone.

When looking from the binary tree root node, we can assess the global tree balance by looking at left and right subtrees heights $h(L)$ and $h(R)$. If the left subtree is deeper than the right subtree, we have a binary tree that leans to the left, making its left backbone closer, measured by the number of required rotations, which is visible in Figure 1.34a. In such case we want to create the left backbone. If the right subtree is deeper than the left subtree, we have a binary tree that leans to the right, making its right backbone closer, measured by the number of required rotations, which is visible in Figure 1.34b.

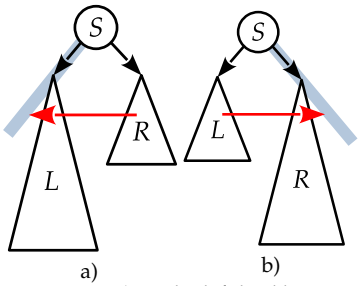


Figure 1.34: a) To the left backbone when $h(L) > h(R)$, b) to the right backbone when $h(L) < h(R)$.

Once we got either a left or a right backbone, we need to create a balanced binary tree by using counter rotations. **Meaning, if we got a right backbone, we need to use left rotations** to get certain nodes from the backbone back into the balanced binary tree structure. How much left rotations we perform, depends on the size of the binary tree. *Vice versa*, if we got a left backbone, we need to use right rotations to get the balanced binary tree. Throughout the rest of this Section, we use the right backbone and left rotations for the binary tree balancing.

Left rotations are performed in two distinct steps. In the first step we form leaves of the balanced binary tree. Since we want to create a **complete** balanced binary tree, this means that only the last level of the binary tree does not need to be populated fully and must be filled from the left side. After that, we create inner nodes in the same manner, repeating left rotations for all inner layers. The DSW algorithm is

procedure DSW($tree, n$)
  $h \leftarrow \lceil \log_2(n+1) \rceil$
  $i \leftarrow 2^{h-1} - 1$
  perform $n - i$ rotations of every second node from the root
  while $i > 1$ **do**
    $i \leftarrow \lfloor i/2 \rfloor$
    perform i rotations of every second node from the root

Algorithm 1.7: DSW algorithm.

For $n = 7$, we have $h = \lceil \log_2 8 \rceil = 3$. The total number of internal nodes is $i = 2^2 - 1 = 3$ and leaves $l = n - i = 7 - 3 = 4$. In the first step, we use left rotations to create leaf nodes. In the case from the previous table, we must rotate 4 rotations of the every second node from the top of the right backbone, marked red on the left side in Figure 1.35. The right side of the same Figure shows the final result of the rotations, with red markings of the nodes that will be future leaves.

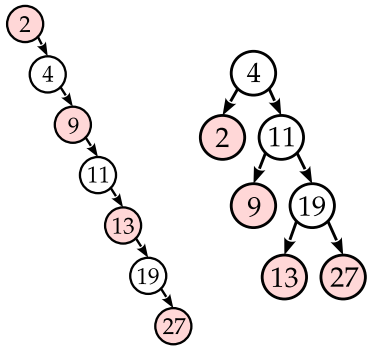


Figure 1.35: DSW rotations to create leaves. The left rotation of the last node in the backbone has no effect.

We continue by rotating the inner nodes, starting from the level above leaves. The number of rotations must be equal to the current level nodes decreased by 1, the node that remains in the backbone slope. We iterate in a while loop, always decreasing the level we are rotating, until we reach the top level where we have only the root node, which we do not rotate since it must remain in the backbone slope. A small example can be seen in Figure 1.36. Green marked nodes are the ones we left rotate, eventually forming the left side of inner nodes in the balanced binary tree.

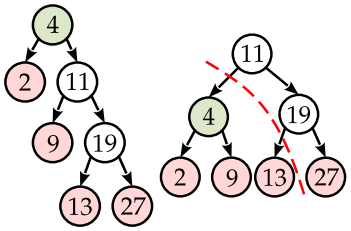


Figure 1.36: DSW rotations to create inner nodes.

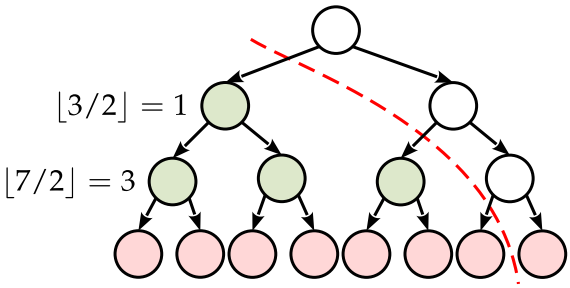


Figure 1.37: A generic DSW rotations.

When looking at the generic example in Figure 1.37, we can see how the while loop works. At the beginning, we have $i = 7$ inner nodes. In the first while loop iteration, we need to perform $\lfloor i/2 \rfloor = \lfloor 7/2 \rfloor = 3$ left rotations, which creates the left side of the level 3 in the balanced binary tree. In the second iteration of the while loop, we need to perform $\lfloor i/2 \rfloor = \lfloor 3/2 \rfloor = 1$ left rotation, which creates the

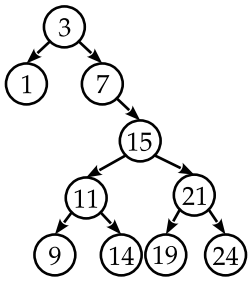
```
def DSW_rotates(tree,rcount):
  node,A=tree.root,None
  while rcount>0:
    A1=node.R
    node.leftrotate(tree)
    rcount=rcount-1
    A=A1
    node=A.R

def DSW(tree,n):
  if tree.root is None: return
  tree.rightbackbone()
  h=ceil(log(n+1,2))
  i=pow(2,h-1)-1
  DSW_rotates(tree,n-i)
  while i>1:
    i=floor(i/2)
    DSW_rotates(tree,i)
```

Listing 1.14: DSW python code that creates the right backbone and uses left rotations to balance the binary tree.

left side of the level 2 in the balanced tree. This ends our balanced tree creation. All unmarked nodes are those that were originally in the starting backbone and were not rotated in the DSW algorithm.

In the presented DSW algorithm we perform $n - h - 1$ left rotations. Since $h \leftarrow \lceil \log_2 n + 1 \rceil$, we can easily take that the complexity of the DSW algorithm is $O(n)$. This sounds quite expensive, especially knowing that DSW algorithm performs global balancing, which means we restructure the whole binary tree each time.

Exercise 1.4.1. Using DSW algorithm, balance the following binary tree. Why are leaves of the DSW balanced binary tree filling the last level from the left side?



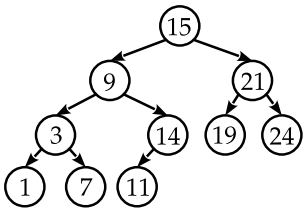


Figure 1.38: Solution of the Exercise 1.4.1.

Adelson-Velski-Landis (AVL) self-balancing tree

When we think about the lifecycle of a binary tree, we see them as highly dynamic rather than static structures. This means that in its lifetime, a binary tree can undergo big number of insertions and removals, which is directly affecting usefulness of the DSW algorithm. Also, for each insertion or removal we expect to introduce only limited and local disbalance in the binary tree, which can be solved by a local balancing approach. There are numerous algorithms for local balancing of binary trees, starting from the Adelson-Velski-Landis (AVL) self-balancing tree.

Before inserting or removing a value from a binary tree, we assume that the binary tree is balanced according to Definition 1.7. It means that insertions and removals can disbalance the binary tree only locally. For measuring the effect of the insertion or removal on the local balance of the binary tree, we define *a balance factor* for each node as

$$BF(S) = h(R) - h(L) \tag{1.24}$$

which is the height difference between the right and left subtree of the node. To call the AVL tree balanced, we need each node to have a balance factor of either -1, 0, or 1, which is according to the basic definition of a balanced binary tree. A node that has a balance factor of 0 is perfectly balanced. However, a node that has a balance factor -1 or 1 can potentially cause problems in higher levels of the AVL tree. Any node having a balance factor -2 or 2 represents a locally disbalanced AVL tree, which requires restructuring.

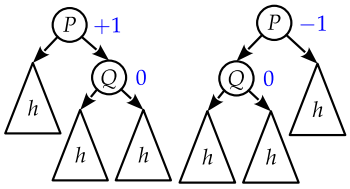


Figure 1.39: Balanced AVL trees and their nodes balance factors.

We start by having a balanced AVL tree as in Figure 1.39. As soon as we insert a new value, it becomes a new leaf in one of the subtrees and potentially increases the height of the subtree, which leads to a local disbalance in the AVL tree. We identify 2 main disbalance cases:

1. **Straight case:** We start by having the AVL tree on the left side of Figure 1.40, where we add a new value (a new leaf) to the right subtree of the node Q.

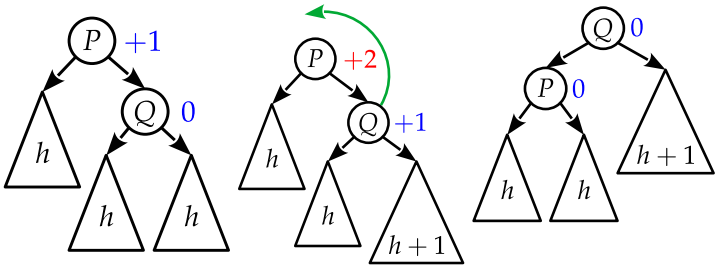


Figure 1.40: Adding a new value to the right subtree of the node Q causes an unbalance on the node P.

After the insertion, the left subtree of the node P has 2 levels less than the right subtree. We can correct it by moving one node on top of the left subtree, simultaneously removing one node from the top of the right subtree. This can be achieved by performing a left rotation of the node Q around the node P, which can be seen in the center of Figure 1.40. After the rotation, we get the node P on the left side of the AVL tree, while the node Q becomes the new root node. After the balance factor recalculation, we can see that the AVL tree is now perfectly balanced, which can be seen in the right side of Figure 1.40.

```
class Node:
  ...
  def updateHeight(self):
    l,r=0,0
    if self.L is not None:
      l=self.L.h
    if self.R is not None:
      r=self.R.h
    self.h=1+max(l,r)
  def bal_factor(self):
    l,r=0,0
    if self.L is not None:
      l=self.L.h
    if self.R is not None:
      r=self.R.h
    return r-l
```

Listing 1.15: Python balance factor calculation.

We can also start by having the inverted case, as in the left side of Figure 1.41, where we add a new value (a new leaf) to the left subtree of the node Q. In this case, all balance factors are of negative values since left subtrees are of higher depth than right subtrees.

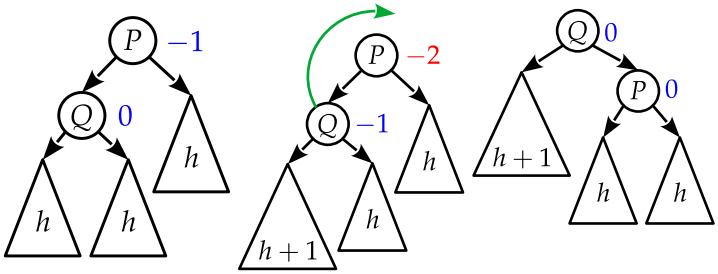


Figure 1.41: Adding a new value to the left subtree of the node Q causes an unbalance on the node P.

We solve this inverted case by performing a right rotation of the node Q around the node P, as seen in the center of Figure 1.41. The final result of the rotation is perfectly balanced AVL tree, having all balance factors of 0, which can be seen in the right side of Figure 1.41.

The straight case can be identified by looking at the node balance factor between parent-child node pairs. If a node has the balance

factor of $+2$ and his right child has the balance factor of 1 (**both are positive**) or 0, this is a right straight case. If a node has the balance factor -2 and his left child has the balance factor of -1 (**both are negative**) or 0, this is a left straight case. Solving a straight case **is done by one rotation**, which is **a left rotation** around the node having the balance factor of 2 **in a right straight case**, or **a right rotation** around the node having the balance factor of -2 **in a left straight case**.

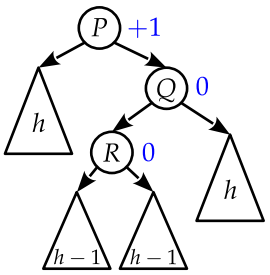


Figure 1.42: An example of a balanced AVL tree.

2. **Broken case:** We start from the balanced AVL tree in Figure 1.42. The left subtree of the node Q is divided into the root node R and the child subtrees of height $h-1$. We insert a new value into the left subtree of the node Q. This new value can be inserted to either of the child subtrees of the node R. If inserted to the right subtree of the node R, we have the following case.

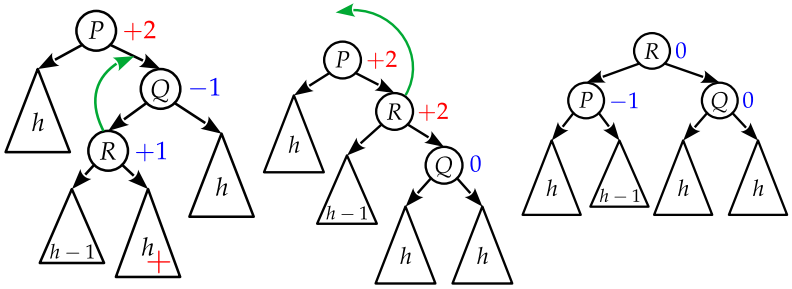


Figure 1.43: Inserting a new value to the right subtree of the node R causes an unbalance on the node P.

Unlike the straight case, broken cases have the balance factor of -1 for the node Q and the balance factor of $+2$ for the node P, which means that the left subtree of the node Q is higher than the right subtree. This prevents us from performing a simple left rotation of the node Q around the node P, as the higher left subtree of the node Q would now end in the left subtree of the AVL tree root node. Together with nodes P and R in the left subtree of the AVL root node, this subtree would be of height $h+2$, which would flip the AVL root node balance factor to -2, and would not solve the disbalance we introduced by the insertion. Instead, first we do a right rotation of the node R around the node Q, as seen in the left side of Figure 1.43. The result, seen in the center of Figure 1.43 is a variant of the straight case, where we do a left rotation of the node R around the node P. The smaller left subtree of the node R now ends in the left subtree of the AVL tree root node. The height of the left subtree of the AVL tree root node is now higher only by 1, or by the node P. This balances the AVL tree, as seen in the right side of Figure 1.43.

Inserting a new value to the left subtree of the node R, as in Figure 1.44, makes the same broken case where the balance factor of the node P is $+2$ and the balance factor of the node Q is -1. We solve this case using the same double rotation as in the previous case, with slightly distinct result, as seen in the right side of Figure 1.44.

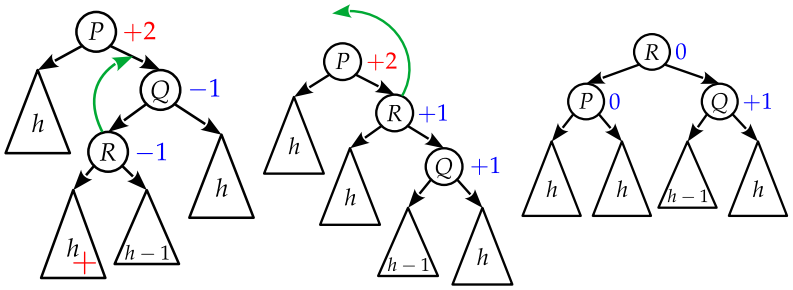


Figure 1.44: Inserting a new value to the left subtree of the node R causes an unbalance on the node P.

We can also start from the balanced AVL tree in Figure 1.45, which is horizontally flipped variant of the AVL tree in Figure 1.42. The right subtree of the node Q is divided into the root node R and the child subtrees of height $h-1$. We insert a new value into the right subtree of the node Q. This new value can be inserted to either of the child subtrees of the node R. If inserted to the left subtree of the node R, we have the following case.

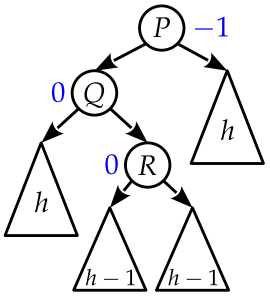


Figure 1.45: An examples of a balanced AVL tree.

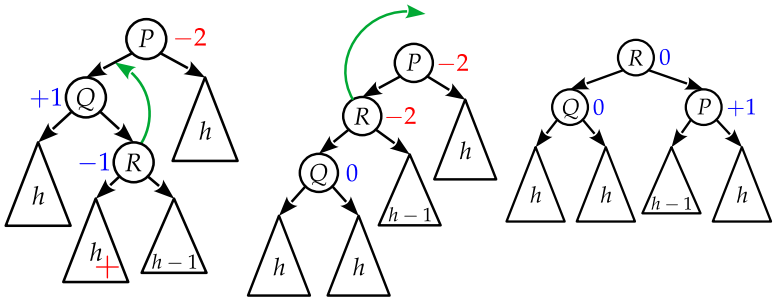


Figure 1.46: Inserting a new value to the left subtree of the node R causes an unbalance on the node P.

This case is solved inversely from the cases in Figures 1.43 and 1.44. We again use double rotation. First is a left rotation of the node R around the node Q, followed by a right rotation of the node R around the node P. The final result can be seen in the right side of Figure 1.46.

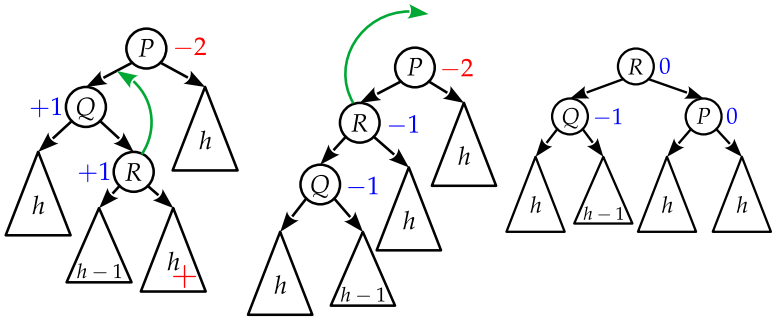


Figure 1.47: Inserting a new value to the right subtree of the node R causes an unbalance on the node P.

If we insert a new value to the right subtree of the node R, we get a similar case as in Figure 1.46, which is again solved by the same double rotation.

The broken case can be identified by looking at the node balance factor between parent-child node pairs. If a node has the balance factor of $+2$ and his right child has the balance factor of -1 (**the parent is positive and the child is negative**), this is a right broken case. If a node has the balance factor of -2 and his left child has the balance factor of $+1$ (**the parent is negative and the child is**

positive), this is a left broken case. Solving a broken case **is done by a double rotation**.

For the right broken case, as seen in Figures 1.43 and 1.44, first we perform a right rotation of the node R around the node Q, followed by a left rotation of the node R around the node P.

For the left broken case, as seen in Figures 1.46 and 1.47, first we perform a left rotation of the node R around the node Q, followed by a right rotation of the node R around the node P.

Removing values

When removing values from the AVL tree, we usually do it by using the *remove by copy* approach, because it doesn't change the structure of the AVL tree dramatically. In such case, we physically remove one of the leaves, which brings us to the same position as with inserting new values, but this time we get a decrease of a subtree height. Remember, we do not remove the node that has the removal value, instead we remove a replacement node, which is always a leaf. In the case of removal we can still identify the 4 basic disbalance cases we introduced when inserting new values. However, since the removal of values decreases subtree height, identifying the appropriate disbalance case is inverse as in the insertion of values.

For example, we can have the same starting AVL tree as in Figure 1.40 for the right straight case. By removing a node from the

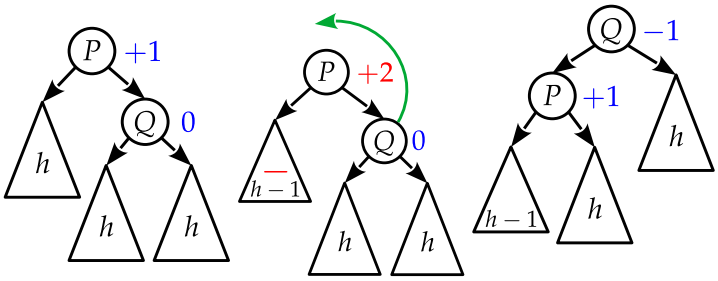


Figure 1.48: Removing value from the left subtree of the node P causes the right straight disbalance case.

left subtree of the node P, as in the center of Figure 1.48, we get the similar disbalance as in the right straight case for the insertion. We solve the disbalance using one left rotation, intended to be used for the right straight case. Although the result is not a perfectly balanced AVL tree as in Figure 1.40, having balance factors in $\{-1, +1\}$ is sufficient according to the balanced binary tree definition.

Exercise 1.4.2. Identify subtrees of the AVL tree whose nodes need to be removed to get the left straight case and broken cases. After we perform rotations to balance the AVL tree, what are the resulting AVL tree balance factors differences between removals and insertions?

The AVL balancing algorithm

We start either by inserting a new node or by removing a replacement node (*remove by copy*) from the AVL tree structure. First, we

need to know the direct parent n_p of the affected (inserted or removed) node n_a. We begin from this starting parent node n_p and ascend up to the AVL tree root node *root*. For each node on the path p_{avl} between n_p and *root*, we need to calculate its balance factor. If any of the nodes balance factors is either -2 or $+2$, we need to identify the involved disbalance case and appropriately balance this part of the AVL tree.

function AVLDETECTROTATE(n)
  if $balanceFactor(n)$ is $+2$ **then**
    $n_1 \leftarrow rightChild(n)$
    if $balanceFactor(n_1)$ is 0 or $+1$ **then**
      left rotate n_1 around n
    if $balanceFactor(n_1)$ is -1 **then**
      $n_2 \leftarrow leftChild(n_1)$
      right rotate n_2 around n_1
      left rotate n_2 around n
  else
    $n_1 \leftarrow leftChild(n)$
    if $balanceFactor(n_1)$ is 0 or -1 **then**
      right rotate n_1 around n
    if $balanceFactor(n_1)$ is $+1$ **then**
      $n_2 \leftarrow rightChild(n_1)$
      left rotate n_2 around n_1
      right rotate n_2 around n
procedure AVLBALANCE(n)
  $p \leftarrow parent(n)$
  if $balanceFactor(n)$ is -2 or $+2$ **then**
    AVLDETECTROTATE(n)
  if p is not *nil* **then**
    AVLBALANCE(p)

Algorithm 1.8: AVL balancing algorithm.

```
class AVLTree(SimpleBinaryTree):
  def updateHeights(self,l):
    for n in l:
      n.updateHeight()
  def AVLDetectRotate(self,n):
    if n.bal_factor()==2:
      n1=n.R
      n1bf=n1.bal_factor()
      if n1 is not None
          and n1bf in [0,1]:
        n.leftrotate(self)
        self.updateHeights([n,n1])
      elif n1 is not None
          and n1bf==-1:
        n1.rightrotate(self)
        n.leftrotate(self)
        self.updateHeights([n1,n,n.P])
    else:
      n1=n.L
      n1bf=n1.bal_factor()
      if n1 is not None
          and n1bf in [0,-1]:
        n.rightrotate(self)
        self.updateHeights([n,n1])
      elif n1 is not None
          and n1bf==1:
        n1.leftrotate(self)
        n.rightrotate(self)
        self.updateHeights([n,n1,n.P])
  def AVLBalance(self,n):
    n.updateHeight()
    p=n.P
    if n.bal_factor() in [-2,2]:
      self.AVLDetectRotate(n)
    if p is not None:
      self.AVLBalance(p)
  def insert(self,v):
    n=super().insert(v)
    if n is not None:
      self.AVLBalance(n)
  def remove(self,v):
    n=super().remove(v)
    if n is not None:
      self.AVLBalance(n)
```

Listing 1.16: Python AVL tree.

Updating node heights and calculating balance factor

The most tricky part of the AVL balancing algorithm is to calculate balance factors for the tree nodes. Doing it in the *wrong way* can significantly increase insertion and removal complexities. For example, we can do it in a naïve way, using a recursion to calculate the maximal height for each node on the path p_{avl}. The recursion descends down the AVL tree, to find a maximally distant leaf node. Doing it for each node in the path p_{avl} significantly increases the complexity, not to mention repeating the same calculations for descendant nodes.

The other way is to store the current height of each node. All leafs are of height 1, which means that the default height value for all newly inserted nodes must be 1. As we traverse the path p_{avl}, we

update the height of each node as

$$\forall n \in p_{avl} : \\ height(n) = 1 + \max(height(leftChild(n)), height(rightChild(n))) \tag{1.25}$$

Such iterative calculation of the node height can be only disrupted when we perform balancing rotations. By doing balancing rotations, we need to update heights for all rotation participating nodes. Since all rotation participating nodes remain *relatives* even after the rotation, we start by updating descendants first. This means that in broken cases we need to update nodes Q and P first, followed by the R node. For straight cases, we update the Q node first, followed by the P node. After updating the rotation participating nodes, we continue ascending the p_{avl} path. Such an incremental approach does not introduce additional complexity to the AVL tree insertions and removals.

The search complexity of the AVL tree remains the same as in the basic binary tree $O(\log_2 n)$. The insertion and removal complexity is not much different. After searching the node to remove, or the place where to insert a new node, we need to ascend back to the AVL tree root using the same search path, which gives the complexity $O(2\log_2 n)$.

Finally, due to the allowed balance factor values and rotations for disbalance cases, the theoretical limits of the AVL tree depth are

$$\log_2(n+1) \leq h \leq 1.44\log_2(n+2) - 0.328 \tag{1.26}$$

The proof can be found in [Drozdek, 2012].

Exercise 1.4.3. In an initially empty AVL tree insert the following numbers

$$16, 29, 18, 34, 26, 15, 45, 33, 6, 37, 49, 48, 40 \tag{1.27}$$

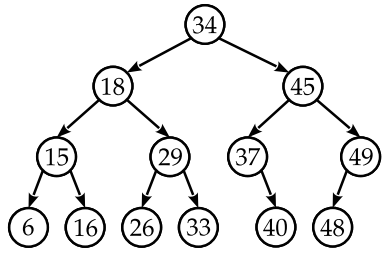


Figure 1.49: The solution of the Exercise 1.4.3.

Exercise 1.4.4. From the AVL tree, result of Exercise 1.4.3, remove the following numbers

$$16, 45, 37 \tag{1.28}$$

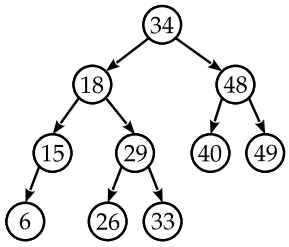


Figure 1.50: The solution of the Exercise 1.4.4.

Exercise 1.4.5. How can we add another number 6 to the AVL tree, result of Exercise 1.4.3? What is the required ordering of the AVL tree structure for duplicate values?

1.5 *B-trees*

We begin by defining Multiway trees

Definition 1.10 (Multiway tree)**.** A multiway tree (an M-tree) is a tree that complies the following properties:

1. Each node has $m-1$ values and m children. We say that the tree is of m-th degree,
2. Node values are sorted ascending,
3. Values in the node first i children are smaller than the i-th value of the node,
4. Values in the node last $m-i$ children are greater than the i-th value of the node.

A binary tree is a 2nd degree M-tree, according to the previous definition. An example of a 3rd degree M-tree can be seen in Figure 1.51.

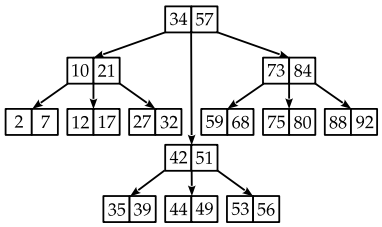


Figure 1.51: An example of an M-tree of the 3rd degree.

When implementing an M-tree, we need to think of each node as a double linked sorted list of node values. It can be implemented otherwise as well. Each value itself is a *node* that can have a left (smaller values) child and a right (greater values) child, which is similar to binary trees. Due to the M-tree searching algorithm, we avoid having duplicate pointers, meaning that each value in an M-tree node linked list has defined the left child, except the last value that has both children. A pointer to a child points to the start of the child's node linked list. An example of such combination of double linked lists and tree structure that constitutes the M-tree can be seen in Figure 1.52.

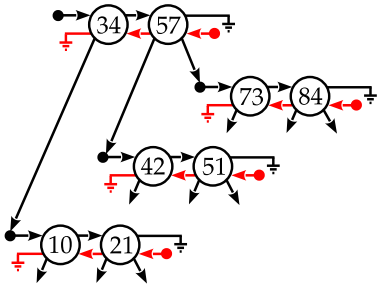


Figure 1.52: An internal implementation of a 3rd degree M-tree by using double linked lists of values.

Definition 1.11 (B-tree)**.** A B-tree is an M-tree of m-th degree that complies with the following additional requirements:

1. The root of the B-tree must have at least 1 value and 2 child nodes, unless it is the only node in the B-tree,
2. Internal nodes must be at least 50% populated. For the B-tree of m-th degree, each inner node needs to have at least $k-1$ values and k children, where $\lceil m/2 \rceil \leq k \leq m$,
3. Leaves must be at least 50% populated. For the B-tree of m-th degree, each leaf needs to have at least $k-1$ values and no child nodes, where $\lceil m/2 \rceil \leq k \leq m$,
4. All leaves need to be at the same level of the B-tree, which makes the B-tree perfect tree, completely filled in all levels, having the balance factor in all nodes 0. If there are pointers to the external source of data in leaves, then we call it a **B$^+$-tree**, which is regularly used in databases.

Notice that node being at least 50% populated refers to the number of pointers, or child nodes. For a B-tree having degree=4 this means at least one value and two child nodes is what we consider 50% populated.

The main case that drove the creation of B-trees was organizing data into a segments, such as in the relational databases. For that

purpose, B$^+$-trees are more used than ordinary B-trees. Let us imagine a sequence of data, organized into segments. This organization was imposed by the structure of magnetic hard drives, which had data organized into sectors. Magnetic medium in a hard drive was divided into an even number of sectors, where each sector was a subdivision of a track on the medium. Each sector stored an equal amount of data and was read in its entirety, which means we could read minimally one sector from the hard drive, and it was 512, 1024, or more bytes. This was done to minimize the read/write magnetic head movements, and such way of reading was the fastest possible. For that reason, a relational databases data store was divided into segments that comprised several sectors on the hard drive. The degree of the B$^+$-tree was directly related to the size of the segment, which means that each leaf of the tree covered one segment of the database data store. Of course, the B$^+$-tree was an external, data organizing structure that helped to search for the related database segment more efficiently, and by that to speed up database queries.

The structure of the B$^+$-tree

As we already established, the first level of the B$^+$-tree needs to have at least 2 pointers. Each inner node needs to have at least $\lceil m/2 \rceil$ pointers, while each leaf has to have at least $\lceil m/2 \rceil - 1$ pointers to data values, which is $\approx \lceil m/2 \rceil$ for large m.

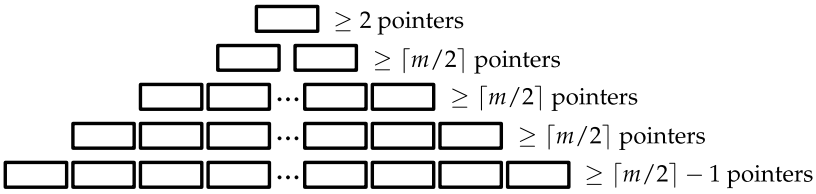


Figure 1.53: The structure of a B$^+$-tree.

Looking at Figure 1.53, we conclude that a B$^+$-tree of the m-th degree and h levels will have

$$2 * \lceil m/2 \rceil * \lceil m/2 \rceil * ... * \lceil m/2 \rceil = 2 * \lceil m/2 \rceil^{h-1} \tag{1.29}$$

Finally, the depth of a B$^+$-tree can be calculated knowing the number of final data values d, which is related to the maximal number of pointers in the leaves. It must be

$$d \leq 2 * \lceil m/2 \rceil^{h-1} \tag{1.30}$$

which means

$$h = \lfloor \log_{\lceil m/2 \rceil}(d/2) + 1 \rfloor \tag{1.31}$$

Exercise 1.5.1. Calculate depth for a B$^+$-tree of degree 70 needed to support 1 million data values.

Solution of Exercise 1.5.1
1. level: ≥2 pointers
2. level: ≥70 pointers
3. level: ≥2 450 pointers
4. level: ≥85 750 pointers
5. level: ≥3 001 250 pointers
The solution is 4 levels, since 5th has too much pointers to be minimally filled by 1 million data values, which can be calculated from $\log_{35} 500000 + 1$.

Searching for values in B-trees

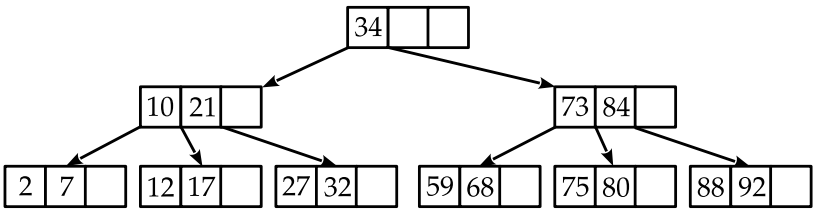


Figure 1.54: An example of the B-tree.

We start by traversing the root node. When traversing the double linked list in the node, we search for the first value that is equal or greater than the searched value v_s. If we did not find the exact value v_s in the node, and we encountered a value greater than v_s, this means that we need to continue searching in the left child of this greater value. If all values in the node are smaller than v_s, we continue searching in the right child of the last value. This continues until we encounter a node that has no right child on the last value, which is always a leaf due to the constructing rules.

```
class BTValue:
  value=None
  prev,next,lc,rc=None,None,None,None
  def __init__(self,value):
    self.value=value
  def setnext(self,v):
    ...
  def setleftchild(self,lc,pn=None):
    ...
  def setrightchild(self,rc,pn=None):
    ...
  def clearleftchild(self):
    ...
  def clearrightchild(self):
    ...
```

Listing 1.17: Python B-tree value node. If we use more complex values than built-in integers, string, floats, we need to implement functions *__lt__*, *__gt__*, *__le__*, and similar, to support value comparisons.

function BTreeSearch(n, v_s)
  $n_v \leftarrow$ starting value of the node n
  while $value(n_v) < v_s$ and $next(n_v)$ is not nil **do**
    $n_v \leftarrow next(n_v)$
  if $value(n_v) = v_s$ **then**
    return (node of the value n_v, n_v)
  else if $next(n_v)$ is nil and $value(n_v) < v_s$ **then**
    if $rightChild(n_v)$ is not nil **then**
      return BTreeSearch($rightChild(n_v), v_s$)
    else
      return (node of the value n_v, nil)
  else
    if $leftChild(n_v)$ is not nil **then**
      return BTreeSearch($leftChild(n_v), v_s$)
    else
      return (node of the value n_v, nil)

Algorithm 1.9: B-tree searching algorithm.

```
class BTNode:
  start,last=None,None
  pn,pv,orphan=None,None,None
  size,deg=0,5
  def __init__(self,value=None,
      start=None,deg=5,initsize=1):
    if value is not None:
      self.start=BTValue(value)
    if start is not None:
      self.start=start
    self.last=self.start
    self.size=initsize
    self.deg=deg
  def search(self,value):
    if self.start is None:
      return (self,None)
    v=self.start
    while v.value<value and
        v.next is not None:
      v=v.next
    if v.value==value:
      return (self,v)
    elif v.next is None and
        v.value<value:
      if v.rc is not None:
        return v.rc.search(value)
      else: return (self,None)
    else:
      if v.lc is not None:
        return v.lc.search(value)
      else: return (self,None)

class BTree:
  root,deg=None,5
  def __init__(self,value,deg=5):
    self.root=BTNode(value,deg=deg)
    self.deg=deg
  def search(self,value):
    (n,v)=self.root.search(value)
    return v
```

Listing 1.18: Python B-tree search code. The code consists of *BTNode* class that represents a multi-valued node, and *BTree* class that represents the B-tree.

Inserting values into B-trees

Inserting a new value v_n into a B-tree begins with the previously described search procedure. The search procedure continues traversing the searched B-tree, until reaching a leaf. If the searched value v_s is not in the leaf of the appropriate range, which is defined by the root node and inner nodes, we can conclude that searched value is not in the B-tree. It is customary to add the new value in the last searched leaf, since the new value belongs to the range of that leaf.

We can encounter the following situations:

1. **The leaf node had less than $m-1$ values before inserting the new value v_n.** In such a case, we normally insert the new value into the leaf, taking in consideration that values of the leaf must be sorted ascending. If using a linked list, this means inserting into the linked list in the appropriate position. Such insertion does not require B-tree restructuring. Figure 1.55 shows such a situation.

2. **The leaf node had $m-1$ values before inserting the new values.** Thus, inserting the new value would violate B-tree definition, since the leaf would have m values. This requires B-tree restructuring. The overflowing leaf, now having m values must be split into two nodes. We separate the overflowing leaf into the left side n_l, having $\lceil m/2 \rceil - 1$ left values, one middle value n_{mv}, and the right side n_r that has the rest of the values. When splitting the leaf, we can encounter two situations;

   (a) **The leaf node has no parent node, meaning that the leaf node is the root node.** We create a new root node that has a single value, which is the middle value of the splitting leaf n_{mv}. This single value n_{mv} in the newly added root node has the left child that is the left side of the splitting node n_l, and the right child that is the right side of the splitting node n_r. Such as situation can be seen in Figure 1.56.

   (b) **The leaf node has a parent node.** The middle value of the splitting node n_{mv} is inserted to the parent node in the appropriate position, which is always between values having the child pointer that points to the splitting node. Then, the left child of the value n_{mv} in the parent node is the left side of the splitting node n_l, and the right child of the value n_{mv} is the right side of the splitting node n_r. Such a situation can be seen in Figure 1.57.

   Taking that the leaf has the parent node, splitting the overflowing leaf (having m values) creates two leafs nodes that have $|n_l| = \lceil m/2 \rceil - 1$ values, $|n_r| = m - 1 - (\lceil m/2 \rceil - 1)$, and the middle value n_{mv}, which is inserted into the parent node. It means that the parent node gets one more value, and can overflow in the same way as the leaf. The whole splitting procedure applies to all inner nodes and the root node as well, meaning that the splitting procedure is recursive and stops when inserting n_{mv} in the parent node doesn't cause overflowing of the parent node or we create a new root node.

We need to notice that B-trees grows by adding new root nodes in the splitting procedure, which is opposite from binary trees, which grow by adding new leaves.

Also, there is a subtle difference between B-trees and B$^+$-trees. B$^+$-trees must comprise all values in leaves, which causes repetition of these values in inner nodes and the root node. The search procedure

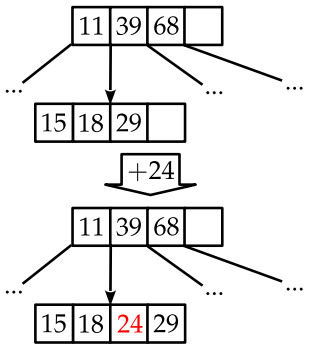


Figure 1.55: Inserting 24 into the leaf of the B-tree of degree=5, when the leaf has less than $m-1$ values. Such insertion does not require B-tree restructuring.

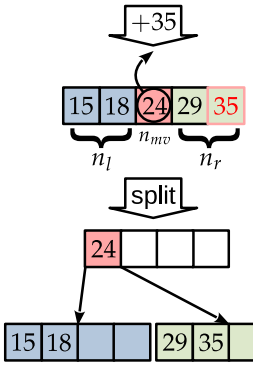


Figure 1.56: Inserting 35 into the leaf of the B-tree of degree=5, when the leaf has exactly $m-1$ values and has no parent node. Such insertion requires splitting of the node where the new value was inserted. Additionally, we add a new parent, root node for in the splitting process.

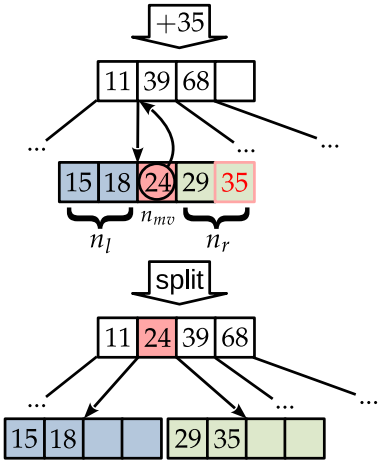


Figure 1.57: Inserting 35 into the leaf of the B-tree of degree=5, when the leaf has exactly $m-1$ values and has the parent node. Such insertion requires splitting of the node where the new value was inserted.

for the B$^+$-tree always must end in a leaf, simply because each value in tree leaves has a pointer to an external data store. From the splitting procedure point of view, this means that the middle value n_{mv} in the B$^+$-tree must remain in either the left child n_l or the right child n_r, which is the cause of the aforementioned repetition of values.

procedure BTreeSplit($btree, n$)
  if $|n| == m$ **then**
    $n_l \leftarrow$ new node having first $\lceil m/2 \rceil - 1$ values in the node n
    $n_{mv} \leftarrow$ the next value in the node n
    $n_r \leftarrow$ new node having the rest of values from the node n
    $n_{last} \leftarrow$ the last value in n_l
    $rightChild(n_{last}) \leftarrow leftChild(n_m)$
    if $parent(n)$ is nil **then**
      $n_{root} \leftarrow$ new node
      $n_{mv}' \leftarrow$ insert $value(n_{mv})$ into the node n_{root}
      $root$ of $btree \leftarrow n_{root}$
      $leftChild(n_{mv}') \leftarrow n_l$
      if $next(n_{mv}')$ is not nil **then**
        $leftChild(next(n_{mv}')) \leftarrow n_r$
      else
        $rightChild(n_{mv}') \leftarrow n_r$
    else
      $n_{mv}' \leftarrow$ insert $value(n_{mv})$ into the parent node of n
      $leftChild(n_{mv}') \leftarrow n_l$
      if $next(n_{mv}')$ is not nil **then**
        $leftChild(next(n_{mv}')) \leftarrow n_r$
      else
        $rightChild(n_{mv}') \leftarrow n_r$
      BTreeSplit($btree$,parent of n)
procedure BTreeInsert($btree, v_n$)
  $(n, n_v) \leftarrow$BTreeSearch($root$ of $btree, v_n$)
  insert v_n into the node n
  BTreeSplit($btree, n$)

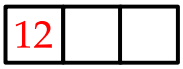
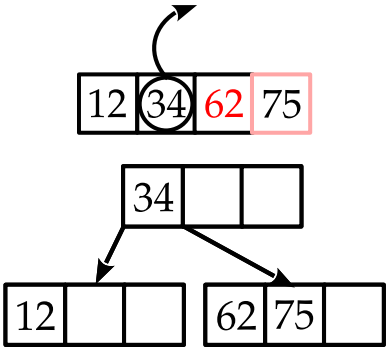
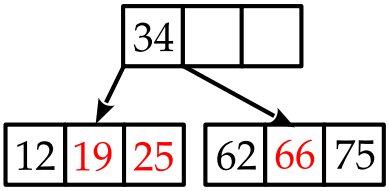
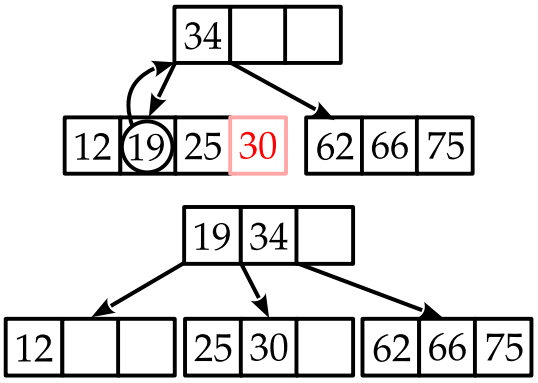
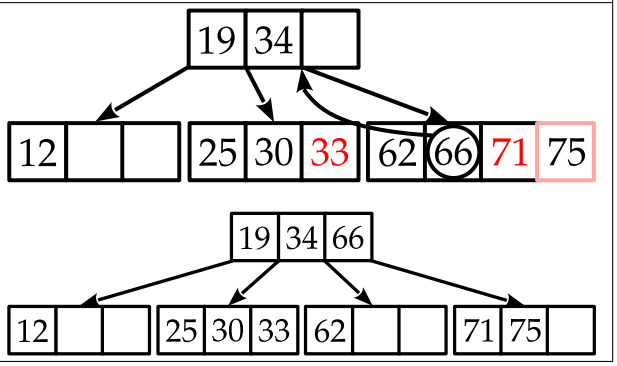
Algorithm 1.10: B-tree inserting algorithm.

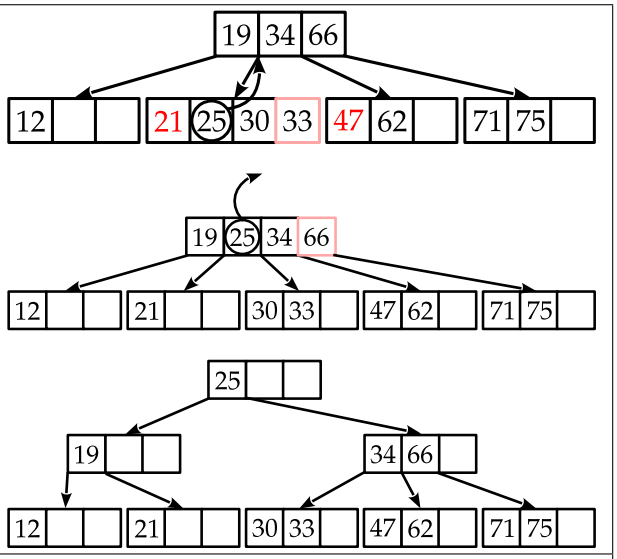
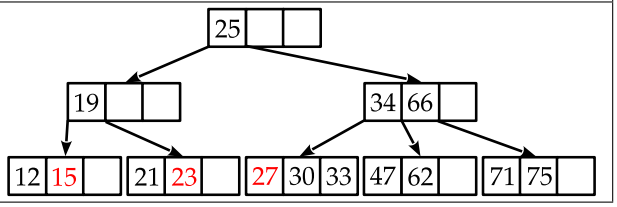
We explain the aforementioned Algorithm 1.10 in an example. Let us say, we start from an empty B-tree of degree=4, and insert the following sequence of numbers

$$12, 75, 34, 62, 19, 25, 66, 30, 33, 71, 47, 21, 15, 23, 27$$

```
class BTNode:
  ...
  def insert(self,value):
    v=self.start
    while v is not None and
          v.value<=value:
      v=v.next
    nv=BTValue(value)
    if v is not None:
      pv=v.prev
      nv.setnext(v)
      if pv is not None:
        pv.setnext(nv)
      else: self.start=nv
    else:
      if self.last is not None:
        self.last.setnext(nv)
      else: self.start=nv
    if nv.next is None:
      self.last=nv
    self.size=self.size+1
    return nv
  def adjustParent(self):
    v=self.start
    while v is not None:
      if v.lc is not None: v.lc.pn=self
      if v.rc is not None: v.rc.pn=self
      v=v.next
  def split(self):
    if self.size>=self.deg:
      s=ceil(self.deg/2)-1
      nl=BTNode(start=self.start,
            deg=self.deg,initsize=s)
      nl.pn=self.pn
      v=self.start
      for i in range(s-1): v=v.next
      nm=v.next
      v.next=None
      v.setrightchild(nm.lc)
      nl.last=v
      nl.adjustParent()
      nr=self
      nr.start,nr.size=nm.next,nr.size-1-s
      nm.next,nr.start.prev=None,None
      if nr.pn is not None: # 2b
        nv=nr.pn.insert(nm.value)
        nv.setleftchild(nl)
        if nv.next is None:
          nv.setrightchild(nv.prev.rc)
          nv.prev.setrightchild(None)
        if nr.pn is not None:
          return nr.pn.split()
      else: # 2a
        nroot=BTNode(value=nm.value,
              deg=self.deg,initsize=1)
        nl.pn,nr.pn=nroot,nroot
        nroot.start.setleftchild(nl)
        nroot.start.setrightchild(nr)
        return nroot
    return None

class BTree:
  ...
  def _insert(self,val):
    (n,v)=self.root.search(val)
    n.insert(val)
    nroot=n.split()
    if nroot is not None:
      self.root=nroot
  def insert(self,value):
    if type(value) is list:
      for v in value:
        self._insert(v)
    else: self._insert(value)
```

Listing 1.19: Python B-tree inserting code.

Step	B-tree
1: We start from an empty root node, where the first number 12 from the sequence is inserted.	
2: Inserting numbers 75 and 34 is done to the same root node. After this, the root node is full.	
3: By inserting 62, the root node overflows, so we need to split it. The left side of the splitted node n_l has $\lceil 4/2 \rceil - 1 = 1$ value, followed by the middle value $n_{mv} = 34$, followed by the right side which comprises the rest of the values in the splitted node. This is the $2a$ case, so we create a new root node. Now the B-tree has two levels.	
4: We insert more numbers $19, 25, 66$ directly into leaves of the B-tree, without overflowing any of the involved nodes.	
5: By inserting the number 30, the left leaf overflows, so we need to split it. The middle value $n_{mv} = 19$ is inserted into the parent node (the current root node). This is the $2b$ case.	
6: We insert number 33 and then number 71, which causes the overflow of the rightmost leaf. We split the rightmost node using the $2b$ case.	

7: We insert number 47 and then number 21, which causes the overflow of the second leaf. We split the second node using the $2b$ case, by inserting the middle value $n_{mv} = 25$ into the root node. However, in this case , the root node overflows as well, so we recursively split the root node as well. Splitting the root node is the $2a$ case, so we need to create a new root node. After that, the B-tree has three levels.	
8: We insert numbers 15, 23, and 27 into leaves of the B-tree. None of the leaves overflows. This is the final B-tree for the given sequence.	

Removing values from B-trees

Removing values from B-trees is somewhat more complex than inserting values. When inserting values, we split overflowed nodes and add new root nodes when necessary. Removing values does the opposite, we need to merge underflowed nodes, which consequently leads to removal of root nodes.

In each step, we remove one value from the B-tree. As with inserting, we start by searching the removed value. If the value is found in the B-tree, it can be positioned as follows:

1. **The value is found in a leaf node.** Such value can be directly removed from the leaf node, without additional actions,

2. **The value is found in an inner node or the root node.** We cannot directly remove such value since it has two dependent child nodes. To remove such value, we can use the remove by copy approach we used in binary trees. As in binary trees, we need to find direct predecessor or successor of the removed value, which is then used as a replacement value. The predecessor is always the rightmost value of the removed value left subtree, while the successor is always the leftmost value of the removed value right subtree. Both are found in the leaves of the B-tree. We copy the replacement value in the place of the removed value, and remove the replacement value. By doing this, we again removed one value from leaves of the B-tree.

   For example, we want to remove 34 from the B-tree in Figure 1.58. This value is found in the root node. The direct predecessor is 32, found as the rightmost value of the left subtree. The direct

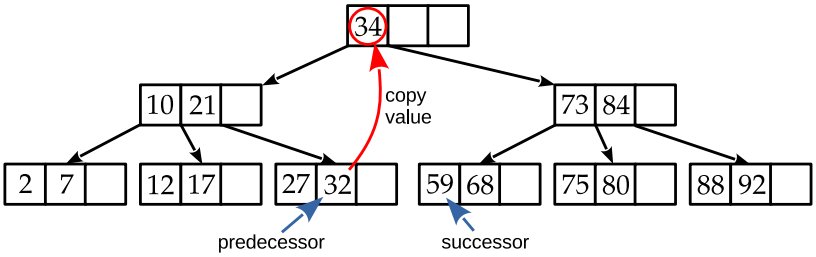


Figure 1.58: Removing value from the root node.

successor is 59, found as the leftmost value of the right subtree. Assuming we are using the predecessor, we copy 32 in the place of the removed value 34, and remove 32 in the leaf of the B-tree.

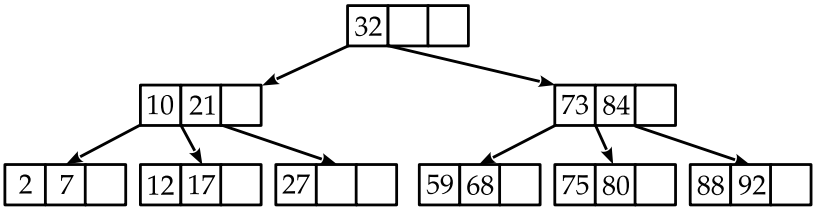


Figure 1.59: The removal final result.

The final removal result is seen can be seen Figure 1.59.

After we remove a value from a leaf, this leaf can underflow, which means it has $< \lceil m/2 \rceil$ child nodes (pointers) and $< \lceil m/2 \rceil - 1$ values. Since we need to have all nodes populated at least 50%, such a situation requires B-tree restructuring. For restructuring we need to know underflowed node siblings. Remember, sibling nodes come from the same parent. By looking at Figure 1.60, we can identify two siblings, one that has predecessor values to the values in the underflowed node, called a *predecessor sibling*, and one that has successor values to the values, called a *successor sibling*. Due to the B-tree structure, each node must have at least one sibling. The underflowed node and its predecessor sibling share 73 as the parent value. Similarly, the underflowed node and its successor sibling share 84 as the parent value. In both situations we have one left node n_l, the shared parent value n_{mv}, and one right node n_r. One of the nodes must be the underflowed node, while the other node must be one of its siblings.

Once we identified the pair of sibling nodes and their shared parent value, we can sum the total number of values contained in the structure made of triplet (n_l, n_{mv}, n_r), which is

$$|n_l| + 1 + |n_r| \tag{1.32}$$

For the pair n_l=*the predecessor sibling*, n_r=*the underflowed node*, and their shared parent value $n_{mv} = 73$, we have total of 5 values, which is $> m - 1$ and cannot be placed in a single node. Contrarily, n_l=*the underflowed node*, n_r=*the successor sibling*, and $n_{mv} = 84$ has 4 values in total and can be placed in a single node.

```
class BTNode:
  ...
  def maxvalue(self):
    if self.last is None:
      return (self,None)
    if self.last.rc is not None:
      return self.last.rc.maxvalue()
    else:
      return (self,self.last)
  def minvalue(self):
    if self.start is None:
      return (self,None)
    if self.start.lc is not None:
      return self.start.lc.minvalue()
    else:
      return (self,self.start)
  def _removevalue(self, v):
    pv,nv=v.prev,v.next
    if pv is None:
      if nv is not None:
        self.start=nv
        nv.prev=None
      else:
        self.start,self.last=None,None
    else:
      if nv is not None:
        pv.next=nv
        nv.prev=pv
      else:
        pv.next=None
        self.last=pv
    self.size=self.size-1
    return (pv,v,nv)
  def remove(self,v):
    if v.lc is not None:
      (n,pv)=v.lc.maxvalue()
      v.value=pv.value
      n._removevalue(pv)
      return n
    else:
      self._removevalue(v)
      return self
```

Listing 1.20: Finding predecessor, successor and value removal python code.

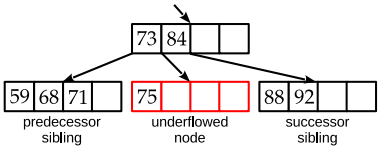


Figure 1.60: Underflowed node siblings.

To resolve the underflow, we can identify two distinct cases for each triplet (n_l, n_{mv}, n_r), which have their own specific solutions:

1. **Redistribution case**, in which the number of values in the sibling node is $> \lceil m/2 \rceil - 1$. This means that the total number of values cannot be placed in a single node. For this reason, we need to redistribute values by shifting them from the sibling node to the underflowed node, until we resolve the underflow. Looking at Figure 1.61 we can see a redistribution example. We shift n_l to n_r as long as n_r underflows. When shifting for one value, we take the shared parent value n_{mv} and add it to the beginning of the n_r linked list. Then we take the last value in n_l and place it as the shared parent value n_{mv}. In this example, shifting for one value was sufficient to resolve the underflow,

2. **Merging case**, in which the number of values in the sibling node is $\leq \lceil m/2 \rceil - 1$. This means that the total number of values can be placed in a single node. In this case, to resolve the underflow, we merge n_l, n_{mv}, and n_r into a single node, remove n_{mv} from the parent node and restructure the pointers to point to the new merged node. The example in Figure 1.62 shows merging of a triplet (n_l, n_{mv}, n_r). The new, merged node, is a sequential addition of the node $n_l = (75)$, the shared parent value $n_{mv} = 84$, and the node $n_r = (88, 92)$. We remove the shared parent value 84, and then update the right child of the value 74 to the new merged node $n_{merged} = (75, 84, 88, 92)$. In Figure 1.63 we removed the shared parent value 73, which has the successor value 84 in the parent node, therefore we needed to update the left child of the successor value to the new merged node.

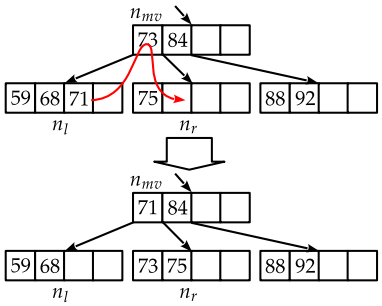


Figure 1.61: Redistribution example.

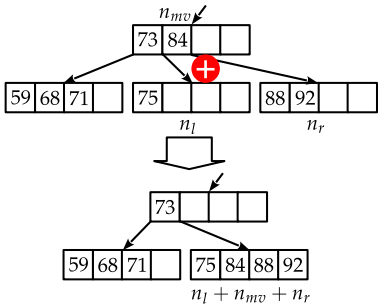


Figure 1.62: Merge example with successor sibling.

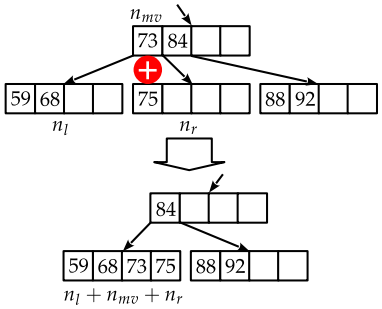


Figure 1.63: Merge example with predecessor sibling. Notice that n_l has one less value than in the previous merge example.

Shifting values

To support redistribution of values and merging of nodes, we can implement shifting actions for triplets (n_l, n_{mv}, n_r). Shifting values are primarily intended for redistribution of values, but can be used for merging as well. If we keep shifting until we empty one node and the shared parent value, meaning until we gather all values in one node, this is essentially the merging action. Depending on the side which we are shifting values in, we can have a *left shift*, or a *right shift*. The left shift is when we shift values from n_r into n_l, while the right shift is when we shift values from n_l to n_r. We need to take in consideration that redistribution and merging can happen on inner nodes, meaning that we need to restructure pointers to child nodes while shifting values.

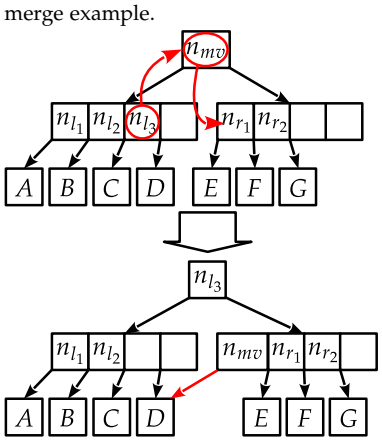


Figure 1.64: The schema for the right shift.

Right shift: Performing a right shift is done by taking the shared parent value n_{mv}, and pushing it as a new start of the linked list in the right node n_r. Simultaneously, we take the last value from the list in the left node n_l and place it as the new shared parent value n_{mv}. As the final action, we need to take the rightmost child node of the

left node n_l, and put it as the leftmost child node of the right node n_r. A right shift schema and example can be seen in Figures 1.64 and 1.65.

Left shift: Performing a left shift is done by taking the shared parent value n_{mv}, and pushing it as a new last value of the linked list in the left node n_l. Simultaneously, we take the start value from the list in the right node n_r and place it as the new shared parent value n_{mv}. As the final action, we need to take the leftmost child node of the right node n_r, and put it as the rightmost child node of the left node n_l. A left shift schema and example can be seen in Figures 1.66 and 1.67.

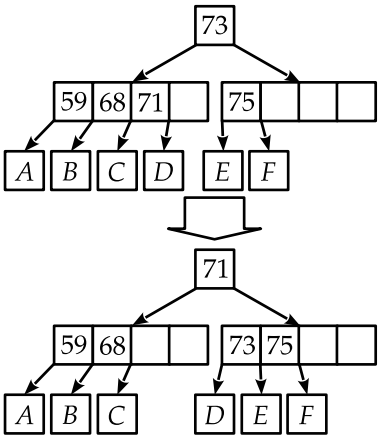


Figure 1.65: A right shift example.

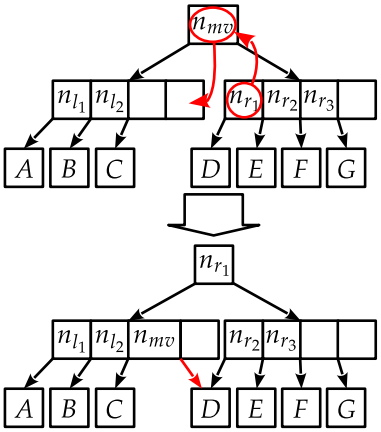


Figure 1.66: The schema for the left shift.

Border cases

In low-degree B-trees (< 5) we can have situations where one value in a node is sufficient to claim that it is at least 50% populated. For example, for a B-tree of degree=4, a single value means two child nodes, which is exactly 50%. However, if we remove this single value, the node remains empty with possibly one *orphan* child node left. In B-trees, we can remove only the root node as part of the merging, since removing any other inner node or leaf leads to an unbalanced B-tree. B-trees always need to be perfectly balanced. For example in Figure 1.68 we have a structure of three nodes that are populated exactly 50%, and each has exactly one value. If we remove 12 from the B-tree, we got into the merging case, which leaves one of the inner nodes empty, having one *orphan* child node. We cannot just remove this inner node, as this would lead to an unbalanced B-tree.

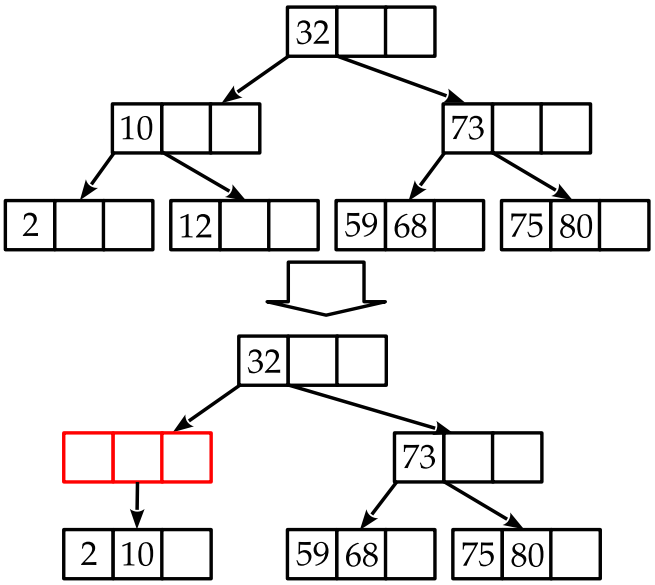


Figure 1.68: An example of the border case removal.

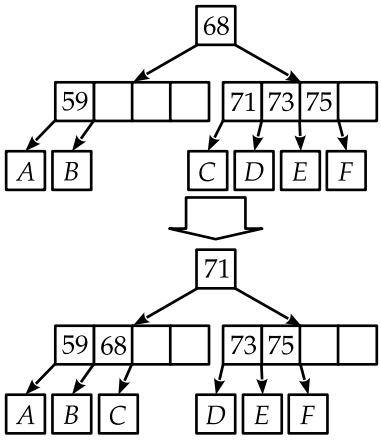


Figure 1.67: A left shift example.

The empty node can be perceived underflowed as any other node that has some values. We only need to take care of the orphan child node while performing shifting. The performed shifting must be as quick as possible, one shift of the shared parent value into the

populated sibling should suffice. Figure 1.69 shows schematas for a left and right shifting of the border cases.

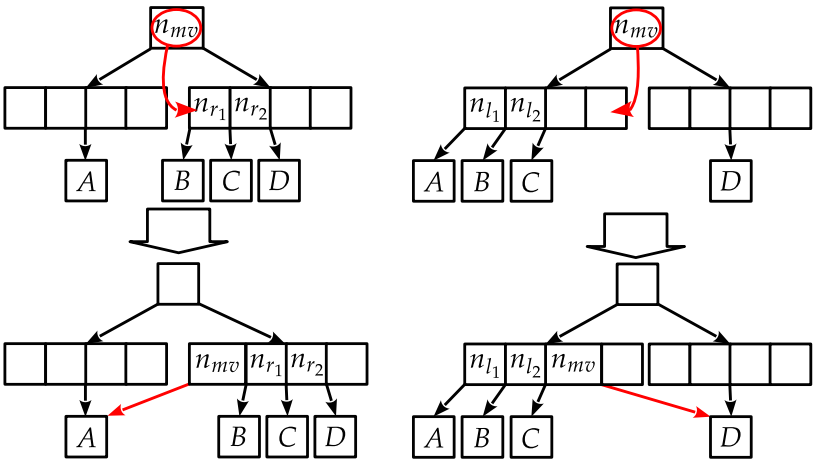


Figure 1.69: Border case shifting with orphan child node restructuring.

The empty node in Figure 1.68 is treated as any other underflowed node. Having a single sibling that is exactly 50% populated, we can only merge them as $n_l = ()$, $n_{mv} = 32$, and $n_r = (73)$. After merging, the old root node becomes empty and can be removed. The merged node becomes new root node, which completely solves our case.

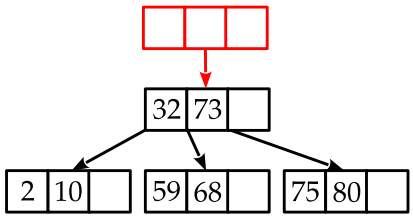


Figure 1.70: The final solution for the border case removal.

Finally, we can pinpoint a high-level B-tree value removal algorithm, which is seems to be significantly more complex than the insertion algorithm.

procedure BTREEREDISTRIBUTE($btree, n_l, n_{mv}, n_r$)
  $n_u \leftarrow$ the underflowed node
  while $|n_u| < \lceil$degree of $btree/2\rceil - 1$ **do**
    if n_u is n_l **then**
      shift left triplet (n_l, n_{mv}, n_r)
    else
      shift right triplet (n_l, n_{mv}, n_r)

```
class BTNode:
  ...
  def siblings(self):
    ps,ss,pv=None,None,self.pv
    if pv is not None:
      if pv.lc==self:
        if pv.prev is not None:
          ps=pv.prev.lc
        if pv.rc is not None:
          ss=pv.rc
        elif pv.next is not None:
          ss=pv.next.lc
      elif pv.rc==self:
        ps=pv.lc
    return (ps,ss)
```

Listing 1.21: Finding sibling nodes python code.

```
function BTreeMerge(btree, n_l, n_mv, n_r)
    n_u ← the underflowed node
    while |n_u| > 0 do
        if n_u is n_l then
            n_merged ← n_r
            shift right triplet (n_l, n_mv, n_r)
        else
            n_merged ← n_l
            shift left triplet (n_l, n_mv, n_r)
    if |parent of n_merged| = 1 then
        if parent of n_merged is the root of btree then
            set n_merged as the new root of btree
            return n_merged
        else
            remove value n_mv from the parent of n_merged
            set n_merged as the orphan child of its parent node
    else
        if there is a successor n_mv_s of the value n_mv then
            remove value n_mv from the parent of n_merged
            update the left child of the successor n_mv_s to the n_merged
            node
        else if there is a predecessor n_mv_p of the value n_mv then
            remove value n_mv from the parent of n_merged
            update the right child of the predecessor n_mv_p to the
            n_merged node
    return nil
procedure BTreeRemovalConsolidation(btree, n)
    if |n| < ⌈degree of btree/2⌉ − 1 then
        ps ← the predecessor sibling
        n_root ← nil
        if ps is not nil then
            n_mv ← the shared parent value between ps and n
            if |ps| > ⌈degree of btree/2⌉ − 1 then
                BTreeRedistribute(btree, ps, n_mv, n)
            else
                n_root ← BTreeMerge(btree, ps, n_mv, n)
        else
            ss ← the successor sibling
            n_mv ← the shared parent value between ss and n
            if |ss| > ⌈degree of btree/2⌉ − 1 then
                BTreeRedistribute(btree, n, n_mv, ss)
            else
                n_root ← BTreeMerge(btree, n, n_mv, ss)
        if n_root is nil and parent of n exists then
            BTreeRemovalConsolidation(btree, parent of n)
```

Algorithm 1.11: B-tree value removal algorithm.

procedure BTreeRemoval($btree, val$)
  $(n_{rem}, n_v) \leftarrow$ BTreeSearch(root of $btree, val$)
  if n_v is not *nil* **then**
    remove value val from the node n_{rem}
    BTreeRemovalConsolidation($btree, n_{rem}$)

Listing 1.22: B-tree value removal algorithm python code.

```
class BTNode:
  ...
  def shiftleft(self, nl, v, nr):
    if v.value is None: return
    nl.insert(v.value)
    if nl.last.prev is not None:
      nl.last.setleftchild(nl.last.prev.rc,nl)
      nl.last.prev.clearrightchild()
    elif nl.orphan is not None:
      nl.last.setleftchild(nl.orphan,nl)
      nl.orphan=None
    if nr.size>0:
      nl.last.setrightchild(nr.start.lc,nl)
      if nr.size==1:
        nr.orphan=nr.start.rc
      v.value=nr.start.value
      nr._removevalue(nr.start)
    else:
      nl.last.setrightchild(nr.orphan,nl)
      v.value=None
      nr.orphan=None
  def shiftright(self, nl, v, nr):
    nr.insert(v.value)
    if nr.orphan is not None:
      nr.start.setrightchild(nr.orphan,nr)
      nr.orphan=None
    if nl.last is not None:
      nr.start.setleftchild(nl.last.rc,nr)
      v.value=nl.last.value
      if nl.last.prev is not None:
        nl.last.prev.setrightchild(nl.last.lc)
      nl._removevalue(nl.last)
    else:
      nr.start.setleftchild(nl.orphan,nr)
      nl.orphan=None
      v.value=None

class BTree:
  ...
  def _remove(self,val):
    (n,v)=self.root.search(val)
    n=n.remove(v)
    nroot=n.removal_consolidate()
    if nroot is not None: self.root=nroot
  def remove(self,value):
    if type(value) is list:
      for v in value:
        self._remove(v)
    else: self._remove(value)
```

```
class BTNode:
  ...
  def redistribute(self, nl, v, nr):
    while self.size<ceil(self.deg/2)-1:
      if nl is self: self.shiftleft(nl, v, nr)
      else: self.shiftright(nl, v, nr)
  def merge(self, nl, v, nr):
    while self.size>0 or v.value is not None:
      if nl is self: self.shiftright(nl, v, nr)
      else: self.shiftleft(nl, v, nr)
    nn=nl
    if nl is self: nn=nr
    if self.pn.size==1:
      if self.pn.pn is None:
        nn.pn,nn.pv=None,None
        return nn
      else:
        self.pn.orphan=nn
        self.pn._removevalue(v)
    elif v.next is None:
      v.prev.setrightchild(nn,self.pn)
      self.pn._removevalue(v)
    else:
      v.next.setleftchild(nn,self.pn)
      self.pn._removevalue(v)
    return None
  def removal_consolidate(self):
    nroot=None
    if self.size<ceil(self.deg/2)-1:
      (ps,ss)=self.siblings()
      pn=self.pn
      if ps is not None:
        if ps.size>ceil(self.deg/2)-1:
          self.redistribute(ps, ps.pv, self)
        else:
          nroot=self.merge(ps,ps.pv,self)
      elif ss is not None:
        if ss.size>ceil(self.deg/2)-1:
          self.redistribute(self, self.pv, ss)
        else:
          nroot=self.merge(self,self.pv,ss)
      if nroot is None and pn is not None:
        nroot=pn.removal_consolidate()
    return nroot
```

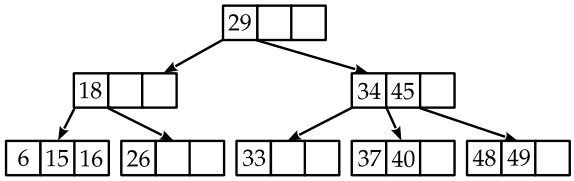
The complexity analysis

The search complexity of a B-tree is directly related to its height (1.31). The worst-case scenario is when all nodes are full and we search for the maximal value stored in the B-tree. Using the height of the B-tree and n as the total number of stored values, we get $O(m \log_{\lceil m/2 \rceil} n)$ as the search complexity. Insertion and removal are similar, since we need to find the appropriate node, and then we return back to the root node to consolidate nodes that overflow (in insertion) or underflow (in removal).

Exercise 1.5.2. Into an empty B-tree of degree=4, insert the following sequence of numbers

$$16, 29, 18, 34, 26, 15, 45, 33, 6, 37, 49, 48, 40$$

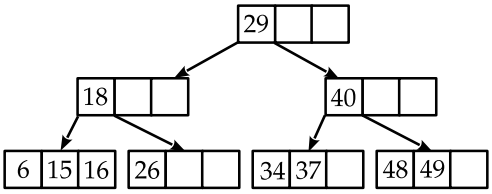
The solution



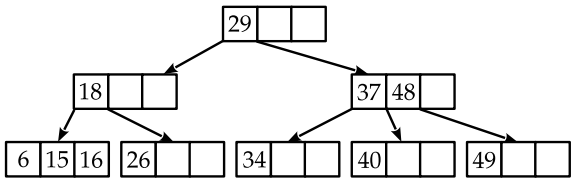
Exercise 1.5.3. From the result of the Exercise 1.5.2, remove the following sequence of numbers

$$45, 33$$

The solution



or



1.6 Red-Black (RB) trees

Red-Black (RB) trees are another variant of self-balancing trees, which uses coloring of the nodes for this purpose.

Definition 1.12 (Red-Black tree)**.** A Red-Black tree is a typical binary tree that adheres to the following rules:

1. Nodes can be colored either red ● or black ●,
2. The root of the tree is colored black ●,
3. All leafs of the tree are black ●,
4. Both children of a red node ● are black ●,
5. Each path from any node of the tree to all his descendant leaves passes through the same number of black nodes ●.

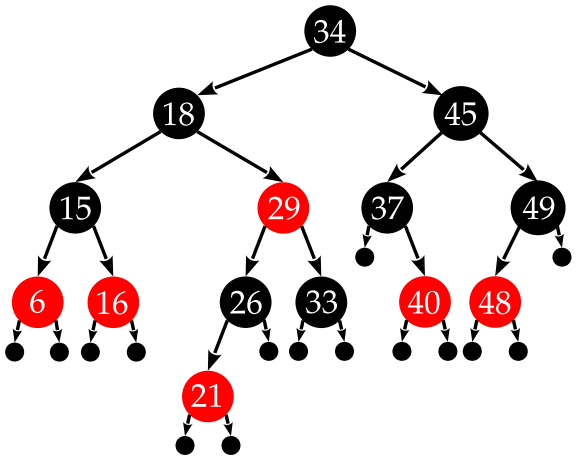


Figure 1.71: An example of the RB tree.

An example of a correctly built RB tree can be seen in Figure 1.71. We can see that the solution for the **Rule 3** is to have all leaves as a *dummy* black nodes. In fact we can create a single empty black node as a synthetic leaf node to which we point whenever we need a leaf node. This is regularly needed when we insert new values into the RB tree, or remove values from the RB tree.

To support **Rule 5**, we create two measure of heights within the RB tree

$$rh(n), bh(n) \tag{1.33}$$

where $rh(n)$ is the red height of the node n, and $bh(n)$ is the black height of the node n. It means the number of the red or black nodes on the path from n (excluding the node n) to any of its descendant leaves.

Due to the red and black coloring of tree nodes, the longest height of the tree is at most twice longer than the shortest height, which ensures that the RB tree is *approximately balanced*.

Theorem 1.1. The height of the RB tree having n inner nodes is $h \leq 2\log_2(n+1)$.

Proof: A binary tree of height h has at most $n = 2^h - 1$ inner nodes. Due to the **Rule 4**, we know that at least 50% of nodes in RB tree vertical paths will be black, or $bh \geq h/2$. We know that the number of inner nodes is greater or equal to the number of black nodes

$$n \geq 2^{bh} - 1 \geq 2^{h/2} - 1$$

finally

$$h \leq 2\log_2(n+1)$$

Searching RB trees is identical to searching any of the previous binary search trees, and is $O(\log_2 n)$ complex.

Inserting new values in RB tree

Inserting new values in the RB tree is the same as in any other binary tree, as explained in Section 1.3. Since in the RB trees we have black ● empty leaf nodes, we need to take this into account, since the empty synthetic node must always end up as the last one in the RB tree structure. A keen eye could notice that we can have all-black tree that would comply all the rules from Definition 1.12. If we would do that, we would loose the self-balancing capability of the RB tree. For that reason, a newly added value **always** ends up in a red node ●.

After inserting a new red node ● containing the new value, we can inspect the RB tree definition rules and determine whether some are being violated. In the case of the RB tree rule violation, we need to restructure the tree, which consequently balances the tree.

To understand the RB tree restructuring, we need to repeat the relationships between nodes of the binary tree:

- The usual parent (**P**)-child (**N**) relationship,
- The sibling (**S**) is the *other* child of the same parent,
- The uncle (**U**) is the sibling the of parent (**P**),
- The grandparent (**G**) is the parent of the parent (**P**).

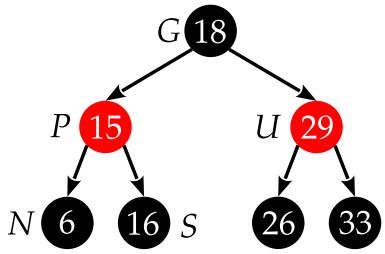


Figure 1.72: A RB tree genealogy structure example.

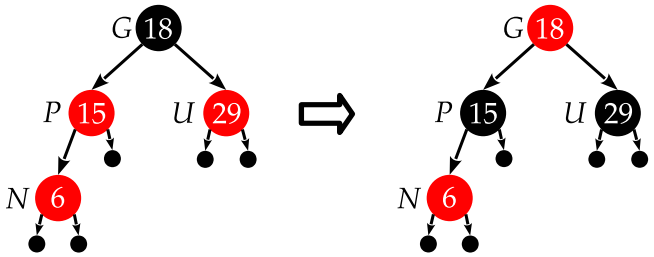
As part of the restructuring we use the same rotations as for the AVL tree.

Violation of the Rule 2
In such case we can only recolor the root node into the black color ●.

Violation of the Rule 4
Which means that the parent of the newly inserted node is also red ●, which violates the **Rule 4**, since the newly added node as the child is not black ● as stated in the RB tree rules. Here we have three distinct cases that we need to solve:

1. **Input pattern:** parent (**P**) and uncle (**U**) are both red ●
   Solution:

- Recolor parent (**P**) and uncle (**U**) to black ●,
- Recolor grandparent (**G**) to red ●,
- Continue checking for RB tree rules violations taking the grandparent node (**G**→**N**).

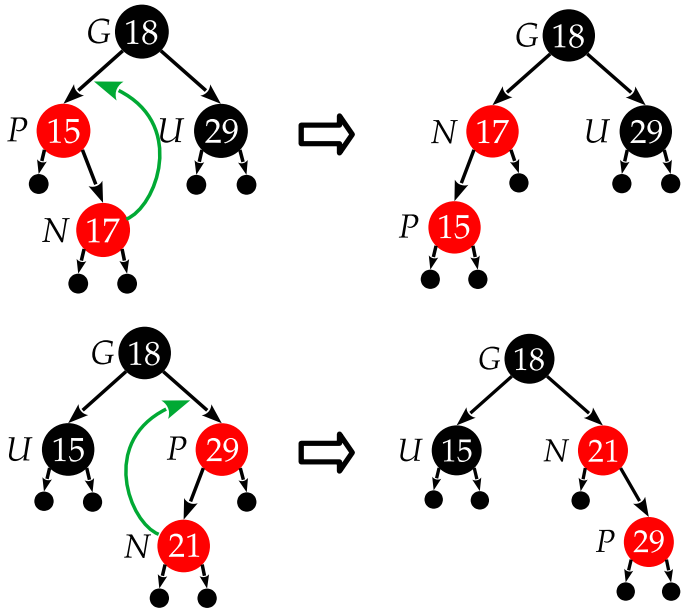


2. **Input pattern:** The parent (**P**) is red ● and the uncle (**U**) is black ●. This is **the broken case**, which means:

- If the uncle (**U**) is the right sibling of the parent (**P**), then the newly inserted node (**N**) is the right child of the parent (**P**),
- If the uncle (**U**) is the left sibling of the parent (**P**), then the newly inserted node (**N**) is the left child of the parent (**P**).

Solution:

- Straighten the case by rotating **N** around **P**,
- Continue checking for RB tree rules violations taking the parent node (**P**) as the newly inserted node (**P**→**N**), which leads us to the case 3.



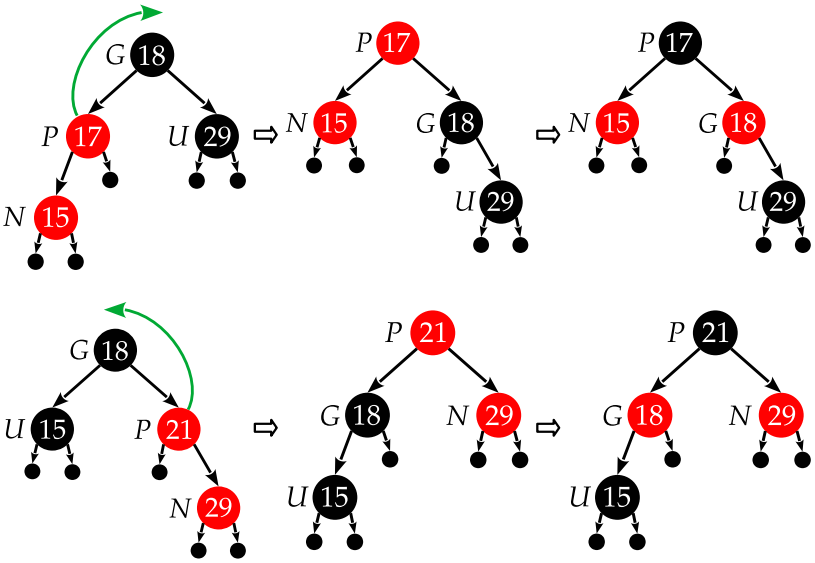
3. **Input pattern:** The parent (**P**) is red ● and the uncle (**U**) is black ●. This is **the straight case**, which means:

- If the uncle (**U**) is the right sibling of the parent (**P**), then the newly inserted node (**N**) is the left child of the parent (**P**),

- If the uncle (**U**) is the left sibling of the parent (**P**), then the newly inserted node (**N**) is the right child of the parent (**P**).

Solution:

- Rotate the parent node **P** around the grandparent node **G**,
- Change the color for parent **P** and grandparent **G** nodes. We know that P was red ● and G was black ●, since the RB tree was complying the rules before the insertion. We end the restructuring of the RB tree here.



Removing values from RB tree

When removing nodes from the RB tree, we need to use remove by copy approach. In such a removing action, we can denote *the replacement node* as **X**. When removing **X** from the RB tree, we can encounter two basic cases:

1. **X** was red ●. By removing a red node, we did not violate any RB rules, thus we do not restructure the tree,
2. **X** was black ●. By removing a black node, we violate the **Rule 5**, as we reduced the black height *bh* for one vertical path, meaning that the root node does not have the same number of black nodes to all leaves after the removal. This requires the RB restructuring, which is rather complex.

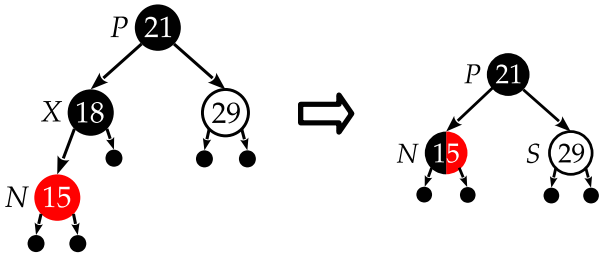
We need to keep the black count consistent by pushing the removed black node **X** up the structure of the RB tree. We need to keep this information somehow. We add this information to the child node **N**=**C**, or the parent node **N**=**P** if **X** had no children, which means that the node **N** now temporarily has two colors.

```
procedure RBTreeInsert(rbtree, N)
    P ← parent(N)
    G ← parent(P)
    while P is ● do
        if P is the left child then
            case ← LL
            if N is the right child then
                case ← LR
            U ← the right child of G
        else
            case ← RR
            if N is the left child then
                case ← RL
            U ← the left child of G
        if U and P are ● then
            P ← U ← ●
            G ← ●
            N ← G
        else if P is ● and U is ● then
            if case ∈ {LL, RR} then                ▷ straight cases
                if case is LL then
                    right rotate P around G
                else
                    left rotate P around G
                switch P and G colors
                break
            else                                   ▷ broken cases
                if case is LR then
                    right rotate N around P
                else
                    left rotate N around P
                N ← P
            P ← parent(N)
            G ← parent(P)
    root(rbtree) ← ●                               ▷ Rule 2
```

Algorithm 1.12: RB-tree insertion restructuring algorithm.

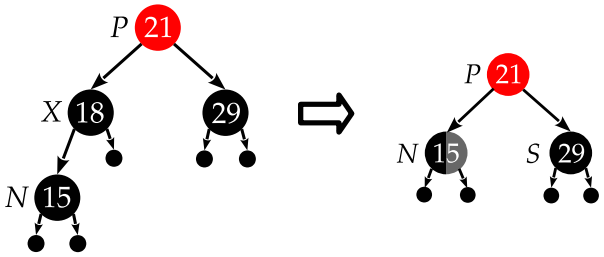
When transferring the black color from **X** to **N** (we see the case where **X** had one child, which is now marked as **N**), we have two cases:

1. The child node **N** is red ●. By removing the black node **X**, the new node **N** is red | black ◐. This case can be simply solved by



removing the red color from the node **N**, which leaves the node **N** black only.

2. The child node **N** is also black ●. By removing the black node **X**, the new node **N** is **double black** ◐. To solve this case, we need to



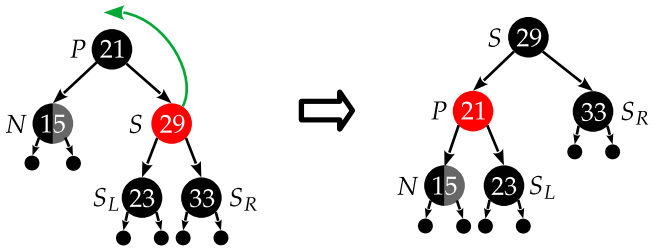
move one black color from the node **N** upwards through the RB tree structure, placing it into a node where it won't violate any RB tree rules.

To solve the double black color in the node **N**, we observe colors of the parent node **P**, the sibling node **S**, and the sibling node children **SL** and **SR**. We have the following cases:

1. **Input pattern:** The sibling node **S** is red ●. We draw the following conclusions:

   - Parent **P** and children **SL** and **SR** are all black ●.

   Solution:



   - Rotate the sibling node **S** around the parent node **P**. If **N** is the left child then this is going to be the right rotation of **S** around **P**. If **N** is the right child, this is going to be the left rotation of **S** around **P**,

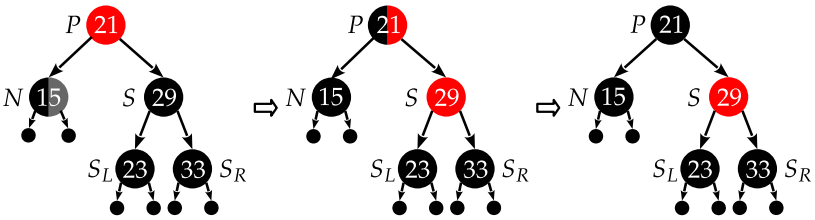
- Change the color for sibling **S** and parent **P** nodes,
- Continue restructuring from the node **N**.

2. **Input pattern:** The sibling node **S** is black ● and both of its children are black ●.
   Solution:

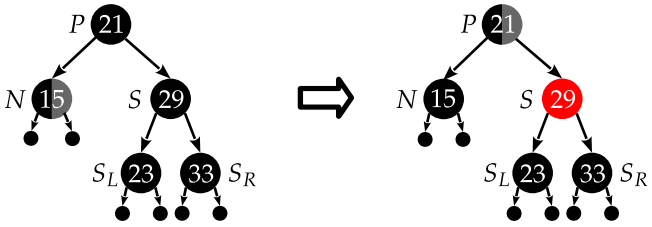
   - Move one black from the node **N** and its sibling **S** to the parent **P**. Since the sibling **S** was black, it becomes red, which is according to the RB tree rules since its children are both black,
   - Depending on the original color of the parent node **P**, now it becomes either red | black ◐ or double black ◐.

   Depending on the final colors of the parent node **P**, we have two sub-cases:

   (a) The parent node **P** is red | black ◐. We remove the red color from it, and it becomes black.

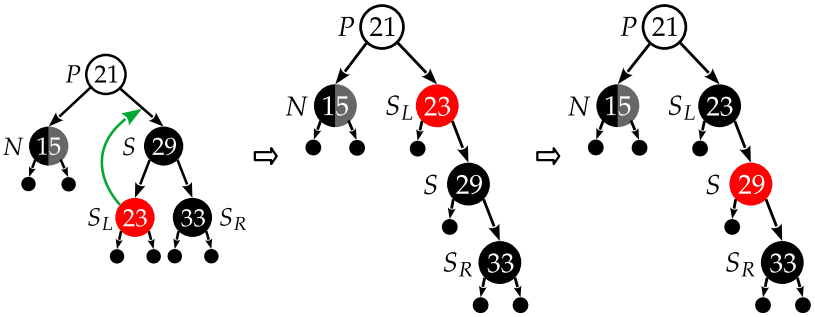


   (b) The parent node **P** is double black ◐. If the node **P** is the tree root node, we simply remove one black color and stop restructuring the tree, otherwise we continue the restructuring by taking the parent node **P** as the new node **N** ($P \rightarrow N$).



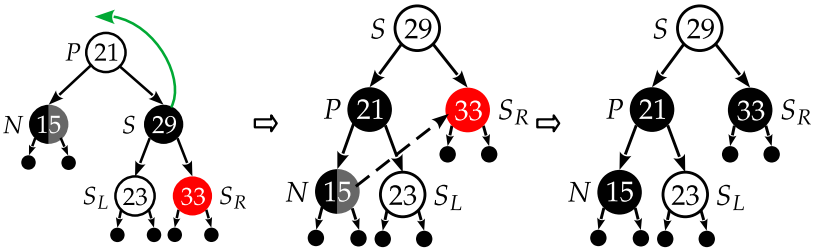
3. **Input pattern:** If the node **N** is the left child, the sibling node **S** is black ●, the sibling left child **SL** is red ●, and the sibling right child **SR** is black ●. The color of the parent node **P** is irrelevant.
   Notice that we have also the symmetric case where the node **N** is the right child, the sibling node **S** is black ●, the sibling right child **SR** is red ●, and the sibling left child **SL** is black ●.
   Solution:

   - Rotate the left child **SL** around its parent node **S** and change their colors,
   - Continue the restructuring with the same node **N**, which is the case 4.



4. **Input pattern:** If the node **N** is the left child, the sibling node **S** is black ●, and the sibling right child **SR** is red ●. Colors of the parent node **P** and the sibling left child **SL** are irrelevant.
   Notice that we have also the symmetric case where the node **N** is the right child, the sibling node **S** is black ●, and the sibling left child **SL** is red ●.
   Solution:

   - Rotate the sibling node **S** around its parent node **P** and change their colors,
   - Move the extra black color from the node **N** to the red node **SR**.



   The final result of this case is correct no matter what color we had in nodes **P** and **SL**.

When developing the RB tree removal algorithm, we need to take into account both symmetric variants for cases 3 and 4.

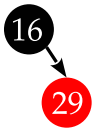
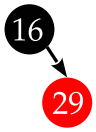
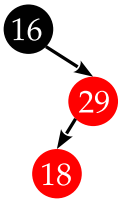
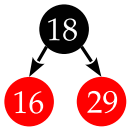
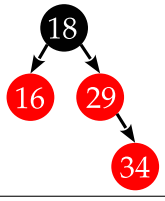
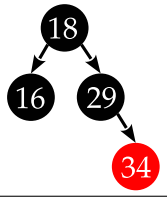
In the RB tree removal algorithm, we need to implement only rules that take the input tree and reproduce the final result, which means we can ignore the whole double coloring scheme and watch only the case 2, which can potentially move of second black color all the way to the root node. Cases 1 and 3 finish in 2 steps, while the case 4 needs only one step.

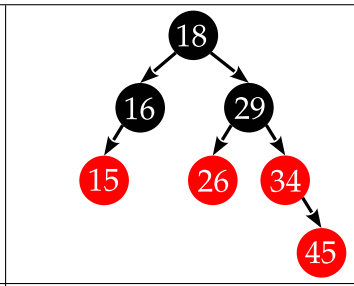
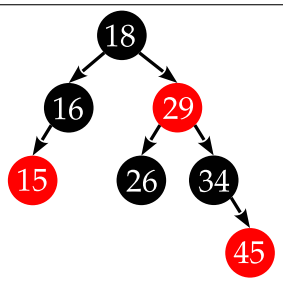
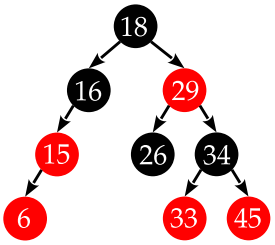
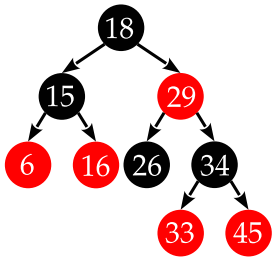
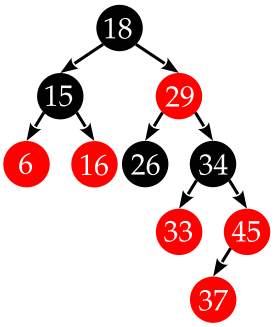
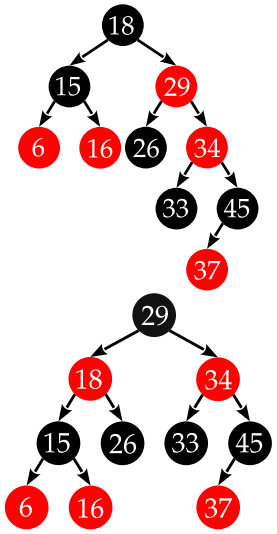
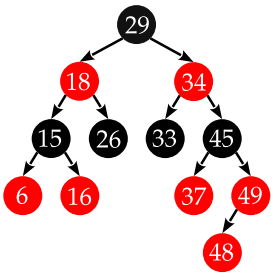
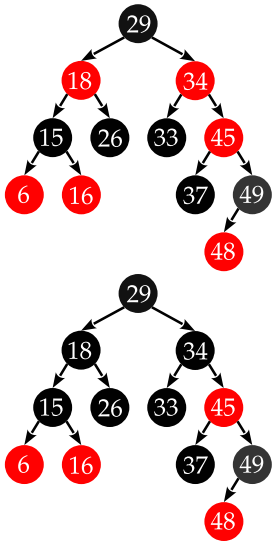
```
procedure RBTreeRemove(rbtree, N)
    while N is not root(rbtree) and N is ● do
        P ← the parent of N
        if N is the left child of P then
            S ← the right child of P
            S_L, S_R ← children of S
            if S is ● then
                S ← ●
                P ← ●
                left rotate S around P
            if S_L is ● and S_R is ● then
                S ← ●
                N ← the parent of N
            else
                if S_R is ● then
                    S_L ← ●
                    S ← ●
                    right rotate S_L around S
                color of S ← color of P
                P ← S_R ← ●
                left rotate S around P
                N ← root(rbtree)
        else                          ▷ implement the symmetrical cases
    N ← ●
```

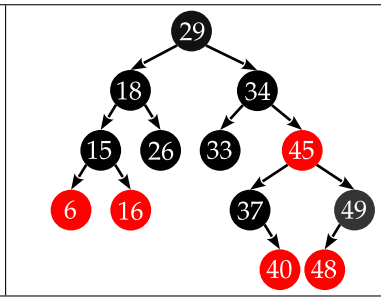
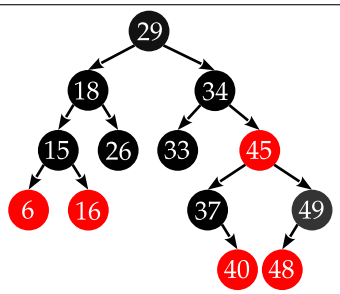
Algorithm 1.13: RB-tree removal restructuring algorithm.

Exercise 1.6.1. In an initially empty RB tree insert the following numbers

$$16, 29, 18, 34, 26, 15, 45, 33, 6, 37, 49, 48, 40$$

Step	After insertion	After restructuring
1: (16) Adding 16 which is colored black as the root		
2: (29) Adding 29 as the right child, colored to red ●		
3: (18) **Case 2:** left rotate 18 around 29 **Case 3:** right rotate 18 around 16 + switch colors		
4: (34) **Case 1:** set 18 to red ● and 16, 29 to black ● set root color to black ●		

5: (26,15,45) **Case 1:** set 29 to red ● and 26, 34 to black ●		
6: (33,6) **Case 3:** right rotate 15 around 16 switch 15 and 16 colors		
7: (37) **Case 1:** set 34 to red ● and 33, 45 to black ● **Case 3:** left rotate 29 around 18 switch 18 and 29 colors		
8: (49,48) **Case 1:** set 45 to red ● and 37, 49 to black ● **Case 1:** set 29 to red ● and 18, 34 to black ● set root to black ●		

9: (40) restructuring is not needed		

We can notice that the final resulting RB tree for the previous exercise is not equally balanced as the AVL tree from Exercise 1.4.3. However, we need to notice that the shortest height of the resulting RB tree is exactly twice shorter that the longest height, which is according to Definition **??**. The black height in all vertical paths from the root 29 is always the same $bh(29) = 2$, which does not stand for the red height $rh(29)$.

Notice that we exclude the starting node from the color height (*bh* and *rh*) of the RB tree.

Listing 1.23: RB tree insertion restructuring python code.

```
class RBTree(SimpleBinaryTree):
  def insertRestructure(self,n):
    if n.P is None: return
    P,G=n.P,None
    if P is not None: G=P.P
    while P is not None and G is not None and P.color=='red':
      if P==G.L:
        case='LL'
        if n==P.R:
          case='LR'
        U=G.R
      else:
        case='RR'
        if n==P.L:
          case='RL'
        U=G.L
      if U is not None and U.color=='red':
        P.color=U.color='black'
        G.color='red'
        n=G
      else:
        if case in ['LL','RR']:
          if case=='LL':
            G.rightrotate(self)
          else:
            G.leftrotate(self)
          tmp=P.color
          P.color=G.color
          G.color=tmp
        else:
          if case=='LR':
            P.rightrotate(self)
          else:
            P.leftrotate(self)
          n=P
      P,G=n.P,None
      if P is not None: G=P.P
    self.root.color='black'
  def insert(self,v):
    p=super().insert(v)
    if p is None:
      n=self.root
    else:
      if p.L is not None and p.L.S==v: n=p.L
      else: n=p.R
    n.color='red'
    self.insertRestructure(n)
```

2 String data structures

In programming, a sequence of characters in the computer memory is called a string. Alphabet used to form string is here denoted as Σ and consists of all characters that can be found in strings. Memory representation of strings can be done by mapping function $\mu : \Sigma \to \mathbb{N}$, which is usually implemented as a mapping table. Standard memory footprint of a string uses character mapping tables for mapping characters to memory values. These mapping tables can use a single memory byte (location) for a character, such as the ASCII mapping table, up to 2 bytes for a character in the Unicode mapping table.

To use a string, we need to know two values: a pointer to the string beginning, and to know where the string is terminated, which is usually marked with NULL character (hexadecimally $0x00$). In the following text, the NULL terminator will be denoted as \$. It is assumed that the NULL terminator \$ is present in the alphabet Σ.

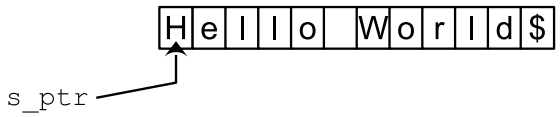


Figure 2.1: Standard string memory representation using a pointer and NULL terminator.

The other representation uses an ordered pair $s_i = (p_i, l_i)$, where p_i is the pointer to the starting position and l_i is the number of characters, or length of the string s_i.

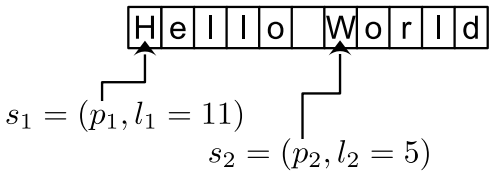


Figure 2.2: String representation using a pointer and length.

where $s_1 =$ "Hello World", and $s_2 =$ "World".

The second representation is more useful when it comes to defining and using a substring of a string. Some data structures prefer the first approach with NULL terminator, while others require the second approach with string lengths.

The basic idea of string data structures is to create indexing structures that will allow fast substring checking and searching on a bigger text corpus. For example, we want to find all occurrences of a word in our document as quickly as possible. Doing it sequentially, character by character, passing the whole document, would take quite a long time. Obviously, paying attention to the processing complexity is important. However, being able to address many substrings in the text corpus (we can structure phonems, morphems, words, phrases, sentences) makes us wondering how is it impacting the space complexity. Can we make string data structures memory preserving as much as possible?

In the following chapter we give several basic string data structures [Crochemore et al., 2007] that allow us store and quickly identify substrings of our interest in a bigger text corpus.

Listing 2.1: Position/length string representation python code.

```
class StringPL:
  def __init__(self, s: str, p: int=0, l: int=None):
    self.s, self.p = s, p
    if l is None: self.l = len(s)
    else: self.l = l
    if self.s is not None and self.p>len(self.s):
      raise 'cloning position greater than the length of the string'
    if self.s is not None and self.p+self.l>len(self.s):
      raise 'cloning position+length greater than the length of the string'
  def substring(self) -> (str):
    return self.s[self.p:self.p + self.l]
  def indexSubstring(self, tp:int, tl:int) -> (str):
    return self[tp:tp+tl]
  def clone(self, tp:int, tl:int) -> (object):
    if self.s is None: return StringPL(None, 0, 0)
    if tp>len(self.s):
      raise 'cloning position greater than the length of the string'
    if tp+tl>len(self.s):
      raise 'cloning position+length greater than the length of the string'
    return StringPL(self.s, tp, tl)
```

2.1 *Trie or Prefix tree*

We start by defining a finite set of strings S used to create a string data structure, or

$$S = \{s_i : 0 < i \leq N\} \tag{2.1}$$

In the context of the processed text corpus, strings in S are substrings of our interest or those that we want to index and check later on. We want to know that a certain query string q exists in the text corpus, which can be partially checked by testing the hypothesis $q \in S$? Why partially? Simply because S is only a subset of all strings of interest in the text corpus.

A naïve implementation would take each string from S and check all characters sequentially, which would take

$$\mathcal{L} = \Sigma_{s_i \in S} |s_i| \tag{2.2}$$

characters to test.

Trie - Reads as "try", comes from word "re**trie**val".

Definition 2.1 (Prefix tree, Trie)**.** A Trie is a rooted tree $T = (N, E, \mu)$, where μ is the edge mapping function $\mu : E \to \Sigma$. The root node of Trie represent the empty string. Each node of Trie can have at most $|\Sigma|$ children. Edges of Trie are transitions that are mapped to the alphabet in Σ. Each node outgoing incident edges that lead to child nodes must have distinct characters from Σ, i.e., no two outgoing incident edges of a node can have the same character from the alphabet Σ, meaning

$$\nexists v_i v_j, v_i v_z \in E : v_j \neq v_z, \mu(v_i v_j) = \mu(v_i v_z) \tag{2.3}$$

Leafs are terminating nodes that are precedent by the NULL terminating character \$. $\forall s_i \in S$ there is a vertical path from the Trie root node to a terminating leaf node.

A Trie is in fact a deterministic finite automata (DFA, state machine) that allows us parsing query string q to determine whether

$q \in S$. The previous definition uses the NULL terminated string representation.

If we take

$$S_1 = \{\text{"bird\$"}, \text{"cat\$"}, \text{"fish\$"}, \text{"zebra\$"}\}$$

Trie constructed from S_1 can be seen in Figure 2.3. The red node in Trie is the root node, from which we start to parse each query string q. We take character by character from the query string q and seek for the next transition in Trie. If we reach any of the terminating leaf nodes, marked blue in the Figure, the query string q is considered to be a member of S_1.

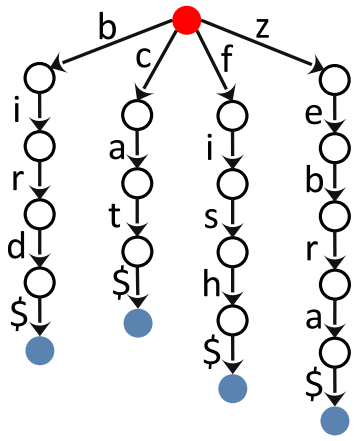


Figure 2.3: Trie for S_1.

Constructing Trie from a typical tree type of a graph is not sufficient. Tree nodes usually have no indexed or ordered outgoing edges (transitions in Trie), so we would need to iterate all of them to find the matching one for the current character, if one exists. This would take $O(|q| * |\Sigma|)$ to search for the query string q in Trie. This search complexity can be reduced down to $O(|q|)$ by *sacrificing* the space complexity. In each node, we reserve an array of pointers for

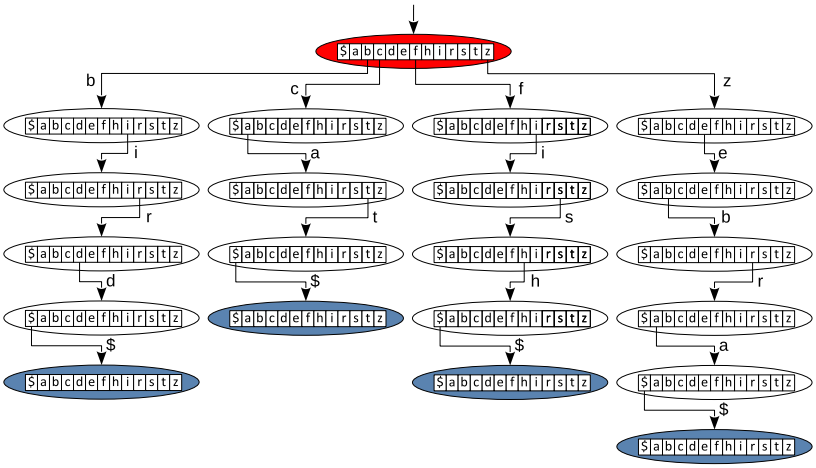


Figure 2.4: Node pointers structure for Trie for S_1.

all alphabet Σ elements. The reserved array and the alphabet must have the same ordering, so it can be directly accessed and checked. As we already mentioned, in computers we have several mapping tables that define alphabets with ordering, such as the ASCII mapping table. For the basic ASCII mapping table, where each character takes one byte, we need an array of 256 pointers, which point to child nodes. If pointer in a specific place in the array (for a specific alphabet element) is not defined (is NULL, or $0x00$ hexadecimally), we say that transition for this alphabet element is not defined. Otherwise, there is a transition we can use to traverse and move forward through Trie. We can see such an example in Figure 2.4, which is the detailed view of Trie in Figure 2.3. In the following text we use the notation

as in Figure 2.3, having in mind the structure of each node needed to achieve the search complexity $O(|q|)$. This approach simply trades the space complexity $O(|\Sigma|)$ for lowering the processing complexity in each Trie node down to $O(1)$. The total space complexity for such Trie is then $O(\mathcal{L} * |\Sigma|)$.

The overall search complexity of Trie is $O(|q|)$.

The other way is to use hash table in each Trie node. This is then called *Hashed Trie*. In such an implementation the space complexity in each node is down to $O(n)$, where n is the number of outgoing transitions in node. We know that the total space complexity for *Hashed Trie* is $O(\mathcal{L})$. *Hashed Trie* is also slower than normal Trie, since hash table processing in each node takes more time than simply accessing an element of a static array.

We need to notice that strings in S_1 have no common prefixes, which makes the Trie tree *uncompressed*, i.e., there are no two strings in the set of strings S_1 that share a common node in the Trie tree. In this case we have Trie that has $|E| = \mathcal{L}$ edges.

As for many words, many substrings of interest can share common prefixes or suffixes, which can be utilized to create less complex string data structure. This means that we can have pairs of strings such that

$$\exists s_i, s_j \in S : s_i \neq s_j \wedge s_i = t_i u_i \wedge s_j = t_j u_j \Rightarrow t_i = t_j \tag{2.4}$$

which share a common prefix, or a common sequence of starting characters. If we have

$$S_2 = \{\text{"analysis\$"}, \text{"analgetic\$"}, \text{"analogy\$"}, \text{"anarchy\$"}, \text{"acetone\$"}, \text{"acronim\$"}, \text{"archaic\$"}\} \tag{2.5}$$

and by placing these strings in an alphabetically sorted array (the NULL terminating character \$ is omitted for the clarity sake) we can observe many strings having a common prefixes, as defined in (2.4).

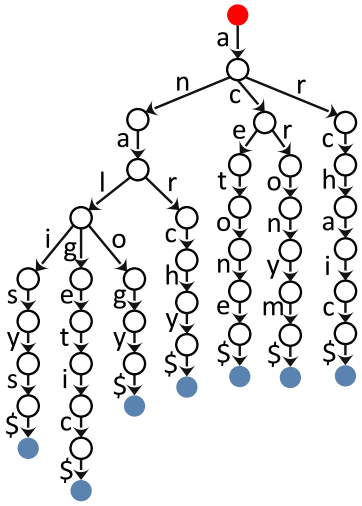


Figure 2.5: Trie for S_2.

a	n	a	l	i	s	y	s	
a	n	a	l	g	e	t	i	c
a	n	a	l	o	g	y		
a	n	a	r	c	h	y		
a	c	e	t	o	n	e		
a	c	r	o	n	i	m		
a	r	c	h	a	i	c		

Table 2.1: S_2 strings placed in an alphabetically sorted array.

In Definition 2.1, the constraint (2.3) is the main requirement to build the prefix tree. Using S_2 and Definition 2.1, we construct Trie as in Figure 2.5. Due to having the same prefix subtree in Trie for multiple strings in S_2 (2.4), we can expect to have $|E| < \mathcal{L}$ edges, which lowers Trie space complexity when comparing to the aforementioned naïve string search.

In Figure 2.5 we had a unique set of strings S_2 where each string had its own vertical path and terminating leaf node. Another ex-

ample is to have a string in S that is a substring of some other string, for example

$$S_3 = \{\text{"ana\$"}\} \cup S_2 \tag{2.6}$$

which can be seen in Figure 2.6. We can notice one more \$ transition after "ana", and the related terminating leaf node, which allows us to parse "ana\$" as a common prefix and a substring of other strings in S_3, such as "analysis\$", "analgetic\$", "analogy\$", and "anarchy\$". This is the real reason for introducing the NULL terminating character \$ into Trie.

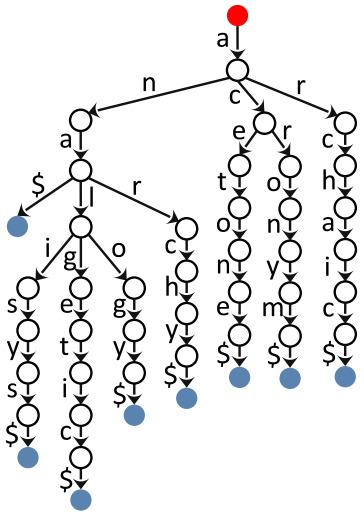


Figure 2.6: Trie for S_3.

Searching

Once we have a Trie tree (remember, it is a DFS), searching for a query string q is as follows

Algorithm 2.1: Search query string in Trie.

```
function SearchTrie(T, q)
    cn ← root(T)
    for ch ∈ q do
        if there is transition from cn for character ch to cn_child then
                                              ▷ array[ch] ≠ NULL in cn
            cn ← cn_child
        else
            return (false, cn)
    if cn is a leaf then
        return (true, cn)
    return (false, cn)
```

Algorithm 2.1 starts traversing the Trie tree from the root, which we assign to the *current node cn*. We take character by character from the input query string q, and match transitions to children of the current node. If there is such a transition, the new current node is the child node of the transition that matches the current character. As soon as there is no transition to a child node that would match the current character, we conclude that there is no string in Trie that would match the query string q. If we match all characters, we must also match the NULL terminator \$. This is needed to prevent successfully matching substrings of strings stored in Trie. For example, the searching algorithm should return $false$ for query string q = "ana\$" in Trie made for S_2. Finally, if the NULL terminator \$ matches as well, we can return $true$, which means that the query string q is contained in Trie T.

Inserting

```
procedure InsertTrie(T, s)
    cn ← root(T)
    for ch ∈ s do
        if there is transition from cn for character ch to cn_child then
                                                ▷ array[ch] ≠ NULL in cn
            cn ← cn_child
        else
            create new node cn_new
            add transition for character ch from cn to cn_new
            cn ← cn_new
```

Algorithm 2.2: Insert a new string into Trie.

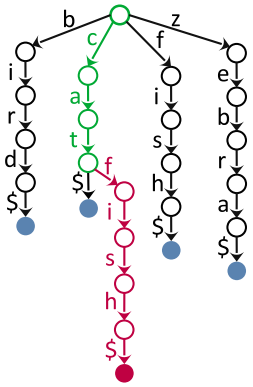


Figure 2.7: Inserting string "catfish$" into Trie for S_1. Green transitions are the common prefix, while purple transitions are newly added ones, to support string "catfish$".

Inserting a new string s to Trie starts by searching $q = s$. If we find the whole string s in Trie, the insertion is not necessary, as Trie already has the string s. The other purpose of searching is to find the common prefix between the string s and the set of strings S contained in Trie T. As soon as there is no transition for a next character in the input string s, we start adding new nodes and transitions until we create the full vertical path for the input string s, including the trailing NULL terminating character \$.

Figure 2.7 shows the final result after inserting string "catfish$" into Trie that for the set of strings S_1. Since we have string "cat$" in the set of strings S_1, which is a substring of the inserted string "catfish$", we transition the existing Trie tree until we encounter the character f. From this character onward, we add new Trie subtree, which must support the string "catfish$".

It must be noticed that we have the same "fish" subtree in the other part of the Trie tree. Although commonization string prefixes in Trie lowers the space complexity, there are still cases which can be optimized even further.

Removal

```
procedure RemoveTrie(T, s)
    (succ, cn) ← SearchTrie(T, s)
    if not succ then
        return                                  ▷ s not found in Trie
    s ← 'reversed(s)
    cn ← parent(cn)
    for ch ∈ s do
        remove transition from cn for character ch
        if cn has no children and there is a parent of cn then
            cn ← parent(cn)
        else
            return
```

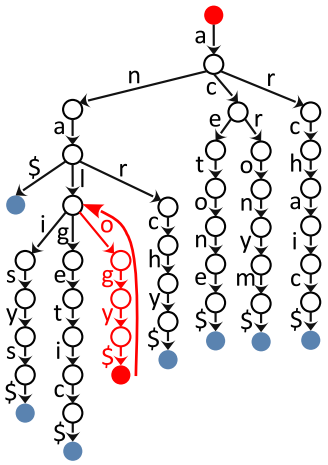


Figure 2.8: Removing the string "analogy$" from Trie that has S_3 set of strings. Traversing up the vertical path of the removed string. We stop at the first node that has more than one child.

Algorithm 2.3: Remove a string from Trie.

Due to the nature of the prefix tree, string removal must be performed backwards, from the terminating leaf node following the removed string s vertical path, up to the root node. Since Trie has common prefix vertical paths, we need to remove only part of the vertical path belonging exclusively to the removed string s, which is always its suffix. This is the reason why we begin from the terminating leaf node.

First, we start by searching the removed string s, which will lead us to the terminating leaf node. Then we start ascending the tree up the removed string s vertical path until we reach the first node having more than one child node. This part of the removed string s vertical path is its unshared suffix that need to be removed.

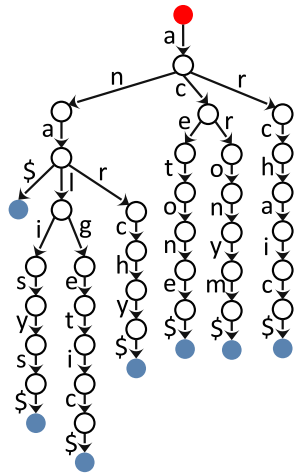


Figure 2.9: Trie after the removal of the string "analogy$".

Listing 2.2: Trie python code.

```
class TrieNode:
  def __init__(self, p=None):
    self.children = [None] * 256 #sigma=ascii
    self.parent, self.cno = p, 0
  def isLeaf(self) -> (bool):
    return self.cno == 0
  def transition(self, c: chr) -> (object):
    return self.children[ord(c)]
  def insert(self, c: chr) -> (object):
    cn = self.transition(c)
    if cn is None:
      ncn = TrieNode(self)
      self.children[ord(c)] = ncn
      self.cno += 1
      return ncn
    else: return cn
  def remove(self, c: chr) -> (bool):
    cn = self.transition(c)
    if cn is not None:
      self.children[ord(c)] = None
      self.cno -= 1
      return True
    return False
```

```
class Trie:
  def __init__(self):
    self.root = TrieNode(None)
  def insert(self, s: str):
    currTN = self.root
    for c in s:
      currTN = currTN.insert(c)
    currTN.insert('$')
  def search(self, q: str) -> (bool, TrieNode):
    currTN = self.root
    for c in q + '$':
      currNTN = currTN.transition(c)
      if currNTN is None: return (False, currTN)
      currTN = currNTN
    return (currTN.isLeaf(), currTN)
  def remove(self, s: str) -> (bool):
    (res, ltn) = self.search(s)
    if res:
      s = '$' + "".join(reversed(s))
      currTN = ltn.parent
      for c in s:
        split = len(currTN.children) > 1
        currTN.remove(c)
        if split: break
        currTN = currTN.parent
        if currTN is None: break
      return True
    return False
```

One way to implement Trie is to avoid having the NULL terminating character $ and to mark final nodes. In such an implementation we would have an additional flag in each node that would mark the final state, i.e., the acceptable state.

2.2 *Patricia tree*

When we look at the Trie tree structure, we can see many redundancies in form of sequentially chained transitions without branching, where nodes have no more than one child, as in Figure 2.10, which shows Trie for

$$S_4 = \{\text{"ana\$"}, \text{"analysis\$"}, \text{"analogy\$"}, \text{"acronym\$"}, \text{"acrobat\$"}\}$$

In a Patricia tree [Morrison, 1968] (i.e., *Practical Algorithm To Retrieve Information Coded In Alphanumeric*, or a *compressed Trie*), we collapse all these sequentially chained transitions into a single transition, containing a concatenated substring, made of all collapsed transitions from Trie. If we collapse red marked sequential chains from Figure 2.10, we get a Patricia tree as in Figure 2.11.

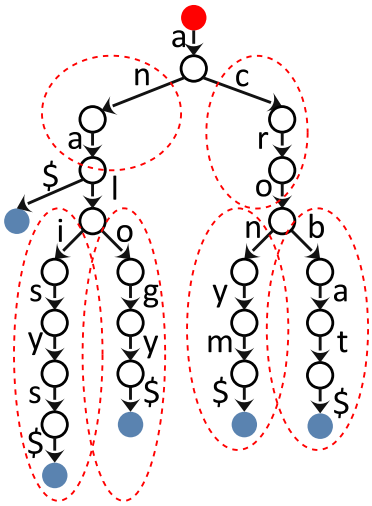


Figure 2.10: Trie for S_4. Red markings denote sequentially chained transitions.

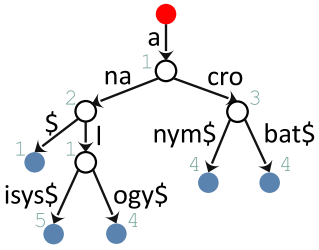


Figure 2.11: Patricia tree for S_4.

Searching a query string q in a Patricia tree is a bit more complex than in Trie. In Trie we could take character by character from the query string q and traverse through Trie, until we encounter either a terminating leaf node, or we cannot find transition for our current character from the query string q. In a Patricia tree, to traverse each transition, we must match full substring of the transition. If we match only a part of the transition substring, we cannot traverse it. Only by reaching a leaf node we can conclude that the query string q is contained in the Patricia tree.

The Patricia tree node contains the same array as Trie node. Here, each array field maps only the first character of the transition substring. Two transitions from the same Patricia tree node cannot start on the same character, since they cannot share the same prefix. Such an approach make the search complexity of the Patricia tree the same as for Trie $O(|q|)$.

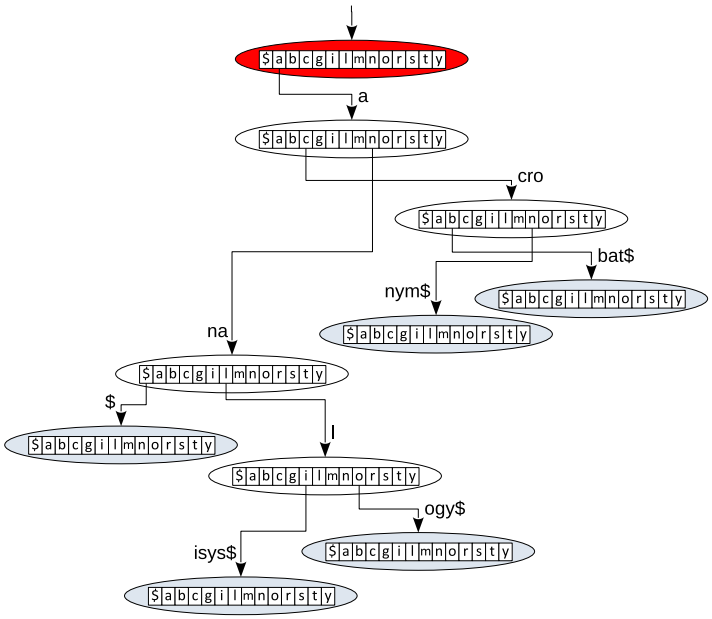


Figure 2.12: Patricia tree node arrays for S_4.

Algorithm 2.4: Searching algorithm for Patricia tree using pointer and length string representation.

```
function SearchPatricia(P, q = (q_p, q_l))
    sc ← 0
    cn ← root(P)
    while cn not leaf do
        if there is transition from cn for character q_p[sc] to cn_c then
                                        ▷ array[q_p[sc]] ≠ NULL in cn
            (t_p, t_l) ← string representation in cn_c
            if q_p[sc : sc + t_l] = t_p[0 : t_l] then
                                ▷ substring matching (Python notation)
                sc ← sc + t_l
                cn ← cn_c
            else
                return false
        else
            return false
    return sc = q_l
```

There are some interesting observations about Patricia trees, which we could find useful:

- All internal nodes of a Patricia tree must have at least two child nodes. Otherwise, this is also a sequential chain of transitions that can be collapsed into a single transition.
- We can use the pointer and length string representation at each node of a Patricia tree, which is then taken to be related to the single incoming transition into the node. Remember, in a directed tree, all nodes except the root node can have only one incoming edge, i.e., each node except the root node can have only one parent node. By using the ordered pair (p, l) in each node of a Patricia tree except the root node, we point to the first character and define the substring length for the incoming transition. Which substrings are being pointed to, depends on the evolution of a Patricia tree.

The evolution of a Patricia tree is taken as a sequence of string insertions and removals.

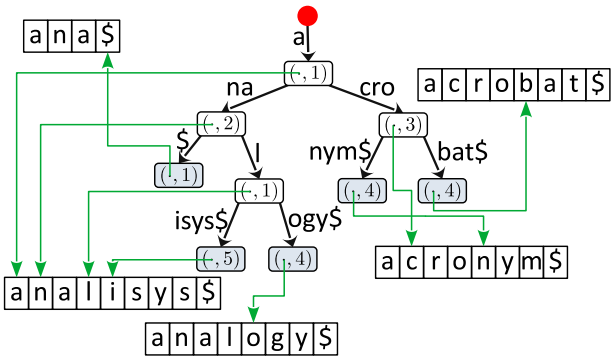


Figure 2.13: Patricia tree for S_4 using the string pointer and length representation.

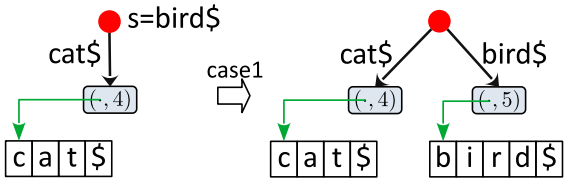
To understand what is the evolution of the Patricia tree in Figure 2.13, we need to understand how to update a Patricia tree each time we insert new string into it.

Inserting

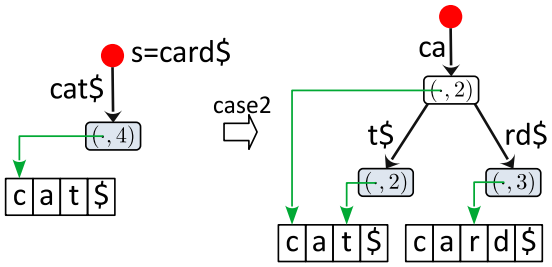
When inserting a new string s, we start by traversing through the existing Patricia tree. At the beginning we have only root node. The purpose of traversing through the Patricia tree is to find the common prefix with all strings already contained in the tree.

We focus on our current node, which could be the root node. There are two cases which could occur:

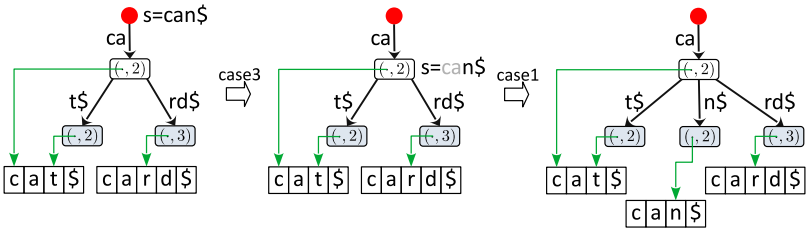
1. We cannot find any outgoing transition that would even partially match the remainder of the string s we want to insert. This means, we cannot match even one character for the outgoing transitions. In this case, we add a new leaf node and a new transition from out current node to the newly added leaf. The transition must contain the remainder of the string s we had when testing outgoing transitions from the current node. This case happens when we have an empty Patricia tree, having only the root node.



2. We can find one outgoing transition that can be partially matched. Then we break this transition bz adding a new node to it. The incoming transition into the newly added node contains the matched part of the substring from the old transition, while the outgoing transition from the newly added node contains the unmatched part of the substring from the old transition. Finally, we add a new leaf node and a new transition from the newly added node to the new leaf node that contains the unmatched part of the string s we want to insert.



3. There is an outgoing transition from the current node that can be fully matched to the substring of the string s we want to insert. We traverse this transition and take the child node as the new current node, taking in consideration that we partially matched the string s, i.e., that part of the string s shares a common prefix with some other string in the Patricia tree.



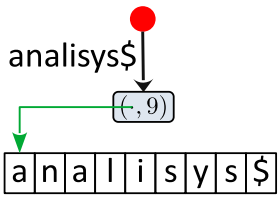
Algorithm 2.5: Inserting algorithm for Patricia tree using pointer and length string representation.

procedure INSERTPATRICIA($P, s = (s_p, s_l)$)
  $sc \leftarrow 0$
  $cn \leftarrow root(P)$
  while cn is root or not leaf **do** ▷ $O(|s|)$
    if there is transition from cn for character $s_p[sc]$ to cn_c **then**
      ▷ $array[s_p[sc]] \neq NULL$ in cn
      $(t_p, t_l) \leftarrow$ string representation in cn_c
      if $s_p[sc : sc + t_l] \subset t_p[0 : t_l]$ **then** ▷ case 2
        ▷ we know for sure that the first character is matched
        $i \leftarrow$ first unmatched character from t_p ▷ $i > 0$
        remove transition between cn and cn_c
        add node cn_{ins} with (t_p, i) ▷ $O(|\Sigma|)$
        update node cn_c with $(t_p + i, t_l - i)$
        add transition between cn and cn_{ins}
        add transition between cn_{ins} and cn_c
        add leaf node cn_{leaf} with $(s_p + (sc + i), s_l - (sc + i))$
        ▷ $O(|\Sigma|)$
        add transition between cn_{ins} and cn_{leaf}
        return
      else if $s_p[sc : sc + t_l] = t_p[0 : t_l]$ **then** ▷ case 3
        $sc \leftarrow sc + t_l$
        $cn \leftarrow cn_c$
    else ▷ case 1
      add leaf node cn_{leaf} with $(s_p + sc, s_l - sc)$ ▷ $O(|\Sigma|)$
      add transition between cn and cn_{leaf}
      return

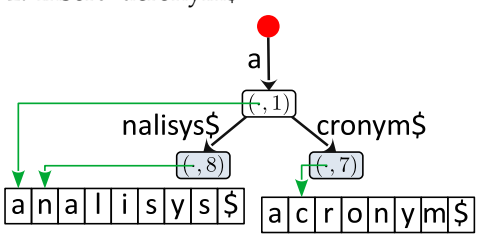
The insertion complexity is $O(|s| + |\Sigma|)$, which consists of traversal to the leaf node and adding at most two new nodes.

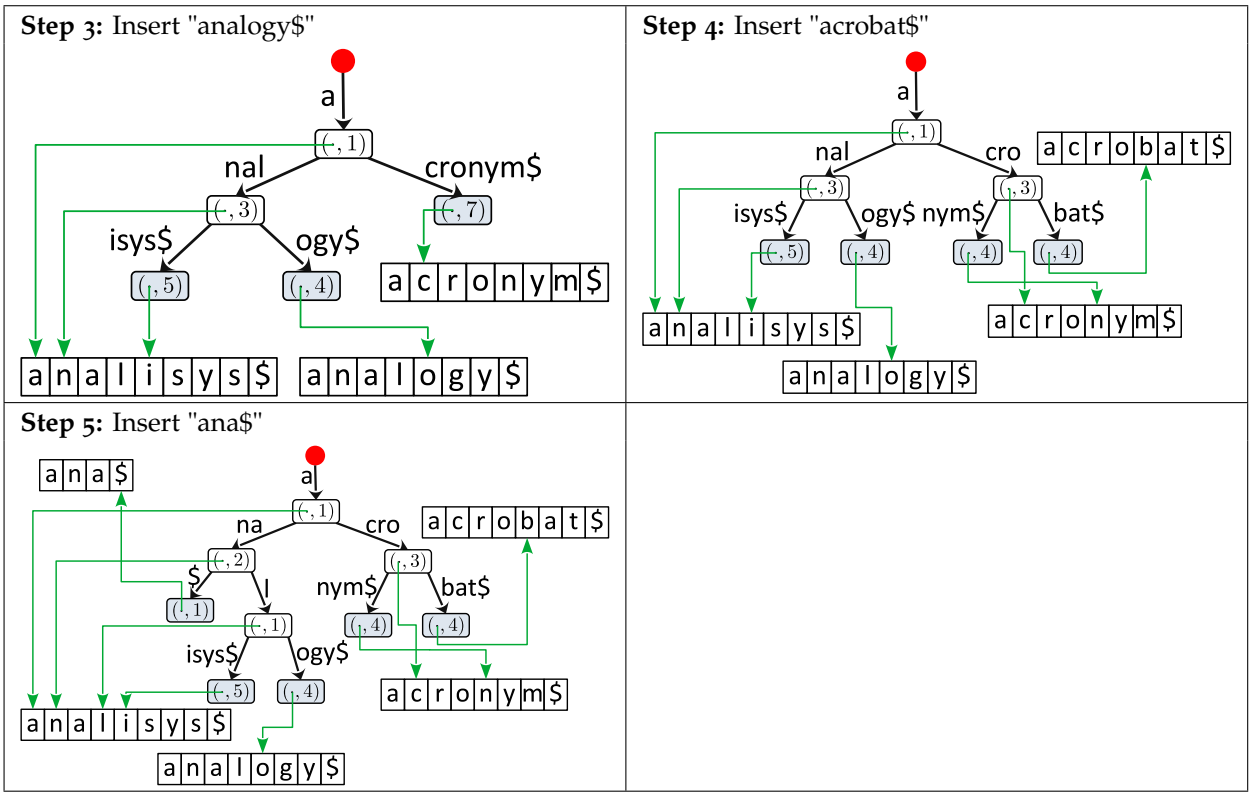
Let us show an evolution of a Patricia tree by inserting the following sequence of strings ⟨"analisys$", "acronym$", "analogy$", "acrobat$", "ana$"⟩.

Step 1: Insert "analisys$"



Step 2: Insert "acronym$"





Removal

Removing strings from Patricia tree is much like in Trie. We first need to find the terminating leaf node for the removed string s and move upwards the traversed vertical path of the string. Unlike in Trie, in Patricia tree we need to move only to the first parent of the terminating leaf node, since only the last transition in the vertical path defines the removed string s unshared suffix.

For example, we want to remove string "analisys$" from S_4. In

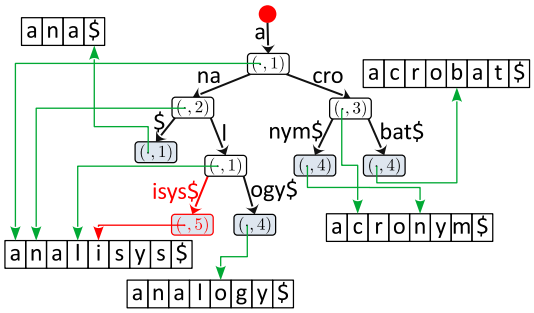


Figure 2.14: Removing the leaf node and the related transition from Patricia tree for S_4.

Figure 2.14 we can see the leaf node and its related transition that need to be removed.

After this removal, we have Patricia tree node that has an internal node with a single child, as seen in Figure 2.15. This is not allowed, as this is then not Patricia tree anymore. To resolve this issue, we need to remove the internal node that has the single child and con-

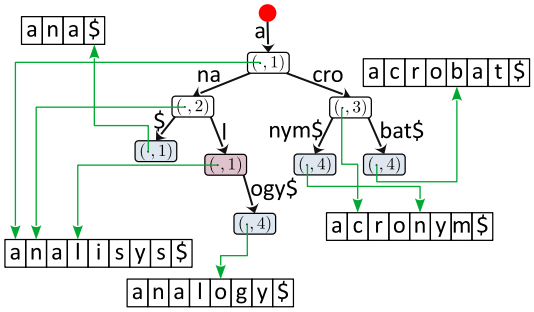


Figure 2.15: After removal we have one internal node with only one child.

catenate its transitions. In our example, two transitions concatenated are "l" and "ogy$". This can be easily achieved by pushing the new child node string pointer towards the start of the string as much characters as the length of the concatenated prefix.

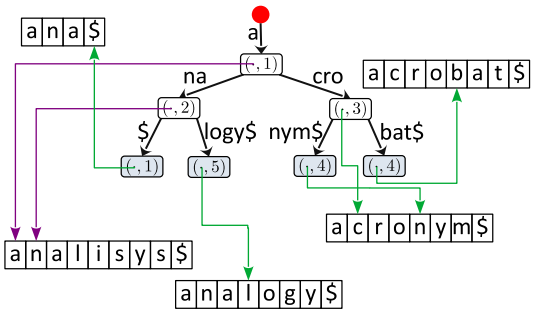


Figure 2.16: After removal of the internal node that had single child.

Algorithm 2.6: Removing algorithm for Patricia tree using pointer and length string representation.

procedure RemovePatricia($P, s = (s_p, s_l)$)
  $(succ, cn_{leaf}) \leftarrow$ SearchPatricia(P, s) ▷ $O(|s|)$
  if not *succ* **then**
    return
  $os \leftarrow$ string instance in cn_{leaf}
  $cn \leftarrow parent(cn_{leaf})$
  if cn is defined **then**
    remove cn_{leaf} from cn and the related transition
    $single \leftarrow$ check whether cn has only one child
    $firstChild \leftarrow$ retrieve the first child of cn ▷ $O(|\Sigma|)$
    if *single* and $parent(cn)$ is defined **then**
      $(cn_p, cn_l) \leftarrow$ string representation in cn
      remove cn from $parent(cn)$
      $(fc_p, fc_l) \leftarrow$ string representation in $firstChild$
      update node $firstChild$ with $(fc_p - cn_l, fc_l + cn_l)$
      add transition between $parent(cn)$ and $firstChild$
    if $firstChild$ is defined **then**
      $cn \leftarrow firstChild$
      while $parent(cn)$ is not NULL **do** ▷ $\approx O(|s|)$
        if string instance in $parent(cn) = os$ **then**
          string instance in $parent(cn) \leftarrow$ string instance in cn
        $cn \leftarrow parent(cn)$

In our example the concatenated prefix is "l" and 1 character long, so we need to move the pointer in the new child node for 1 character towards the start of the string, and increase the length of the substring for 1 character, which can be seen in Figure 2.16.

However, we encounter another type of a problem with this removal. In Figure 2.16 we can see that some nodes still point to the removed, now *orphaned* string. These nodes are on the search vertical path for the removed string, so we can easily traverse this path and replace these pointers to one of the remaining *sibling* strings.

The removal complexity is $O(|s| + |\Sigma\|)$, which consists of traversal to the leaf node and checking child nodes.

Listing 2.3: Patricia tree python code.

```
class PatriciaTreeNode:
  def __init__(self, s: str, p: int, l: int):
    self.s, self.p, self.l = s, p, l
    self.children = [None] * 256 #sigma=ascii
    self.cno, self.parent = 0, None
  def isLeaf(self) -> (bool):
    return self.cno == 0
  def transition(self, c: chr) -> (object):
    return self.children[ord(c)]
  def insert(self, cnc: object):
    if self.children[ord(cnc.s[cnc.p])] is None:
      self.cno += 1
      cnc.parent = self
    self.children[ord(cnc.s[cnc.p])] = cnc
  def remove(self, cnc: object):
    child = self.children[ord(cnc.s[cnc.p])]
    if child is not None:
      child.parent = None
      self.cno -= 1
      self.children[ord(cnc.s[cnc.p])] = None
  def checkChildren(self):
    if self.cno > 0:
      for i in range(0,255):
        if self.children[i] is not None:
          return (self.cno == 1, self.children[i])
    return (False, None)

class PatriciaTree:
  def __init__(self):
    self.root = PatriciaTreeNode(None, 0, 0)
  def search(self, q: str) -> (bool):
    (succ, cn) = self._search(q)
    return succ
  def _search(self, q: str):
    q = q + '$'
    q_p, q_l = 0, len(q)
    sc = 0
    cn = self.root
    while not cn.isLeaf():
      cnc = cn.transition(q[sc])
      if cnc is not None:
        t_p, t_l = cnc.p, cnc.l
        if q[sc:sc + t_l] == cnc.s[t_p:t_p + t_l]:
          sc, cn = sc + t_l, cnc
        else: return (False, None)
      else: return (False, None)
    return (sc == q_l, cn)
  ...
  ...
  def insert(self, s: str):
    s = s + '$'
    s_p, s_l = 0, len(s)
    sc = 0
    cn = self.root
    while cn==self.root or not cn.isLeaf():
      c = s[sc]
      cnc = cn.transition(c)
      if cnc is not None:
        t_p, t_l = cnc.p, cnc.l
        if s[sc:sc + t_l] == cnc.s[t_p:t_p + t_l]: #case 3
          cn,sc=cnc,sc+t_l
        else: #case 2
          #the first char is always matched
          for i in range(1, t_l):
            if s[sc + i] != cnc.s[cnc.p + i]: break
          cnins=PatriciaTreeNode(cnc.s,cnc.p,i)
          cnc.p,cnc.l=cnc.p+i,cnc.l-i
          cn.insert(cnins)
          cnins.insert(cnc)
          cnleaf=PatriciaTreeNode(s,sc+i,s_l-sc-i)
          cnins.insert(cnleaf)
          return
      else: #case 1
        cnleaf=PatriciaTreeNode(s,sc,s_l-sc)
        cn.insert(cnleaf)
        return
  def remove(self, s: str):
    (succ, cnleaf) = self._search(s)
    if not succ: return
    os = cnleaf.s
    cn = cnleaf.parent
    if cn is not None:
      cn.remove(cnleaf)
      (single, firstChild) = cn.checkChildren()
      if single and cn.parent is not None:
        cnpp = cn.parent
        cnpp.remove(cn)
        firstChild.p -= cn.l
        firstChild.l += cn.l
        cnpp.insert(firstChild)
      if firstChild is not None:
        cn = firstChild
        while cn.parent is not None:
          if cn.parent.s == os:
            cn.parent.s = cn.s
          cn = cn.parent
```

Using Patricia tree for IP lookup

The Patricia tree structure is widely used for a quick prefix lookup. Some generalized overview of Patricia tree and radix tree can be further studied in [Knuth, 2014, Sedgewick, 2002]. Some notable areas where Patricia trees are extremely useful are bioinformatics (genome and protein sequence storing and retrieval) and IP networking (lookup, filtering, routing, blocking, etc..). For IP networking, we usually use $\Sigma = \{0, 1\}$ Patricia tree, which is a binary radix tree. For the IP lookup purposes, we transform our IP addresses into a binary sequence. Each IPv4 address consists of 4 bytes as follows

$$192.168.11.45/32$$
$$11000000.10101000.00001011.00101101$$

or a subnet mask as

$$192.168.0.0/16$$
$$11000000.10101000.*.*$$

We start creating an IPv4 radix tree as in Figure 2.17 where the leaf

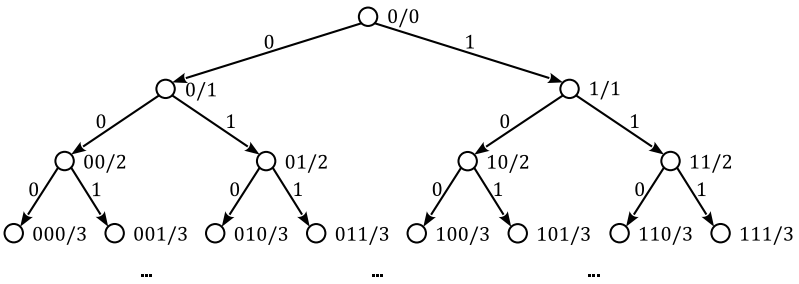


Figure 2.17: IPv4 radix tree.

nodes are IPv4 addresses at 32nd level. Since we do not want have all IPv4 addresses in our radix tree (only those that are of our interest), not having all leaf nodes on the 32nd level makes the higher levels of the IPv4 radix tree more sparse. IP subnet masks are on higher levels from the 32nd level. Starting from the root node, we

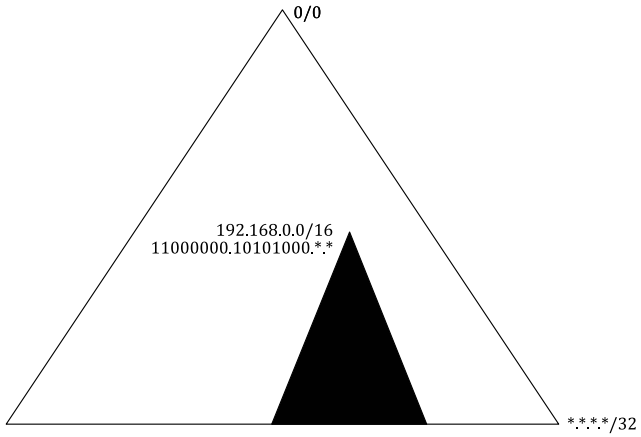


Figure 2.18: IPv4 radix tree subnet selection.

traverse down the IPv4 radix tree to the node 1100000010101000/16,

or 192.168.0.0/16, whose subtree represents all included subnets on levels 17-31, and all IP addresses on level 32.

So, we are talking about the IPv4 radix tree, but how do we compress such a tree? Or, where is the Patricia tree here? Following the previous work about the usage of Patricia trees in the network routing [1], we can show this concept on IP routing for the following 4 distinct subnets. We do not need to create full IPv4 radix tree with 32

[1] Keith Sklower. A tree-based packet routing table for berkeley unix. In *USENIX Winter*, volume 1991, pages 93–99. Citeseer, 1991

192.168.10.0/24	11000000.10101000.00001010	fer1.hr
192.168.11.0/24	11000000.10101000.00001011	fer2.hr
192.167.45.0/24	11000000.10100111.00101101	fer3.hr
192.167.92.0/24	11000000.10100111.01011100	fer4.hr

levels to store these subnets. Also, after the compression, the related Patricia tree looks as in the following Figure. In the leaf nodes we

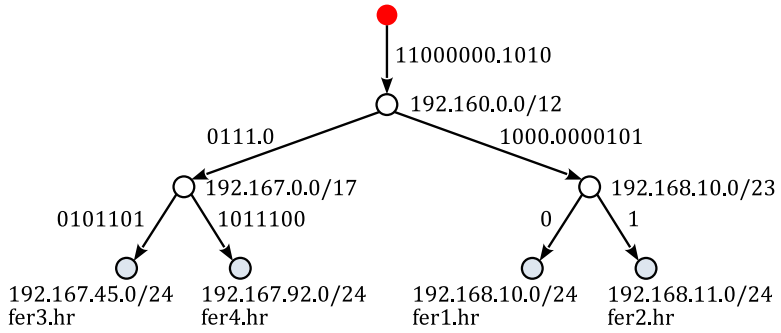


Figure 2.19: Patricia tree routing table.

have pointers to external data, which is then used for routing purposes. The Patricia tree is only a way of fast subnet lookup, which is extremely important in IP network routing. So, why aren't we using hash tables for this purpose? It turns out that calculating key hash takes more time than Patricia tree lookup.

2.3 *Suffix arrays*

Imagine we have a string t and we want to check whether some arbitrary query string q is contained in t. In other words, we want to check whether q is a substring of t. There is a naïve way to do it: we simply iterate through t and q in two nested iterations and check character by character, which gives us the complexity of $O(|t| * |q|)$, or quadratic complexity. Is it possible to do reduce this check complexity to linear? We start by introducing suffix arrays.

Definition 2.2 (Suffix array)**.** Let us define a string $t = t[1], t[2], ..., t[i], ..., t[n]$, where $t[i]$ is an i-th character of this string. Let us define $t_i = t[i], t[i+1], ..., t[n]$ as i-th suffix of t. Then $SA = \langle t_{A_1}, t_{A_2}, ..., t_{A_i}, ..., t_{A_n} \rangle$ is an alphabetically ordered array of all suffixes of t, or a suffix array, and $A = \langle A_1, A_2, ..., A_i, ..., A_n \rangle$ is the array of suffix array indices.

Let us define a string $t = program\$$ as follows

1	2	3	4	5	6	7	8
p	r	o	g	r	a	m	$

We can obtain all suffixes from t by iterating from each character to the end of the string. The pointer and length string representation, as used in Patricia trees, makes this process easy, as we need only to create $|t|$ ordered pairs (p, l), where each pair points to its own character with decrementing length. The initial array of suffixes looks as

1	p	r	o	g	r	a	m	$
2	r	o	g	r	a	m	$	
3	o	g	r	a	m	$		
4	g	r	a	m	$			
5	r	a	m	$				
6	a	m	$					
7	m	$						
8	$							

Table 2.2: Unordered suffix array.

which is then alphabetically ordered into

i	A_i	t_{A_i}							
1	8	$							
2	6	a	m	$					
3	4	g	r	a	m	$			
4	7	m	$						
5	3	o	g	r	a	m	$		
6	1	p	r	o	g	r	a	m	$
7	5	r	a	m	$				
8	2	r	o	g	r	a	m	$	

we get $A = \langle 8, 6, 4, 7, 3, 1, 5, 2 \rangle$, and suffix array $SA = \langle \$, \text{am}\$, \text{gram}\$, \text{m}\$, \text{ogram}\$, \text{program}\$, \text{ram}\$, \text{rogram}\$ \rangle$.

A naïve way of creating a suffix array from the string t is to use any $O(n \log n)$ sorting algorithm. After we have all suffixes of the string t, we need to compare pairs of suffixes to establish alphabetical ordering of the suffix array. Knowing that only one suffix has the length $n = |t|$, the complexity of the naïve creation is $O(nk \log n)$, where $k \leq n$. Only one suffix has $k = n$, all others are $k < n$, so we can approximate the complexity for $O(n^2 \log n)$. We can create a suffix array in linear time using algorithm such as *skew algorithm*[2], which is not going to be detailed here.

[2] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *International colloquium on automata, languages, and programming*, pages 943–955. Springer, 2003

Searching

The basic search over a suffix array is done by using the binary search algorithm. Let us say we want to search for a query string q. The

complexity of the binary search algorithm is $O(\log n)$. Each comparison takes checking $|q|$ characters. This makes the complexity of the binary search of the query string q over a suffix array $O(|q| \log n)$.

Longest Common Prefix (LCP)

Let us define a substring of the string t starting at the i-th character and ending at the j-th character. We denote such substring as $t[i, j]$.

Definition 2.3 (Longest Common Prefix (LCP)). Let us define two strings u and w. The number of characters $H = lcp(u, w)$ constitutes the longest common prefix of u and w only if $u[1, H] = w[1, H]$ and $u[1, H+1] \neq w[1, H+1]$.

Theorem 2.1. Let us define two suffixes that are non-subsequent (distant more than 1) in the alphabetically ordered suffix array, having indices i and j as follows

$$SA = \langle t_{A_1}, ..., t_{A_i}, t_{A_i+1}, ..., t_{A_j-1}, t_{A_j}, ..., t_{A_n} \rangle \tag{2.7}$$

where $j > i+1$ and $lcp(t_{A_i}, t_{A_j}) > 0$. For any suffix k in the suffix array, such that $i < k < j$, it is valid to consider that $lcp(t_{A_i}, t_{A_k}) \geq lcp(t_{A_i}, t_{A_j}) \wedge lcp(t_{A_k}, t_{A_j}) \geq lcp(t_{A_i}, t_{A_j})$.

Proof: If we take that the LCP has $H = lcp(t_{A_i}, t_{A_j})$ characters, then from Definition 2.3 we know that $t_{A_i}[1, H] = t_{A_j}[1, H]$. Let us consider that $lcp(t_{A_i}, t_{A_k}) < lcp(t_{A_i}, t_{A_j}) \vee lcp(t_{A_k}, t_{A_j}) < lcp(t_{A_i}, t_{A_j})$ is valid. This would mean that there is at least one character $t_{A_k}[l]$, where $1 \leq l \leq H$, is $t_{A_k}[l] < t_{A_i}[l]$ or $t_{A_k}[l] > t_{A_j}[l]$, which means that the suffix array is not alphabetically ordered, since the ordering of the indices is $i < k < j$. Finally, this means that for an alphabetically ordered suffix array we always must have $lcp(t_{A_i}, t_{A_k}) \geq lcp(t_{A_i}, t_{A_j}) \wedge lcp(t_{A_k}, t_{A_j}) \geq lcp(t_{A_i}, t_{A_j})$.

We can test the previous theorem on an example. Let us say we have $t = banana\$$. This result in the following suffix array

i	A_i	t_{A_i}						
1	7	$						
2	6	a	$					
3	4	a	n	a	$			
4	2	a	n	a	n	a	$	
5	1	b	a	n	a	n	a	$
6	5	n	a	$				
7	3	n	a	n	a	$		

Table 2.3: LCP between t_{A_2} and t_{A_4}, marked as red, and then LCP between t_{A_3} and t_{A_4}. $lcp(t_{A_3}, t_{A_4}) \geq lcp(t_{A_2}, t_{A_4})$.

If we calculate $i = 2, j = 4, H = lcp(t_{A_2}, t_{A_4})$, we get $H = 1$. Then all suffixes between $i = 2$ and $j = 4$ share the same prefix as in Table 2.3.

Fast binary search with ℓ-matrix

We can sped up the binary search on a suffix array from $O(|q| \log n)$ to $O(|q| + \log n)$. This can be only done by separating comparison between substrings and binary search suffix array traversal. To prepare, along with suffix array creation, we need to create an ℓ-matrix, which basically stores values $\ell_{i,j} = lcp(t_{A_i}, t_{A_j})$ between all pairs of suffixes in the suffix array.

The algorithm uses two indices i and j, such that $1 \leq i < j \leq n$. Then we calculate an index in the middle of this range $m = \lfloor (j - (i-1))/2 \rfloor + i$. We need the matching prefix length between q and suffixes at indices i and j as $a = lcp(q, t_{A_i}), b = lcp(q, t_{A_j})$.

For example, we have the following alphabetically ordered string array (not a suffix array, the same principle applies), where we want to search $q = programmer\$$.

$$
\begin{aligned}
&\ldots \\
s_i =& pro gramm able\$ && a = 8 \\
&\ldots \\
s_m =& pro \ldots \$ \\
&\ldots \\
s_j =& protect\$ && b = 3 \\
&\ldots
\end{aligned}
$$

The extended binary algorithm is recursive and takes four parameters (i, j, a, b), where a is the number of matched prefix characters between s_i and q, and b is the number of matched prefix characters between s_j and q. If $a = |q| \vee b = |q|$ we already found the queried string q as s_i or s_j, otherwise we start searching for the string q using the initial parameters $(1, n, a, b)$. The algorithm is recursive, which is then stopped when we find the string q or we come to $j = i + 1$.
In each pass, we have three cases:

1. $\ell_{i,m} < a$: This means that $s_m[\ell_{i,m} + 1] > s_i[\ell_{i,m} + 1]$, and by that $s_m[\ell_{i,m} + 1] > q[\ell_{i,m} + 1]$. This means that q can be found somewhere between s_i and s_m. We recursively call with new parameters $(i, m, a, \ell_{i,m})$. An example would be $s_m = progress\$$,

2. $\ell_{i,m} > a$: This means that $s_m[a + 1] = s_i[a + 1]$. Since $s_i[a + 1] < q[a + 1]$, we conclude that also $s_m[a + 1] < q[a + 1]$, therefore q must be found somewhere between s_m and s_j. We recursively call with new parameters (m, j, a, b). An example would be $s_m = programmatically\$$.

3. $\ell_{i,m} > a$: The matched a characters exists in s_m as well, and we cannot decide which half of strings in the array contains the query

string q without further analysis. Therefore, we try to match further k characters after the already matched a characters. We can stop at the $(k+1)$-th character, or after we have no more characters to match, and the outcome can be:

(a) $q[a+k+1] < s_m[a+k+1]$: $(a+k+1)$-th character of s_m is ***after*** the $(a+k+1)$-th character of q, which means that q can be found somewhere between s_i and s_m. We recursively call with new parameters $(i, m, a, a+k)$. An example would be $s_m = programmes$...\$,

(b) $q[a+k+1] > s_m[a+k+1]$: $(a+k+1)$-th character of s_m is ***before*** the $(a+k+1)$-th character of q, which means that q can be found somewhere between s_m and s_j. We recursively call with new parameters $(m, j, a+k, b)$. An example would be $s_m = programmed$...\$,

(c) $a+k = |q|$, we expect that $q = s_m$, which means we found the queried string q. An example would be $s_m = programmers$\$.

Notice that \$ is sorted before other characters. Encountering \$ in the tested s_m would be the case $3b$. This occurs always when $|s_m| < |q|$.

One can say that calculating the ℓ-matrix would be impractical, given the fact that we sparsely use its values. Without getting into details, we can immediately notice that $\ell_{j,i}, i < j$ makes no sense, as we only need i to be alphabetically **before** j. Also, we can focus to calculate only LCPs that we can encounter in the algorithm pass, such as $\ell_{i,m}$. In this manner, we get a sparsely populated ℓ-matrix, which lowers down the matrix calculation efforts.

Algorithm 2.7: Fast search for a suffix array using ℓ-matrix.

```
function SearchSA(i, j, a, b, array, q, ℓ)
    if j = i + 1 then
        return false
    m ← ⌊(j − (i − 1))/2⌋ + i
    if ℓ[i, m] < a then
        return SearchSA(i, m, a, ℓ[i, m], array, q, ℓ)
    else if ℓ[i, m] > a then
        return SearchSA(m, j, a, b, array, q, ℓ)
    else
        s_m ← array[m]
        for k in range a + 1 to min(|s_m|, |q|) do
            if q[k] < s_m[k] then
                return SearchSA(i, m, a, a + k, array, q, ℓ)
            else if q[k] > s_m[k] then
                return SearchSA(m, j, a + k, b, array, q, ℓ)
            else
                if k = |q| then
                    return true
        return false
```

Listing 2.4: Suffix array python code.

```
class SuffixArray:
  def __init__(self, s: str):
    self.str = StringPL(s, 0, len(s))
    tmp = []
    for i in range(0,len(self.str.s)):
      tmp.append(self.str.clone(i,len(self.str.s)-i))
    self.SA = sorted(tmp)
    self._initSA()
  def _initSA(self):
    self.n = len(self.SA)
    self.l = [[0]*self.n for i in range(self.n)]
    # naive construction of the l-matrix
    for i in range(0,self.n):
      for j in range(i+1,self.n):
        lcp = self.SA[i].lcp(self.SA[j])
        self.l[i][j]=self.l[j][i] = lcp
  def search(self, q: str) -> (bool):
    if q[len(q)-1] != '$': q = q + '$'
    i, j = 0, self.n - 1
    a, b = self.SA[i].lcp(q), self.SA[j].lcp(q)
    if a==len(q) or b==len(q): return True
    return self._searchSA(i, j, a, b, q)
  ...
```

```
...
def _searchSA(self, i: int, j: int, a: int, b: int,
              q: str) -> (bool):
  if j==i+1: return False
  m = floor((j-(i-1))/2)+i
  if self.l[i][m] < a:
    return self._searchSA(i,m,a,self.l[i][m],q)
  elif self.l[i][m] > a:
    return self._searchSA(m,j,a,b,q)
  else:
    sm = self.SA[m]
    for k in range(a,min(len(self.SA[m]),len(q))):
      ch, smk = q[k], sm[k]
      if ch=='$': return True
      if ch < smk: return self._searchSA(i,m,a,k,q)
      elif ch > smk: return self._searchSA(m,j,k,b,q)
      else:
        if ch!='$' and k==len(q)-1: return True
    return False
```

Constructing LCP array and ℓ-matrix in linear time

First, let us review some LCP properties, which are useful for calculating LCP array faster than $O(n \log n)$ [Kasai et al., 2001]. We start by repeating the suffix array for $t = banana\$$.

i	A_i	t_{A_i}						
1	7	$						
2	6	a	$					
3	4	a	n	a	$			
4	2	a	n	a	n	a	$	
5	1	b	a	n	a	n	a	$
6	5	n	a	$				
7	3	n	a	n	a	$		

$j = A_i$	$R_j = i$	t_j						
1	5	b	a	n	a	n	a	$
2	4	a	n	a	n	a	$	
3	7	n	a	n	a	$		
4	3	a	n	a	$			
5	6	n	a	$				
6	2	a	$					
7	1	$						

Lemma 2.1 (Enclosing LCP). Let us define the following alphabetically ordered suffix indices $i < k < j - 1$, for which we claim

$$lcp(t_{A_i}, t_{A_j}) \leq lcp(t_{A_k}, t_{A_{k+1}}) \tag{2.8}$$

Generally, for any $i < j - 1$, we can calculate its LCP as

$$lcp(t_{A_i}, t_{A_j}) = min\{lcp(t_{A_i}, t_{A_{i+1}}), ..., lcp(t_{A_{j-1}}, t_{A_j})\} \tag{2.9}$$

as the minimal LCP for all alphabetically ordered adjacent suffixes.

An additional proof for Lemma 2.1 can be found in the proof of Theorem 2.1. As the range between alphabetically ordered suffixes increases, the LCP between them becomes lower.

Lemma 2.2. Let us define a suffix index i, such that $0 < i \leq n$. If we have

$$lcp(t_{A_{i-1}}, t_{A_i}) > 1 \tag{2.10}$$

then we can claim

$$lcp(t_{A_{i-1}}, t_{A_i}) = lcp(t_{A_{i-1}+1}, t_{A_i+1}) - 1 \tag{2.11}$$

For the alphabetical ordering, we know that

$$R_{A_{i-1}+1} < R_{A_i+1} \tag{2.12}$$

For $t = banana\$$, if we take $i = 4$, then we have $lcp(t_{A_3}, t_{A_4}) = lcp(t_4, t_2) = 3$, which is

i	A_i	t_{A_i}						
1	7	$						
2	6	a	$					
3	4	a	n	a	$			
4	2	a	n	a	n	a	$	
5	1	b	a	n	a	n	a	$
6	5	n	a	$				
7	3	n	a	n	a	$		

since $lcp(t_{A_3}, t_{A_4}) > 1$, then we have $lcp(t_{A_3+1}, t_{A_4+1}) = lcp(t_5, t_3) = lcp(t_{A_3}, t_{A_4}) - 1 = 2$, or as follows

i	A_i	t_{A_i}						
1	7	$						
2	6	a	$					
3	4	a	n	a	$			
4	2	a	n	a	n	a	$	
5	1	b	a	n	a	n	a	$
6	5	n	a	$				
7	3	n	a	n	a	$		

	$j = A_i$	$R_j = i$	t_j						
	1	5	b	a	n	a	n	a	$
	2	4	a	n	a	n	a	$	
$A_i + 1$	3	7	n	a	n	a	$		
	4	3	a	n	a	$			
$A_{i-1} + 1$	5	6	n	a	$				
	6	2	a	$					
	7	1	$						

Definition 2.4 (LCP array). We define an array $H = \langle H_1 = 0, H_2, ..., H_n \rangle$ as an array of LCP values, where $H_i = lcp(t_{i-1}, t_i)$ for all $i > 1$.

i	A_i	H_i	t_{A_i}						
1	7	0	$						
2	6	0	a	$					
3	4	1	a	n	a	$			
4	2	3	a	n	a	n	a	$	
5	1	0	b	a	n	a	n	a	$
6	5	0	n	a	$				
7	3	2	n	a	n	a	$		

Lemma 2.3. If $lcp(t_{A_j-1}, t_{A_l-1}) > 1$, we have $lcp(t_{A_k}, t_{A_l}) \geq lcp(t_{A_j}, t_{A_l})$ and $R_j \leq R_k = R_l - 1 = q - 1$.

Proof: By Lemma 2.2 we know that if $lcp(t_{A_j-1}, t_{A_l-1}) > 1$ then $R_j < R_l$. Then by Lemma 2.1 we know that $lcp(t_{A_k}, t_{A_l}) \geq lcp(t_{A_j}, t_{A_l})$ for $R_j < R_k < R_l$.

	i	A_i		t_{A_i}							
	1	7		\$							
	2	6		a	\$						
$p-1$	3	4	$j-1 = A_{q-1}-1$	a	n	a	\$				t_{j-1}
p	4	2	$l-1 = A_q-1$	a	n	a	n	a	\$		t_{l-1}
	5	1		b	a	n	a	n	a	\$	
$q-1$	6	5	$j = k = A_{q-1} = A_{p-1}+1$	n	a	\$					$t_j = t_k$
q	7	3	$l = A_q = A_p+1$	n	a	n	a	\$			t_l

Theorem 2.2 (Fast LCP array calculation). If we know $H[p]$, then we have

$$H_p = lcp(t_{j-1}, t_{l-1}) = lcp(t_{A_p}, t_{A_{p-1}}) > 1 \tag{2.13}$$

By knowing this, we also know that

$$H_q = lcp(t_k, t_l) = lcp(t_{A_q}, t_{A_{q-1}}) \geq H_p - 1 \tag{2.14}$$

Proof:
By using Theorem 2.2 we can construct the following algorithm

Algorithm 2.8: Linear LCP array constructing algorithm.

```
function ConstructLCPArray(t, A, n)
    H ← array of size n having values 0
    for i in range from 0 to n do
        R_{A_i} ← i
    h ← 0
    for i in range from 0 to n do
        if R_i > 0 then
            k ← A_{R_i − 1}
            while t[i + h] == t[k + h] do
                h ← h + 1
            H_{R_i} ← h
            if h > 0 then
                h ← h − 1
    return H
```

The previous algorithm constructs an LCP array from a suffix array in $O(n)$ time.

Once we have the LCP array for the suffix array, we want to construct its ℓ-matrix. As mentioned, we do not need to calculate all n^2 values, but only those $\ell_{i,m}$ that are used for the linear suffix array binary search algorithm, presented in Algorithm 2.7.

For this algorithm, we need the values that are power of 2, since the algorithm is an adaptation of the binary search algorithm. Therefore, we can have the following pairs of values for $\ell_{i,m}$

$$\begin{aligned} i &= 1 + q2^p \\ m &= min(n, i + 2^p) \end{aligned} \tag{2.15}$$

for all

$$\begin{aligned} p &= \{0, ..., \lceil \log n \rceil - 1\} \\ q &= \{0, ..., \lceil n/2^p \rceil - 1\} \end{aligned} \tag{2.16}$$

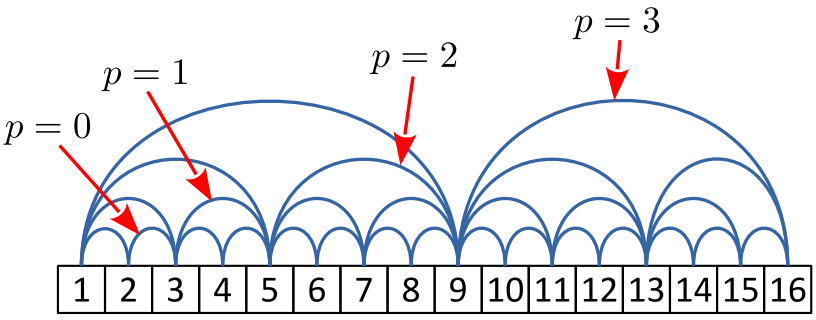


Figure 2.20: ℓ-matrix pairs needed for the binary search algorithm.

Using pairs in Figure 2.20 and Lemma 2.1, especially (2.9), we get the following small algorithm for calculating the ℓ-matrix.

Algorithm 2.9: ℓ-matrix constructing algorithm from LCP array.

function CONSTRUCTLMATRIX(H, n)
  $\ell \leftarrow$ matrix of size nxn having values 0
  for p in range from 0 to $\lceil \log n \rceil - 1$ **do**
    for q in range from 0 to $\lceil n/2^p \rceil - 1$ **do**
      $i \leftarrow q2^p$
      $m \leftarrow min(n, i + 2^p)$
      $\ell_{i,m} \leftarrow min(H_{i+1}, ..., H_m)$
  return ℓ

Listing 2.5: Linear creation suffix array python code.

```
def LinearLCPArray(t: str, A: array) -> (array):
  n = len(t)
  R, H = [0]*n, [0]*n
  for i in range(0, n):
    R[A[i]]=i
  h = 0
  for i in range(0, n):
    if R[i] > 1:
      k = A[R[i]-1]
      while t[i+h] == t[k+h]:
        h = h + 1
      H[R[i]] = h
      if h > 0: h = h - 1
    return H

def CreateLMatrix(n: int, H: array) -> (array):
  lM = [[0]*n for i in range(n)]
  for p in range(0,ceil(log2(n))-1):
    for q in range(0,ceil(n/pow(2,p))-1):
      v1 = q*pow(2,p)
      v2 = min(n, v1 + pow(2,p))
      Hslice=H[v1+1:v2+1]
      lM[v1][v2] = lM[v2][v1] = min(Hslice)
  return lM
```

```
class SuffixArray:
  def __init__(self, s: str):
    self.str = StringPL(s, 0, len(s))
    self.n = len(s)
    tmp1 = []
    for i in range(0,len(self.str.s)):
      tmp1.append((i,self.str.clone(i,len(self.str.s)-i)))
      tmp2 = sorted(tmp1,key=lambda x: x[1],reverse=False)
    A, self.SA = [0]*self.n, ['']*self.n
    for i in range(0,self.n):
      A[i]=tmp2[i][0]
      self.SA[i]=tmp2[i][1]
    H = LinearLCPArray(self.str.s, A)
    self.l = CreateLMatrix(self.n, H)
```

2.4 *Suffix trees*

Finding substrings in a string t using suffix arrays approach can be further extended by adding a tree structure to it. The final result are *Suffix Tries* and *Suffix trees*.

Definition 2.5 (Suffix Trie). A Trie (Definition 2.1) transformed from all suffixes of a string t, as seen in Table 2.2, is called a *Suffix Trie*. Each leaf node of the Suffix Trie contains a number that represents the suffix first character i of the vertical path representing t_i (as in Definition 2.2), ending in the leaf node.

Each edge of the Suffix Trie contains one character from the alphabet Σ, which means that Suffix Trie is uncompressed, like the original generic Trie. Suffix based substring matching data structures, such as Suffix tree, are often used in bioinformatics, for example in genome storing and retrieval. For that reason, many examples are based on the alphabet $\Sigma = \{C, A, G, T, \$\}$.

All suffixes from the string $t = CAGAGG\$$ can be seen in Table 2.4. Using these suffixes, we construct the following Suffix Trie. As seen

C	A	G	A	G	G	$
A	G	A	G	G	$	
G	A	G	G	$		
A	G	G	$			
G	G	$				
G	$					
$						

Table 2.4: All suffixes for $t = CAGAGG\$$.

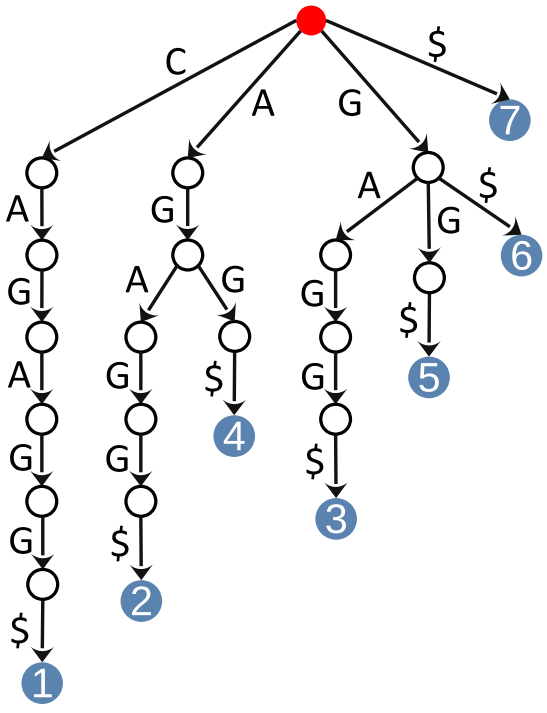


Figure 2.21: Suffix Trie for $t = CAGAGG\$$.

in Figure 2.21, each leaf node has a number that represents the first character of the suffix ending in this leaf node. For example, $t_4 = AGT\$$ ends in the leaf node marked with number 4.

Definition 2.6 (Suffix tree). A compressed Suffix Trie is called Suffix tree. The compression is achieved the same way as for Patricia trees.

Definition 2.7 (Implicit Suffix tree). An implicit Suffix tree is obtained by the following transformation:

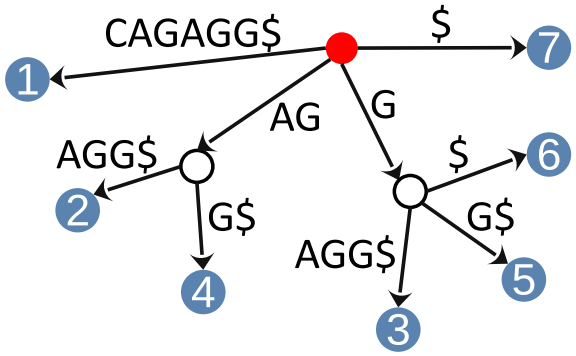


Figure 2.22: Suffix tree for $t = CAGAGG\$$.

1. Remove all NULL-terminating characters \$ from edges of the Suffix tree,
2. Remove all edges that are left with an empty string ϵ, including their subtrees,
3. Compress the Suffix tree again. All nodes except the root node that less than two children are removed and their incoming and outgoing edges are concatenated.

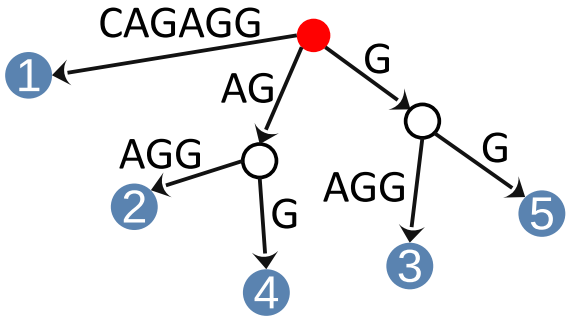


Figure 2.23: Implicit Suffix tree for $t = CAGAGG\$$.

Ukkonen's algorithm for linear Suffix tree construction

The naïve attempt to create a Suffix tree from all suffixes of the string t takes $O(n^2)$ to create. This has been improved multiple times with algorithms given by McCreigh [McCreight, 1976] and Weiner [Weiner, 1973]. Ukkonen's algorithm [Ukkonen, 1995] is an extension of much simpler Weiner's algorithm.

Given the input string t having length $n = |t|$, Ukkonen's algorithm is executed in n phases, each for one character of the input string t. For that reason the algorithm is called an *online Suffix tree construction algorithm*, and has the complexity $O(n)$. For each phase of the algorithm, we start with **implicit Suffix tree** S_i, which is then extended by the $i+1$-th phase (character $t[i+1]$) to the implicit Suffix tree S_{i+1}. $i+1$-th phase is then performed in $i+1$ steps, where in the first i steps we add the character $t[i+1]$ to all suffixes of the prefix $t[1,i]$ in the implicit Suffix tree. In the $i+1$-th step we add the character $t[i+1]$ from the root node of the implicit Suffix tree.

Phase 1, $i+1=1, t[i+1]=A$							
Step 1	A						
Phase 2, $i+1=2, t[i+1]=B$							
Step 1, $j=1$	A	B					
Step 2	B						
Phase 3, $i+1=3, t[i+1]=A$							
Step 1, $j=1$	A	B	A				
Step 2, $j=2$	B	A					
Step 3	A						
Phase 4, $i+1=4, t[i+1]=B$							
Step 1, $j=1$	A	B	A	B			
Step 2, $j=2$	B	A	B				
Step 3, $j=3$	A	B					
Step 4,	B						
...							
Phase 7, $i+1=7, t[i+1]=C$							
Step 1, $j=1$	A	B	A	B	A	B	C
Step 2, $j=2$	B	A	B	A	B	C	
...							
Step 6, $j=6$	B	C					
Step 7	C						

Table 2.5: Phases and steps of the Ukkonen's algorithm for $t = ABABABC\$$.

For $t = ABABABC\$$, an example would be as in Table 2.5. We start by having an empty implicit Suffix tree, containing only the root node. Obviously, in the first phase we add only one leaf and one transition from the root node to the newly added leaf node that has only $t[1]$. In each subsequent step $(i, j), 1 \leq j \leq i$, we traverse the Suffix tree for the all suffixes of the prefix $t[1, i]$, previously added in the tree, and then add the character $t[i+1]$. We can encounter three cases:

1. We traversed to a leaf node, which means that suffix $t[j, i]$ spans the full traversed vertical path, and we can simply add the character $t[i+1]$ to the last transition before the leaf node,

2. We stopped in the middle of a transition in the implicit Suffix tree and the following character of the transition is not $t[i+1]$. We insert $t[i+1]$ into the tree in the same manner as inserting a character in Patricia tree (Algorithm 2.5),

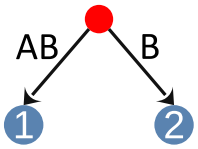
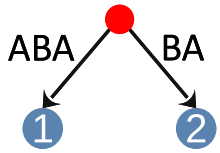
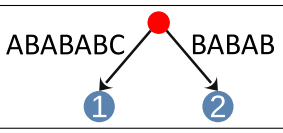
3. We stopped in the middle of a transition in the implicit Suffix tree and the following character of the transition is $t[t+1]$. We do nothing, as the substring $t[j, i]t[i+1]$ is already in the tree.

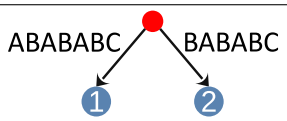
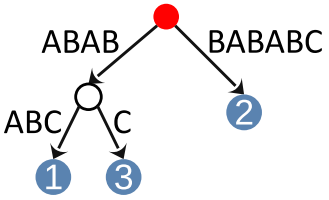
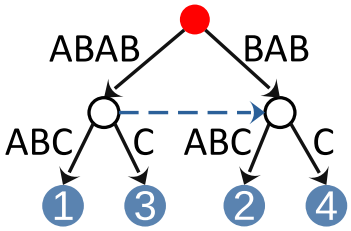
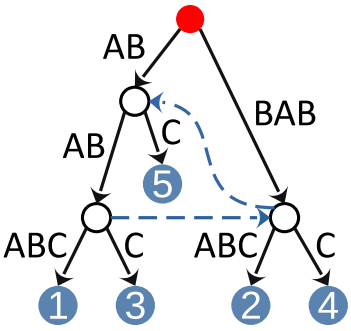
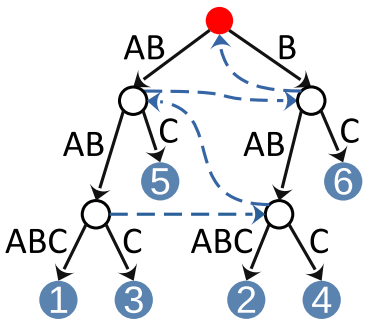
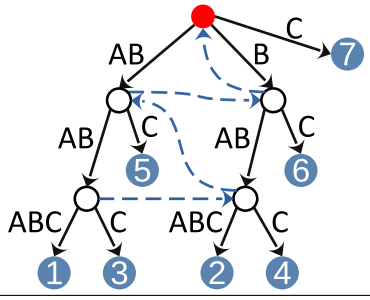
After we finish the implicit Suffix tree using the Ukkonen's algorithm, we transform in into an explicit Suffix tree, by adding the NULL terminating end \$ in the same manner as all other characters.

Suffix links

Suffix links are very important in Ukkonen's algorithm, as they are in fact shortcuts between suffixes of the prefix $t[1,i]$. Using suffix links we decrease the number of steps in each phase, by knowing all nodes where traversal for suffixes of the prefix $t[1,i]$ are ending, and extending the implicit Suffix tree from multiple nodes simultaneously.

Suffix links are created between internal nodes. Remember, internal nodes are only added when we add character $t[i+1]$ in the middle of transition. If we had a suffix $t[j,i]$, where $j > 1$, for which we traversed from the root node to some internal node n_1, then there might be some other non-leaf node n_2 which is at the end of traversal for the suffix $at[j,i]$. n_2 is the root node in case when $a = \epsilon$, which is when $j = i$. A Suffix link connects node n_2 to n_1 to let us know that traversal from the root node to the node n_2 is a sub-suffix of the traversal from the root node to the node n_1. Which means that for all phases $\geq i$ we need to start from both nodes n_1 and n_2, thereby reducing the number of traversal steps for at least two steps in all future phases.

<table>
<tr><td colspan="2">Phase 1: $i+1=1, t[i+1]=A$</td></tr>
<tr><td>Step 1: We have only root node. The initial Suffix tree S_0 is extended by adding one leaf node and a transition A leading to the leaf node.</td><td> 
</td></tr>
<tr><td colspan="2">Phase 2: $i+1=2, t[i+1]=B$</td></tr>
<tr><td>Step 1, Suffix: $t[1,1]=A$: Traversing A lead us to the leaf node. This is case 1, and we add B to the transition.</td><td rowspan="2"> 
</td></tr>
<tr><td>Step 2: Since the traversing suffix is ϵ and we cannot traverse for the character $t[i+1]=B$, we add new transition B to the root node, leading to the newly added leaf node.</td></tr>
<tr><td colspan="2">Phase 3: $i+1=3, t[i+1]=A$</td></tr>
<tr><td>Step 1, Suffix: $t[1,2]=AB$: Traversing AB lead us to the leaf node 1. This is case 1, and we add A to this transition.</td><td rowspan="3"> 
</td></tr>
<tr><td>Step 2, Suffix: $t[2,2]=B$: Traversing B lead us to the leaf node 2. This is case 1, and we add A to this transition.</td></tr>
<tr><td>Step 3: Since the traversing suffix is ϵ and we can traverse for the character $t[i+1]=A$ towards the leaf node 1, we do nothing as this is the case 3.</td></tr>
<tr><td colspan="2">Skipping multiple repetitive phases similar to Phase 3 ...</td></tr>
<tr><td colspan="2">Phase 7: $i+1=7, t[i+1]=C$</td></tr>
<tr><td>Step 1, Suffix: $t[1,6]=ABABAB$: Traversing $ABABAB$ lead us to the leaf node 1. This is case 1, and we add C to this transition.</td><td> 
</td></tr>
</table>

Step 2, Suffix: $t[2,6] = BABAB$: Traversing $BABAB$ lead us to the leaf node 2. This is case 1, and we add C to this transition.	
Step 3, Suffix: $t[3,6] = ABAB$: Traversing $ABAB$ lead us to the middle of transition, where the next character is not $t[i+1] = C$ but rather A. This is case 2, and we add internal node to the transition leading to the leaf node 1, splitting the traversal paths for $ABABABC$ and $ABABC$.	
Step 4, Suffix: $t[4,6] = BAB$: Traversing BAB lead us to the middle of transition, where the next character is not $t[i+1] = C$ but rather A. This is case 2, and we add internal node to the transition leading to the leaf node 2, splitting the traversal paths for $BABABC$ and $BABC$. At this moment, we have two internal nodes, where n_1 has the traversal path BAB from the root node, and n_2 has the traversal path $ABAB$ from the root node. We connect n_2 to n_1 with suffix link.	
Step 5, Suffix: $t[5,6] = AB$: Traversing AB lead us to the middle of transition, where the next character is not $t[i+1] = C$ but rather A. This is case 2, and we add internal node to the transition leading to the internal node, splitting the traversal paths for $\{ABABABC, ABABC\}$ and ABC. At this moment, we have three internal nodes. One internal node n_1 has the traversal path AB from the root node, and n_2 has the traversal path BAB from the root node. We connect n_2 to n_1 with suffix link.	
Step 6, Suffix: $t[6,6] = B$: Traversing B lead us to the middle of transition, where the next character is not $t[i+1] = C$ but rather A. This is case 2, and we add internal node to the transition leading to the internal node, splitting the traversal paths for $\{BABABC, BABC\}$ and BC. At this moment, we have three internal nodes. One internal node n_1 has the traversal path B from the root node, and n_2 has the traversal path AB from the root node. We connect n_2 to n_1 with suffix link. We also add a suffix link between internal node with traversing path B from the root node and the root node.	
Step 7: Since the traversing suffix is ϵ and we cannot traverse for the character $t[i+1] = C$, we add new transition C to the root node, leading to the newly added leaf node.	

At this moment we can transform to the explicit Suffix tree, by adding the NULL-terminating character, as seen In Figure 2.24.

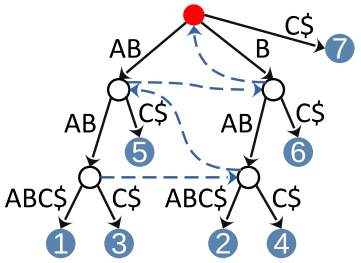


Figure 2.24: Suffix tree for $t = ABABABC\$$.

Finally, to show importance of suffix links, we can add another character G to the implicit Suffix tree. By adding a new character we need to perform *Phase 8:* $i+1 = 8, t[i+1] = D$ of the implicit Suffix tree S_7. The prefix for the Phase 8 is $t[1,7] = ABABABC$. We start by traversing the longest suffix from it $t[1,7] = ABABABC$. At some point during this traversal, for $ABAB$ precisely, we encounter an internal node that has suffix link to another internal node, which has a suffix link leading to another internal node, and so on. We encountered 4 internal nodes and the root node being connected with suffix links after traversing the path $ABAB$, stopping at the 5th character. Therefore, we know that all sub-suffixes of the suffix $ABAB$, or $\{ABAB, BAB, AB, B\}$, can be continued traversing from these 5 nodes. We know that all these sub-suffixes end at the 4th character, which means that our traversal needs to start from the 5th character. However, we see that from the suffix $t[3,7]$ we can skip traversal due to suffix links all the way to the 6th character. When implementing Ukkonen's algorithm, it is wise to have a map that maps a suffix character index to the set of internal nodes that offer traversal skipping, so we can always skip to the farthest sub-suffix and save as more traversal work as possible.

Looking at Table 2.6, we get to these 5 nodes by traversing the suffix $t[1,7] = ABABABC$. Later on, when traversing $t[3,7]$ we can push these 5 nodes to the 6th character. Once we get these skipping nodes, we can skip traversing all sub-suffixes, which are marked as red in the table.

$t[1,7]$	A	B	A	B	A	B	C
$t[2,7]$		B	A	B	A	B	C
$t[3,7]$			A	B	A	B	C
$t[4,7]$				B	A	B	C
$t[5,7]$					A	B	C
$t[6,7]$						B	C
$t[7,7]$							C

Table 2.6: Suffix traversal skipping using suffix links for $t = ABABABCD$.

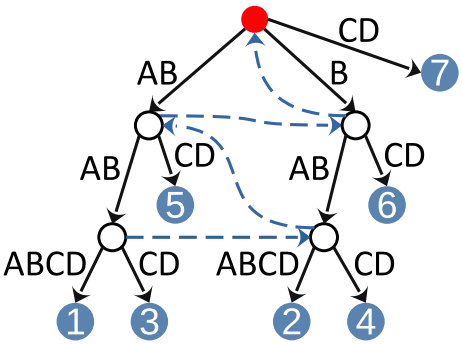


Figure 2.25: Implicit Suffix tree for $t = ABABABCD$.

Looking at Table 2.6 and Figure 2.25, we can see that once we skip all sub-suffixes by using suffix links, all we need to traverse are transitions to leaf nodes. All three cases can happen in these transitions, whereas only the case 2 can result in creation of new suffix links.

Python code that contains Suffix tree and Ukkonen's algorithm implementation is extensive and can be found in the lecture book python codes on github.

3 Geometric algorithms

Geometry is a branch of mathematics that has been used for a long time. It studies positions, shapes, sizes, angles, and dimensions of objects in space. Until recently, Euclidean geometry was the only form of geometry. Its non-Euclidean counterparts that appeared in the last century are differential geometry, algebraic geometry, computational geometry, algebraic topology, combinatorial geometry, and so on. We focus on computational geometry [De Berg et al., 2008], which started in the 1970s and can be defined as *a systematic approach of studying data structures and algorithms in geometry*. Many practical geometric problems require a systematic approach and usage of computers, which consequently resulted in the creation of numerous data structures and algorithms for their solution. There are numerous use cases for computational geometry. To enumerate a few:

- **Computer graphics** is one of the most developed usages of geometry in computers. It covers many things we see on the computer screen, from computer games, modelling tools, visualizations, projections, and so on. Nowadays, we cannot imagine a modern movie without computer-generated imagery (CGI) effects. Computer graphics is a wide field that comprises many mathematical formalisms and algorithms, and as such is the subject for itself.
- **Spatial product design**, which represents procedures for optimizing physical object placement, to support optimal physical product layout, such as circuit board design, integrated chip design, product packaging, and similar. **CAD/CAM** are generic design tools that utilize geometric algorithms for their work.
- **Robotics**. For calculating robotic movements and trajectories.
- **Geographic information systems** require strong geometrical mathematical background. Spatial projections, object placing, calculating trajectories, spatial searching, and many other features of GIS require geometrical data structures and algorithms.

As part of this chapter, we give selected geometric algorithms, which can be used in other algorithms in this book or are their natural extensions.

3.1 Points, segments, polygons

To start defining basic geometrical elements we start from the Euclidean space, which can be defined as a vector space E having an inner product

$$\begin{aligned} \vec{E} \times \vec{E} &\to \mathbb{R} \\ (x, y) &\mapsto \langle x, y \rangle \end{aligned} \tag{3.1}$$

The Euclidean vector space is an abstract space, which is usually laid over an infinite set of numbers, $\mathbb{R}$ or $\mathbb{C}$ for example. We can say that the n-dimensional Euclidean space over the set of real numbers is $\mathbb{R}^n$. There is an isomorphism from the Euclidean space to $\mathbb{R}^n$ that associates each point in the Euclidean space to the element from $\mathbb{R}^n$.

Definition 3.1 (Point)**.** Point is an infinitesimal element of the space $\mathbb{R}^n$ characterized by an n-tuple of real numbers $\mathbb{R} \times \mathbb{R} \times ... \times \mathbb{R}$ (n times), known as *Cartesian coordinates* of the point $p = (x_1, x_2, ..., x_n)$.

The inner product in $\mathbb{R}^n$ is a dot product that can be expressed as

$$\begin{aligned} \vec{x} = \begin{bmatrix} x_1 & x_2 & ... & x_n \end{bmatrix}, \vec{y} = \begin{bmatrix} y_1 & y_2 & ... & y_n \end{bmatrix} \\ (\vec{x}, \vec{y}) \mapsto \langle x, y \rangle = x_1 y_1 + x_2 y_2 + ... + x_n y_n \end{aligned} \tag{3.2}$$

where $\vec{x}$ an $\vec{y}$ are n-dimensional vectors, characterized by their n-tuples.

The Euclidean space $\mathbb{R}^n$ is an abstract construct, which can be displayed with help of the *Cartesian coordinate system*. In the Cartesian coordinate system, each dimension is displayed with an axis line having a specific name, such as *x axis*, *y axis*, *z axis*, and similar. Axes are perpendicular to each other, which means they intersect once and at 90° angle. The whole Cartesian coordinate system has one origin. Each axis has its units, which in our case follows the set of numbers that define the axis dimension.

Figure 3.1 shows two vectors $\vec{x} = [x_1, x_2]$, $\vec{y} = [y_1, y_2]$ in $\mathbb{R}^2$ space presented in the Cartesian coordinate system. Figure 3.2 shows two points in $\mathbb{R}^2$ space. For the simplicity sake, we use $\mathbb{R}^2$ throughout the chapter, having in mind that **all mathematical formalisms and algorithms can be extended to higher dimensional spaces**.

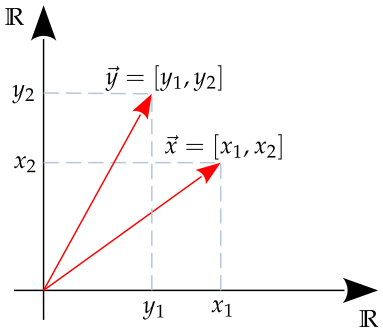


Figure 3.1: Vectors in $\mathbb{R}^2$ space.

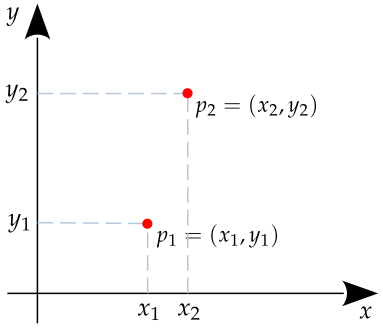


Figure 3.2: Points in $\mathbb{R}^2$ space.

Definition 3.2 (Line)**.** A line is a set of Euclidean space points whose coordinates comply the following linear equation

$$L = \{(x, y) : (x, y) \in \mathbb{R}^2, ax + by = c\} \tag{3.3}$$

resulting in an infinite set of points.

The line equation can be redefined to

$$y = \frac{c - ax}{b} \tag{3.4}$$

For $b = 1$ this is

$$y = c - ax \tag{3.5}$$

Such a line can be seen in Figure 3.3. Every line can be defined by two points, where

$$\begin{aligned} p_1 = (x_1, y_1), p_2 = (x_2, y_2) \\ y_1 = c - ax_1, y_2 = c - ax_2 \end{aligned} \tag{3.6}$$

where the line parameters can be calculated as

$$\begin{aligned} a = \frac{y_2 - y_1}{x_1 - x_2} \\ c = y_1 + \frac{y_2 - y_1}{x_1 - x_2} x_1 \end{aligned} \tag{3.7}$$

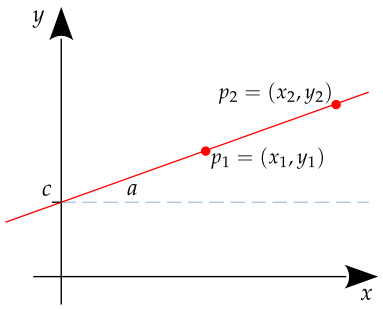


Figure 3.3: A line in the $\mathbb{R}^2$ space.

The final linear equation for a line that passes two exact points becomes

$$\begin{aligned} \Delta y = y_2 - y_1, \Delta x = x_2 - x_1 \\ y = y_1 - \frac{\Delta y}{\Delta x} x_1 + \frac{\Delta y}{\Delta x} x \end{aligned} \tag{3.8}$$

A line can also be written as dimensional parametric equations

$$\begin{aligned} y = y_1 + t\Delta y \\ x = x_1 + t\Delta x \end{aligned} \tag{3.9}$$

where for $t = 0$, the point is $(x, y) = p_1 = (x_1, y_1)$ and for $t = 1$, the point is $(x, y) = p_2 = (x_2, y_2)$, which brings us to the following definition

Definition 3.3 (Line segment)**.** A line segment is a part of a line bounded by two points $A = (x_a, y_a)$ and $B = (x_b, y_b)$. For a scaling parameter $t \in [0, 1]$, we can express the line segment $\overline{AB}$ as the following set of dimensional linear equations

$$x = x_a + t(\frac{x_b}{x_a}), y = y_a + t(\frac{y_b}{y_a}) \tag{3.10}$$

which can be expressed also as the following set of points

$$\overline{AB} = \{p = (x, y) : |p - A| + |p - B| = |B - A|\} \tag{3.11}$$

where $|x|$ is the Euclidean distance expressed as

$$|x| = \sqrt{x^2 + y^2} \tag{3.12}$$

Since Euclidean space is also a vector space, the line segment can be written as a scaled combination of vectors. If we write end points as vectors

$$\vec{A} = \begin{bmatrix} x_a & y_a \end{bmatrix}, \vec{B} = \begin{bmatrix} x_b & y_b \end{bmatrix} \tag{3.13}$$

and we take

$$\vec{B} = \vec{A} + \vec{C} \tag{3.14}$$

then

$$\overline{AB} = \{\vec{A} + t\vec{C} : t \in [0, 1]\} \tag{3.15}$$

is the set of vectors that point to the line segment between end points $\vec{A}$ and $\vec{B}$. We call a line segment **open** for $t \in (0, 1)$ and **closed** for $t \in [0, 1]$.

Combining line segments together, we can create other geometrical shapes and areas. We start by defining a space plane.

Definition 3.4 (Plane)**.** A plane is flat, infinite two-dimensional surface in a space. The plane is equivalent to a point (0-dimensions) on a line (1-dimension) in a space that has 3 or more dimensions. Planes can be usually described by the following linear equation

$$P = \{(x, y, z) : (x, y, z) \in \mathbb{R}^3, ax + by + cz = d\} \tag{3.16}$$

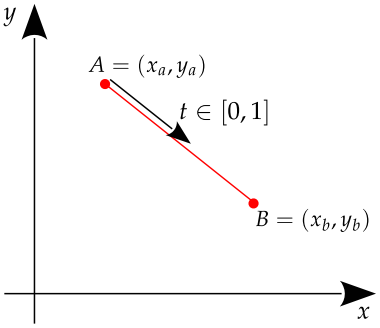


Figure 3.4: Line segment $\overline{AB}$ in the $\mathbb{R}^2$ space.

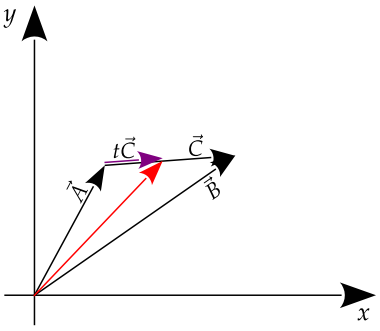


Figure 3.5: Line segment $\overline{AB}$ in the vectorial form.

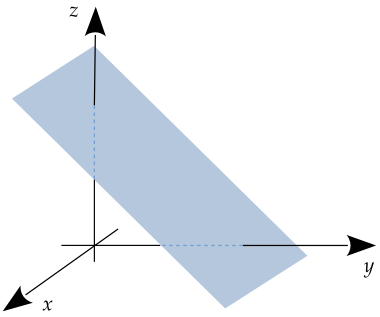


Figure 3.6: A plane in the $\mathbb{R}^3$ space.

We get shapes and areas by selecting a subset of a plane.

Definition 3.5 (Polygon)**.** A subset of a plane $\mathcal{P} \subset P$ defined by a set of points and their line segments is called a polygon. More generally, a polygon made of n points is called an n-gon. Polygon line segments are closed and are forming the *polygon chain* or *edge*. The interior of the polygon is called *polygon body*. Polygon points are usually enumerated clockwise. The sum of the polygon interior angles between incident line segments can be calculated as

$$S = (n-2) * 180^\circ \tag{3.17}$$

For example, for triangle the sum of interior angles is $(3-2) * 180^\circ = 180^\circ$. There are more distinct types of polygons:

- **Convex or non-convex** polygons. We call polygon $\mathcal{P}$ convex if all pairs of its points have line segments that are fully in the polygon, or

  $$\forall p_1, p_2 \in \mathcal{P} : \overline{p_1 p_2} \in \mathcal{P} \tag{3.18}$$

  Contrary, polygon $\mathcal{P}$ is non-convex if there is a pair of points in the polygon whose line segment is partially outside the polygon, or

  $$\exists p_1, p_2 \in \mathcal{P} \exists \overline{p_3 p_4} \subseteq \overline{p_1 p_2} : \overline{p_3 p_4} \notin \mathcal{P} \tag{3.19}$$

  Figure 3.7 shows examples of convex and non-convex polygons.

- **Simple** polygons, or those that don't self-interact. Self-interaction would be crossing its own edge, as in Figure 3.9.

- **Regular** polygons, which are *equiangular* and *equilateral*, meaning their all internal angles are the same, and their edge line segments are the same. Their convex variants have unique names: triangle, square, pentagon, hexagon, heptagon, octagon, and so on.

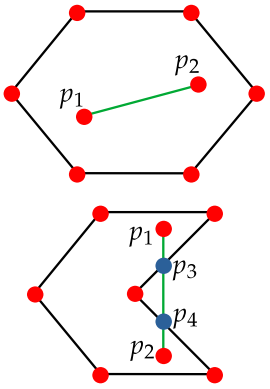


Figure 3.7: Polygon convexity.

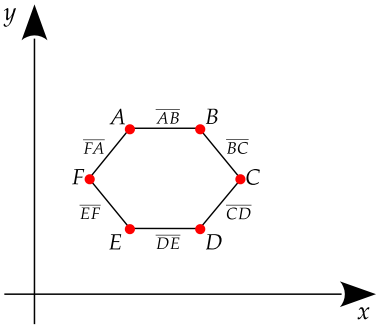


Figure 3.8: A simple convex polygon.

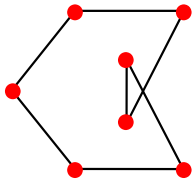


Figure 3.9: A complex non-convex polygon.

Line segment intersection

Two lines can be defined through the following system of linear equations

$$\begin{aligned} a_1 x + b_1 y &= c_1 \\ a_2 x + b_2 y &= c_2 \end{aligned} \tag{3.20}$$

which can be written in the matrix format as

$$\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} \tag{3.21}$$

How do we calculate the intersection of these two lines (x_s, y_s)? Using *Cramer's rule*, we get

$$\begin{aligned} x_s = \frac{\begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}, y_s = \frac{\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} \\ x_s = \frac{c_1 b_2 - b_1 c_2}{a_1 b_2 - b_1 a_2}, y_s = \frac{a_1 c_2 - c_1 a_2}{a_1 b_2 - b_1 a_2} \end{aligned} \tag{3.22}$$

which gives the intersection in all cases when $a_1b_2 - b_1a_2 \neq 0$. If $a_1b_2 - b_1a_2 = 0$ then two line are parallel and have no intersection.

When we limit a line to a segment, two line segments not necessarily intersect if their lines intersect, which can be see in Figure 3.10. Having four points $A = (x_a, y_a), B = (x_b, y_b), C = (x_c, y_c), D = (x_d, y_d)$, we can combine them into two line segments $\overline{AB}, \overline{CD}$. These two line segments can be written as the following dimensional linear equation system

$$\begin{aligned} x_s &= x_a + s(x_b - x_a), y_s = y_a + s(y_b - y_a) \\ x_t &= x_c + t(x_d - x_c), y_t = y_c + t(y_d - y_c) \end{aligned} \tag{3.23}$$

where s is the position of the intersection on the line segment $\overline{AB}$ and t is the position of the intersection on the line segment $\overline{CD}$. This can be rewritten as

$$\begin{aligned} s(x_b - x_a) - t(x_d - x_c) &= x_c - x_a \\ s(y_b - y_a) - t(y_d - y_c) &= y_c - y_a \\ \begin{bmatrix} x_b - x_a & -(x_d - x_c) \\ y_b - y_a & -(y_d - y_c) \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} &= \begin{bmatrix} x_c - x_a \\ y_c - y_a \end{bmatrix} \end{aligned} \tag{3.24}$$

Using Cramer's rule, we get the solution as

$$\begin{aligned} s &= \frac{(x_d - x_c)(y_c - y_a) - (x_c - x_a)(y_d - y_c)}{(x_d - x_c)(y_b - y_a) - (x_b - x_a)(y_d - y_c)} \\ t &= \frac{(x_b - x_a)(y_c - y_a) - (x_c - x_a)(y_b - y_a)}{(x_d - x_c)(y_b - y_a) - (x_b - x_a)(y_d - y_c)} \end{aligned} \tag{3.25}$$

We get the correct intersection only when $(x_b - x_a)(y_d - y_c) - (x_d - x_c)(y_b - y_a) \neq 0$. Also, two line segments $\overline{AB}, \overline{CD}$ intersect only if $s \in [0, 1]$ and $t \in [0, 1]$. If any of the intersection positions s, t gets outside the interval $[0, 1]$, as on the bottom side of Figure 3.10, we can conclude that line segments do not intersect, unlike their lines, which always have an intersection point if not parallel.

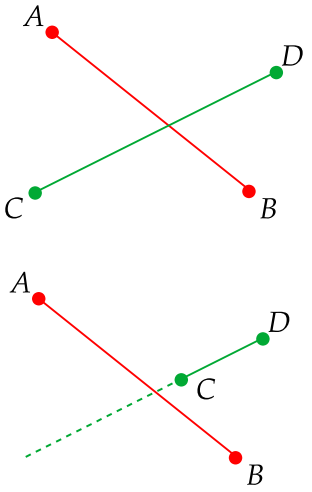


Figure 3.10: Line segments intersection.

Plane sweeping line segment intersection algorithm

When having a set of line segments, how do we find all intersections among them? For example, we can have multiple infrastructural maps: roads, railways, water distribution, sewage, and so on. How do we find their intersections? This is called *a map overlay* problem. Any naïve solution would require testing all pairs of n line segments for intersections, which is $O(n^2)$ complex. If we want to reduce this quadratic complexity, we need to find a better solution. As the input in the line segment intersection algorithm, we have only a set of line segments

$$\mathcal{S} = \{s_1, s_2, ..., s_n\} \tag{3.26}$$

where each segment is

$$s_i = \overline{p_1 p_2} \tag{3.27}$$

First, we need to avoid checking all pairs of line segments s_i, s_j that are distant from each other, and we are certain that there could be no overlapping. We can define such line segments as

$$\begin{aligned}[y(p_1(s_i)), y(p_2(s_i))] \setminus [y(p_1(s_j)), y(p_2(s_j))] = \emptyset \\ [x(p_1(s_i)), x(p_2(s_i))] \setminus [x(p_1(s_j)), x(p_2(s_j))] = \emptyset\end{aligned} \tag{3.28}$$

where there is no overlapping between line segments in one of their dimensions. Figure 3.11 shows that line segment s_1 does not overlap with any other line segments neither on x and y axis. According to (3.28), s_1 cannot intersect with any other line segments. However, line segments s_2, s_3, s_4 overlap both on x and y axes, which means there could be a potential intersection in $\Delta x_2, \Delta y_2$ plane segment.

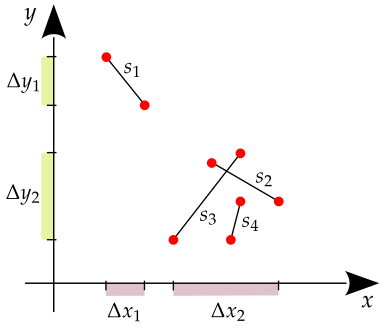


Figure 3.11: Overlapping line segments.

To test which line segments are belonging to the same segment on the y axis, we introduce an $x - y$ plane sweeping line l. All line segments from $\mathcal{S}$ that intersect the plane sweeping line l belong to the same y axis segment. Imagine we are moving the plane sweeping line l in one direction, for example as in Figure 3.12, the $x - y$ sweeping line is orthogonal on the y axis and moved (is sweeping) in the axis descending direction.

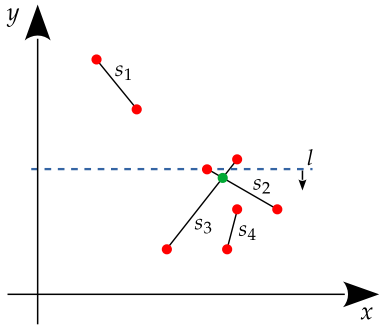


Figure 3.12: Plane sweeping line l intersecting line segments s_2 and s_3.

According to (3.28), only line segments intersecting the plane sweeping line l can intersect each other. As we sweep the plane, some line segments can start or stop intersecting the sweeping line. This happens on the line segment endpoints p_1 or p_2. In the moment when a new line segment starts intersecting the sweeping line, we need to be aware of this, as the new line segment can intersect other line segments currently intersecting the sweeping line. This intersection between line segments can occur on the sweeping line l or below. If we look at the example in Figure 3.13, we can identify the following events:

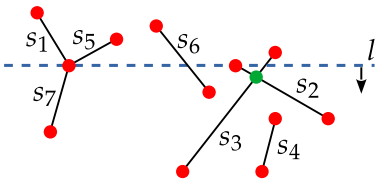


Figure 3.13: Plane sweeping line l intersecting multiple line segments.

1. **Line segment s_7 starts intersecting the sweeping line**. s_7 intersects with line segments s_1 and s_5 on their end points. The joint intersection for line segments s_1, s_5, and s_7 occurs on the sweeping line. No other line segments currently intersecting the sweeping line $\{s_6, s_3\}$ intersect with s_7.

2. **Line segment s_2 starts intersecting the sweeping line**. We can see that s_2 intersects with line segment s_3, which is also intersecting the sweeping line. The intersection between line segments s_2 and s_3 occurs below the sweeping line.

3. **Line segments s_1 and s_5 stop intersecting the sweeping line** and cannot intersect and new line segments as we proceed with plane sweeping.

Obviously line segment end points are important. We call them *events*, as they *notify* our algorithm which line segments are intersecting the sweeping line

$$\mathcal{S}_l \subseteq \mathcal{S} \tag{3.29}$$

When a line segment s_i starts intersecting the sweeping line, we add it as

$$\mathcal{S}_l^{k+1} = \mathcal{S}_l^k \cup \{s_i\} \tag{3.30}$$

where $\mathcal{S}_l^k$ is the set of line segments intersecting the sweeping line after k processed events. After some number of events m, when the same line segment s_i stops intersecting the sweeping line, we remove it as

$$\mathcal{S}_l^{k+m+1} = \mathcal{S}_l^{k+m} \setminus \{s_i\} \tag{3.31}$$

At some point, a big number of line segments can intersect the sweeping line, or $|\mathcal{S}_l^k| \approx |\mathcal{S}|$. When introducing a new line segment as in (3.30), do we need to check line segment intersections between s_i and the whole $\mathcal{S}_l^k$? This would bring us to the original $O(n^2)$ complexity.

Taking that $s_i \in \mathcal{S}_l^k$ and $y_{s_i} = l$, we denote the x_{s_i} as the value where line segment s_i intersects the sweeping line l as

$$x_{s_i} = s_i^{-1}(l) \tag{3.32}$$

One of the solutions is to horizontally sort line segments that intersect the sweeping line. We define an ordered *sequence*

$$\tau = sort(\mathcal{S}_l^k, l) = (s_i : s_i \in \mathcal{S}_l^k) \tag{3.33}$$

satisfying

$$\forall 1 \leq i, j \leq |\mathcal{S}_l^k| : i < j, s_i^{-1}(l) \leq s_j^{-1}(l) \tag{3.34}$$

Figure 3.14 shows two distinct positions of the sweeping line, denoted as l_1 and l_2, where l_2 is chronologically later than l_1, or $l_2 > l_1$. In l_1 we introduce the line segment s_3. After addition, we have the following horizontally sorted sequence of line segments

$$sort(\mathcal{S}_l^k, l_1) = (s_1, s_2, s_5, s_6, s_3, s_7) \tag{3.35}$$

Since we added new line segment s_3 to the sweeping line intersecting line segments, we need to check whether the new line segment intersects only adjacent line segments, as they are the first that could be intersected with. So, we need to test whether s_3 intersects with s_6 and s_7. In the case in Figure 3.14, we indeed have an intersection between s_3 and s_6 below the sweeping line at the position l_1. We denoted this intersection as i_2. Then we proceed with sweeping to the position l_2. The horizontal sorting is now slightly different

$$sort(\mathcal{S}_l^k, l_1) = (s_1, s_2, s_5, s_3, s_6, s_7) \tag{3.36}$$

We noticed that line segments s_3 and s_6 swapped their places in the horizontally sorted sequence of line segments. As a consequence, s_3 is now adjacent to line segments s_5 and s_6, having another intersection with s_5 below the sweeping line at the position l_2, denoted as i_3. It means that each swap of the line segments in the horizontally ordered sequence, due to their intersection, changed adjacency between line segments in $\mathcal{S}_l^k$. We need to take line segment intersections as events, as we need to notify the algorithm that horizontal adjacency has changed and that some checking needs to be done.

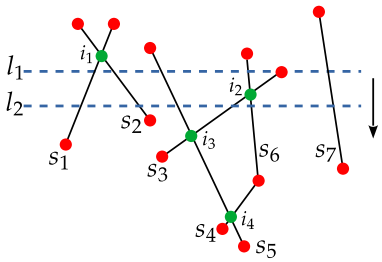


Figure 3.14: Horizontal view on the sweeping line.

Events
Events in this algorithm are points in the $x - y$ plane. These are:

- **Line segment end points**. They are known beforehand, and are at disposal when the algorithm starts.
- **Line segment intersections**. We discover them in the plane sweeping process, and they are the output that algorithm needs to produce. They are important, as they change the line segment adjacency.

To handle these events, we need to construct a sorted sequence Q that helps us organizing them.

$$Q = (e_i = (x, y)) \tag{3.37}$$

where e_i is an event point. The *event sequence* Q must satisfy the following order

$$\forall e_i, e_j \in Q : i < j, y(e_i) < y(e_j) \vee (y(e_i) = y(e_j) \wedge x(e_i) \leq x(e_j)) \tag{3.38}$$

This is also the plane sweeping order. The plane is swept downwards on the y axis. For all events that are on the same sweeping line, we sweep upwards on the x axis. The event sequence Q for the example in Figure 3.15 is the following

$$Q = (p_1(s_5), p_1(s_2), p_1(s_1), p_1(s_6), i_1, i_2, p2(s_1), p_1(s_3), i_3, p_1(s_4), i_4, p_2(s_2), p_2(s_3), p_2(s_6), i_5, p_2(s_4), p_2(s_5)) \tag{3.39}$$

Of course, intersections i_1 to i_5 are added as we discover them. Nonetheless, they need to be correctly sorted in the event sequence Q. Of course, in the algorithm, we do not sweep the plane continuously, as this would be extremely slow. Instead, we jump from event to event, performing further calculations, which can be described as line segment intersection detection right on the sweeping line or below the current event. Horizontal sorting of the event sequence Q for $x(e_i) = x(e_j)$ is introduced to support horizontal line segments, such as s_3 in Figure 3.15.

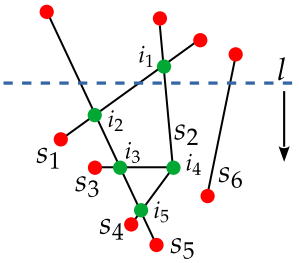


Figure 3.15: An example for the event sequence Q.

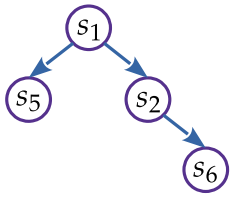


Figure 3.16: The binary tree τ for the sweeping line position l in Figure 3.15.

Implementing horizontally sorted sequence of line segments
In (3.33) we defined a horizontally sorted sequence of line segments τ, which can be use to know:

- Which line segments are intersecting the sweeping line after we processed k events from the event sequence Q.
- What are the adjacent line segments of a queried line segment s_q, through horizontal sorting of line segments.

The natural way to implement the horizontally sorted sequence of line segments τ is a binary tree. Based on the binary tree in Figure 3.16, we can tell that:

- The sweeping line in the position l is intersected by line segments $\mathcal{S}_l^5 = \{s_1, s_2, s_5, s_6\}$.
- That the horizontally sorted sequence of line segments is $sort(\mathcal{S}_l^5, l) = \{s_5, s_1, s_2, s_6\}$. If we try to find adjacent line segments for $s_q = s_2$, these are s_1 and s_6, and this is found with complexity $O(n \log n)$.

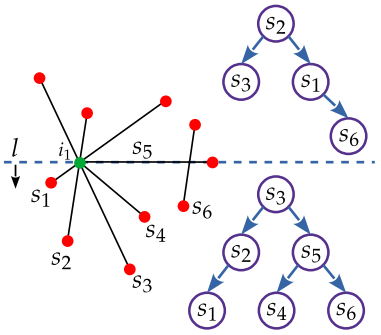


Figure 3.17: Binary trees before and after the event i_1.

Since we are not performing the continuous plane sweep, our algorithm will perform calculations only in event points, such as line segment intersections. Looking at Figure 3.17 we can see a sweeping line passing through a single intersection point, where line segments $\{s_1, s_2, s_3\}$ intersect in their interior, s_4 has the upper point, and s_5 has the left point as the horizontal line segment. Generating a binary tree τ in such sweeping line is not possible, as there is no horizontal ordering of the line segments. Instead, we generate a binary tree τ for the sweeping areas between the currently processing event and the next event in Q. Notice that according to (3.38), the next event can lie on the same sweeping line, right from the current event. For the situation as in Figure 3.17, we adhere to the following rules when creating a binary tree τ for the next event in the event sequence Q:

1. For all non-horizontal line segments intersecting in the event $e = i_1$, their order in the resulting binary tree τ is taken just below the sweeping line l.
2. For all horizontal line segments, having their left point in the event $e = i_1$, they get ordered as the utmost right line segments from all exiting the event e.

After the event $e = i_1$ we have the following horizontally ordered sequence of line segments

$$sort(\mathcal{S}_l^k, l) = (s_1, s_2, s_3, s_4, s_5, s_6) \tag{3.40}$$

which is reflected in the binary tree τ in Figure 3.17. The binary tree interface must support the following operations:

- $insert(e, \mathcal{S}(e) \subseteq \mathcal{S})$: Inserts a subset of line segments that intersect the event e, using ordering in the previously defined rules, which must be part of the binary tree implementation. This operation returns an ordered pair (s_l, s_r), where $s_l \in \mathcal{S}(e)$ is the utmost left newly inserted line segment and $s_r \in \mathcal{S}(e)$ is the utmost right newly inserted line segment.
- $remove(e, \mathcal{S}(e) \subseteq \mathcal{S})$: Removes a subset of line segments from the binary tree.
- $query(e \in Q)$, $query(p \in \mathbb{R}^2)$: Queries an event or an point supplied in the parameter and returns a tuple (s_q, s_l, s_r), where $s_q \subseteq \mathcal{S}$ represents a set of line segments that contain the event e or the point p, s_l is an adjacent line segment immediately left from the event e or the point p, and s_r is an adjacent line segment immediately right from the event e or the point p.

Programming tips: A correct implementation of the binary tree that supports the horizontal line segment sorting is the most demanding part implementing this algorithm. First and foremost, **the implemented binary tree must be balanced**, as this is the only way to achieve $O(\log n)$ complexity for each *query*, *insert*, and *delete*.

The *query* method: We usually query the τ tree on event points. This can happen for example in i_1 in Figure 3.18. Having the planar sweep line l intersecting the event i_1, we have a number of nodes in the τ tree containing line segments that are intersecting the event i_1. The query method starts from the root node s_1, finding that the event i_1 intersects the containing line segment, which is taken as $i_1 = s_1$, and s_1 is added to the set $s_q = \{s_1\}$. Because of the equality, we need to descend into both child nodes. We find that $i_1 > s_3$, which makes $s_l = s_3$ and we descend to the right child node. Again, $i_1 = s_2$ because s_2 intersects the event i_1, and we add s_2 to the set $s_q = \{s_1, s_2\}$. On the right side of the τ tree, we find $i_1 < s_5$, which makes $s_r = s_5$. Then we descend to the left child node, where we find $i_1 < s_4$, which makes $s_r = s_4$. The final result of the query method is $(s_q = \{s_1, s_2\}, s_l = s_3, s_r = s_4)$. Supporting equality gives a bit higher complexity, which is $O(\log n) \leq t(\tau) < O(n)$.

If we query a point p that is not an event and by that not present in the event sequence Q, this can be a point that does not intersect any line segments currently in the τ tree. In Figure 3.19 we can see such an example. When the plane sweeping line is in the position l_1, the query returns $query(p_1) = (s_q = \{\}, s_l = s_3, s_r = s_2)$. Moving forward to the position l_2, the query returns $query(p_2) = (s_q = \{\}, s_l = nil, s_r = s_3)$.

The *remove* method: Once we detected which line segments must switch their positions in the τ tree, we need to remove them. In the case in Figure 3.18 these are the line segments $s_q = \{s_1, s_2\}$. Removing them from the τ tree using *delete by copy* principle for example, can make the tree unbalanced. This requires constant rebalancing. The best option is to implement a locally balanced tree, such as *AVL tree*. The final result of the removal can be seen in Figure 3.20.

The *insert* method: After deleting line segments that intersect in the event i_1 from the τ tree, we need to reinsert them again, but in the reverse order. We also add all line segments that have the upper point in the event i_1, as seen in Figure 3.21. The main problem when inserting the reversed line segments is knowing their horizontal order, as they all intersect the event i_1. For that reason, we look for the order of the line segments between the current event i_1 and the next event, which is in our case i_2. All horizontal line segments, in our case s_6, come the most right in the ordering, as we want to see whether there are line segments right of the event i_1 that intersect the horizontal line. We can see the inserting result in Figure 3.21 on

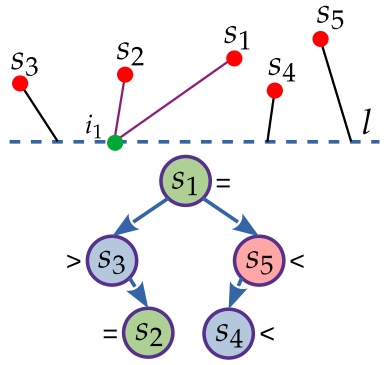


Figure 3.18: A planar line intersecting the event i_1, and related τ tree. *Query* method results marked on the τ tree nodes.

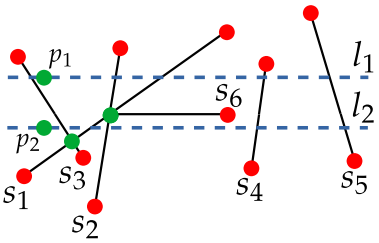


Figure 3.19: A change of the adjacent line segments between points p_1 and p_2, where $x(p_1) = x(p_2)$.

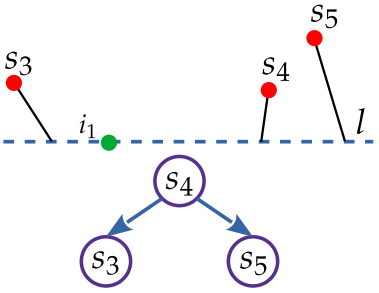


Figure 3.20: A removal of the line segments $s_q = \{s_1, s_2\}$ and re-balancing the τ tree.

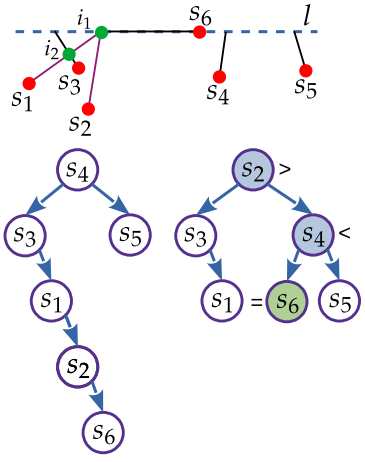


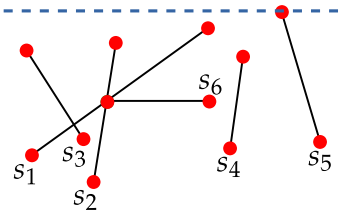
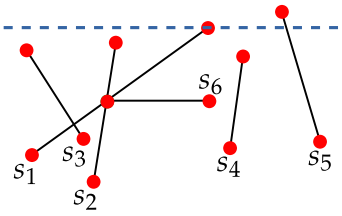
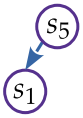
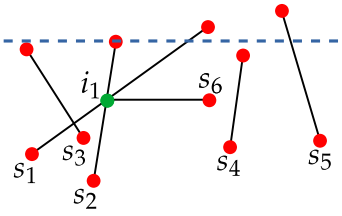
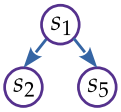
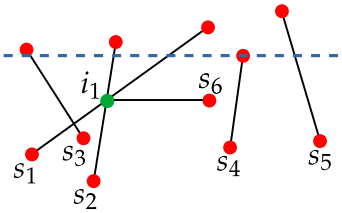
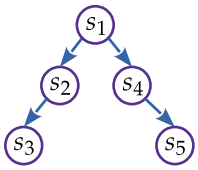
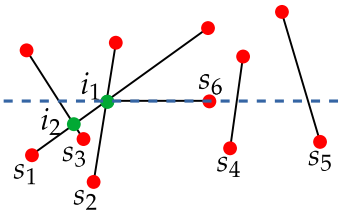
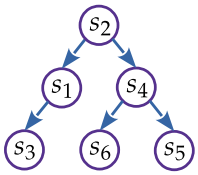
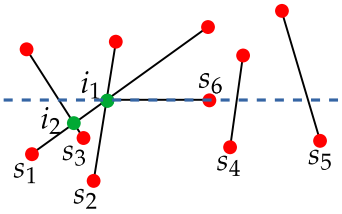
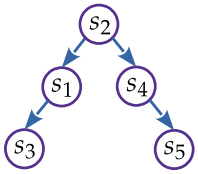
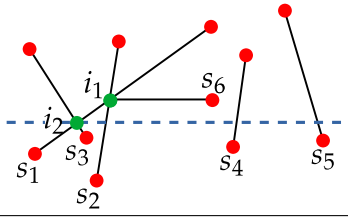
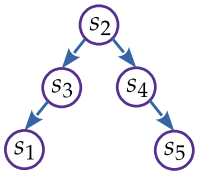
Figure 3.21: Inserting the line segments s_1, s_2, and s_6 into the τ tree and re-balancing it.

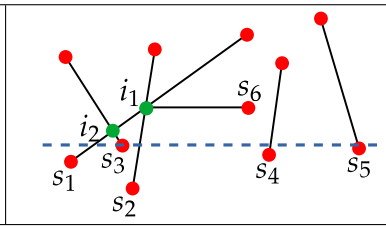
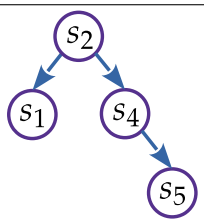
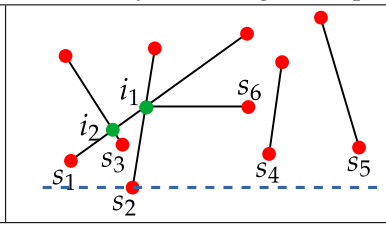
the left side, which also shows how unbalanced the τ tree became. The re-balanced variant of the τ tree can be seen on the right side of the same Figure.

Again, querying the final balanced τ tree in Figure 3.21 on the lower s_6 point returns $query(p_2(s_6)) = (s_q = \{s_6\}, s_l = s_2, s_r = s_4)$.

Algorithm 3.1: Plane sweep line segment intersection algorithm.

function PLANESWEEPLSINTERSECTION($\mathcal{S}$)
  $Q \leftarrow$ new priority queue ▷ Event sequence (3.38)
  for $s_i \in \mathcal{S}$ **do**
    $Q.insert(p_1(s_i))$
    $Q.insert(p_2(s_i))$
  $\mathcal{I} \leftarrow \emptyset$
  $\tau \leftarrow$ new empty binary tree
  while Q is not empty **do**
    $e \leftarrow Q.pop()$
    PROCESSEVENT($e, Q, \tau, \mathcal{I}, \mathcal{S}$)
  return $\mathcal{I}$
procedure PROCESSEVENT($e, Q, \tau, \mathcal{I}, \mathcal{S}$)
  $U(e) \leftarrow$ all line segments in $\mathcal{S}$ having e as upper point or left point for horizontal line segments
  $C(e) \leftarrow$ all line segments in τ having e in the interior
  $L(e) \leftarrow$ all line segments in τ having e as lower point or right point for horizontal line segments
  if $|U(e) \cup C(e) \cup L(e)| > 1$ and e is not in $\mathcal{I}$ **then**
    Add e to $\mathcal{I}$ along with all line segments intersecting it: $U(e) \cup C(e) \cup L(e)$
  $\tau.remove(e, L(e) \cup C(e))$
  $(s_{i1}, s_{i2}) \leftarrow \tau.insert(e, U(e) \cup C(e))$
    ▷ τ is prepared for the next event
  if $U(e) \cup C(e) = \emptyset$ **then**
    $(s_l, s_r) \leftarrow \tau.query(e)$
    if s_l and s_r intersect at point i **then**
      $Q.insert(i)$
  else
    $(s_l, s_r) \leftarrow \tau.query(s_{i1})$
    if s_l and s_{i1} intersect at point i **then**
      $Q.insert(i)$
    $(s_l, s_r) \leftarrow \tau.query(s_{i2})$
    if s_{i2} and s_r intersect at point i **then**
      $Q.insert(i)$

Step	The plane and the sweep line	The resulting τ tree
$Q = \{p_1(s_5), p_1(s_1), p_1(s_2), p_1(s_3), p_1(s_4), p_1(s_6), p_2(s_6), p_2(s_3), p_2(s_5), p_2(s_4), p_2(s_1), p_2(s_2)\}$		
1: $e = p_1(s_5)$ $U(e) = \{s_5\}$ $L(e) = \emptyset$ $C(e) = \emptyset$ $s_{i1} = s_5, s_{i2} = s_5$		
2: $e = p_1(s_1)$ $U(e) = \{s_1\}$ $L(e) = \emptyset$ $C(e) = \emptyset$ $s_{i1} = s_1, s_{i2} = s_1, s_l = nil, s_r = s_5$		
3: $e = p_1(s_2)$ $U(e) = \{s_2\}$ $L(e) = \emptyset$ $C(e) = \emptyset$ $s_{i1} = s_2, s_{i2} = s_2, s_l = nil, s_r = s_1$ intersection i_1 between s_2 and s_1, which is not added to Q since $i_1 = p_1(s_6)$		
... continues by adding upper points ...		
5: $e = p_1(s_4)$ $U(e) = \{s_4\}$ $L(e) = \emptyset$ $C(e) = \emptyset$ $s_{i1} = s_4, s_{i2} = s_4, s_l = s_1, s_r = s_5$		
6: $e = i_1 = p_1(s_6)$ $U(e) = \{s_6\}$ $L(e) = \emptyset$ $C(e) = \{s_1, s_2\}$ $s_{i1} = s_1, s_{i2} = s_6, s_l = s_3, s_r = s_4$ intersection i_2 between s_3 and s_1		
$Q = \{p_2(s_6), i_2, p_2(s_3), p_2(s_5), p_2(s_4), p_2(s_1), p_2(s_2)\}$		
7: $e = p_2(s_6)$ $U(e) = \emptyset$ $L(e) = \{s_6\}$ $C(e) = \emptyset$ $s_l = s_2, s_r = s_4$		
8: $e = i_2$ $U(e) = \emptyset$ $L(e) = \emptyset$ $C(e) = \{s_1, s_3\}$ $s_{i1} = s_1, s_{i2} = s_3, s_l = nil, s_r = s_2$		

9: $e = p_2(s_3)$ $U(e) = \emptyset$ $L(e) = \{s_3\}$ $C(e) = \emptyset$ $s_l = s_1, s_r = s_2$		
... continues by removing lower points ...		
13: $e = p_2(s_2)$ $U(e) = \emptyset$ $L(e) = \{s_2\}$ $C(e) = \emptyset$ $s_l = nil, s_r = nil$		empty tree

Complexity assessment

Originally we estimated that each-with-each strategy would be $O(n^2)$ complex. Anything less complex than that is an improvement. Using binary trees lowers our complexity for querying, inserting, and removing events down to $O(\log n)$ per each operation. If we look at the ProcessEvent procedure, we can see that $m(e) = |U(e)| + |L(e)| + |C(e)|$ is the most line segments involved in the event we currently process. We query $|L(e)| + |C(e)|$ events, insert $|U(e)| + |C(e)|$ events, and remove $|L(e)| + |C(e)|$ events. The worst case scenario is to have all events in the $C(e)$ set, which would mean that all line segments in the binary tree intersect the current event (intersection). The total sum for all events is taken as $m = \sum_{e \in Q} m(e)$, which gives the complexity $O(m \log n)$. It is obvious that $m = O(n + I)$, where n is the total number of line segments, and I is the total number of reported intersections, which finally ends as $O(n \log n + I \log n)$.

Listing 3.1: Plane sweep line segment intersection algorithm python code.

```
def PlaneSweepLSIntersection(S):
  Q,SS=PriorityQueue(),set()
  for ls in S:
    Q.put(Event(ls.p1))
    SS.add(ls.p1)
    Q.put(Event(ls.p2))
    SS.add(ls.p2)
  I=[]
  tau=BinaryTree()
  while not Q.empty():
    e=Q.get()
    ProcessEvent(e,Q,tau,I,S,SS)
  return I
```

```
def ProcessEvent(e,Q,tau,I,S,SS):
  Ue=[]
  for ls in S:
    if ls.isUpperPoint(e):
      Ue.append(ls)
  tauQuery=tau.query(e)
  Le,Ce=tauQuery['Le'],tauQuery['Ce']
  sl,sr=tauQuery['Sl'],tauQuery['Sr']
  if len(Ue)+len(Le)+len(Ce)>1 and (e.x,e.y) not in I:
    I.append({(e.x,e.y):Le+Ue+Ce})
  tau.remove(e,Le+Ce)
  (s1,s2)=tau.insert(e,Ue+Ce)
  if len(Ue+Ce)==0:
    if sl is not None and sr is not None:
      i=sl.intersection(sr)
      if i is not None and i not in SS:
        Q.put(Event(i))
        SS.add(i)
  else:
    if sl is not None:
      i=s1.intersection(sl)
      if i is not None and i not in SS:
        Q.put(Event(i))
        SS.add(i)
    if sr is not None:
      i=s2.intersection(sr)
      if i is not None and i not in SS:
        Q.put(Event(i))
        SS.add(i)
```

Exercise 3.1.1. Implement the remainder of the classes needed for the PLANESWEEPLSINTERSECTION algorithm in the python programming language. Classes must have the following declaration:

class Point:	A point class.
x=0	The point x value.
y=0	The point y value.
def __init__(self,x=0,y=0):	A default constructor.
class Event(Point):	An inherited class that represents an event.
def __init__(self,p=None,x=0,y=0):	A constructor taking `p` as `Point` or `(x,y)` as positions.
class LineSegment:	A line segment class.
p1,p2=Point(),Point()	Line segment end points.
def __init__(self,p1,p2):	A default constructor taking end points.
def intersection(self,other):	A method that returns `Point` of the intersection, or `None` if there is no intersection.
def isUpperPoint(self,e):	A method that takes `Event` parameter and returns `True/False` depending on whether the event `e` is the upper point of this line segment.
def isLowerPoint(self,e):	A method that takes `Event` parameter and returns `True/False` depending on whether the event `e` is the lower point of this line segment.
def isHorizontal(self):	A method that returns `True/False` depending on whether this line segment is horizontal.
def isInteriorPoint(self,e):	A method that takes `Event` parameter and returns `True/False` depending on whether the event `e` is in the interior of this line segment.
class BinaryTree:	A τ binary tree class
rootNode=None	The tree *root* node
def insert(self,e,Se):	A method for inserting line segments into the τ tree, where `e` is the current `Event` and `Se` is the set of line segments that needs to be inserted.
def query(self,e):	A method for querying line segments from the τ tree, where `e` is the current `Event`. The method must return a dictionary with elements: `Le`-the set of line segments whose lower end points intersecting the event `e`, `Ce`-the set of line segments that have the event `e` in their interior, `Sl`-the adjacent left line segment that does not intersect the event `e` or `None` is there is none, `Sr`-the adjacent right line segment that does not intersect the event `e` or `None` is there is none.
def remove(self,e,Se):	A method for removing line segments supplied in the parameter `Se`.

Test your implementation using the following testing code:

```
ls1=LineSegment(p1=Point(2,2),p2=Point(10,10))
ls2=LineSegment(p1=Point(2,10),p2=Point(10,2))
ls3=LineSegment(p1=Point(6,8),p2=Point(6,4))
ls4=LineSegment(p1=Point(9,7),p2=Point(10,4))
ls5=LineSegment(p1=Point(6,6),p2=Point(20,6))
ls6=LineSegment(p1=Point(8,6.5),p2=Point(13,6.5))
ls7=LineSegment(p1=Point(6,1),p2=Point(10,4))
ls8=LineSegment(p1=Point(7,5),p2=Point(11,5))
ls9=LineSegment(p1=Point(0,3),p2=Point(15,3))
I=PlaneSweepLSIntersection([ls8,ls1,ls6,ls2,ls3,ls4,ls9,ls7,ls5])
```

3.2 *Planar convex hulls*

We start by defining a set of points $P_p = \{p_1, p_2, ..., p_n\}$ in a plane P. There is a subset of points

$$P_b \subseteq P_p \tag{3.41}$$

such that

$$bound(\mathcal{P}) = \{\overline{p_i p_j} : p_i, p_j \in P_b\} \tag{3.42}$$

is the bound of the polygon $\mathcal{P}$. We add the restriction that no other point from the set P_p can be outside the polygon $\mathcal{P}$

$$\nexists p \in P_p : p \notin \mathcal{P} \tag{3.43}$$

If the polygon $\mathcal{P}$ is convex, according to (3.18), we call it a *convex hull*. In Figure 3.22 we can see that a polygon containing all the points in P_p is not necessarily convex. A convex hull can be seen on the bottom of the same Figure.

Imagine a wooden board with nails sticking out of it. What will you get when you wrap an elastic rubber band around the nails? A planar convex hull. We can have high-dimensional convex hulls, such as 3-dimensional convex hull, which can be identified with gift wrapping. For the simplicity sake, here we describe only planar convex hulls and related algorithms, which can be used in the linear programming.

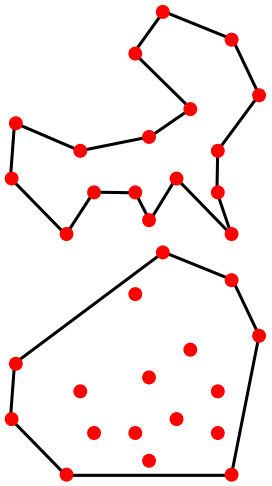


Figure 3.22: A non-convex polygon containing all his points and a convex hull.

Definition 3.6 (Planar convex hull)**.** Given a plane P and a set of points $P_p = \{p_1, p_2, ..., p_n\}$ in it, a polygon $\mathcal{P}$ that includes all points from P_p, has a bound made of these same points $bound(P_p)$, and is convex, is called a *planar convex hull*, denoted as $\alpha(P_p)$, and defined as a sequence of **clockwise ordered** points that from the bound, or

$$\alpha(P_p) = \{p_{b_i} : p_{b_i} \in bound(P_p)\} \tag{3.44}$$

For Figure 3.23, we have

$$P_p = \{p_1, p_2, ..., p_{17}\}$$
$$\alpha(P_p) = \{p_1, p_{11}, p_9, p_{14}, p_7, p_{10}, p_8\}$$

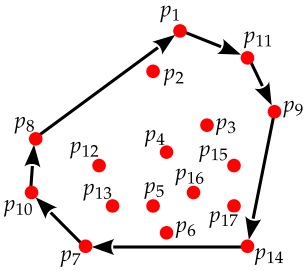


Figure 3.23: A clockwise ordered bound of a convex hull.

If we observe each line segment comprising the bound of a convex hull, such as in Figure 3.24, we can see that all points $p \in P_p$ are placed on the convex hull bound line segment or right from it. There are no points placed left from the convex hull bound line segments. We can use this property of the convex hull, to compute its bound. However, how to we detect that a point $p \in P_p$ is placed left or right from a line segment $\overline{p_{b_1} p_{b_2}}$?

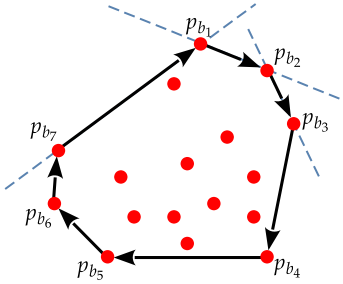


Figure 3.24: All points are positioned right from a convex hull bound line segments.

Given two points $A = (x_a, y_a), B = (x_b, y_b)$ and its line segment $\overline{AB}$, we can define a vector

$$\vec{AB} = \begin{bmatrix} x_b - x_a & y_b - y_a \end{bmatrix} \tag{3.45}$$

We have an orthogonal vector

$$\begin{aligned} \vec{AB} \cdot \vec{n}^\top = 0, \begin{bmatrix} x_b - x_a & y_b - y_a \end{bmatrix} \cdot \begin{bmatrix} y_b - y_a \\ x_a - x_b \end{bmatrix} = 0 \\ (x_b - x_a)(y_b - y_a) + (y_b - y_a)(x_a - x_b) = 0 \\ x_b y_b - x_b y_a - x_a y_b + x_a y_a + x_a y_b - x_b y_b - x_a y_a + x_b y_a = 0 \end{aligned} \tag{3.46}$$

which holds. Therefore

$$\vec{n} = \begin{bmatrix} y_b - y_a \\ x_a - x_b \end{bmatrix} \tag{3.47}$$

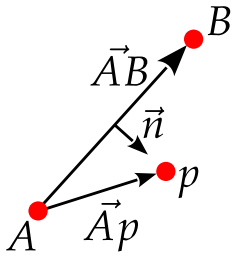


Figure 3.25: Determining the side of a point related to a line segment.

The orthogonal distance of the point p from the line segment $\overline{AB}$ is then calculated as

$$\begin{aligned} D &= \vec{n} \cdot \vec{Ap}^{\top} \\ D &= \begin{bmatrix} y_b - y_a & x_a - x_b \end{bmatrix} \cdot \begin{bmatrix} x - x_a \\ y - y_a \end{bmatrix} \\ D &= (x - x_a)(y_b - y_a) + (y - y_a)(x_a - x_b) \\ &= (x - x_a)(y_b - y_a) - (y - y_a)(x_b - x_a) \end{aligned} \tag{3.48}$$

For $D > 0$, the point p is right from the line segment $\overline{AB}$, for $D = 0$, the point is on the line segment, and for $D < 0$, the point is left from the line segment.

function SIMPLECONVEXHULL(P_p)
  $\alpha(P_p) \leftarrow \emptyset$
  for each pair of points $p_i, p_j \in P_p$ **do**
    $bound \leftarrow 1$
    for each point $p_k \in P_p$ such that $p_k \notin \{p_i, p_j\}$ **do**
      if p_k is left from the line segment $\overline{p_i p_j}$ **then**
        $bound \leftarrow 0$
    if *bound* is 1 **then**
      add $\overline{p_i p_j}$ to $\alpha(P_p)$ taking care of the clockwise order
  return $\alpha(P_p)$

Algorithm 3.2: A simple convex hull finding algorithm.

Complexity is $O(n^3)$

In a naïve attempt to find the convex hull for the input set of points P_p, we use the fact that all points from P_p must be on or right from line segments forming the convex hull bound. In the previous algorithm, we take pairs of points and check this property. If only one point from P_p is left from the line from the chosen pair of points p_i, p_j, this line segment is not on the bound of the convex hull.

Although simple, this approach has one major disadvantage. Iterating through all pairs of points from P_p is $O(n^2)$ complex. Then, for each pair, we also check the orthogonal distance from their line segment to all other points from P_p (checking whether they are on the right side), which brings us up to $O(n^2 * n) = O(n^3)$. Any algorithm having a cubical complexity seeks improvement.

```
def SimpleConvexHull(Pp):
  ch=[]
  for pi in Pp:
    for pj in
       [i for i in Pp
          if i is not pi]:
    v=True
    ls=LineSegment(pi,pj)
    for pk in
       [i for i in Pp
          if i not in [pi,pj]]:
      if ls.position(pk) is
         RelativePosition.LeftOf:
        v=False
        break
    if v: ch.append(ls)
  return ch
```

Listing 3.2: Simple convex hull python code. Extended from exercise 3.1.1.

Additional declarations:

```
class LineSegment:
  def position(self,point): - A method
  that takes point and returns whether
  the point is left, right or on the line
  segment.
class RelativePosition(Enum):
  On=0
  LeftOf=1
  RightOf=2
```

There are ways to improve the previous simple algorithm. We divide this process into finding two major parts of the convex hull boundary: the upper $\alpha_{upper}(P_p)$ and the lower hull $\alpha_{lower}(P_p)$. These two open line segment sequences combined together give the convex hull boundary $\alpha(P_p) = \alpha_{upper}(P_p) \cup \alpha_{lower}(P_p)$. Again, both line segment sequences, representing the upper and the lower hull, must respect the clockwise orientation. To work with upper and lower hulls, we need to horizontally sort points in P_p. For this purpose we define the following sequence

$$h(P_p) = \{p_{h_i} : p_{h_i} \in P_p\} \tag{3.49}$$

sorted as

$$\forall p_{h_i}, p_{h_j} \in h(P_p) : i < j, x(p_{h_i}) \leq x(p_{h_j}) \tag{3.50}$$

The most left point $p_{h_1} \in h(P_p)$ and the most right point $p_{h_n} \in h(P_p)$ are starting and ending points of the upper and lower hulls, where the upper hull begins at the most left point and ends at the most right point $\alpha_{upper}(P_p) = \{p_{h_1}, ..., p_{h_n}\}$, and the lower hull is in the opposite direction $\alpha_{upper}(P_p) = \{p_{h_n}, ..., p_{h_1}\}$. Both hulls meet at points $p_{h_1} \in h(P_p)$ and $p_{h_n} \in h(P_p)$. When looking at Figure 3.26, we can identify the following sequence, upper, and lower hulls

$$\begin{aligned} P_p &= \{p_1, p_2, ..., p_{15}\} \\ h(P_p) &= \{p_{10}, p_8, p_{12}, p_7, p_{13}, p_4, p_6, p_2, p_1, p_5, p_3, p_{15}, p_{11}, p_{14}, p_9\} \\ \alpha_{upper}(P_p) &= \{p_{10}, p_8, p_1, p_{11}, p_9\} \\ \alpha_{lower}(P_p) &= \{p_9, p_{14}, p_7, p_{10}\} \\ \alpha(P_p) &= \{p_{10}, p_8, p_1, p_{11}, p_9, p_{14}, p_7\} \end{aligned}$$

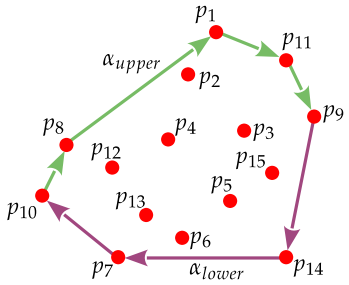


Figure 3.26: Upper and lower hulls.

To have a correct convex hull, both upper and lower hulls must be ***right inclining***. Figure 3.27 shows the definition of the left and right inclining sets of line segments. To determine the type of inclination, we need to have at least three distinct points. If we look at the Figure 3.26 upper and lower hulls, they are both right inclining.

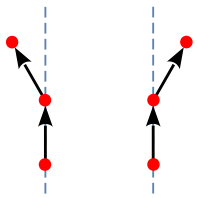


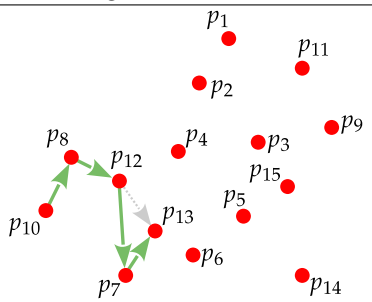
Figure 3.27: Left and right inclining sets of line segments..

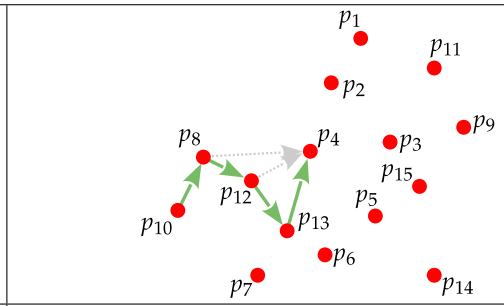
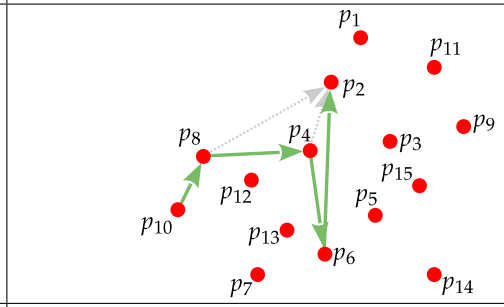
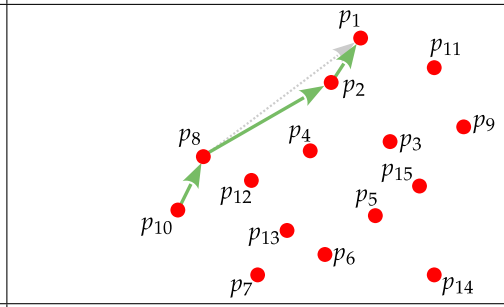
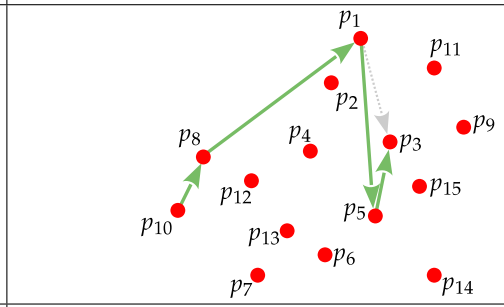
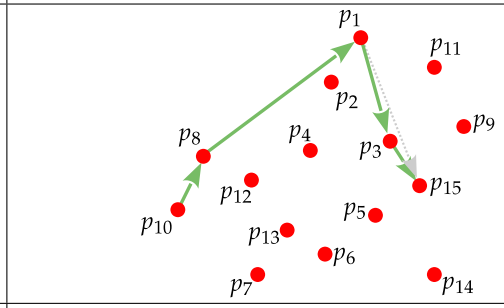
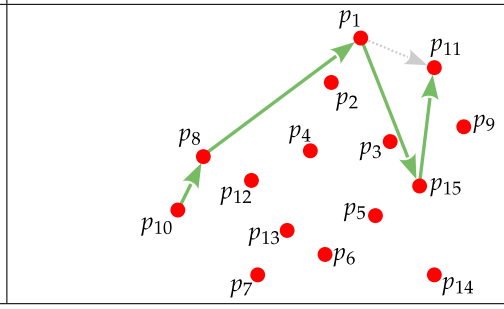
Algorithm 3.3: A convex hull finding algorithm.

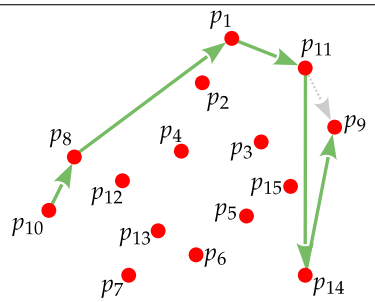
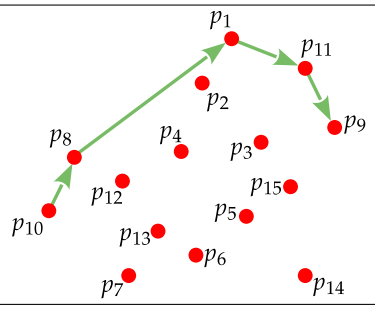
Complexity is $O(n \log n)$

function ConvexHull(P_p)
  create $h(P_p)$ as the sorted list of points from P_p
  $\alpha_{upper}(P_p) \leftarrow \{p_{h_1}, p_{h_2}\}$ from $h(P_p)$
  for $i \leftarrow 3$ **to** n **do**
    add p_{h_i} to $\alpha_{upper}(P_p)$
    while $|\alpha_{upper}(P_p)| \geq 3$ and last three points incline left **do**
      remove the point before the last from $\alpha_{upper}(P_p)$
  $\alpha_{lower}(P_p) \leftarrow \{p_{h_n}, p_{h_{n-1}}\}$ from $h(P_p)$
  for $i \leftarrow n-2$ **downto** 1 **do**
    add p_{h_i} to $\alpha_{lower}(P_p)$
    while $|\alpha_{lower}(P_p)| \geq 3$ and last three points incline left **do**
      remove the point before the last from $\alpha_{lower}(P_p)$
  return $\alpha_{upper}(P_p) \cup \alpha_{lower}(P_p)$

Based on the example in Figure 3.26, we get the following course of the ConvexHull(P_p) execution.

Step	Upper hull line segments
1: We start by progressing forward through $h(P_p)$. We reach $\alpha_{upper}(P_p) = \{p_{10}, p_8, p_{12}, p_7, p_{13}\}$, where the last three points p_{12}, p_7, p_{13} make the left inclination. By removing p_7 from the upper hull, we return back to the right inclining last three points p_8, p_{12}, p_{13}.	

2: We add p_4 to the upper hull $\alpha_{upper}(P_p) = \{p_{10}, p_8, p_{12}, p_7, p_{13}, p_4\}$. The last three points p_{12}, p_{13}, p_4 make the left inclination. By removing p_{13} from the upper hull, the last three points become p_8, p_{12}, p_4, which is also left inclining. By removing p_{12} from the upper hull we return back to the right inclining last three points.	
3: We add p_6 and p_2 to the upper hull $\alpha_{upper}(P_p) = \{p_{10}, p_8, p_4, p_6, p_2\}$. The last three points p_4, p_6, p_2 make the left inclination. By removing p_6 from the upper hull, the last three points become p_8, p_4, p_2, which is also left inclining. By removing p_4 from the upper hull we return back to the right inclining last three points.	
4: We add p_1 to the upper hull $\alpha_{upper}(P_p) = \{p_{10}, p_8, p_2, p_1\}$. The last three points p_8, p_2, p_1 make the left inclination. By removing p_2 from the upper hull, the last three points become p_{10}, p_8, p_1, we return back to the right inclining last three points.	
5: We add p_5 and p_3 to the upper hull $\alpha_{upper}(P_p) = \{p_{10}, p_8, p_1, p_5, p_3\}$. The last three points p_1, p_5, p_3 make the left inclination. By removing p_5 from the upper hull, the last three points become p_8, p_1, p_3, which is right inclining.	
6: We add p_{15} to the upper hull $\alpha_{upper}(P_p) = \{p_{10}, p_8, p_1, p_3, p_{15}\}$. The last three points p_1, p_3, p_{15} make the left inclination. By removing p_3 from the upper hull, the last three points become p_8, p_1, p_{15}, which is right inclining.	
7: We add p_{11} to the upper hull $\alpha_{upper}(P_p) = \{p_{10}, p_8, p_1, p_{15}, p_{11}\}$. The last three points p_1, p_{15}, p_{11} make the left inclination. By removing p_{15} from the upper hull, the last three points become p_8, p_1, p_{11}, which is right inclining.	

8: We add p_{14} and p_9 to the upper hull $\alpha_{upper}(P_p) = \{p_{10}, p_8, p_1, p_{11}, p_{14}, p_9\}$. The last three points p_{11}, p_{14}, p_9 make the left inclination. By removing p_{14} from the upper hull, the last three points become p_1, p_{11}, p_9, which returns back to the right inclining last three points.	
The final upper hull is $\alpha_{upper}(P_p) = \{p_{10}, p_8, p_1, p_{11}, p_9\}$	

The same procedure is then done for the lower hull, where we get $\alpha_{lower}(P_p) = \{p_9, p_{14}, p_7, p_{10}\}$ for the final result. When combined together, the result $\alpha(P_p) = \{p_{10}, p_8, p_1, p_{11}, p_9, p_{14}, p_7\}$ is the same as in Figure 3.26.

When it comes to the complexity assessment, Algorithm 3.3 is obviously less complex than Algorithm 3.2. We have one iteration passing through all points in P_p both for upper and lower hulls, which is $O(2n)$, or simply $O(n)$. Due to the sorting step in the algorithm, the complexity rises to $O(n \log n)$.

Listing 3.3: Convex hull python code.

```
def ConvexHull(Pp):
  hPp=sorted(Pp,key=lambda x:x.x)
  lu=[hPp[0],hPp[1]]
  for i in range(2,len(hPp)):
    lu.append(hPp[i])
    while len(lu)>2 and LineSegment(lu[-3],lu[-2]).position(lu[-1]) is RelativePosition.LeftOf:
      lu.pop(-2)
  ll=[hPp[-1],hPp[-2]]
  for i in range(3,len(hPp)+1):
    ll.append(hPp[-i])
    while len(ll)>2 and LineSegment(ll[-3],ll[-2]).position(ll[-1]) is RelativePosition.LeftOf:
      ll.pop(-2)
  ll.pop(0)
  ll.pop(-1)
  return lu+ll
```

3.3 Geometric queries

It is a fairly common situation to encounter a big number of database records that carry some sort of geometric and positional data. Databases naturally cover data accessing through their B-indexing trees, which is a generic approach. Accessing geometric data sometimes requires slightly modified strategies. In this section we mostly deal with specific tree structures that allow us to perform range, interval and segment queries on geometric data.

Range queries

In database data we can encounter a number of continuous attributes, which combined together can constitute a multidimensional space having each record as a point in the space, as seen in Figure 3.28.

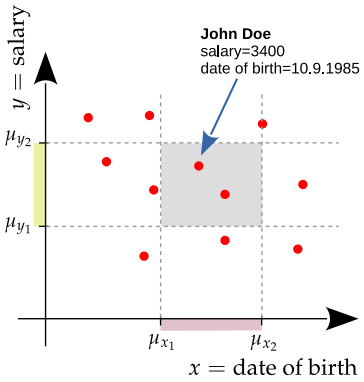


Figure 3.28: An example of database records in 2-dimensional space.

1-dimensional range queries

We start with some simple examples. Let us imagine a set of data records having continuous values, for example people's heights. Again, we can define this as a set of points $P_p = \{p_1, ..., p_n\}$, which are 1-dimensional in our specific case. Our specific query requires finding a subset of records that are in the range $[\mu_1, \mu_2]$, let us say $\mu_1 = 165, \mu_2 = 184$. Structure that allows efficient query execution, or data search, used also in the range queries, is the well-known balanced binary tree τ.

Binary trees can be used in two distinct ways. The first approach, as seen in Algorithm 3.1, is to mix values with search guiding internal nodes. In such a binary tree, internal nodes are not only guiding the search, but are also keeping the values. Such an approach can be used when the binary tree is self-contained, meaning that values are present only in the binary tree and there are no other external pointers to these values exist. This approach was valid in the aforementioned algorithm, as the binary tree τ is only a byproduct of the algorithm, which helps to speed up the line segment search.

The other approach is to separate guiding internal nodes from the values, where the values become leaves in the binary tree. In such an approach, the guiding internal nodes are only used for searching and finding the appropriate values in leaves. This is used when we have values as external data structures, in databases for example, and where the binary tree is only a structure that is used to speed up the search. Figure 3.29 shows such a binary tree, which will be used throughout this section.

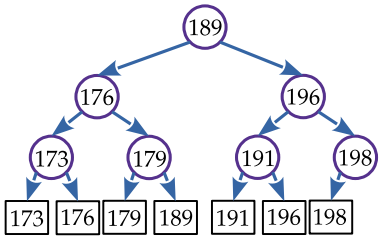


Figure 3.29: Binary tree having values in leaves.

When traversing down the binary tree τ, first we want to find the *splitting node* n_{split}. The splitting node can be defined as the first node in the binary tree which is in the searched range. If we start from the *root node* that is left from the lower range bound, or $n_{root} < \mu_1$, we traverse to the right subtree. Contrarily, if we start from the *root node*

that is right from the upper range bound, or $\mu_2 < n_{root}$, we traverse to the left subtree. Once we encounter the first node $\mu_1 \leq n_{split} \leq \mu_2$, we start traversing in both directions, including both child subtrees.

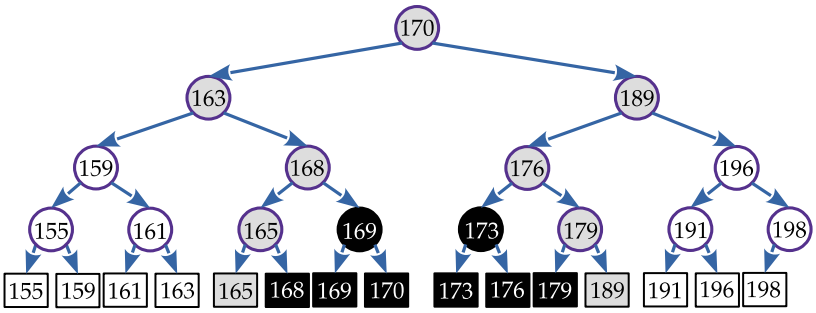


Figure 3.30: Binary tree traversing in search for the range [165, 184]. Gray nodes are in the search path, while black nodes are in the pruned subtrees.

Here we split the binary tree search traversal in two parts. First, we traverse from the splitting node n_{split} to the left, in search for the lower range bound μ_1. In this left traversal we have two situations:

1. **The current node n is in the range**, meaning $\mu_1 \leq n$, then we are confident that its right subtree is completely in the range. Such subtree can be pruned from the search, and only leaves are taken. We continue this left traversal to the left child of the node n.

2. **The current node n is not in the range**, meaning $n < \mu_1$, then we are confident that its left subtree is not in the range at all, which is then skipped. We continue this left traversal to the right child of the node n.

Second, we traverse from the splitting node n_{split} to the right, in search for the upper range bound μ_2. In this right traversal we have two situations:

1. **The current node n is in the range**, meaning $n \leq \mu_2$, then we are confident that its left subtree is completely in the range. Such subtree can be pruned from the search, and only leaves are taken. We continue this right traversal to the right child of the node n.

2. **The current node n is not in the range**, meaning $\mu_2 < n$, then we are confident that its right subtree is not in the range at all, which is then skipped. We continue this right traversal to the left child of the node n.

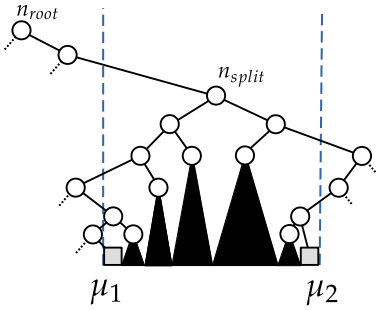


Figure 3.31: Binary tree traversing and pruning in search of the queried range $[\mu_1, \mu_2]$.

```
function FindSplittingNode(τ, μ1, μ2)
    n ← root(τ)
    while n is not leaf and (μ1 ≥ v(n) or μ2 ≤ v(n)) do
        if μ2 ≤ v(n) then
            n ← leftChild(n)
        else
            n ← rightChild(n)
    return n
```

```
function 1DRangeQuery(τ, μ1, μ2)
    rv ← ∅
    n_split ← FindSplittingNode(τ, μ1, μ2)
    if n_split is leaf then
        if v(n_split) is in the range [μ1, μ2] then
            add n_split to rv
    else
                                                    ▷ left traversal
        n ← leftChild(n_split)
        while n is not leaf do
            if μ1 ≤ v(n) then
                add ReportSubtree(rightChild(n)) to rv
                n ← leftChild(n)
            else
                n ← rightChild(n)
        if n is leaf and v(n) is in the range [μ1, μ2] then
            add n to rv
                                                    ▷ right traversal
        n ← rightChild(n_split)
        while n is not leaf do
            if v(n) ≤ μ2 then
                add ReportSubtree(leftChild(n)) to rv
                n ← rightChild(n)
            else
                n ← leftChild(n)
        if n is leaf and v(n) is in the range [μ1, μ2] then
            add n to rv
    return rv
```

Algorithm 3.4: Binary tree range search algorithm.

Complexity is $O(k + \log n)$

Both traversals are continued until we reach a leaf node. An example of such range search traversal can be seen in Figure 3.31.

The ReportSubtree procedure simply traverses through the whole subtree to find all leaves and return them in a list. This reporting of a subtree takes $O(k)$ to complete, where k is the number of leaves we seek, as the number of subtree's internal nodes is smaller than the number of leaves. Because of then traversal split, the searching part of the algorithm has complexity $O(\log n) \leq t(query) \leq O(2 \log n)$. The final complexity of the 1-dimensional range query algorithm can be expressed as $O(k + \log n)$.

Creating balanced binary tree for 1-dimensional case

We started describing how we need to navigate the balanced tree for 1-dimensional range queries. Now to the procedure of creating the balanced binary tree. We want to create a balanced binary tree for a set of values $P_p = \{p_1, ..., p_n\}$ (points in multidimensional case). For this purpose we use premeditated approach, where the set of values are pre-ordered and equally divided, to achieve the balance in the

Listing 3.4: 1-dimensional range query algorithm python code.

```
def ReportSubtree(rv,n):
  if n is not None:
    if isinstance(n,Value):
      rv.append(n)
    else:
      ReportSubtree(rv,n.leftChild)
      ReportSubtree(rv,n.rightChild)
```

```
def FindSplittingNode(tau,mu1,mu2):
  n=tau.rootNode
  while not isinstance(n,Value) and
    ((mu1 is not None and mu1>=n.value) or
     (mu2 is not None and mu2<=n.value)):
    if mu2<=n.value:
      if n.leftChild is not None: n=n.leftChild
      else: return None
    else:
      if n.rightChild is not None: n=n.rightChild
      else: return None
  return n
```

```
def OneDRangeQuery(tau,mu1,mu2):
  rv=[]
  nsplit=FindSplittingNode(tau,mu1,mu2)
  if nsplit is None: return rv
  if isinstance(nsplit,Value):
  if isInRange(mu1,nsplit.value,mu2):
    rv.append(nsplit)
  else:
    n=nsplit.leftChild
    while n is not None and not isinstance(n,Value):
      if mu1 is not None and mu1<=n.value:
        ReportSubtree(rv,n.rightChild)
        n=n.leftChild
      else: n=n.rightChild
    if n is not None and isinstance(n,Value) and isInRange(mu1,n.value,mu2): rv.append(n)
    n=nsplit.rightChild
    while n is not None and not isinstance(n,Value):
      if mu2 is not None and n.value<=mu2:
        ReportSubtree(rv,n.leftChild)
        n=n.rightChild
      else: n=n.leftChild
    if n is not None and isinstance(n,Value) and isInRange(mu1,n.value,mu2): rv.append(n)
  return rv
```

```
def isInRange(q1,v,q2):
  if q1 is not None and q2 is not None and q1<=v<=q2: return True
  elif q1 is None and q2 is not None and v<=q2: return True
  elif q1 is not None and q2 is None and q1<=v: return True
  return False
```

generated binary tree. In range queries we use the median value to divide the set of values.

$$\begin{aligned} v_m &= median(P_p) \\ P_{p_{left}} \cup P_{p_{right}} &= P_p \\ P_{p_{left}} \setminus P_{p_{right}} &= \emptyset \end{aligned} \tag{3.51}$$

The original set of values P_p is divided into two disjunct subsets, such that

$$\begin{aligned} \forall v_l \in P_{p_{left}} &: v_l \leq v_m \\ \forall v_r \in P_{p_{right}} &: v_r > v_m \end{aligned} \tag{3.52}$$

where v_m is the median value that creates the current node, $P_{p_{left}}$ is the subset of values that are lesser or equal to the median and constitute the left subtree from the current node, and $P_{p_{right}}$ is the subset of values that are greater than the median and constitute the right subtree from the current node. Finally, all values from the set of values P_p must have their own leaf.

Figure 3.32 shows an example of 1-dimensional balanced binary tree creation for $P_p = \{160, 163, 169, 172, 173, 180, 188, 192, 193, 198\}$. First median is calculated for $v_m = l_1 = 179$. The left subtree must contain values $P_{p_{left}} = \{160, 163, 169, 172, 173\}$, and the right subtree values $P_{p_{right}} = \{180, 188, 192, 193, 198\}$. The next median is calculated on the left subtree $v_m = l_1 = 167$, after which we repeat the

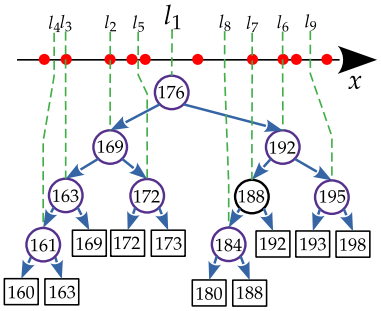


Figure 3.32: An example of 1-dimensional balanced binary tree creation.

value set separation. This requires a recursive algorithm. The recursion stops when we reach a single value from the initial set P_p, which is then used to create a leaf.

function CREATE1DRANGETREE(P_p)
  if $|P_p| = 1$ **then**
    return $n \leftarrow$ leaf having the value from P_p
  else
    $v_m \leftarrow median(P_p)$
    $P_{p_{left}} \leftarrow \{p_i : p_i \in P_p \wedge p_i \leq v_m\}$
    $P_{p_{right}} \leftarrow \{p_i : p_i \in P_p \wedge p_i > v_m\}$
    $n_{left} \leftarrow$ CREATE1DRANGETREE($P_{p_{left}}$)
    $n_{right} \leftarrow$ CREATE1DRANGETREE($P_{p_{right}}$)
    $n \leftarrow$ create node having value v_m
    $leftChild(n) \leftarrow n_{left}$
    $rightChild(n) \leftarrow n_{right}$
  return n

Algorithm 3.5: Balanced binary tree creating algorithm.

Complexity is $O(n \log n)$

```
def create1DTree(self,Pp):
  if len(Pp)==1: return Value(Pp[0])
  else:
    vm=median(Pp)
    Ppleft=list(
        filter(lambda p:p<=vm,Pp))
    Ppright=list(
        filter(lambda p:p>vm,Pp))
    nl=create(Ppleft)
    nr=create(Ppright)
    n=Node(vm)
    n.leftChild,n.rightChild=nl,nr
  return n
```

Listing 3.5: 1-dimensional balanced binary tree creation python code.

The most complex part of the tree creation algorithm is finding the median value for separating left and right subtrees. If we perform an one-time presorting using *quicksort* algorithm, this will cost us $O(n \log n)$. Then we find the median by selecting the middle value for P_p having an odd cardinality, or the arithmetic mean for the middle two values for P_p having an even cardinality. This is repeated in each recursive call, so there is no need for performing multiple sorts as recursion proceeds. Creating binary tree internal nodes takes $O(\log n)$. If we add leaves, this get up to $O(n + \log n)$, or simply $O(n)$. Together, the complete complexity is $O(n + n \log n)$, which is approximately $O(n \log n)$.

Multidimensional range queries

After explaining 1-dimensional range queries, we must ask ourselves what to do in case of multidimensional spaces. Using previous 1-dimensional principle, we could have multiple balanced binary trees, each representing its own dimension. However, querying in such an approach would have higher complexity than needed, $O(k \log n)$ to be precise. In this section, for the sake of simplicity, we explain 2-dimensional range queries.

For the purpose of 2-dimensional range queries, we can create a multileveled binary tree structure. We start by creating 1-dimensional binary tree for one arbitrary dimension, let us say dimension represented by the x axis denoted, as τ_x. Each node in τ_x can have a pointer to the auxiliary binary tree that supports another dimension, in our case represented by the y axis, meaning that each node $n_i \in \tau_x$ can point to its own auxiliary binary tree τ_{y_i}. All leaf values (2-dimensional points) in the n_i subtree must be present in τ_{y_i} as well, differently organized. In this multileveled binary tree structure, the first level represents the x axis, the second level y axis, and so on.

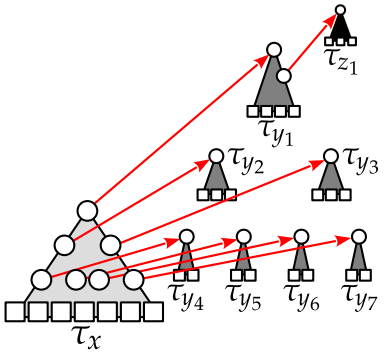


Figure 3.33: An example of multileveled balanced binary tree.

We can extend this structure to support high dimensional spaces. Figure 3.33 shows a 3-dimensional example.

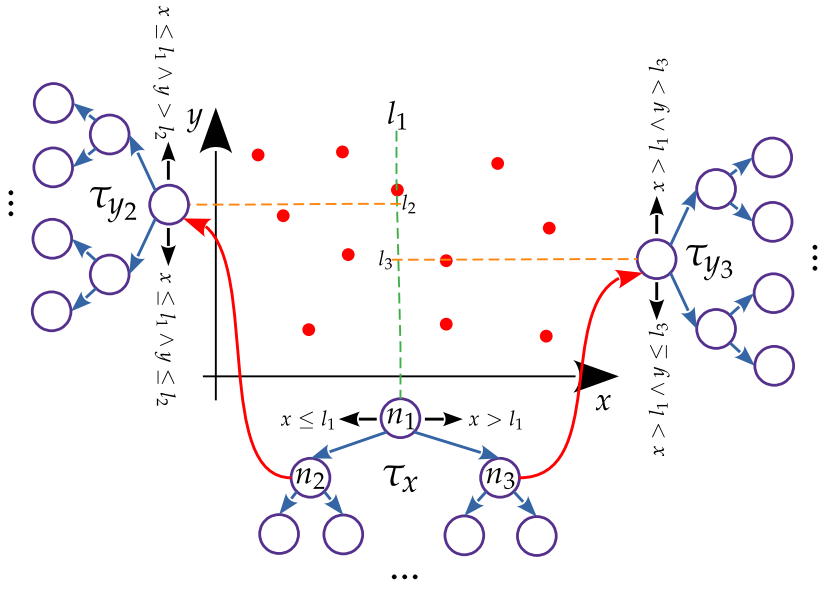


Figure 3.34: Separation of points in subtrees and auxiliary trees in a multileveled binary tree.

To create a multileveled binary tree, we use the same principle of the median division, as in the 1-dimensional case. The top level principle is the same. We find first median value l_1 for the x axis and create a new internal node for this value. The initial set of points P_p is the divided in two subsets $P_{p_{left}} = \{p_i : p_i \leq l_1\}$ and $P_{p_{right}} = \{p_i : p_i > l_1\}$, used to constitute left subtree under node n_2 and the right subtree under node n_3 accordingly. From the same set of points $P_{p_{left}}$ that we used to create the left subtree under node n_2, we create a new auxiliary binary tree τ_{y_2} that represents the second level, or y axis level. Here we determined l_2 as the median value, which is then used to create left and right subtrees. From the set of points $P_{p_{right}}$ that we used to create the right subtree under node n_3, we create a new auxiliary binary tree τ_{y_3} that represent the y axis level. Also, for this binary tree, we determined l_3 as the median value, which is then guiding us to create lower levels of the τ_{y_3} binary tree.

The major problem with such multileveled approach arises when we have more than one points sharing the same dimension value, for example, we have two or more points having the the same x-coordinate, or y-coordinate. When using the multileveled approach, this would lead to having two or more points in the same leaf, which is not included in the basic binary tree definition. This can be solved with some smart pointer magic. We solve this problem from the mathematical point of view, a little further in this Section.

The algorithm for creating a multileveled binary tree is mostly similar to the 1-dimensional version. Here we need to take care of the level we are currently creating, which is determined by the *axis* variable. We descend to as many levels as there are dimensions in the P_p points.

kd-tree
A kd-tree is a similar structure to the multileveled binary tree described here. Instead of multiple levels, a kd-tree can be used for 2-dimensions only. Both dimensions are stored in the single binary tree. Odd levels of the kd-tree are used to store vertical separations (x axis), while even levels of the kd-tree are used to store horizontal separations (y axis). These separations alternate as we descend through the kd-tree structure. For this reason, querying of the kd-tree is somewhat more complex than querying multileveled binary tree.

Algorithm 3.6: Multileveled balanced binary tree creating algorithm.

function CREATE2DRANGETREE($P_p, axis$)
  if more dimensions than *axis* are in the P_p points **then**
    $root(\tau_{assoc}) \leftarrow$CREATE2DRANGETREE($P_p, axis + 1$)
  if $|P_p| = 1$ **then**
    $n \leftarrow$ leaf having the value $P_p[0]$
  else
    $v_m \leftarrow median(\{p_i[axis] : p_i \in P_p\})$
    $P_{p_{left}} \leftarrow \{p_i : p_i \in P_p \wedge p_i[axis] \leq v_m\}$
    $P_{p_{right}} \leftarrow \{p_i : p_i \in P_p \wedge p_i[axis] > v_m\}$
    $n_{left} \leftarrow$ CREATE2DRANGETREE($P_{p_{left}}, axis$)
    $n_{right} \leftarrow$ CREATE2DRANGETREE($P_{p_{right}}, axis$)
    $n \leftarrow$ create node having value v_m
    $leftChild(n) \leftarrow n_{left}$
    $rightChild(n) \leftarrow n_{right}$
  if more dimensions than *axis* are in the P_p points **then**
    $nlevel(n) \leftarrow root(\tau_{assoc})$
  return n

Complexity is $O(n \log n)$

Algorithm 3.7: 2-dimensional binary tree range search algorithm.

function 2DRANGEQUERY($\tau_x, [x_1, x_2] \times [y_1, y_2]$)
  $rv \leftarrow \emptyset$
  $n_{split} \leftarrow$ FINDSPLITTINGNODE(τ_x, x_1, x_2)
  if n_{split} is leaf **then**
    if $v(n_{split})$ is in the range $[x_1, x_2] \times [y_1, y_2]$ **then**
      add n_{split} to rv
  else
    $n \leftarrow leftChild(n_{split})$
    while n is not leaf **do**
      if $x_1 \leq x(v(n))$ **then**
        $\tau_y \leftarrow nlevel(rightChild(n))$
        add 1DRANGEQUERY(τ_y, y_1, y_2) to rv
        $n \leftarrow leftChild(n)$
      else
        $n \leftarrow rightChild(n)$
    if n is leaf and $v(n)$ is in the range $[x_1, x_2] \times [y_1, y_2]$ **then**
      add n to rv
    $n \leftarrow rightChild(n_{split})$
    while n is not leaf **do**
      if $x(v(n)) \leq x_2$ **then**
        $\tau_y \leftarrow nlevel(leftChild(n))$
        add 1DRANGEQUERY(τ_y, y_1, y_2) to rv
        $n \leftarrow rightChild(n)$
      else
        $n \leftarrow leftChild(n)$
    if n is leaf and $v(n)$ is in the range $[x_1, x_2] \times [y_1, y_2]$ **then**
      add n to rv
  return rv

Complexity is $O(k + \log^2 n)$

The complexity of the multileveled binary tree creation algorithm follows the same logic as with 1-dimensional creation. Having linked pre-sorted lists of points for each dimension can help both with finding median and cutting lists for the left and right subtrees, as well as for the next level. Finally, this results in $O(n \log n)$ complexity.

Range querying of the multileveled binary tree is an extension of the 1-dimensional range query. For all nodes, we need to consult the associated next level binary tree. This can happen only when we need to report the whole subtree (ReportSubtree) under a node in the parent level, which means that all leaves in the subtree belongs to the range in this dimension. However, this does not need to be the case in the next level, representing the other dimension, which is the reason why we need to descend and query next levels.

Listing 3.6: 2-dimensional range query and creation algorithm python code. We store lists of points in leaves, which allows us to have more points having the same x or y value.

```
class BinaryTreeWithLeafValues:
  rootNode=None
  def __init__(self,Pp,axis=0):
    self.rootNode=self._create(Pp,axis)
  def _create(self,Pp,axis):
    if len(Pp)==0: return None
    tau_asoc=None
    if axis+1<2:
      tau_asoc=BinaryTreeWithLeafValues(Pp,axis+1)
    dist_vals=set(map(lambda p:p[axis],Pp))
    if len(Pp)==1 or len(dist_vals)==1:
      n=Value(Pp)
      if tau_asoc is not None: n.nlevel=tau_asoc
    else:
      a_vals=list(map(lambda p:p[axis],Pp))
      vm=median(a_vals)
      Ppleft=list(filter(lambda p:p[axis]<=vm,Pp))
      Ppright=list(filter(lambda p:p[axis]>vm,Pp))
      nl,nr=self._create(Ppleft,axis),self._create(Ppright,axis)
      n=Node(vm)
      n.leftChild,n.rightChild=nl,nr
      if tau_asoc is not None:
        n.nlevel=tau_asoc
    return n
  def TwoDRangeQuery(self,x1,x2,y1,y2):
    rv=[]
    nsplit=FindSplittingNode(self,x1,x2)
    if nsplit is None: return rv
    if isinstance(nsplit,Value):
      if nsplit.isInRange(x1,x2,y1,y2): rv.append(nsplit)
    else:
      n=nsplit.leftChild
      while n is not None and not isinstance(n,Value):
        if x1 is None or (x1 is not None and x1<=n.value):
          rv=rv+OneDRangeQuery(n.rightChild.nlevel,1,y1,y2)
          n=n.leftChild
        else: n=n.rightChild
      if n is not None and isinstance(n,Value) and n.isInRange(x1,x2,y1,y2):
        rv.append(n)
      n=nsplit.rightChild
      while n is not None and not isinstance(n,Value):
        if x2 is None or (x2 is not None and n.value<=x2):
          rv=rv+OneDRangeQuery(n.leftChild.nlevel,1,y1,y2)
          n=n.rightChild
        else: n=n.leftChild
      if n is not None and isinstance(n,Value) and n.isInRange(x1,x2,y1,y2):
        rv.append(n)
    return rv
```

The complexity of querying the multileveled binary tree is somewhat different than in the 1-dimensional case. In each internal node of the τ_x binary tree, we have an association to the next level binary tree τ_{y_i}, which produces $O(\log n \log n)$, or $O(\log^2 n)$. Adding the k reported points from the initial set P_p, we get $O(k + \log^2 n)$.

The proposed multileveled binary tree is obviously not too efficient memory-wise, as we can create multiple structures that can access to the same leaves on deeper levels, the second level and on. This also influences the complexity, which seems to be tied to the total number of dimensions involved. The complexity can be lowered down to $O(\log n)$ by using a strategy names *fractional cascading* from the second level forth.

Composite-number spaces

We already determined that there is a problem to create a multileveled binary tree when two or more point have the same dimensional value, for example the same x-coordinate. These points were placed in the real space $\mathbb{R}^2$. Instead of real numbers, we can use composite numbers $(x|y)$ having the following lexicographical ordering

$$(x_1|y_1) < (x_2|y_2) \Leftrightarrow x_1 < x_2 \vee (x_1 = x_2 \wedge y_1 < y_2) \tag{3.53}$$

We can transform points in our $\mathbb{R}^2$ space into the points in the composite-number 2-dimensional place on the following way

$$p = (p_x, p_y) \Rightarrow \hat{p} = ((p_x|p_y), (p_y|p_x)) \tag{3.54}$$

A range can be also transformed as

$$R = [x_1, x_2] \times [y_1, y_2] \rightarrow \hat{R} = [(x_1|-\infty), (x_2, \infty)] \times [(y_1|-\infty), (y_2|\infty)] \tag{3.55}$$

Lemma 3.1. Let p be a point and R a range in $\mathbb{R}^2$. Then we can conclude that

$$p \in R \Leftrightarrow \hat{p} \in \hat{R} \tag{3.56}$$

holds true.

Proof: For the point in the real space we can take $p = (p_x, p_y)$ and for the range $R = [x_1, x_2] \times [y_1, y_2]$. The point p is in the range R if

$$x_1 \leq p_x \leq x_2 \wedge y_1 \leq p_y \leq y_2 \tag{3.57}$$

Transformed to the composite-number space, the point is

$$\hat{p} = ((p_x|p_y), (p_y|p_x)) \tag{3.58}$$

and the range is

$$\hat{R} = [(x_1|-\infty), (x_2, \infty)] \times [(y_1|-\infty), (y_2|\infty)] \tag{3.59}$$

Using the lexicographical order in (3.53), we can write the following

$$(x_1, -\infty) \leq (p_x|p_y) \leq (x_2, \infty) \wedge (y_1, -\infty) \leq (p_y|p_x) \leq (y_2, \infty) \tag{3.60}$$

which can be translated to

$$\begin{aligned} x_1 < p_x < x_2 \vee ((p_x = x_1 \vee p_x = x_2) \wedge -\infty < p_y < \infty) \wedge \\ y_1 < p_y < y_2 \vee ((p_y = y_1 \vee p_y = y_2) \wedge -\infty < p_x < \infty) \end{aligned} \tag{3.61}$$

This is equal to (3.57).

Segment queries

Another type of geometrical queries are so called *window queries*. The principle of geometrical window queries is similar to the range queries from the geometrical perspective. In the window queries we want to find all geometrical objects that are fully or partially contained in the query window. In a 2-dimensional real space $\mathbb{R}^2$ (a plane), a query window is expressed the same as the range $W_q = [q_{x_1}, q_{x_2}] \times [q_{y_2}, q_{y_2}]$.

Let us imagine having a set of line segments in the plane. This set of line segments can resemble for circuit board lines, roads, railways, air lines, or similar. Querying such set of line segments means detecting all line segments that are fully or partially contained in the query window. Figure 3.35 shows an example of a set of line segments and a query window W_q. Based on the example, we can identify two major situations:

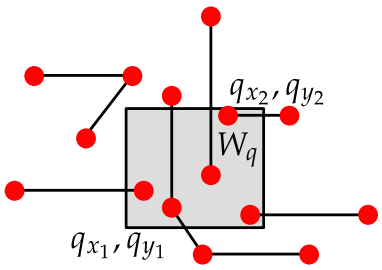


Figure 3.35: Interval query generic example.

1. **A line segment has at least one of its end points in the query window**. We can add line segments that have both end points in the query window to this case.

2. **A line segment that only intersects the query window**. Such line segments has both end points outside the query window, and are only intersecting two of the query window bounds.

To support quick searching of line segments that belong to the query window, we organize them in a binary tree like structures.

Creating interval tree

To simplify the explanation, we start from a parallel interval example. Let us imagine a set of parallel intervals defined as $I = \{s_i = ([x_{i_1}, x_{i_2}], y_i)\}$, such as in Figure 3.36. These intervals are orthogonal to the y-axis and parallel with the x-axis. Each interval has the left end point (x_{i_1}, y_i) and the right end point (x_{i_2}, y_i). To organize these intervals we use a data structure called *interval tree*. We start by creating a binary tree to organize x-axis intervals $[x_{i_1}, x_{i_2}]$. First, we calculate the median value for all end points of all intervals, which is denoted as l_1 in Figure 3.36. Some intervals intersect the value l_1, meaning they have one end point of the left side and one on the right side from the value l_1. We create node n_1 having the value $x_{med}(n_1) = l_1$ and in the node store intervals that intersect the value l_1, which are $I(n_1) = \{s_1, s_2, s_3\}$. All intervals that have both end points left of the value l_1 belong to the left subtree of the node n_1, and all intervals that have both end points right of the value l_1 belong to the right subtree of the node n_1. Again, when creating the left subtree of the node n_1, we take only the intervals s_4 and s_5 and calculate the median value of their end points, which comes out as l_2. For the right subtree we take intervals s_6 and s_7 and calculate their median value, which comes out as l_3. This creation divided to the left and to the right subtree recursively continues as long as there as some intervals left for the creation of subtrees.

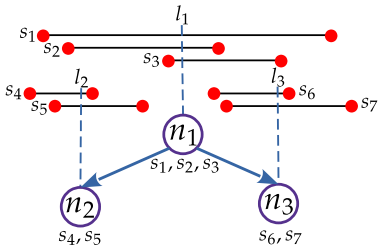


Figure 3.36: 1-dimensional interval tree creation basic structure.

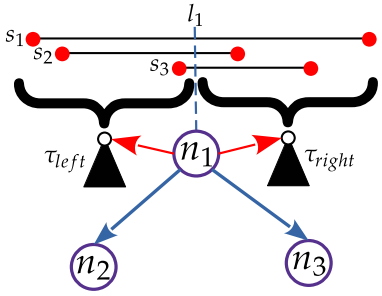


Figure 3.37: 1-dimensional interval tree with end points organized in range trees.

We need to organize end points of intervals that intersect the median value $x_{med}(n_i)$, stored in each node n_i. This can be done by using various data structures, such as the ***multidimensional range tree*** described in the previous Section. As seen in Figure 3.37, each node has two helping data structures that help structuring end points:

- τ_{left} - A data structure that contains all left end points of the node intersecting intervals.
- τ_{right} - A data structure that contains all right end points of the node intersecting intervals.

There needs to be a back reference from the end point stored in the τ_{left} and τ_{right} data structures to the originating interval stored in the related node. Once we perform a query on one of the helping data structures τ_{left} or τ_{right}, we immediately need to have the end point related interval reference, to avoid any additional unnecessary iterations.

Algorithm 3.8: The interval tree creating algorithm.

function CREATEINTERVALTREE(I)
  if $I = \emptyset$ **then**
    $n \leftarrow$ empty leaf
  else
    $x_{med} \leftarrow median(endpoints(I))$
    $I_{left} \leftarrow \{s_i : s_i \in I \wedge x_{i_1}(s_i) < x_{med} \wedge x_{i_2}(s_i) < x_{med}\}$
    $I_{right} \leftarrow \{s_i : s_i \in I \wedge x_{i_1}(s_i) > x_{med} \wedge x_{i_2}(s_i) > x_{med}\}$
    $I_{med} \leftarrow I \setminus (I_{left} \cup I_{right})$
    $n \leftarrow$ create a node having value x_{med}
    $intervals(n) \leftarrow I_{med}$
    $leftChild(n) \leftarrow$ CREATEINTERVALTREE(I_{left})
    $rightChild(n) \leftarrow$ CREATEINTERVALTREE(I_{right})
    if $I_{med} \neq \emptyset$ **then**
      $P_{left} \leftarrow \{((x_{i_k}(s_i), y_i(s_i)), s_i) : s_i \in I_{mid} \wedge x_{i_k}(s_i) \leq x_{med}\}$
      $P_{right} \leftarrow \{((x_{i_k}(s_i), y_i(s_i)), s_i) : s_i \in I_{mid} \wedge x_{i_k}(s_i) > x_{med}\}$
                ▷ We include the back reference to s_i
      $\tau_{left}(n) \leftarrow$ CREATE2DRANGETREE(P_{left})
      $\tau_{right}(n) \leftarrow$ CREATE2DRANGETREE(P_{right})
  return n

Complexity is $O(n \log^2 n)$

We are using 2-dimensional range trees for τ_{left} and τ_{right}, since intervals in $I(n_i)$ are 2-dimensional and querying the y-axis is done through these range trees.

Based on the previous cases, such as the range tree creation, we estimate that creation of the interval tree is closer to the 2-dimensional range tree, due to the associated range trees. Knowing that due to the sorting needs, the basic interval tree takes $O(n \log n)$ to create. When we add two range trees to each node, it rises up to $O((n \log n)(2 \log n)) = O(2n \log^2 n)$, which is $O(n \log^2 n)$.

Listing 3.7: The interval tree creation algorithm python code.

```
class IntervalTree:
  rootNode=None
  def __init__(self,I):
    self.rootNode=self._create(I)
  def _create(self,I):
    if len(I)==0: return None
    else:
      epoints=set(map(lambda si:si[0][0],I)).union(set(map(lambda si:si[0][1],I)))
      xmed=median(epoints)
      n=Node(xmed)
      Ileft=list(filter(lambda si:si[0][0]<xmed and si[0][1]<xmed,I))
      Iright=list(filter(lambda si:si[0][0]>xmed and si[0][1]>xmed,I))
      Imed=list(filter(lambda si:si not in Ileft+Iright,I))
      n.intervals=Imed
      nl,nr=self._create(Ileft),self._create(Iright)
      n.leftChild,n.rightChild=nl,nr
      if len(Imed)>0:
        Pleft=list(map(lambda si:{'point':(min(si[0][0],si[0][1]),si[1]),'interval':si},Imed))
        Pright=list(map(lambda si:{'point':(max(si[0][0],si[0][1]),si[1]),'interval':si},Imed))
                # Here we encode both the point and the back reference to the interval
        n.taul=BinaryTreeWithLeafValues(Pp=Pleft)
        n.taur=BinaryTreeWithLeafValues(Pp=Pright)
      return n
```

Querying interval tree

Querying of an interval tree is very similar to querying a range tree, as seen in Figure 3.31. In this case, we have a query window $W_q = [q_{x_1}, q_{x_2}] \times [q_{y_1}, q_{y_2}]$. We can be sure that any node n_i in the interval tree being in the query interval $q_{x_1} \leq x_{med}(n_i) \leq q_{x_2}$ contains intervals that are in the x-axis query interval $[q_{x_1}, q_{x_2}]$. They need to be additionally checked for the y-axis interval $y_i \in [q_{y_1}, q_{y_2}]$, which is done by consulting the associated range trees $\tau_{left}(n_i)$ and $\tau_{right}(n_i)$. We take the same approach as in range queries, by finding a first splitting node (if possible), whose value is $q_{x_1} \leq x_{med}(n_{split}) \leq q_{x_2}$, and then building two paths from the splitting node n_{split}, one to the left and one to the right.

For the path to the left, we have the following situations:

1. $\forall n_i \in \tau : q_{x_1} \leq x_{med}(n_i) \leq q_{x_2}$ - We must check which of the end points for intervals in this node are in the query window W_q, so we query $\tau_{right}(n_i)$ and $\tau_{left}(n_i)$ for $W_q = [q_{x_1}, q_{x_2}] \times [q_{y_1}, q_{y_2}]$. All intervals that have end points in the query window W_q are added to the resulting intervals. We move to the left child node $n_i = leftChild(n_i)$.

2. $\forall n_i \in \tau : x_{med}(n_i) < q_{x_1}$ - We must check which of the right end points for intervals in this node are in the query window W_q, so we query $\tau_{right}(n_i)$ for $W_q = [q_{x_1}, q_{x_2}] \times [q_{y_1}, q_{y_2}]$. All intervals that have end points in the query window W_q are added to the resulting intervals. We move to the right child $n_i = rightChild(n_i)$.

For the path to the right, we have the following situations:

3. $\forall n_i \in \tau : q_{x_1} \leq x_{med}(n_i) \leq q_{x_2}$ - We must check which of the end points for intervals in this node are in the query window W_q, so we query $\tau_{right}(n_i)$ and $\tau_{left}(n_i)$ for $W_q = [q_{x_1}, q_{x_2}] \times [q_{y_1}, q_{y_2}]$. All intervals that have end points in the query window W_q are added to the resulting intervals. We move to the right child node $n_i = rightChild(n_i)$.

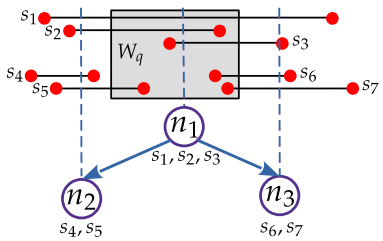


Figure 3.38: A query window W_q for the interval tree.

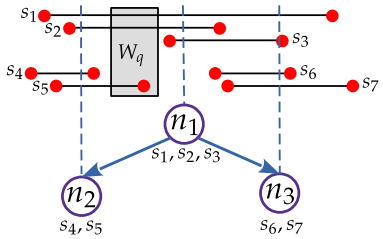


Figure 3.39: A query window W_q for the interval tree, having some intervals fully intersecting the query interval $[q_{x_1}, q_{x_2}]$.

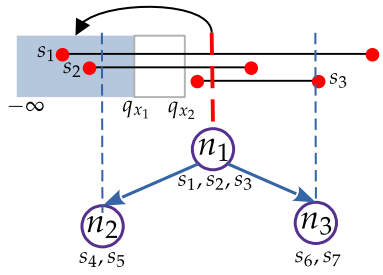


Figure 3.40: A left range query for intervals fully intersecting the query interval.

4. $\forall n_i \in \tau : q_{x_2} < x_{med}(n_i)$ - We must check which of the left end points for intervals in this node are in the query window W_q, so we query $\tau_{left}(n_i)$ for $W_q = [q_{x_1}, q_{x_2}] \times [q_{y_1}, q_{y_2}]$. All intervals that have end points in the query window W_q are added to the resulting intervals. We move to the left child $n_i = leftChild(n_i)$.

In case we don't find the splitting node n_{split}, the goal is find one or both adjacent nodes, as in Figure 3.39. In such case, we require to check the associated range trees, as explained in points 2 and 4.

The only situation left, which is still not covered by the previous analysis, is when intervals fully intersect the query window W_q without having end points in it, as seen in Figure 3.39. We can see that intervals s_1 and s_2 are fully intersecting the query range $[q_{x_1}, q_{x_2}]$, yet their node n_1 is not in it. To find such intervals, we have the following situations:

- $\forall n_i \in \tau : x_{med}(n_i) < q_{x_1}$ - We must check which of the right end points for intervals in this node are right from q_{x_2}, so we query $\tau_{right}(n_i)$ for the query window $W_q^r = [q_{x_2}, \infty) \times [q_{y_1}, q_{y_2}]$. All intervals that have end points in the query window W_q^r are added to the resulting intervals.

- $\forall n_i \in \tau : q_{x_2} < x_{med}(n_i)$ - We must check which of the left end points for intervals in this node are left from q_{x_1}, so we query $\tau_{left}(n_i)$ for the query window $W_q^l = (-\infty, q_{x_1}] \times [q_{y_1}, q_{y_2}]$. All intervals that have end points in the query window W_q^l are added to the resulting intervals.

Figure 3.40 shows one of such situations, where the query window W_q is left from the node n_1 and intervals from the node n_1 are fully intersecting the query window. The query window $W_q^l = (-\infty, q_{x_1}] \times [q_{y_1}, q_{y_2}]$ on the range tree $\tau_{left}(n_1)$ reveals all such intervals. We must notice, that the same end point query on the node n_3 is not effective, which results in a conclusion that only a query window adjacent leaf node and its parents (up to the root node) can contain intervals that can fully intersect the query window W_q.

Lemma 3.2. For a leaf node n_i adjacent to the right of the query window $W_q = [q_{x_1}, q_{x_2}] \times [q_{y_1}, q_{y_2}]$ it is sufficient to find all intervals that have end points in range tree $\tau_{left}(n_i)$ and are in the query window $W_q \cup W_q^l = (-\infty, q_{x_2}] \times [q_{y_1}, q_{y_2}]$. Equivalently, for a leaf node n_j adjacent to the left of the same query window W_q, it is sufficient to find all intervals that have end points in range tree $\tau_{right}(n_j)$ and are in the query window $W_q \cup W_q^r = [q_{x_1}, \infty) \times [q_{y_1}, q_{y_2}]$.

Proof: For the adjacent leaf nodes n_i and n_j we can say that their intervals either:

- End in the query window W_q, meaning that one of their end points is in the query window $W_q = [q_{x_1}, q_{x_2}] \times [q_{y_1}, q_{y_2}]$ or

- They fully intersect the query window W_q, meaning that one of their end points end up in the query window W_q^l for the node n_i and in the query window W_q^r for the node n_j.

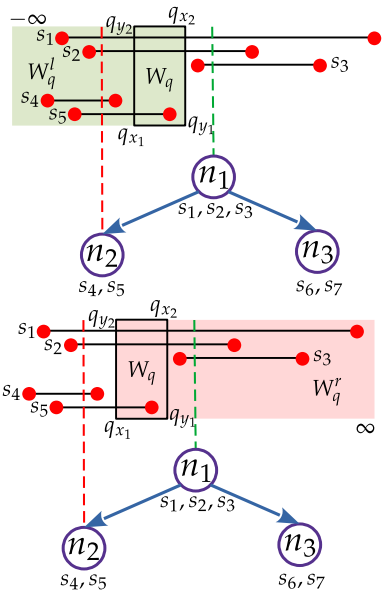


Figure 3.41: A lemma 3.2 graphical example.

Algorithm 3.9: Interval tree query algorithm.

function IntervalQuery($n, [q_{x_1}, q_{x_2}] \times [q_{y_1}, q_{y_2}]$)
  $iv \leftarrow \emptyset$
  if $q_{x_1} \leq x_{med}(n) \leq q_{x_2}$ **then**
    $p \leftarrow$ 2DRangeQuery($\tau_{left}(n), (-\infty, q_{x_2}] \times [q_{y_1}, q_{y_2}]$)
    $p \leftarrow p \cup$ 2DRangeQuery($\tau_{right}(n), [q_{x_1}, \infty) \times [q_{y_1}, q_{y_2}]$)
    $iv \leftarrow$ all intervals of the endpoints in p
    $move \leftarrow both$
  else if $x_{med}(n) < q_{x_1}$ **then**
    $p \leftarrow$ 2DRangeQuery($\tau_{right}(n), [q_{x_1}, \infty) \times [q_{y_1}, q_{y_2}]$)
    $iv \leftarrow$ all intervals of the endpoints in p
    $move \leftarrow right$
  else if $q_{x_2} < x_{med}(n)$ **then**
    $p \leftarrow$ 2DRangeQuery($\tau_{left}(n), (-\infty, q_{x_2}] \times [q_{y_1}, q_{y_2}]$)
    $iv \leftarrow$ all intervals of the endpoints in p
    $move \leftarrow left$
  if $move \in \{both, left\}$ and exists $lc \leftarrow leftChild(n)$ **then**
    $iv \leftarrow iv \cup$ IntervalQuery($lc, [q_{x_1}, q_{x_2}] \times [q_{y_1}, q_{y_2}]$)
  if $move \in \{both, right\}$ and exists $rc \leftarrow rightChild(n)$ **then**
    $iv \leftarrow iv \cup$ IntervalQuery($rc, [q_{x_1}, q_{x_2}] \times [q_{y_1}, q_{y_2}]$)
  return iv

Complexity is $O(\log^3 n)$

Listing 3.8: The interval tree query algorithm python code.

```
class IntervalTree:
  ...
  def IntervalQuery(self,qx1,qx2,qy1,qy2):
    if self.rootNode is not None: return self._IntervalQuery(self.rootNode,qx1,qx2,qy1,qy2)
    else: return []
  def _IntervalQuery(self,n,qx1,qx2,qy1,qy2):
    iv,move=set(),None
    if isInRange(qx1,n.value,qx2):
      ivs=n.taur.TwoDRangeQuery(qx1,None,qy1,qy2)
      ivs+=n.taul.TwoDRangeQuery(None,qx2,qy1,qy2)
      for tmp in ivs: iv.add(tmp[0]['interval']) # We retrieve the back reference to the interval
      move='both'
    elif qx1 is not None and n.value<qx1:
      ivs=n.taur.TwoDRangeQuery(qx1,None,qy1,qy2)
      for tmp in ivs: iv.add(tmp[0]['interval'])
      move='right'
    elif qx2 is not None and n.value>qx2:
      ivs=n.taul.TwoDRangeQuery(None,qx2,qy1,qy2)
      for tmp in ivs: iv.add(tmp[0]['interval'])
      move='left'
    if move in ['both','left'] and n.leftChild is not None:
      iv=iv.union(self._IntervalQuery(n.leftChild,qx1,qx2,qy1,qy2))
    if move in ['both','right'] and n.rightChild is not None:
      iv=iv.union(self._IntervalQuery(n.rightChild,qx1,qx2,qy1,qy2))
    return iv
```

The complexity is similar to the 2-dimensional range tree, as we have nested range trees in each node. This specifically means that the complexity is $O((2\log^2 n)(\log n))$, where we have two 2-dimensional range trees per one interval tree node. It is obvious that $2 \ll n$, which is the reason why we set this implementation of the interval tree complexity to $O(\log^3 n)$.

Using priority tree

In the previous section we used 2-dimensional range trees to find interval end points. For a query window $W_q = [q_{x_1}, q_{x_2}] \times [q_{y_1}, q_{y_2}]$, we searched intervals having end points in query windows $W_q \cup W_q^l = (-\infty, q_{x_2}] \times [q_{y_1}, q_{y_2}]$ and $W_q \cup W_q^r = [q_{x_1}, \infty) \times [q_{y_1}, q_{y_2}]$. These are specific queries, for which there are better data structures than range trees. For this purpose, we can use heaps. The main quality of a heap is the sorted tree structure. To improve querying end points in the interval tree for the query window $(-\infty, q_{x_2}] \times [q_{y_1}, q_{y_2}]$, we can use an ascending sorted heap tree, hereby called a *priority tree*.

Let us remember how to build an ascending heap. Let us say we have an ascending sorted set of numbers $P = \{4, 6, 9, 15, 19, 22, 29\}$. We start by creating a *root* node for the smallest member of the set $4 \in P$. Then we divide the reminder of the set $P \setminus \{4\}$ into two equally sized partitions. Partitions must be equally sized if we want to have a balanced heap tree. Partitioning process is arbitrary, and we have some freedom to choose partitions, for example $P_L = \{6, 15, 22\}, P_R = \{9, 19, 29\}$. Using the first partition P_L we create the left subtree, and using the second partition P_R we create the right subtree. This process is repeated recursively until we have the heap tree, which can be seen in Figure 3.42. Notice that the heap tree is sorted ascending from the root node, which allows us to do the first part of the query $(-\infty, q_{x_2}]$.

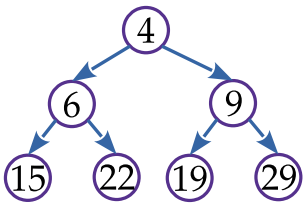


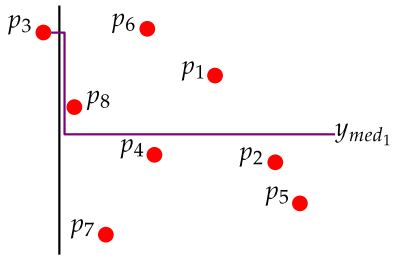
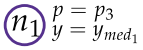
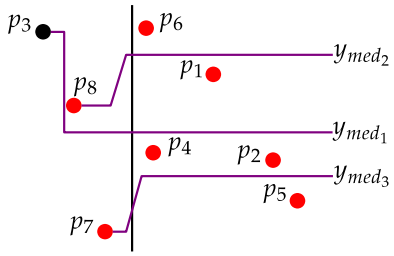
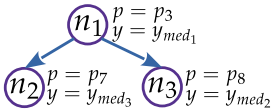
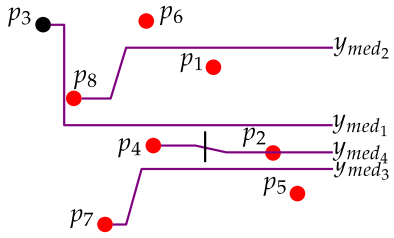
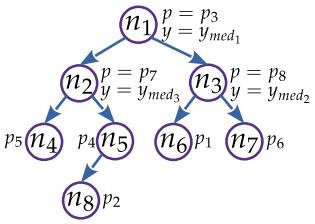
Figure 3.42: An heap example.

Querying a heap is simple, as the recursion occurs until we reach a node that has value greater than the queried value $value(n) > q_{x_2}$, which means that the algorithm has so many recursive calls as many as values in the range query, or $O(k+1)$. The worst case scenario is to have all nodes of the heap in the result, which goes up to the linear complexity $O(n)$. Knowing that the range tree has the complexity $O(k + \log n)$, having $O(k+1)$ certainly seems to be a bit less complex. This complexity is equal to searching through a sorted list.

Algorithm 3.10: Create priority tree algorithm.

function CREATEPRIORITYTREE(P)
  $p_{min} = \underset{p_i \in P}{arg\ min}\ x(p_i)$
  $n \leftarrow$ new node
  $p(n) \leftarrow p_{min}$
  if $P \setminus p_{min} \neq \emptyset$ **then**
    $y_{med} \leftarrow median(P \setminus p_{min})$
    $P_L \leftarrow \{p : p \in P \setminus p_{min}, y(p) \leq y_{med}\}$
    $P_R \leftarrow \{p : p \in P \setminus p_{min}, y(p) > y_{med}\}$
    $y(n) \leftarrow y_{med}$
    if $P_L \neq \emptyset$ **then**
      $leftChild(n) \leftarrow$ CREATEPRIORITYTREE(P_L)
    if $P_R \neq \emptyset$ **then**
      $rightChild(n) \leftarrow$ CREATEPRIORITYTREE(P_R)
  return n

So why bother using a heap? The key lies in the partitioning for the heap. We can embed a point value in each node of the priority tree. Additionally, we can embed a binary tree structure based on the y axis value. Such heap is sorted vertically, which supports the $(-\infty, q_{x_2}]$ part of the window query. On the other side, the y value partitioning that is embedded into the heap structure makes the same structure a binary tree. Combined together, it represents a *priority tree* structure that can be created by Algorithm 3.10.

Description	Points	Priority tree
We select p_3 as the root node, since it is the first point based on its x value. We partition the remainder of the points into two partitions $P_L = \{p_7, p_4, p_2, p_5\}$ and $P_R = \{p_8, p_6, p_1\}$, based on their y value.		
We select p_7 as the left subtree root node and p_8 as the right subtree root node. Our priority tree is now vertically sorted by the x value. We partition the remainder of the points into four partitions $P_{L_1} = \{p_5\}, P_{L_2} = \{p_4, p_2\}, P_L = P_{L_1} \cup P_{L_2}$ and $P_{R_1} = \{p_1\}, P_{R_2} = \{p_6\}, P_R = P_{R_1} \cup P_{R_2}$, based on their y value.		
We continue creating the priority tree through the recursive procedure and partitioning. The final priority tree is vertically sorted by the x value, and horizontally structured as the y value binary tree.		

Finally, the priority tree query is quite similar to the range tree query. To understand it, we must understand Figure 3.31. The main problem in the priority tree structure is it disparity between the point that is stored in each node $p(n)$ and the y median value for the child subtrees, stored as $y(n)$. These values can greatly differ $y(p(n)) \neq y(n)$, which can result in $y(p(n)) \in [q_{y_1}, q_{y_2}], y(n) \notin [q_{y_1}, q_{y_2}]$ or vice versa. This means that the path finding part of the querying algorithm must adhere to the binary tree part of the priority tree, following values $y(n)$, while the heap part of the priority tree must use points stored in the tree nodes. As we proceed by finding path to the range $[q_{y_1}, q_{y_2}]$, we must test all traversed nodes points for the condition $p(n) \in (-\infty, q_{x_2}] \times [q_{y_1}, q_{y_2}]$, and add it to the final result if the condition is satisfied, no matter of the $y(n)$ value. The example can be seen the previous table. The node n_1 has stored the point p_3, which has $y(p_3) \gg y(n_1) = y_{med_1}$. For all $q_{y_1} > y_{med_1}$, we will have the splitting node $n_{split} = n_3, p(n_3) = p_8$. However, it is obvious that

$p(n_1) = p_3$ which was merely on the path to find the splitting node n_3, can also be inside the range query, and must be added to the final result.

Algorithm 3.11: Priority subtree query algorithm.

```
function QueryPrioSubtree(n, q_{x_2})
    res ← ∅
    if n is not leaf and x(p(n)) ≤ q_{x_2} then
        res ← n
        res ← res ∪ QueryPrioSubtree(leftChild(n), q_{x_2})
        res ← res ∪ QueryPrioSubtree(rightChild(n), q_{x_2})
    return res
```

Complexity is $O(k+1)$

Algorithm 3.12: Priority query algorithm.

```
function PriorityTreeQuery(τ, q_{x_2}, q_{y_1}, q_{y_2})
    rv ← ∅
    n_split, nr ← FindSplittingNode(τ, q_{y_1}, q_{y_2})
    for n in nr do
        if p(n) ∈ (−∞, q_{x_2}] × [q_{y_1}, q_{y_2}] then
            add n to rv
    if n_split does not exist then
        return rv
    if n_split is leaf then
        add QueryPrioSubtree(n_split, q_{x_2}) to rv
    else
        if p(n_split) ∈ (−∞, q_{x_2}] × [q_{y_1}, q_{y_2}] then
            add n_split to rv
        n ← leftChild(n_split)
        while n exists and x(p(n)) ≤ q_{x_2} do
            if p(n) ∈ (−∞, q_{x_2}] × [q_{y_1}, q_{y_2}] then
                add n to rv
            if q_{y_1} ≤ y(n) ≤ q_{y_2} then
                add QueryPrioSubtree(rightChild(n), q_{x_2}) to rv
                n ← leftChild(n)
            else
                n ← rightChild(n)
        n ← rightChild(n_split)
        while n exists and x(p(n)) ≤ q_{x_2} do
            if p(n) ∈ (−∞, q_{x_2}] × [q_{y_1}, q_{y_2}] then
                add n to rv
            if q_{y_1} ≤ y(n) ≤ q_{y_2} then
                add QueryPrioSubtree(leftChild(n), q_{x_2}) to rv
                n ← rightChild(n)
            else
                n ← leftChild(n)
    return rv
```

Complexity is $O(k + \log n)$

```
function FindSplittingNode(τ, q_{y_1}, q_{y_2})
    n ← root(τ)
    nr ← ∅
    while n exists and (q_{y_1} > y(n) or q_{y_2} < y(n)) do
        add n to nr
        if q_{y_2} < y(n) then
            n ← leftChild(n)
        else
            n ← rightChild(n)
    return n, nr
```

The complexity of the query algorithm is basically the same as for the range tree. We have a set of reported points and nodes k, and we traverse through the priority tree structure in two paths, which ends up as $O(k + 2\log n)$. Finally, this is $O(k + \log n)$.

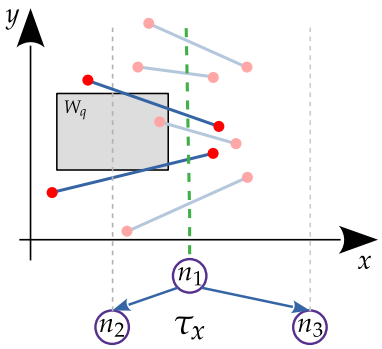


Figure 3.43: Issues when using intervals trees for a generic line segment example.

Segment trees

Parallel intervals and line segments are quite easy to structure and query. What about having generic line segments $S = \{s_i = \overline{p_{i_1}p_{i_2}} : p_{i_1} = (x_{i_1}, y_{i_1}), p_{i_2} = (x_{i_2}, y_{i_2})\}$ that are oriented in all directions, as in Figure 3.35? Can we use interval trees for this purpose?

In the example in Figure 3.43 we can encounter two line segments that would not be found using interval trees. Since the node n_1 is adjacent to the query window W_q from the right side, we need to query the associated intervals left end points. We are looking for all left end points that are in the query window $W_q \cup W_q^l = (-\infty, q_{x_2}] \times [q_{y_1}, q_{y_2}]$. Two highlighted lines do not satisfy this condition, yet they are fully intersecting the query window W_q.

We can circumvent these issues by focusing our queries on the edges of the query window $W_q = [q_{x_1}, q_{x_2}] \times [q_{y_1}, q_{y_2}]$, for example $I_{q_1} = q_{x_1} \times [q_{y_1}, q_{y_2}]$, $I_{q_2} = q_{x_2} \times [q_{y_1}, q_{y_2}]$, and so on. To simplify, we use a single query $I_q = q_x \times [q_{y_1}, q_{y_2}]$ throughout the text. To allow such a query, we need to create a data structure called a *segment tree*, denoted as τ in the following text. The segment tree uses so called *locus approach*, which is partitioning of the problem space into several subspaces where *nothing happens* from the problem point of view. From the geometrical point of view, we can partition our space into regions where we have the same set of line segments. If we focus on the x-axis alone, we can partition the x-axis into partitions where we have the same set of line segments. This can be done by decomposing the x-axis into elementary intervals with respect to the set of input line segments $S = \{s_1 = \overline{p_1p_7}, s_2 = \overline{p_3p_8}, s_3 = \overline{p_2p_4}, s_4 = \overline{p_5p_6}\}$, as seen in Figure 3.44.

Taking the orthogonal projection of each line segment end point to the x-axis, we use

$$x_i = x(p_i) \tag{3.62}$$

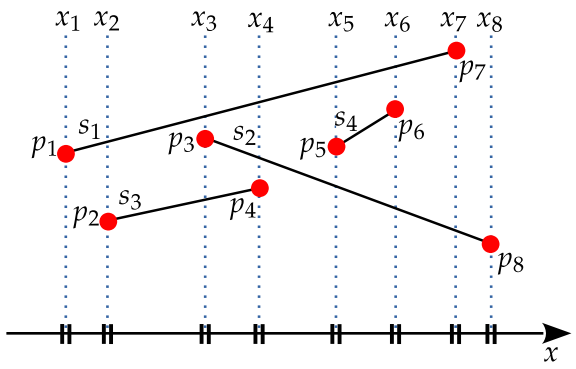


Figure 3.44: Line segments x-axis partitioning.

Using the set of line segments S, we partition the x-axis into elementary intervals

$$(-\infty, x_1), [x_1, x_1], (x_1, x_2), [x_2, x_2], ..., (x_7, x_8), [x_8, x_8], (x_8, \infty) \quad (3.63)$$

Then we turn each elementary interval into leaves of a balanced binary tree. In the nodes (including leaves) of the binary tree, we keep the associated intervals, denoted as $I(n)$. Understandingly, leaves keep only elementary intervals, which represents partitions where we have the same set of line segments from the x-axis point of view. As we proceed to the higher levels of the binary tree, intervals get aggregated, until we reach the root node, which covers $(-\infty, \infty)$.

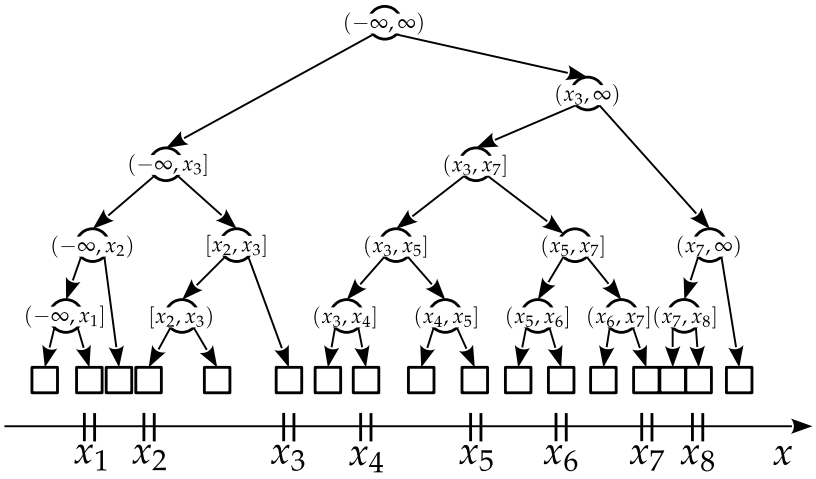


Figure 3.45: The segment tree structure and node-associated intervals.

Figure 3.45 show the segment tree structure and intervals associated with nodes of the tree. We can see two important properties of the segment tree:

1. The whole x-axis is covered by each level of the tree, starting from the root, and there is no gap - part of the x-axis that is not included in the intervals of the segment tree,
2. There is no overlapping of the intervals between sibling nodes (or at the same level of the binary tree).

We know that each elementary interval is having a set of line segments passing through them, and this set of line segments does not

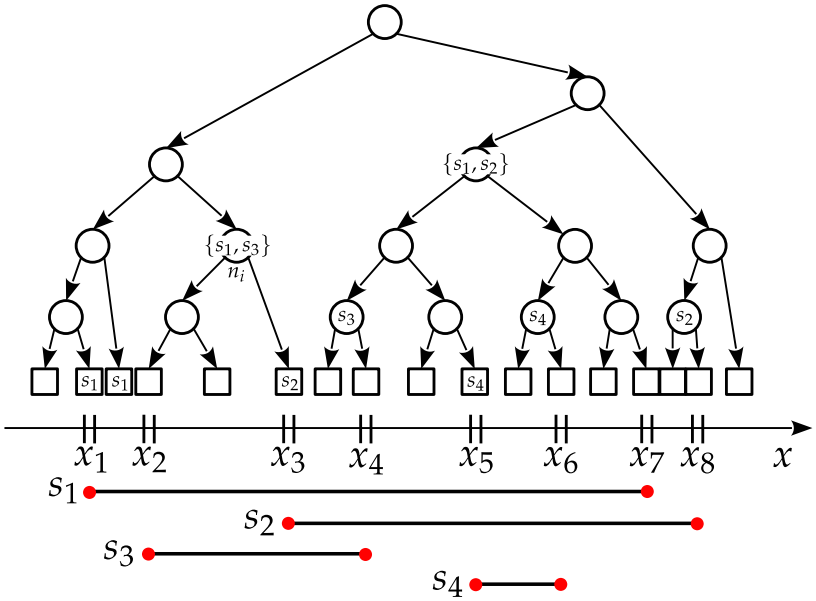


Figure 3.46: Line segments and segment tree nodes association.

change within the elementary interval. All infinitesimal elementary intervals, such as $[x_i, x_i]$, are here to support vertical line segments.

We can assign line segments to leaves of the segment tree and leave it that way. However, in case we have a big number of elementary intervals, this would not be the optimal use of memory. Instead, we can push line segments up through the segment tree structure. To add a line segment s_i to the segment tree, we start descending from its root node. When we find a node n_i whose interval is completely covered by the line segment s_i x-axis orthogonal projection, or $I(n_i) \subseteq [x(p_1(s_i)), x(p_2(s_i))]$, we associate the line segment s_i to the node n_i. The canonical set of line segments associated with the node n_i is denoted as $S(n_i)$.The final result can be seen in Figure 3.46. If we look at the node n_i, we can see that it has the canonical set of line segments $S(n_i) = \{s_1, s_3\}$. This means that both s_1 and s_3 are passing through all descendant leaves of the node n_i. However, the same cannot be said for its sibling node.

Algorithm 3.13: Segment tree line segment insertion procedure.

```
procedure InsertSegment(n, [x1, x2])
    if I(n) ⊆ [x1, x2] then
        add [x1, x2] to S(n)
    else
        if I(leftChild(n)) ∩ [x1, x2] ≠ ∅ then
            InsertSegment(leftChild(n), [x1, x2])
        if I(rightChild(n)) ∩ [x1, x2] ≠ ∅ then
            InsertSegment(rightChild(n), [x1, x2])
```

Let us return to the querying problem. As said before, we are querying the set of line segments for $I_q = q_x \times [q_{y_1}, q_{y_2}]$. Using the segment tree, we can cover the x-axis part of the query, by finding all line segments passing through the vertical line q_x.

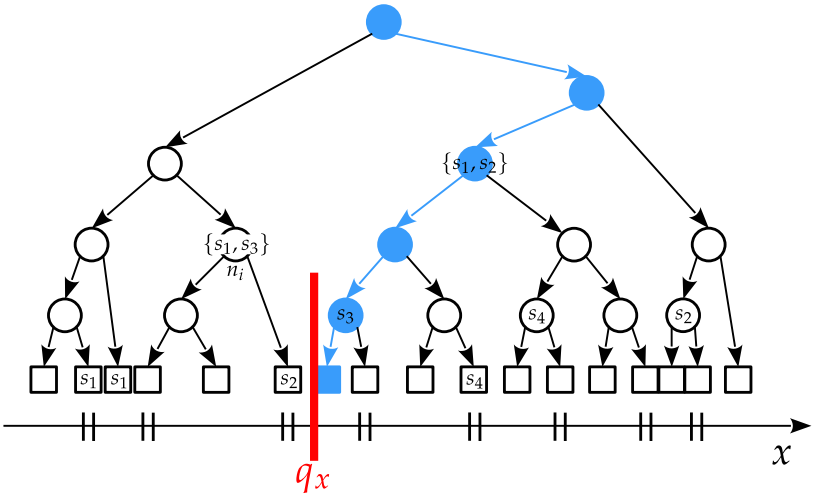


Figure 3.47: Querying the vertical line q_x in the segment tree.

We start querying from the root node and choose child nodes that have $q_x \in I(n)$, until we reach a leaf node, which represents the elementary interval that contains the queried value q_x. The resulting set of line segments is the union of all line segments for all nodes on the vertical path $\mathcal{V}$, or

$$\bigcup_{n \in \mathcal{V}} S(n) \subseteq S \tag{3.64}$$

Each node n_i in the segment tree corresponds to a vertical block bounded by the node interval $V_b(n_i) = I(n_i) \times (-\infty, \infty)$. If there are some line segments in the canonical subset of the node n_i, or $S(n_i) \neq \emptyset$, they fully intersect the vertical block of the node $V_b(n_i)$. In Figure 3.48 we can see that line segments s_1 and s_2 fully intersect the vertical block of the node n_1. Additionally, in Figure 3.48 can be seen that sibling vertical blocks are adjacent and that together they make the vertical block of the parent $V_b(n_1) = V_b(n_2) \cup V_b(n_3)$.

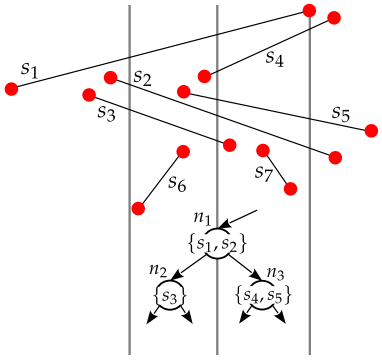


Figure 3.48: Vertical blocks for nodes n_2 and n_3.

It is possible to determine which line segments can be found in an interval $[q_{x_1}, q_{x_2}] \times (-\infty, \infty)$, which is the horizontal part of the query window W_q, by stacking adjacent vertical blocks together, finishing with vertical blocks that contain values q_{x_1} and q_{x_2}.

We use the approach we used in range queries, as in Figure 3.31. Our query interval $I_{q_x} = [q_{x_1}, q_{x_2}]$ will obviously be completely contained in the root node interval $I_{q_x} \subseteq I(root)$. We continue descending through the segment tree until we reach a node n_{split} whose children do not contain the interval I_{q_x} fully, or

$$\exists n_{split} \in \tau : q_{x_1} \in I(leftChild(n_{split})), q_{x_2} \in I(rightChild(n_{split})) \tag{3.65}$$

This means that the interval of the left child of the splitting node $I(lc(n_{split}))$ is ending within the query interval I_{q_x} and the interval of the right child of the splitting node $I(rc(n_{split}))$ is beginning within the query interval I_{q_x}. If we have I_{q_x} that is contained within an elementary interval of the segment tree, then we reach a leaf node before finding the splitting node. If a splitting node is found, we proceed to the left to find an elementary interval containing q_{x_1} and

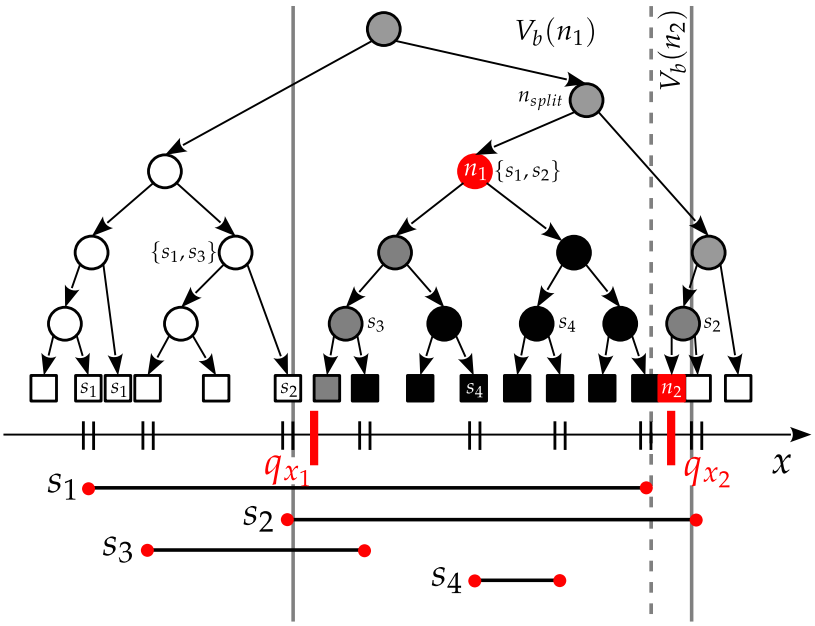


Figure 3.49: Querying the interval $I_q = [q_{x_1}, q_{x_2}]$ in the segment tree.

to the right to find an elementary interval containing q_{x_2}. Similar to performing a range query, all subtrees between the left and right vertical paths can be pruned. In Figure 3.49 we see left and right vertical paths marked as gray nodes, and all pruned subtrees as black nodes. If we denote $\mathcal{V}_L$ as the left vertical search path, $\mathcal{V}_R$ as the right vertical search path, and P_T as the set of pruned subtrees, then we can form the query result as follows

$$S(I_{q_x}) = \bigcup_{n \in \mathcal{V}_L \cup \mathcal{V}_R \cup P_T} S(n) \tag{3.66}$$

which is the set of line segments contained in nodes of both search paths and all pruned subtrees.

All nodes of the segment tree whose subtrees are fully included in the query interval I_{q_x} are the nodes that are forming vertical blocks of the query result. In Figure 3.49 we see these nodes marked as red. If we mark all these nodes as N_T, we can form an unique vertical block as

$$V_b(I_{q_x}) = \bigcup_{n \in N_T} V_b(n) \tag{3.67}$$

and the list of segments

$$S' = \bigcap_{n \in N_T} S(n) \tag{3.68}$$

gives all line segments that fully intersect the query interval I_{q_x}. All other line segments that are in the result of the query are partially intersecting the query interval I_{q_x} and are contained in the nodes of the subtrees from N_T.

What about the vertical part of the query window W_q? We already defined that for a node n_i in the segment tree, the canonical set of line segments $S(n_i) \neq \emptyset$ fully intersects the vertical block of the

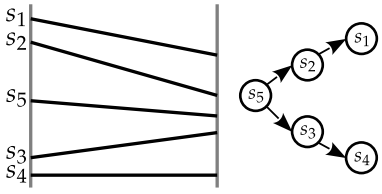


Figure 3.50: An example of a binary tree that organizes vertical block intersecting line segments to facilitate vertical queries.

node $V_b(n_i)$. Since we know line segments that need to be vertically queried, we need to organize them to facilitate the vertical query. If we generally **disallow crossing line segments**, we can organize line segments $S(n_i)$ in a simple binary tree, seen in Figure 3.50, which is then associated to the node n_i of the segment tree, denoted as $\tau(n_i)$. To find all line segments from $S(n_i)$ that are within the vertical query interval $I_{q_y} = [q_{y_1}, q_{y_2}]$, we use the same approach as for range queries: find the splitting node and search in two vertical paths, to find the subset of line segments $S_{W_q}(n_i) \subseteq S(n_i)$.

In Figure 3.51 we can see how a line segment can be outside of the query window W_q, yet performing the vertical query against vertical blocks boundaries without taking in consideration intersection of the line segment with q_{x_1} and q_{x_2} could produce a *false positive*. This means that vertical queries must be performed only within the horizontal interval $I_{q_x} = [q_{x_1}, q_{x_2}]$ of the query window W_q. The leftmost and rightmost vertical blocks in $V_b(I_{q_x})$ can be partially outside the horizontal interval I_{q_x}, which needs to be taken in account.

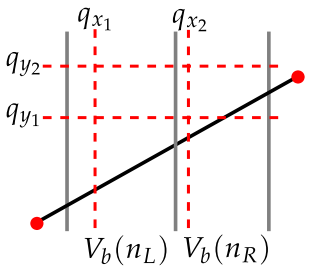


Figure 3.51: Vertical query correction for the leftmost and rightmost vertical blocks.

Algorithm 3.14: Segment tree window query algorithm.

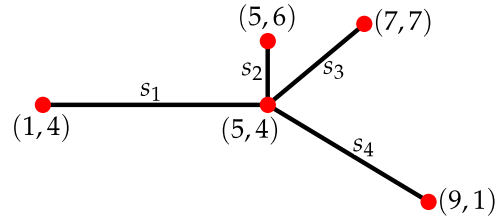
```
function SegmentTreeQuery(τ, W_q = [q_x1, q_x2] × [q_y1, q_y2])
    rv ← ∅
    n_split ← FindSplittingNode(τ, [q_x1, q_x2])
    if n_split is leaf then
        if S(n_split) ≠ ∅ then
            add VerticalQuery(τ(n_split), n_split, W_q) to rv
    else
        n ← leftChild(n_split)
        while n is not leaf do
            add VerticalQuery(τ(n), n, W_q) to rv
            lc ← leftChild(n), rc ← rightChild(n)
            if q_x1 ∈ I(lc) then
                add ReportSubtree(rc, W_q) to rv
                n ← lc
            else
                n ← rc
        add VerticalQuery(τ(n), n, W_q) to rv
        n ← rightChild(n_split)
        while n is not leaf do
            add VerticalQuery(τ(n), n, W_q) to rv
            lc ← leftChild(n), rc ← rightChild(n)
            if q_x2 ∈ I(rc) then
                add ReportSubtree(lc, W_q) to rv
                n ← rc
            else
                n ← lc
        add VerticalQuery(τ(n), n, W_q) to rv
    return rv
```

```
function ReportSubtree(n, W_q = [q_{x_1}, q_{x_2}] × [q_{y_1}, q_{y_2}])
    rv ← ∅
    if n is not nil then
        add VerticalQuery(τ(n), n, W_q) to rv
        add ReportSubtree(leftChild(n), W_q) to rv
        add ReportSubtree(rightChild(n), W_q) to rv
    return rv
```

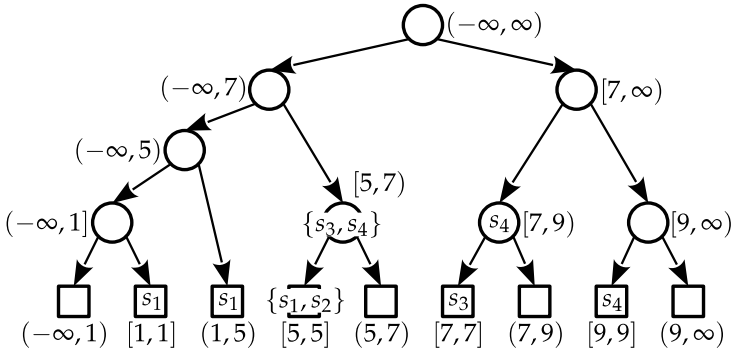
```
function FindSplittingNode(τ, I_{q_x} = [q_{x_1}, q_{x_2}])
    n ← root(τ)
    while n is not leaf do
        lc ← leftChild(n), rc ← rightChild(n)
        if q_{x_1} ∈ I(lc) and q_{x_2} ∈ I(rc) then
            return n
        else
            if I_{q_x} ⊆ I(lc) then
                n ← lc
            else
                n ← rc
    return n
```

Python code for the segment tree is extensive and can be found in the lecture book python codes on github.

Exercise 3.3.1. Create a segment tree for the following set of line segments.



Solution:



Interval and segment trees takeaway

In the previous section we extensively described several selected ad-

vanced geometrical data structures in form of trees that support geometrical querying. We intentionally omitted some structures and related algorithms, such as *kd-trees*, *R-trees*, *M-trees*, and more advanced structures. The idea of this section is not to give all solutions used in the computational geometry, but to select those that generally fit into this lecture book.

3.4 Geometric algorithms python codes

Listing 3.9: Priority tree query python code.

```
class PriorityTree:
  rootNode=None
  def __init__(self,Pp):
    Pp=list(sorted(Pp,key=lambda p:p[0]))
    self.rootNode=self._create(Pp)
  def _create(self,Pp):
    pmin=Pp[0]
    n=Node(pmin)
    del Pp[0]
    if len(Pp)>0:
      ymed=median(map(lambda it:it[1],Pp))
      n.y=ymed
      Pl=list(filter(lambda it:it[1]<=ymed,Pp))
      if len(Pl)>0: n.leftChild=self._create(Pl)
      Pr=list(filter(lambda it:it[1]>ymed,Pp))
      if len(Pr)>0: n.rightChild=self._create(Pr)
    return n
  def _findSplittingNode(self,qy1,qy2):
    n,rn=self.rootNode,[]
    while n is not None and (qy1>n.y or qy2<n.y):
      rn.append(n)
      if qy2<n.y:
        if n.leftChild is not None: n=n.leftChild
        else: return None,rn
      else:
        if n.rightChild is not None: n=n.rightChild
        else: return None,rn
    return n,rn
  def _queryPrioritySubtree(self,n,qx2):
    res=[]
    if n is not None and n.p[0]<=qx2:
      res.append(n.p)
      res=res+self._queryPrioritySubtree(n.leftChild,qx2)
      res=res+self._queryPrioritySubtree(n.rightChild,qx2)
    return res
  def Query(self,qx2,qy1,qy2):
    rv=[]
    nsplit,rn=self._findSplittingNode(qy1,qy2)
    for n in rn:
      if n.p[0]<=qx2 and isInRange(qy1,n.p[1],qy2): rv.append(n.p)
    if nsplit is None: return rv
    if nsplit.leftChild is None and nsplit.rightChild is None:
      rv=rv+self._queryPrioritySubtree(nsplit,qx2)
    else:
      if nsplit.p[0]<=qx2 and isInRange(qy1,nsplit.p[1],qy2): rv.append(nsplit.p)
      n=nsplit.leftChild
      while n is not None and n.p[0]<=qx2:
        if isInRange(qy1,n.p[1],qy2): rv.append(n.p)
        if isInRange(qy1,n.y,qy2):
          rv=rv+self._queryPrioritySubtree(n.rightChild,qx2)
          n=n.leftChild
        else: n=n.rightChild
      n=nsplit.rightChild
      while n is not None and n.p[0]<=qx2:
        if isInRange(qy1,n.p[1],qy2): rv.append(n.p)
        if isInRange(qy1,n.y,qy2):
          rv=rv+self._queryPrioritySubtree(n.leftChild,qx2)
          n=n.rightChild
        else: n=n.leftChild
    return rv
```

4 Linear programming

Linear programming is a very important algorithmic tool for both applications and theoretical considerations. Its deceptively simple structure can model many applications in supply chain management, resource allocation problems, etc. On the theoretical side, it is the basis of the field of combinatorial optimization and advanced algorithm design and analysis. Linear programming will be utilized in the chapters on approximation algorithms and network flows. Also, its geometrical interpretation on polyhedra (polytopes) and convex sets are related to the computational geometry chapter.

Please, be warned that linear programming and polyhedral geometry were researched in several separate research fields (such as operations research, mathematical optimization, economics, etc.) and that basic terms and definitions might slightly differ!

4.1 Linear programs

Definition 4.1. Mathematical program is constrained optimization problem of the form:

$$\min_{x \in S} f(x)$$

where $S \subseteq \mathbb{R}^n$.

Definition 4.2. Linear program is any mathematical program of the form:

$$\begin{array}{ll} \min & c^T x \\ \text{subject to} & A\,x \leq b \\ & D\,x = e \end{array}$$

where $x \in \mathbb{R}^n$ is the vector of **decision variables** and $k, m, n \in \mathbb{N}_0, c \in \mathbb{R}^n, b \in \mathbb{R}^m, e \in \mathbb{R}^k, A \in \mathbb{R}^{m \times n}, D \in \mathbb{R}^{k \times n}$ are **parameters of the problem**.

In the upper definition, k and m are the number of equality and inequality constraints, respectively. n is the number of decision variables. You will notice that all the constraints include equality, that is, the boundary is included in the feasible area. Hence, we have closed the feasible set. If we used even one strict inequality, we would possibly be in the situation where we would not be able to find the exact optimum, but could only asymptotically approach it.

Definition 4.3. Linear program is in **canonical form** when it fits the following form:

$$\begin{array}{ll} \min & c^T x \\ \text{subject to} & A\,x \leq b \\ & x \geq 0 \end{array}$$

The canonical form is the most intuitive form for interpretation. We shall use it in the graphical method for developing intuition.

Definition 4.4. Linear program is in **standard form** when it fits the following form:

$$\begin{array}{ll} \min & c^T x \\ \text{subject to} & A\,x = b \\ & x \geq 0 \end{array}$$

where the right-hand side of equalities is non-negative (i.e. $b \geq 0$).

Standard form is the most convenient form for computations in simplex (subsection 4.3).

Reductions to standard form

As mentioned all linear programs can be reduced to standard form. We shall cover all the cases that need reductions:

1. *Elimination of negative right-hand-side b*: Multiply constraint by -1.
2. *Elimination of maximization*: Transform into minimization using the following equivalence:

$$\max f(x) \iff \min -f(x)$$

3. *Elimination of free variable*: If decision variable x_j is of unrestricted domain in $\mathbb{R}$, we introduce new two decision variables $x_j^+, x_j^- \geq 0$, and introduce the following substitution $x_j := x_j^+ - x_j^-$.
4. *Elimination of unaligned restricted domain variable*: If j-th decision variable is $x_j \leq z$, we:
   (a) multiply the constraint by -1 to put it into form $-x_j \geq -z$,
   (b) move everything to the left side: $-x_j + z \geq 0$,
   (c) introduce new variable with substitution $x_j^{'} := -x_j + z$,
   (d) reorder items and substitute into the linear program

   If the $x_j \geq z \neq 0$, then use the similar procedure as above, starting from step b.
5. *Elimination of inequality constraints*: Inequality constraints are translated to equality constraints using the introduction of non-negative variables.
   - Each constraint of the form $\sum_{j=1}^{n} a_{i,j}x_j \leq b_i$ is substituted by the following: $\sum_{j=1}^{n} a_{i,j}x_j + s_i = b_i, s_i \geq 0$, where s_i is the *slack* variable.
   - Each constraint of the form $\sum_{j=1}^{n} a_{i,j}x_j \geq b_i$ is substituted by the following: $\sum_{j=1}^{n} a_{i,j}x_j - s_i = b_i, s_i \geq 0$, where s_i is the *surplus* variable.

4.2 Graphical method

Linear programs with $dim(x) \leq 3$ can be represented graphically and we shall use that property to build the intuition for solving these problems. First, we shall present a graphical method. After that, we shall analyze the graphical case to extract as many facts and properties that are interesting for building algebraic approaches that can solve problems of higher dimensionality.

Solve the following linear program:

$$\begin{aligned} \max \quad & x_1 + 5x_2 \\ \text{subject to} \quad & \begin{bmatrix} 5 & 6 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} 30 \\ 12 \end{bmatrix} \\ & x \geq 0 \end{aligned} \tag{4.1}$$

Obviously, the problem is, except for the maximization, in the canonical form, which is suitable for intuitive analysis. Each inequality constraint "selects" all the points in the half-space, and that the final set of feasible points is the conjunction/intersection of all the sets selected by individual constraints.

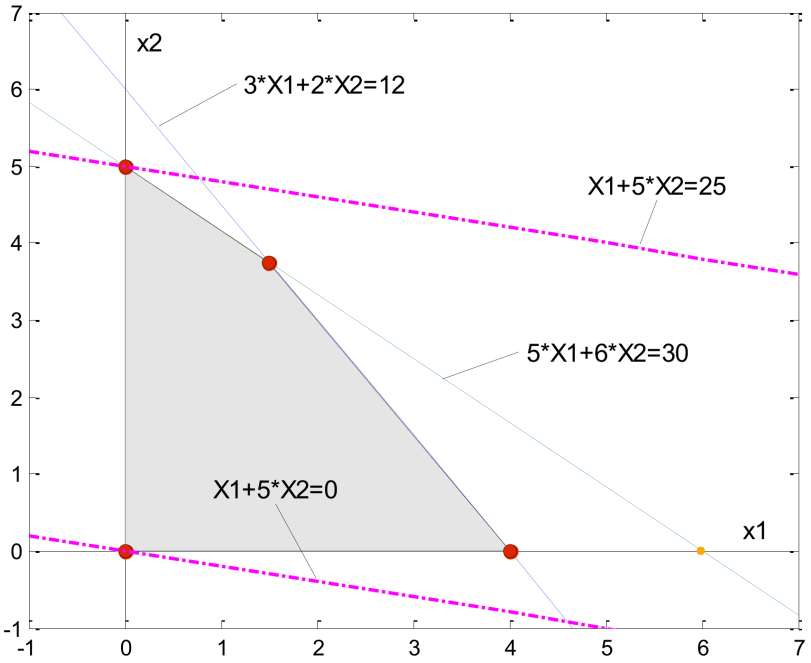


Figure 4.1: Graphical representation and solution to the example (4.1)

If we plot the constraints onto the 2D Euclidean space, we get the situation in Figure 4.1. The gray area (including the edges) is the set of feasible points. As we can see, the feasible set is a **convex polytope**, a body with linear edges. In 2D, the objective function defines a set of parallel lines with different objective values. Intersections of lines and polytope are linear segments.

In a 2D case, a solution to the LP is a point from an objective line such that:

- that line contains at least one point from feasible polytope

- there is no line with "better" objective value that still contains points from the feasible polytope

We can see two objective isolines (in purple) on Figure 4.1. The lower one is minimal and the upper one is maximal. We can easily see that $x = [0, 5]$ is the solution to our LP, with the objective value of 25. Had the program instead been minimization, the optimum would be in $x = [0, 0]$ with an objective value of 0.

The graphical method (applicable only in 2D, at best in 3D):

1. Draw feasible polytope P. In P is empty, stop - there is no solution to LP.
2. Draw initial isoline that intersects P
3. "Slide" the isoline in parallel in the direction of gradient if maximizing, or opposite if minimizing.
4. Stop when you have the last isoline that still intersects P (in fact, it touches it on some face or in vertice. Let E be that isoline, with objective value O.
5. Pick some $x \in E \cap P$ as the solution.

From little observation on the LP example, it is evident that no matter what objective function and the optimization direction we would use, "the best" isoline would always touch some vertex of P and sometimes even a face (which contains two vertices in 2D). This is the observation that will be instrumental for simplex algorithm.

More generally, the objective function, linear itself, defines a set of parallel hyperplanes with different values. The intersection of polytope P and objective hyperplane O is a polytope itself. The solution to LP is a point in an objective hyperplane that:

- has $P \cap O \neq \emptyset$
- there exists no other objective hyperplane that satisfies the previous condition and has a "better" objective value.

Definition 4.5. Convex polytope in d-dimensional space is set of vectors (points)

$$\{x \in \mathbb{R}^d | Ax \leq b\}$$

From now on, in this section when we talk of polytope we will assume convex polytope unless stated otherwise. Below are some examples of convex polytopes for better intuition. A bounded polytope is a polytope that can fit into some sphere of finite diameter.

Definition 4.6. Active constraint in some point x is any constraint that is satisfied with equality.

Definition 4.7. Polytope vertex. In n-dimensional space, **polytope vertex** is any intersection of n active linearly independent constraints, while all other constraint must be satisfied.

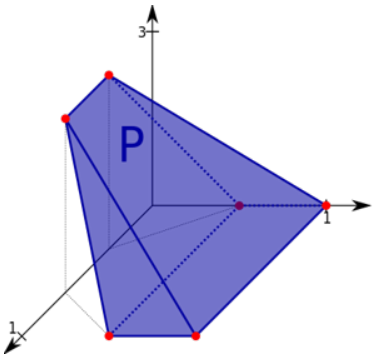


Figure 4.2: Bounded polytope

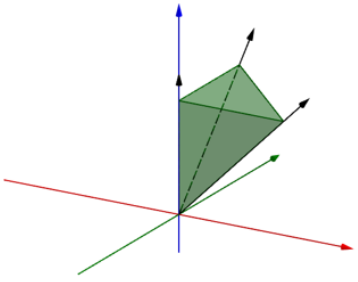


Figure 4.3: Unbounded polytope

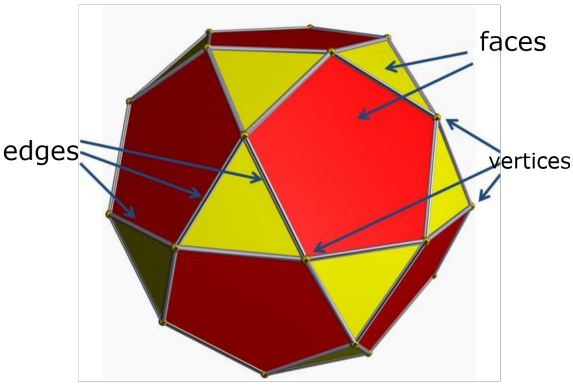


Figure 4.4: Polyhedron - 3D polytope with different subspaces

Notice the difference between the polytope vertex and just an intersection of n active constraints. The set containing all the latter is a superset of polytope vertices. Check Fig.4.1 where red dots mark polytope vertices. The orange dot marks the intersection of n active constraints, but this is obviously not a polytope vertex. The difference is that red dots, in addition to having at least n active constraints, also keep the rest of the constraints satisfied. The orange dot obviously breaks the constraint $3x_1 + 2x_2 \leq 12$.

From the geometric observations on 2D and 3D polytopes, we list all the important regularities:

- vertex is any feasible intersection of n active linearly independent constraints
- neighboring vertices (connected by an edge) differ only in one active constraint
- solution to LP is in one of the vertices of the feasible polytope

So far, we have defined geometric concepts, and have connected geometry to algebra by definition of active constraints which led to the definition of a vertex in algebraic terms. Let us add several other definitions of algebraic nature.

Definition 4.8 (Convex set). Set Θ is convex set if it contains all the points on linear segments between any two points from Θ. That is,
$\forall x, y \in \Theta, \forall \alpha \in (0,1) : \alpha x + (1-\alpha)y \in \Theta$

Obviously, convex polytopes are convex sets.

Definition 4.9. Extreme of convex set Θ is any point $x \in \Theta$ which is not on linear segment connecting some other points from Θ. That is,
$(\nexists x_1, x_2 \in \Theta \setminus \{x\})(\nexists \alpha \in (0,1))x = \alpha x_1 + (1-\alpha)x_2$

Extremes in polytopes are vertices!

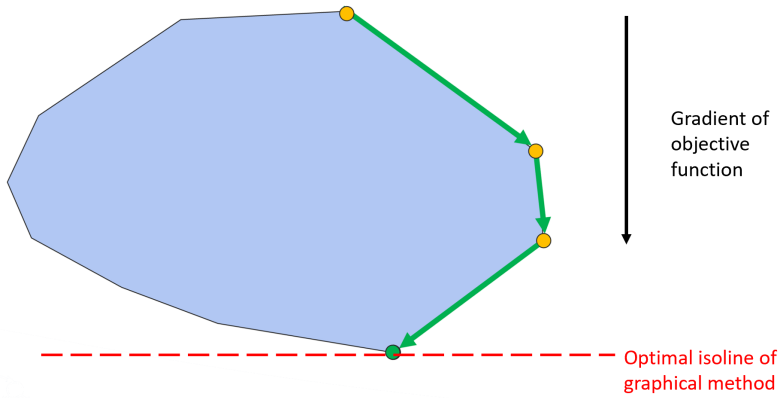


Figure 4.5: Simplex method compared to the graphical method

4.3 *Simplex*

Simplex method is an **algebraic method** that is, unlike the graphical, capable of solving LP of any dimensionality. In Fig.4.5 we have contrasted the ways of finding optimum by graphical method and by simplex method on 2D example. Graphical method slides isoline in the direction (or opposite) of the gradient of the objective function. Simplex, on the other hand, iteratively goes across the boundary of the polytope, jumping between the vertices, whilst always improving the objective value (following the trend of gradient, though not directly). The walk along the boundary between the neighboring vertices will include exchanging one active constraint for another to make a jump. This will result in fast linear algebra operations.

We assume you know Gauss-Jordan elimination from linear algebra, for solving systems of linear equations. Simplex algorithm is an extension of Gauss-Jordan elimination from linear equality to inequality constraints and to working with rectangular matrices.

In short, simplex method works as follows:

1. translate LP into **standard form**
2. we shall have rectangular matrices A, obviously non-invertible.
3. We shall select invertible square submatrices of A which will correspond to vertices in feasible polytope
4. We do a greedy local search over vertices which is monotonically improving until finding optimum which we will know how to recognize.

For the rest of the work with simplex, we shall assume LP is in standard form and that A is full rank. Simplex algorithm needs to start from some polytope vertex, which must be chosen for it by some independent initialization procedure.

"Naïve" simplex

"Naïve" simplex assumes that the initial polytope vertex is immediately given. If this is not the case, we must use a more sophisticated method such as **two-phase simplex** or **generalized simplex method** to find such vertex.

Let us assume that A contains an identity submatrix of the same rank as A. This is, for example, the case when the feasible polytope of problem in canonical form contains the origin. In this case, simplex is initialized from the origin.

We are within n-dimensional space (n is the number of columns in A). Polytope vertices are defined by n active constraints (def.4.7). Equality constraints form m active constraints.

This means that all the polytope vertices can be found by choosing arbitrary $n - m$ active constraints out of n non-negativity constraints. Making non-negativity constraints active means fixing values of some variables to 0. The variables that are fixed to 0 are called **non-basic**. Substituting non-basic variables back into the problem reduces the problem of solving m linear equations for m unknowns - which is simple (Gauss-Jordan elimination).

In effect, set of all variables is **partitioned** into **non-basic** (fixed to 0) and **basic** (calculated from the reduced linear equations) variables.

Paraphrased to simplify, by choosing $(n - m)$ variables out of n in standard form LP and setting them to 0 selects intersections of n active constraints. If those active constraints are linearly independent, we can calculate the values of the rest of the m variables. If the values of those m variables are non-negative, we have landed onto a polytope vertex. The following definition is a formalization of the previous discussion.

Definition 4.10. Basic solution of system $Ax = b, x \geq 0$ is a vector $x = [x_B^T, 0]$ where $x_B = B^{-1}b$ and $B \in \mathbb{R}^{m \times m}$ is selected non-singular basis matrix in A.

Basic solution is the *potential* description of polytope vertex (see def.4.7) for standard form LPs. Notice that only feasible basic solutions correspond to polytope vertices. Feasible basic solution is any basic solution that has $x_B = B^{-1}b \geq 0$. Obviously, the detection of feasibility is easy! Infeasible basic solutions correspond to the points that are outside of the polytope in question.

We shall organize the LP from standard form in the following way:

c^T	$-obj.value$
A	b

Furthermore, we shall partition matrix A into basic and nonbasic submatrices $A = [B, N]$.

c_B^T	c_N^T	$-obj.value$
B	N	b

The table above already defines some basic solution. However, we cannot read off its coordinates, nor its objective value. Let us multiply the constraint block with B^{-1} from the left. And additionally,

eliminate cost coefficients above basis vectors. Then, the following definition sets the look of simplex tableau at each iteration.

Definition 4.11. Simplex tableau. With the partition of decision variables into basic and nonbasic, the following is the look of simplex tableau (in practice, columns may be permuted)

0^T	$c_N^T - c_B^T B^{-1} N$	$-c^T B^{-1} b$
I	$B^{-1} N$	$B^{-1} b$

Different blocks in simplex tableau have different meaning, which is shown in the overlaid table below:

basic reduced costs	nonbasic reduced costs	-obj.value
basis	edges to neighbors	values of basic variables

Basic variables by construction have 0 **reduced costs**. **Nonbasic variables have reduced costs** of $c_N^T - c_B^T B^{-1} N$ at the current basic solution. **Objective value** is given in the top right corner of the table, but it is multiplied with -1 (artifact of the procedure). **Basis** consists of the column vectors belonging to basic variables. In simplex tableau, the basis block will be column vectors of identity matrix. The reason is that we are doing Gauss-Jordan elimination (pivot operations) to iteratively obtain such form to be able to read off the coordinates of the basic solution, so we can check if it is feasible (i.e. with all components non-negative). Edges to neighbors in polytope are expressed in the currently used basis (i.e. multiplied with B^{-1} from the left). Column vectors express coefficients of the direction towards other neighboring basic solutions (feasible and infeasible). Finally, values of basic variables are given in the rightmost column, below the cell that contains objective value. Values are assigned to basic variables by locations of unit vectors e_i in the basis - in an identical way as it is done in solving systems of linear equations by Gauss-Jordan elimination. Unit vector e_i has 0 in all components, except for i^{th} element which is 1. For example, if the column j contains e_i vector, then the value of x_j is read from the i^{th} row.

Simplex algorithm over simplex tableau works as follows:

1. Optimality? If all nonbasic variables have a nonnegative reduced cost, the solution is optimal. STOP
2. Attempt transition to better polytope vertex
   (a) Select entering nonbasic var. Out of nonbasic vars with negative reduced cost, select some x_j. Let $u = B^{-1} A_j$ be its column in the tableau.
   (b) Select exiting basic variable. For all basic variables x_i for which $B^{-1} A_{i,j} > 0$ select the one with minimal ratio x_i / u_i. If there is no basic variables that satisfy condition, the problem is infeasible - STOP.
   (c) Gauss-Jordan elimination around the pivot element. Do row operations in a way to get a unit vector in the column of entering nonbasic variable with 1 in the place of the pivot.

3. Return to step 1.

4.4 Two-phase simplex

If the initial tableau contains no identity submatrix (which makes the initial polytope vertex evident), we shall construct auxiliary LP' which:

- has the origin on the LP' polytope
- can find polytope vertex for our LP of interest if such exists

We shall solve two linear programs, which are called two phases:

1. **PHASE 1** - We construct and solve auxiliary LP' using "naive" simplex to find initialization for the second phase if such exists.
2. **PHASE 2** - We initialize the original LP from the previous phase. Solve it using "naive" simplex to find the final solution. If we could not find solve LP', the second phase is not done because the original LP is infeasible.

Obviously, we use naive simplex twice, so the only novelties in two-phase simplex are:

- the construction of auxiliary problem LP',
- initialization of the original LP.

Constructing auxiliary problem

Assume we have problem in standard form:

$$\begin{array}{ll} \min & c^T x \\ \text{subject to} & A\,x = b \\ & x \geq 0 \end{array}$$

We must change the constraints by adding an artificial variable for each constraint, and we must change the objective function to the sum of artificial variables.

The auxiliary problem is then:

$$\begin{array}{lll} \min & \mathbb{1}^T x_{n+1:n+m} & \\ \text{subject to} & [A|I_m] & [x_{1:n}|x_{n+1:n+m}]^T = b \\ & & x_{1:n+m} \geq 0 \end{array}$$

The auxiliary problem can be trivially initialized and solved using naive simplex as covered in the previous section.

The outcome of the first phase is one of:

1. objective value is not 0. The original problem is infeasible.
2. objective value is 0.
   (a) All artificial variables are non-basic. The simplex tableau is ready for the second phase.

(b) Some artificial variables are basic, and the auxiliary problem has degeneration. Repeat simplex iterations to remove all the artificial variables from the basis.

Initialization of second phase

We assume we have simplex tableau from the first phase with optimal value 0 and all artificial variables non-basic.

We initialize the second phase in the following way:

- Delete columns belonging to artificial variables
- Substitute the original objective function into the tableau
- Do Gauss-Jordan elimination on the row of the objective function to get reduction factors (and hence valid simplex tableau)

From this point, you can proceed with "naive" simplex to get the solution to the original LP.

4.5 *Duality*

Duality is an outgrowth of the Lagrange multiplier method used in calculus to solve minimization problems subject to equality constraints. All the derivations of duals follow the path:

1. Introduce the constraints into the objective using Lagrange multipliers, with suitable constraints on multipliers. This composite function is called Lagrangian.
2. Derive the dual linear program that optimizes only over the multiplier variables.

Duality is a very useful concept, with applications in:

- sensitivity analysis
- game theory - e.g. application in minimax theorem in zero-sum games [1]
- basis for constructing a family of algorithms called primal-dual, etc.

[1] J. v. Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, December 1928. ISSN 1432-1807. DOI: 10.1007/BF01448847. URL https://doi.org/10.1007/BF01448847

You have already witnessed primal-dual relationship! Min-cut problem (i.e. finding the minimal cut in a network) is the dual of the max flow problem. You can think of these problems as a game between two players: A and B, where A is trying to maximize the flow of information in the communication network and B is the saboteur who is trying to cut the network at the lowest cost to themselves to stop the flow completely.

For example, assume we have linear program in canonical form:

$$\begin{array}{ll} \min & c^T x \\ \text{subject to} & A\,x \leq b \\ & x \geq 0 \end{array}$$

then its dual is (using decision variables y):

$$\begin{array}{ll} \max & b^T\, y \\ \text{subject to} & A^T y \geq c \\ & y \geq 0 \end{array}$$

Derivation of the dual of standard form LP

Let us derive the dual from the primal in the standard form. First, put everything on the left side of constraints.

$$\begin{array}{ll} \min & c^T x \\ \text{subject to} & A\, x - b = 0 \\ & x \geq 0 \end{array}$$

Put all constraints (other than non-negativity) into the objective function to get Lagrangian $L(x, y)$:

$$L(x, y) = c^T x + y^T (b - Ax)$$

Let us look at the Lagrange dual function:

$$\begin{aligned} g(y) &= min_{x \geq 0} L(x, y) && \text{(4.2a)} \\ &= min_{x \geq 0} [c^T x + y^T (b - Ax)] && \text{(4.2b)} \\ &= y^T b + min_{x \geq 0} (c^T - y^T A) x && \text{(4.2c)} \end{aligned}$$

By carefully looking, we see that $g(y)$ is the lower bound on optimal value of primal:

$$g(y) = min_{x \geq 0} [c^T x + y^T (b - Ax)] \leq c^T x^* + y^T (b - Ax^*) = c^T x^*$$

Also, $g(y)$ achieves trivial value of $-\infty$ for any y such that $c^T - y^T A < 0$.

Let us search for the greatest $g(y)$ to find the best bound on objective value of primal. In that case, we can look only at nontrivial cases, when $c - y^T A \geq 0$ which makes the second term in 4.2c equal to 0. Hence, we get that the dual of LP in standard form is:

$$\begin{array}{ll} \max & b^T\, y \\ \text{subject to} & A^T y \leq c \end{array}$$

The derivation of the dual of canonical form LP follows a similar route. The only difference is the additional preceding step of transforming inequality constraints to equality constraints (with exception of non-negativity constraints on variables).

Relationship between primal and dual

There is a profound relationship between the primal and dual. For example, there is a cyclical relationship between the two, as claimed in the following theorem.

Theorem 4.1. (Cyclic property of duality) If we make the dual of the dual LP, we obtain a problem equivalent to the original primal LP.

In the case of minimizing primal, the value of the dual is the lower bound on the solution of primal (due to the constraints) and we look for the greatest lower bound.

Theorem 4.2. (Weak duality) If x is a feasible solution to the primal problem and y is the feasible solution to the dual problem, then $b^T y \leq c^T x$.

Obviously, if the optimal cost in the primal is $-\infty$, the dual is infeasible. Also, if the optimal cost in dual is ∞, the primal is infeasible. Furthermore, there is an even stronger result for duals of linear programs (the appropriate generalization holds for convex programs):

Theorem 4.3. (Strong duality) If the linear program has an optimal solution, the same holds for its dual, and the optimal costs are identical.

Additionally, there is one more strong result that ties the primal and dual.

Theorem 4.4. (Complementary slackness) Let x and y be feasible solutions for primal and dual problem. The vectors x and y are optimal for their respective problems iff the following holds:

$$y_i(A_{i,:}x - b_i) = 0, \forall i$$

and

$$(c_j - y^T A_{:,j})x_j = 0, \forall j$$

.

Complementary slackness, paraphrased, states that for optimal solutions x and y, the only vector components that are non-zero are:

- dual components for active constraints in primal
- primal components with zero reduced cost (i.e. basic variables).

Dual simplex method

After the introduction of duality theory, we can construct a new method that can solve general LPs (as a substitute for the two-phase simplex method).

We can think of primal simplex method as the one that keeps primal feasibility and works towards dual feasibility until they are both achieved at optimum (strong duality). Any method that works in this way is primal algorithm.

An alternative idea would be to proceed in the opposite direction; start with a dual feasible solution and work towards primal feasibility, which would make for the dual algorithm.

When we have primal LP in standard form within simplex tableau, we also have immediate access to the data from its dual on which we

can act. Hence, we get the dual simplex algorithm which enables us to deal with the infeasibility of the current primal basic solution (i.e. that has negative components) - something that primal simplex cannot achieve.

Iteration of dual simplex algorithm:

1. We assume that all reduced costs in simplex tableau are non-negative.

2. If the right-hand side is non-negative we are at the optimal solution - STOP. Otherwise, we are in an infeasible basic solution (i.e. not on the edge of primal polytope).

3. Pick some tableau row i with a negative value of a basic variable. If all other elements v_j in the row are positive, the primal is infeasible (and the optimal dual cost is ∞) - STOP.

4. For each negative element in the row i, pick the column with minimal ratio $\frac{r_j}{v_j}$. The pivot is element (i, j).

5. Do the pivoting operation like in primal simplex.

The above dual simplex method assumes non-negative reduced costs. If this is not the case, you can alternate between the dual and primal simplex iterations until reaching a conclusive solution. Primal simplex iterations deal with negative reduced costs, while dual simplex iterations deal with negative components in the basic solution (RHS). This method that uses both types of iterations is **generalized simplex method**.

Dual simplex method is an important tool in combinatorial optimization as it can lead to fast resolving of the problems after adding additional constraints, which is how cut generation and branch&bound methods work.

Side-issues

Up so far, we made, sometimes implicitly, convenient assumptions.

- We assumed that matrix A in standard form is full rank. If that is not the case, we can detect the linearly dependent rows and remove them without loss of generality.

- We ignored numerical issues.

- We did not deal with situations called degeneracy, i.e. when some basic variables have value 0. In such cases, the simplex methods we presented can get into the infinite loop (cycling) and it does not find the solution. The specialized solutions are to use Bland's lexicographical rule for selecting pivots (the first method that resolved the issue). Mostly, commercial solvers use perturbation methods that perturb the alignment between some constraints - which is the geometrical interpretation of degeneracy.

- The presented method is not actually used in commercial software, but instead a revised simplex method which keeps only the track of matrix B^{-1} and usually LU-decomposition is used for increasing efficiency.

- Simplex method is an exponential algorithm in the worst case, as demonstrated on Klee-Minty cube [2]. Klee-Minty cube is the unit hypercube with corners perturbed in such an intricate way to force simplex to visit all the vertices on the way to reaching the optimum. The number of vertices in n-dimensional unit hypercube is 2^n. Such traversal is called **spanning path**. However, in practice simplex achieves polynomial complexity and its numeric precision is first-class. The explanation of such behavior is offered by using the smoothed analysis that incorporates the perturbation around the worst cases that shows that the probability of working with the worst-case problem is vanishing [3]. Smoothed analysis is a very important tool for research in the analysis of algorithms.

[2] V. Klee and G. Minty. How good is the simplex algorithm? In *Inequalities III - Proceedings of the Third Symposium on Inequalities*, pages 159–175, University of California, Los Angeles, USA, 1972. Academic Press

[3] Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM*, 51(3):385–463, May 2004. ISSN 0004-5411. DOI: 10.1145/990308.990310. URL https://doi.org/10.1145/990308.990310; and Roman Vershynin. Beyond Hirsch Conjecture: Walks on Random Polytopes and Smoothed Complexity of the Simplex Method. *SIAM Journal on Computing*, 39(2):646–678, January 2009. ISSN 0097-5397. DOI: 10.1137/070683386. URL https://locus.siam.org/doi/abs/10.1137/070683386. Publisher: Society for Industrial and Applied Mathematics

4.6 Ellipsoid method*

The first proof that linear programs are in P class was given by Leonid Kantorovich using the ellipsoid method. It is useful only for proof of polynomiality and it is not used in practice for solving LPs as it is impractically slow.

In short, and simplified form, ellipsoid algorithm generates a sequence of ellipsoids E_t with centers x_t such that LP polytope is contained in E_t. The method can be used to prove feasibility or find the optimal solution.

What follows is a crude description of ellipsoid method for optimization.

1. Consider the LP_{primal} and its dual LP_{dual}
2. Construct the new feasibility problem FP that uses all the constraints of primal and dual, with additional constraint (by strong duality) that $c^T x = b^T y$.
3. Construct a ball E_0 that contains the extreme points of polytope of FP' (FP perturbed by small value to ensure full-dimensionality)
4. Repeat until proving the infeasibility or feasibility of FP':
   (a) If we reached finite precalculated iteration t^* without resolution, STOP - the problem is infeasible.
   (b) Check if the center x_t of E_t, belongs to the FP'. If yes - STOP, we have found the optimum.
   (c) If x_t is not in FP', find violating constraint and use it to construct a new ellipsoid E_{t+1} that contains only the half of E_t that satisfies that constraint.

The complexity of ellipsoid method is $O(n^6 log(nU))$ where n is the number of decision variables and U is the lowest absolute upper bound on elements of A and b. While this is better than the worst-case for simplex family of methods, in practice simplex methods are much better performing.

4.7 *Interior point methods*

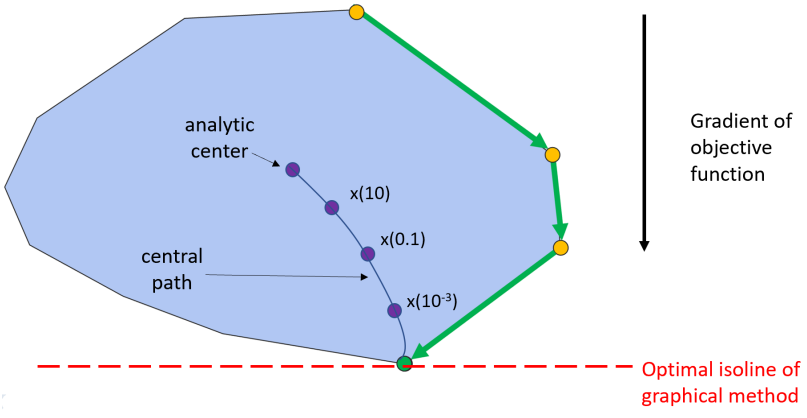


Figure 4.6: Interior point method compared to simplex and graphical method

Geometrically, simplex method solves the problem by moving through extreme points, on the boundary of the feasible area (green path with yellow points in Fig.4.6). In the mid-1980s new algorithms were devised that moved through the interior of the feasible space (blue path with violet points in Fig.4.6). They often outperform simplex method for large degenerate problems (such as in scheduling) and have a better potential for parallelization. Interior point methods, unlike simplex, can also be used for solving nonlinear constrained optimization problems.

Computationally, simplex and interior-point methods differ in the distribution of computational work. Simplex method involves many inexpensive iterations. On the other hand, interior point methods need a smaller number of computationally expensive iterations.

Here we shall cover path following algorithms, which are considered the most successful.

We shall need to define Karush-Kuhn-Tucker (KKT) conditions before proceeding. In our notation, for some function $\mathbf{g}(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^m$, $D\mathbf{g}(\mathbf{x})$ will be its Jacobian matrix.

Definition 4.12. (Karush-Kuhn-Tucker (KKT) conditions). Assume general optimization problem of the form:

$$\begin{gathered} \text{minimize } f(\mathbf{x}) \\ \text{subject to} \\ \mathbf{g}(\mathbf{x}) \leq 0 \\ \mathbf{h}(\mathbf{x}) = 0 \end{gathered}$$

with $f, \mathbf{g}, \mathbf{h}$ differentiable.

Then for some primal point x and dual point (ξ, λ) KKT conditions are:

- ***(Stationarity)***

  $\nabla f(x) + D\mathbf{g}(x)^\top \boldsymbol{\xi} + D\mathbf{h}(x)^\top \boldsymbol{\lambda} = \mathbf{0}$

- ***(Primal feasibility)***

  $\mathbf{g}(x) \leq \mathbf{0}$

  $\mathbf{h}(x) = \mathbf{0}$

- ***(Dual feasibility)***

  $\boldsymbol{\xi} \geq \mathbf{0}$

- ***(Complementary slackness)***

  $\boldsymbol{\xi}^\top \mathbf{g}(x) = 0$

KKT conditions are necessary (first-order) conditions for optimality in nonlinear programming. Under certain circumstances, KKT are also sufficient conditions. For example, KKT are both necessary and sufficient for optimality in convex optimization problems, which includes linear programming.

Theorem 4.5. Let x^* and (ξ^*, λ^*) be primal and dual optimal points with zero duality gap for our general optimization problem. Then these points satisfy KKT conditions.

Theorem 4.6. If f and $\mathbf{g}$ are convex and $\mathbf{h}$ is affine (i.e. the optimization problem is convex) and x, ξ, λ is a point that satisfies KKT conditions, then x and (ξ, λ) are primal and dual optimal with zero duality gap.

We want to solve linear program in the standard form:

$$\begin{array}{ll} \text{min} & c^T x \\ \text{subject to} & A\,x = b \\ & x \geq 0 \end{array}$$

and we also need its dual:

$$\begin{array}{ll} \text{max} & b^T \;\; y \\ \text{subject to} & A^T y + s = c \\ & s \geq 0 \end{array}$$

Inequality constraints are problematic for optimization, so we will convert problems to the ones with equality constraints only. Instead of non-negativity constraints, we shall introduce logarithmic barrier functions of the following form:

$$\begin{array}{ll} B^p_\mu(x) = c^T x - \mu \mathbb{1}^T \log(x) & \text{(for primal)} \\ B^d_\mu(y, s) = b^T y + \mu \mathbb{1}^T \log(s) & \text{(for dual).} \end{array}$$

B^p_μ is defined to be infinity if $x_j \leq 0$ for some j. Similarly B^d_μ is defined to be negative infinity if $s_j \leq 0$ for some j.

We will consider the family of **nonlinear programming problems** - barrier problems which are parameterized by $\mu > 0$. Barrier problems are created from the original LP in standard form by changing objective function by barrier function and removing inequality constraints.

Primal barrier problem is:

$$\begin{array}{ll} \min & c^T x - \mu \mathbb{1}^T \log(x) \\ \text{subject to} & A \quad x = b \end{array} \tag{4.3}$$

Dual barrier problem is:

$$\begin{array}{ll} \max & b^T y + \mu \mathbb{1}^T \log(s) \\ \text{subject to} & A^T y + s = c \end{array} \tag{4.4}$$

Assume that for all $\mu > 0$, barrier problem (4.3) has an optimal solution $x(\mu)$. This holds due to the strict convexity of barrier function (logarithmic terms). As we will vary μ, sequence of minimizers $x(\mu)$ forms a central path, as illustrated in Fig.4.6. The minimizer corresponding to the $\mu = \infty$ is called *analytic center*. The analytic center can be found by solving the following optimization problem:

$$\begin{array}{ll} \min & -\mathbb{1}^T \log(x) \\ \text{subject to} & Ax = b \end{array} \tag{4.5}$$

As $\mu \to 0$ we approach the optimum of the LP. Also, let $y(\mu), s(\mu)$ be optimal solutions to the dual barrier problem (4.4) for selected parameter $\mu > 0$.

Barrier problems are convex optimization problems, i.e. we are optimizing convex function over the convex feasible set (in this case, polytope). Hence, KKT conditions are necessary and sufficient and we can use them to construct an interior-point gradient-based algorithm.

KKT conditions for our barrier problems are:

$$\begin{aligned} Ax(\mu) &= b \\ x(\mu) &\geq 0 \\ A^T y(\mu) + s(\mu) &= c \\ s(\mu) &\geq 0 \\ X(\mu) S(\mu) \mathbb{1} &= \mathbb{1}\mu \end{aligned} \tag{4.6}$$

wherein $X(\mu) = diag(x(\mu))$ and $S(\mu) = diag(s(\mu))$ and $\mathbb{1}$ is vector of all components equal to 1. Also, let $\epsilon > 0$ be a desired numerical precision (that is tolerance of numerical error).

Theorem 4.7. If some x^*, y^*, s^* satisfy conditions (4.6), those values are optimal solutions to problems (4.3) and (4.4) for some selected μ.

The barrier problem is still hard to solve due to the objective function. The system of equations (4.6) is hard to solve directly because is non-linear due to the equations $X(\mu)S(\mu)\mathbb{1} = \mathbb{1}\mu$.

The algorithm given in the next section is based on Taylor approximation of barrier functions in primal with terms up to quadratic, inclusively. The problem is then solved iteratively with Newton's

method. Since the method is based on primal, we have to keep the feasibility of the current solution throughout the procedure and the procedure must be initialized with a primal feasible point.

The alternative method called *primal-dual path-following algorithm* uses Newton's method directly on the system of KKT condition equations (4.6). Variants of this method need not keep the feasibility of the current solution and can be initialized with primal and dual infeasible points.

Primal path following algorithm

In order to obtain an efficient algorithm, the following is done:

- Substitute barrier function $B^p(x)$ with their first three terms of Taylor series expansion around x.
- Obtained approximate problem can be solved in closed form using the method of Lagrangian multipliers
- We obtain a system of $m + n$ linear equations with $m + n$ unknowns. Unknowns are gradient direction and dual solution.
- Normally, for Newton's method, for each μ we would need to do several iterations to converge to $x(\mu)$. However, here we shall do only one iteration for each value of μ. In order to guarantee the efficient and effective working of the algorithm, we must be careful with the selected value of parameter α which decreases μ.

  The algorithm we obtain is given below.

1. (Initialize) k=0, set $x^0 > 0, s^0 > 0, y^0$ to primal and dual feasible values
2. (Test optimality) if $(s^k)^T x^k < \epsilon$, STOP.
3. (Prepare iteration)

$$\begin{aligned} X_k &= diag(x^k) \\ \mu^{k+1} &= \alpha\mu^k \end{aligned}$$

4. (Compute directions) by solving system for y and Newton direction d:

$$\begin{aligned} \mu^k X_k^{-2} d - A^T y &= \mu^{k+1} X_k^{-1} \mathbb{1} - c \\ Ad &= 0 \end{aligned} \tag{4.7}$$

5. (Step)

$$\begin{aligned} x^{k+1} &= x^k + d \\ y^{k+1} &= y \\ s^{k+1} &= c - A^T y \end{aligned}$$

6. $k := k + 1$, goto 2.

Primal path following algorithm produces feasible, near-optimal (terminated-asymptotic in step 2), primal, and dual solutions.

Complexity of the algorithm is $O(n^3\sqrt{n}log\frac{\epsilon_0}{\epsilon})$. With appropriate initialization of ϵ_0, this is polynomial in n, $log(1/\epsilon)$ and logarithm of upper bound on size all numerical parameters of the problem (in A,b,c).

4.8 Applications

Approximation algorithms

Check the chapter for applications of linear programming in approximation algorithms. There is given an example for a minimal vertex cover problem that uses LP as a building block for designing an approximation algorithm.

Matching in dating sites

PROBLEM. On a dating site, in the straight section (just because it is easier to model), there exist M males and F females. Compatibilities (Table 4.1) have been calculated for all potential couples using different models that were calibrated on filled questionnaires. The site wants to match all couples in such a way that the total sum of normalized compatibilities is maximized (which amounts to *utilitarianism*).

Table 4.1: Data for dating site

	Compatibilities			
	M1	M2	M3	M4
F1	9	1	8	7
F2	1	2	1	7
F3	8	2	4	8
F4	2	4	6	4

MODEL. Let x_{ij} be 1 if (i,j) pairing is selected and 0 otherwise. Also, let c_{ij} be the compatibility for matching (i,j).Then, the aforementioned problem is modeled by the following LP:

$$\begin{array}{lll} \max & \sum_{i,j} c_{ij}x_{ij} & \\ \text{subject to} & \sum_j & x_{ij} = 1, \forall i \\ & \sum_i & x_{ij} = 1, \forall j \\ & & x \geq 0 \end{array}$$

Transport problem

PROBLEM. Pfizer is producing vaccines for COVID-19 and it has production facilities in three locations: T1...3 with available capacities given in Table 4.2. Customers are on three locations O1...4 with demands specified in the last row of the Table 4.2. Unit transportation costs for all combinations of production facilities and customers are also in the table. How to satisfy the demand most efficiently?

	Transportation costs				
	O1	O2	O3	O4	Capacity
T1	10	9	14	8	432
T2	7	11	9	11	138
T3	8	12	12	9	35
Ordered	500	200	115	100	

Table 4.2: Data for vaccine transportation

MODEL. Let x_{ij} be the quantity delivered from i^{th} facility to the j^{th} customer. Also, let c_{ij} be the unit transportation cost between i and j. s_i is the available capacity at facility i and d_j is the demand at customer j.

Then, the aforementioned problem is modeled by the following LP:

$$\begin{array}{lll} \text{min} & \sum_{i,j} c_{ij} x_{ij} & \\ \text{subject to} & \sum_j & x_{ij} \leq s_i, \forall i \\ & \sum_i & x_{ij} = d_j, \forall j \\ & & x \geq 0 \end{array}$$

Exact combinatorial optimization

Often combinatorial problems can be modeled or approximated by a mixed-integer-linear program, which is an LP with integrality constraint on some decision variables. In such cases, LP relaxations solved with dual simplex are used as a heuristic within branch&bound method that does implicit enumeration. Going into more details is out of scope. Interesting modern applications within this area include verification of neural networks for safety and other properties [4], optimization in supply chains, electric power distribution, scheduling, etc.

[4] Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher Strong, Clark Barrett, and Mykel J. Kochenderfer. Algorithms for Verifying Deep Neural Networks. *Foundations and TrendsÂő in Optimization*, 4(3-4):244–404, February 2021. ISSN 2167-3888, 2167-3918. DOI: 10.1561/2400000035. URL https://www.nowpublishers.com/article/Details/OPT-035. Publisher: Now Publishers, Inc

4.9 Conclusion

We have seen in this chapter that **linear programs** are quite a general problem family that can model many real-world situations. It is one of the simplest subfamilies of convex optimization problems, which all fall into the class of problems P. We have learned how to model, transform and solve LPs. We have shown the low-dimensional geometric interpretation to build the intuition about convex polytopes - spaces described by LPs constraints. We have used the graphical method as the first specialized method for solving low-dimensional LPs. The first algebraic method for solving was "naive" simplex, which was capable of solving only specific LPs. Simplex solves LPs in a round-about fashion by traversing extreme points on the boundary of the feasible area. We upgraded this method into the first full-blown LP solving algorithm: **two-phase simplex**, which is simply achieved by running "naive" simplex twice paired with a few simple tricks. Afterward, we introduced the idea of duality which is a very profound concept. It connects many ideas from sensitivity analysis,

economics, game theory, etc. It enables not only rich interpretation, but it also gave us a second full-blown LP solving algorithm: **generalized simplex method**. That algorithm follows from the dual simplex method that works on dual feasible, but a primal infeasible solution. Generalized simplex method can work on both primal and dual infeasible solutions. Then we encountered a dangerous counter-example: **Klee-Minty cube** that demonstrates that simplex family of algorithms is in the worst case of exponential complexity. Smoothed analysis explains that such examples are rare and in practice, simplex is a very powerful, often used, algorithm for solving LPs in commercial solvers. It is also much simpler to learn than interior point methods. Then, we addressed the ellipsoid method that was the first approach to prove that LPs fit into the P complexity class. Finally, we have introduced the latest advancement in the LP-solving technology which came about in the 1980s: **interior point methods**. They are complementary to simplex and are also used in commercial solvers. These methods do not go in a round-about way to the solution, but get straight to the matter and approach the optimum from the interior of the feasible area. This is achieved by coding all the inequality constraints as logarithmic barriers and putting them into a changed objective of the problem. These barriers in objective function act as a force-field that keeps gradient steps within the feasible area. The problem is still convex, but it becomes a nonlinear optimization problem.

5 Graph algorithms

Graph as a non-Euclidean structure has become some of the basic structures used in science and industry. Graph's main purpose and strength is to model relationships and structures. Some of the typical domains where graphs are extensively used are:

- **Network modelling** - Graphs are quite often used to model networks. Having a network modelled as a graph allows us to perform some advanced calculations and analyses of the modelled network. Some examples are electrical distribution networks, computer networks, telecommunication networks, social networks, and so on. Graphs in network modelling are quite often used for calculating maximal network flow, discovering network weak spots, and their resolution. Social networks can be used to detect people's relationships, for marketing and sales, tracking virus spread in the epidemiology, and similar. A genealogy tree is a kind of social network.
- **Geometric graphs** - Also called *spatial graphs*, geometric graphs can be used to define geometric positions and relationships between objects. One example is a facial feature spatial graph that can be used for facial recognition. Some of the well-known problems, such as the *Chinese postman problem* and *traveling salesman problem*, use the capability of graphs to model spatial features. Such graphs can be used to optimize traveling, flying routes, car navigation routes, and similar.
- **General data** - The entity-relationship data model is graph-friendly, which means we can transform data from a database into a graph. Having data in form of a graph can provide some additional ways of data processing, especially data relationships and structure.
- **Linguistic** - Graphs are quite useful in linguistic. Language syntax and compositional semantic are traditionally modelled in a form of hierarchical graphs. Graphs are also used to model vocabularies and synonyms.
- **Biology** - In molecular biology, graphs are often used to model biological molecules, such as genomes (DNA, RNA), protein chains, and similar. Graphs are also used to model evolution trees, species inherited features, ecological systems, species movements, and similar.
- **Physics and chemistry** - Various molecules and crystal structures are modelled in a form of a graph. Graphs are used in particle physics, for example to model collision events.
- **Computer science** - Many general-purpose algorithms use graph structure to establish relationships between their parts and components, such as artificial neural networks, automata, and data processing networks, such as the Petri-nets. The whole business process automation is based on algorithms and solutions that have graph-like structures.

In this chapter, we bring selected graph algorithms [Agnarsson and Greenlaw, 2006, Cormen et al., 2009, Drozdek, 2012] that can be used to solve some of the problems in the aforementioned domains.

5.1 Graph theory

Definition 5.1 (Graph). A graph is a non-Euclidean structure defined as an ordered pair

$$G = (V, E) \tag{5.1}$$

where $V \neq \emptyset$ is a non-empty set of vertices, and $E \supseteq \emptyset$ is a set of edges.

Edges add structure to the graph. An edge represents a relationship between two vertices, defined through a product

$$E \subseteq V \times V \tag{5.2}$$

An edge is an ordered pair of vertices

$$\begin{aligned} e = (a, b) &\in E \\ a, b &\in V \end{aligned} \tag{5.3}$$

shortly denoted as $e = ab$. Graphically, we draw edges as lines that connect two vertices. Vertices a and b are called *endpoints* for the edge $e = (a, b)$. We say that the edge $e = (a, b)$ is **incident** (*connected*) to vertices a and b. Vertices a and b are **adjacent** (*neighbors*) when they are connected by the edge $e = (a, b)$. Also, vertices are adjacent when they are incident to the same edge.

The undirected graph in Figure 5.1 is defined as

$$\begin{aligned} V_1 &= \{v_1, v_2, v_3, v_4, v_5\} \\ E_1 &= \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_4), (v_3, v_5), (v_4, v_5)\} \\ G_1 &= (V_1, E_1) \end{aligned} \tag{5.4}$$

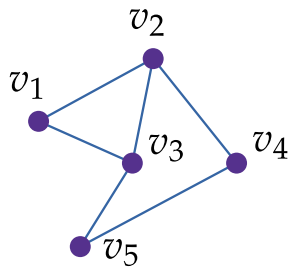


Figure 5.1: An example of an undirected graph.

Definition 5.2 (Undirected graph). We say that a graph is **undirected** when the order of vertices in edges is not important

$$(a, b) = (b, a) \tag{5.5}$$

The undirected graph in Figure 5.1 we can be also defined having the following set of edges

$$E_1 = \{(v_2, v_1), (v_3, v_1), (v_3, v_2), (v_4, v_2), (v_5, v_3), (v_5, v_4)\} \tag{5.6}$$

where we simply turned the order of vertices in edges.

Definition 5.3 (Directed graph). We say that a graph is **directed** when the order of vertices in edges is important

$$(a, b) \neq (b, a) \tag{5.7}$$

which means that an edge (a, b) is not the same as (b, a). Since we differentiate these two edges, we can define the set of edges as

$$E = \{(a, b), (b, a)\} \tag{5.8}$$

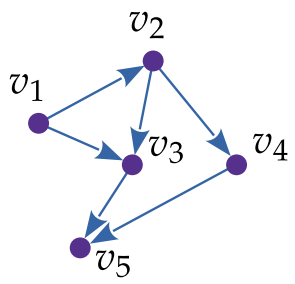


Figure 5.2: An example of a directed graph.

The directed graph in Figure 5.2 is formally defined as

$$\begin{aligned} V_2 &= \{v_1, v_2, v_3, v_4, v_5\} \\ E_2 &= \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_4), (v_3, v_5), (v_4, v_5)\} \\ G_2 &= (V_2, E_2) \end{aligned} \tag{5.9}$$

For directed graphs, a vertex a is considered to be adjacent to the vertex b only if there is an edge $e = (a, b) = ab$. The edge $e = (a, b) = ab$ is an **outedge** for the vertex a and an **inedge** for the vertex b.

Definition 5.4 (Graph order). The number of vertices in a graph is called graph order.

The order of graph G_2 in Figure 5.2 is $|V_2| = 5$.

Definition 5.5 (Graph size). The number of edges in a graph is called graph size.

The size of graph G_2 in Figure 5.2 is $|E_2| = 6$.

Definition 5.6 (Loop). An edge that starts and ends in the same vertex, incident to the same vertex, is called loop

$$e = (a, a) = aa \tag{5.10}$$

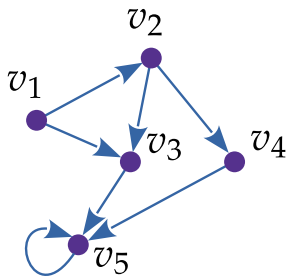


Figure 5.3: An example of loop $e = v_5v_5$ in the directed graph.

Definition 5.7 (Vertex degree). The total number of edges incident to the vertex $v \in V$ is called vertex degree and is denoted as $deg(v)$.

By decomposing incident edges of the vertex $v \in V$ we get

$$\begin{aligned} E_b(v) &= \{ab \in E \mid v = a \veebar v = b\} \subseteq E \\ E_p(v) &= \{ab \in E \mid v = a \wedge v = b\} \subseteq E \\ deg(v) &= |E_b(v)| + 2|E_p(v)| \end{aligned} \tag{5.11}$$

where $E_b(v)$ are incident non-looping edges and $E_p(v)$ are incident loops of the vertex v. We count each incident non-looping edge from $E_b(v)$ once, and each incident loop from $E_p(v)$ twice.

Lemma 5.1. (*Handshaking*) The sum of all vertex degrees in an undirected graph is

$$\sum_{v \in V} deg(v) = 2|E| \tag{5.12}$$

since each non-looping edge is incident to two distinct vertices, and each loop is incident twice to the same vertex.

Figure 5.4: A vertex in a directed graph that has two inedges and one loop (counts for one inedge and one outedge), which results in $indeg(v_1) = 3$, $outdeg(v_1) = 1$. The final vertex degree is $deg(v_1) = 4$.

In a directed graph, vertex degree can be additionally broken down. The number of inedges of the vertex $v \in V$ is called **indegree**, while the number of outedges is called **outdegree**

$$\begin{aligned} E_{ub}(v) &= \{ab \in E \mid v = b\} \subseteq E, indeg(v) = |E_{ub}(v)| \\ E_{ib}(v) &= \{ab \in E \mid v = a\} \subseteq E, outdeg(v) = |E_{ib}(v)| \\ deg(v) &= indeg(v) + outdeg(v) \end{aligned} \tag{5.13}$$

where $E_{ub}(v)$ are inedges and $E_{ib}(v)$ are outedges of the vertex v.

Graph types

Definition 5.8 (Simple graph). A graph $G = (V, E)$ (usually undirected) where we can have at most one edge connecting two distinct vertices

$$\forall a, b \in V \exists E' = \{ab, ba\} \cap E \mid a \neq b, |E'| \leq 1 \tag{5.14}$$

and there are no loops

$$\forall a \in V \nexists aa \in E \tag{5.15}$$

is called simple graph.

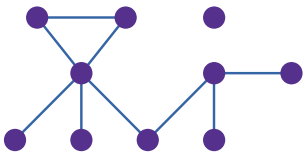


Figure 5.5: A simple graph example.

Definition 5.9 (Multigraph). A graph where we can have multiple edges connecting two distinct vertices and no loops is called a multigraph.

Limitation imposed by (5.14) is no longer valid for multigraph.

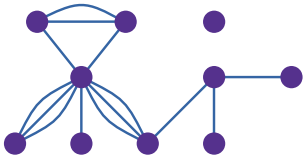


Figure 5.6: A multigraph example.

Definition 5.10 (Pseudograph). A graph where we can have multiple edges connecting two distinct vertices and loops are allowed is called a pseudograph.

Limitations imposed by both (5.14) and (5.15) are no longer valid for pseudograph

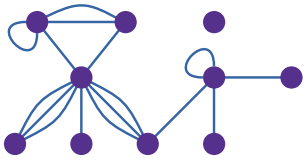


Figure 5.7: A pseudograph example

Definition 5.11 (Complete graph). A simple undirected graph that has exactly one edge between each pair of vertices is called a complete graph. A complete graph of order n is denoted as K_n. For $K_n = (V, E)$ we must have

$$\forall a, b \in V \exists! e \in E \mid a \neq b, e = ab \vee e = ba \tag{5.16}$$

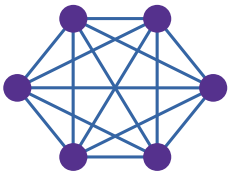


Figure 5.8: An example of a complete graph K_6 having $E(K_6) = \frac{6(6-1)}{2} = 15$ edges.

Since the complete graph is simple, it does not comprise any loops. The number of edges in a complete graph correlates with the number of vertex pairs. We can calculate the number of edges in a complete graph as

$$E(K_n) = \binom{|V|}{2} = \frac{|V|!}{(|V|-2)! * 2!} = \frac{|V|(|V|-1)}{2} \approx |V|^2 \tag{5.17}$$

Definition 5.12 (Subgraph). Let us define a graph $G = (V, E)$, and subsets of its vertices $V' \subset V$ and edges $E' \subseteq E \cap (V' \times V')$. Then, the graph $G' = (V', E')$ is a subgraph of the graph G. For the subgraph G' the following holds

$$\forall ab \in E' \mid a \in V', b \in V' \tag{5.18}$$

meaning that each edge in the subgraph G' can be incident only to vertices from the subset V'.

Regrading the aforementioned definition, there can be some reminder of edges $E \setminus E'$ from the graph G that are incident to vertices from the subset V', which means

$$\exists ab \in E \setminus E' \mid a \in V', b \in V' \tag{5.19}$$

Definition 5.13 (Graph isomorphism). Let us define graphs G and H such that there is a bijective mapping function between their vertices defined as

$$f : V(G) \to V(H) \tag{5.20}$$

This bijection is called graph isomorphism iff for each pair of adjacent vertices a and b in the graph G, there is an equivalent pair of adjacent vertices $f(a)$ and $f(b)$ in the graph H.

$$\forall ab \in E(G) \exists f(a)f(b) \in E(H) \tag{5.21}$$

If there is graph isomorphism between graphs G and H, then these graphs are called *isomorphic*, which is denoted as $G \simeq H$.

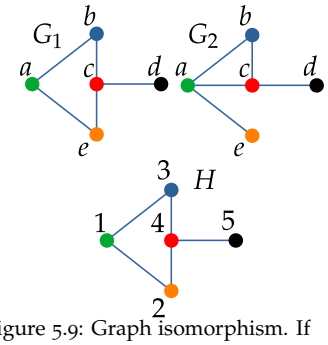


Figure 5.9: Graph isomorphism. If we have $f(a) = 1, f(b) = 3, f(c) = 4, f(d) = 5, f(e) = 2$ from the mapping function f, then $G_1 \simeq H$ and $G_2 \not\simeq H$.

Definition 5.14 (Induced subgraph). Let us define a graph $G = (V, E)$ and a subset of its vertices $V' \subset V$. Then, a graph $G[V'] = (V', E')$ is called an induced subgraph of the graph G, where each pair of adjacent vertices in the graph G are also adjacent in the induced subgraph $G[V']$

$$\forall a, b \in V' \exists ab \in E(G) \mid ab \in E(G[V']) \tag{5.22}$$

which defines the resulting set of edges E' of the induced subgraph $G[V']$ as

$$E' = E \cup (V' \times V') \tag{5.23}$$

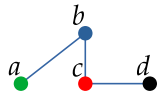


Figure 5.10: Induced subgraph $G_1[V']$ from Figure 5.9 for the subset $V' = \{a, b, c, d\}$.

We say that a graph $H = (V_H, E_H)$ is **subgraph isomorphic** to a graph $G = (V_G, E_G)$ if there is a subgraph $G_0 = (V_0, E_0), V_0 \subset V_G, E_0 \subseteq E_G \cap (V_0 \times V_0)$ of the graph G such that $G_0 \simeq H$. According to the aforementioned definitions, $G_0 = G[V_0]$ is actually an induced subgraph of the graph G.

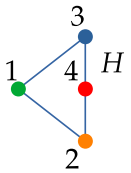


Figure 5.11: H is subgraph isomorphic to the graph G_1 in Figure 5.9.

Walks, paths and cycles

Real-life problems often require finding paths in graphs that adhere some requirements or have specific properties. For example, we want to know all flights between two cities, their durations, costs, layovers.

Definition 5.15 (Walk). A sequence of vertices and edges between a starting vertex a and an ending vertex b is called a walk.

Let us define a graph $G = (V, E)$ that has the following vertices and edges

$$\begin{aligned} \{a, v_1, ..., v_n, b\} &\subseteq V \\ \{e_1, e_2, ..., e_n, e_{n+1}\} &\subseteq E \\ e_1 = av_1, e_2 = v_1v_2, ..., e_{n+1} &= v_nb \end{aligned} \tag{5.24}$$

A walk w is defined as a sequence of vertices and edges

$$w = (a, v_1, e_1, v_2, e_2, ..., v_n, e_{n+1}, b) \tag{5.25}$$

that can be shorter denoted as

$$w = av_1v_2...v_nb \tag{5.26}$$

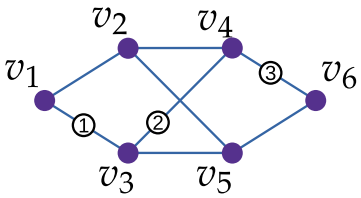


Figure 5.12: An undirected graph having walk $w = v_1v_3v_4v_6$ from v_1 to v_6.

Some of the open walks from v_1 to v_6 in Figure 5.12 are

$$w_1 = v_1v_2v_4v_6, w_2 = v_1v_2v_5v_6,$$
$$w_3 = v_1v_3v_4v_2v_5v_6, ...$$

Another interesting walk is

$$w_4 = v_1v_3v_5v_2v_4v_3v_5v_6$$

Closed walks from v_4 to v_4 in Figure 5.12 are

$$w_1 = v_4v_6v_5v_2v_4, w_2 = v_4v_6v_5v_3v_4,$$
$$w_3 = v_4v_2v_1v_3v_4, w_4 = v_4v_3v_5v_6v_4, ...$$

Walk length depends on the number of edges contained in the walk. We have the following important cases:

- Walks having length 0 that comprise a single vertex, for example $w = a$,
- Walks having length greater than 0 that comprise at least one edge.

In directed graphs, walks need to follow edge directions. A walk w from the vertex a to the vertex b is **open** if $a \neq b$, or **closed** if $a = b$.

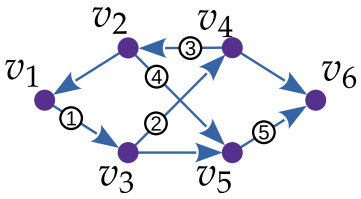


Figure 5.13: Directed graph having a walk $w = v_1v_3v_4v_2v_5v_6$ from v_1 to v_6.

Clearly, a graph $G = (V, E)$ can consist of numerous walks. A set of all walks in the graph G is defined as

$$W(G) = \{w_1, w_2, ..., w_m\} \tag{5.27}$$

We define the following relation between the starting vertex a and the ending vertex b in the graph G

$$aW^Gb \equiv av_1...v_nb \in W(G) \tag{5.28}$$

We say that the vertex b is **reachable** from the vertex a in the graph G if aW^Gb, which means that there is at least one walk $av_1...v_nb$ in the set of all walks $W(G)$ of the graph G.

Definition 5.16 (Trail). An open walk that does not contain the same edge twice is called a trail.

$$\forall e_k, e_l \in w \mid e_k \neq e_l \tag{5.29}$$

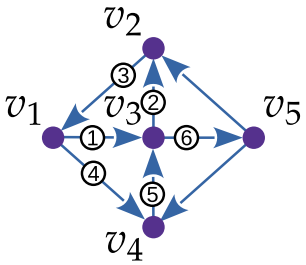


Figure 5.14: Directed graph having a trail $w = v_1v_3v_2v_1v_4v_3v_5$ from v_1 to v_5. Notice that we did not walk over the same edge more than once. However, we visited vertices v_1 and v_3 twice.

Definition 5.17 (Path). A trail that does not contain the same vertex twice is called a path. Considering trail definition, a path is an open walk that does not repeat vertices and edges, which means that besides the aforementioned limitation (5.29) the following limitation for vertices must be added

$$\forall v_i, v_j \in w \mid v_i \neq v_j \tag{5.30}$$

In the directed graph in Figure 5.14 we have:

- A path $w = v_1v_4v_3v_5$ from v_1 to v_5
- A circuit $w = v_1v_3v_5v_4v_3v_2v_1$ from v_1 to v_1
- A cycle $w = v_1v_4v_3v_5v_2v_1$ from v_1 to v_1

Definition 5.18 (Circuit). A closed trail is called a circuit.

Definition 5.19 (Cycle). A closed path is called a cycle.

Graph connectedness

Definition 5.20 (Connected undirected graph). Based on the walk definition, an undirected graph $G = (V, E)$ is considered to be connected if each pair of vertices $a, b \in V$ is connected, which is true only if the vertex b is reachable from the vertex a.

$$\forall a, b \in V \mid aW^Gb \tag{5.31}$$

Since the aforementioned definition is focused on undirected graphs, based on (5.5) we can consider the relation W^G to be symmetrical

$$\forall a, b \in V \mid aW^G b \wedge bW^G a \tag{5.32}$$

meaning, if b is reachable from a, then surely a is also reachable from b.

For directed graphs, connectedness definition is somewhat more complex than for undirected graphs. In the example in Figure 5.16 we can observe the following

$$\nexists u \in V \mid v_7 W^G u \tag{5.33}$$

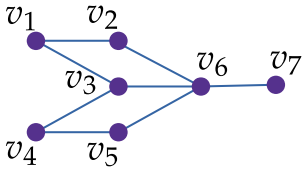


Figure 5.15: Connected undirected graph where (5.31) and (5.32) hold.

Definition 5.21 (**Weakly connected directed graph**). A directed graph is considered to be weakly connected if its undirected equivalent is connected according to definition 5.20.

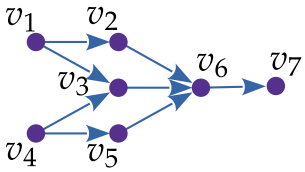


Figure 5.16: Weakly connected directed graph. Its undirected equivalent can be seen in Figure 5.15.

Definition 5.22 (**Strongly connected directed graph**). A directed graph $G = (V, E)$ is considered to be strongly connected if each pair of vertices $a, b \in V$ is mutually reachable, which means that the vertex b must be reachable from the vertex a and *vice versa*, similarly to definition 5.20.

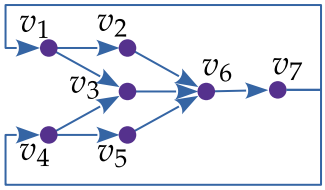


Figure 5.17: Strongly connected directed graph.

A graph that has a single vertex and no edges $K_1 = (\{a\}, \emptyset)$ is considered to be connected since it has one walk of length 0 comprising only one vertex, having $W(K_1) = \{w_1 = a\}$.

Definition 5.23 (**Disconnected graph**). Let us define a graph $G = (V, E)$ in which there is a subset of vertices $V' \subset V$ such that the induced subgraph $G[V']$ is connected and there is no walk that connects vertices from the set V' and the remainder of vertices $V \setminus V'$

$$\nexists a \in V' \nexists b \in V \setminus V' \mid aW^G b \vee bW^G a \tag{5.34}$$

then we can consider graph G to be disconnected.

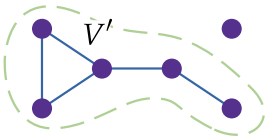


Figure 5.18: Disconnected undirected graph.

Generally, if we split graph $G = (V, E)$ to vertex partitions $P(V) = \{V_1, V_2, ..., V_n\}$ such that the induced subgraph for each vertex partition $G[V_i]$ is connected, and there is no walk that would connect vertices from distinct partitions

$$\forall V_i, V_j \in P(V) \nexists a \in V_i \nexists b \in V_j \mid aW^G b \vee bW^G a \tag{5.35}$$

then we can draw the following conclusions:

- Graph G is disconnected,
- Graph G made of $|P(V)|$ subgraphs that are internally connected,
- Graph G can be losslessly decomposed to $|P(V)|$ induced subgraphs $G[V_1], G[V_2], ..., G[V_n]$.

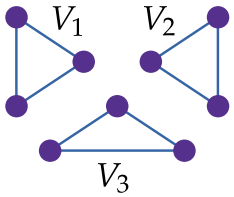


Figure 5.19: Disconnected undirected graph made of three vertex partitions $P(V) = \{V_1, V_2, V_3\}$ whose induced subgraphs $G[V_1], G[V_2], G[V_3]$ are internally connected.

Definition 5.24 (**Articulation point**). A vertex in a graph G whose removal makes the graph disconnected is called an articulation point.

Definition 5.25 (**Bridge**). An edge in a graph G whose removal makes the graph disconnected is called a bridge.

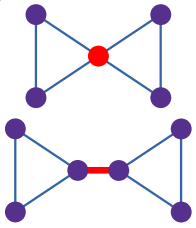


Figure 5.20: Articulation point and bridge examples.

Graph components

Undirected graphs are composed of *biconnected components*.

Definition 5.26 (Biconnected component (block)). A subgraph $G' = (V', E')$ of a connected graph $G = (V, E)$ that has no articulation points and bridges is called a biconnected component (or a block) of the graph G.

Each directed graph that does not satisfy definition 5.23 can be considered for connected. There are two kinds of directed graphs regarding existing cycles:

- **Directed acyclic graph** (DAG) - A directed graph that doesn't contain cycles. An example can be seen in Figure 5.16,
- **Directed cyclic graph** (DCG) - A directed graph that contains cycles. An example can be seen in Figure 5.17.

Definition for strongly connected directed graphs requires that each pair of vertices $a, b \in V$ are mutually reachable, which means that b must be reachable from a and *vice versa*. This means that for each pair of vertices in a directed graph G, there must be two open walks

$$\begin{aligned} w_1 &= au_1u_2...u_kb \\ w_2 &= bv_1v_2...v_la \end{aligned} \tag{5.36}$$

whose combination forms the following closed walk

$$w = au_1u_2...u_kbv_1v_2...v_la \tag{5.37}$$

If we cannot find such a walk for at least one pair of vertices in G, this means that G is weakly connected in whole. However, this does not mean that in the directed graph G there are no pair of vertices that are mutually reachable, and according to definition 5.22.

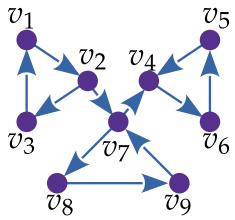


Figure 5.21: Directed graph that is weakly connected in whole since not all pairs of vertices are mutually reachable, for example v_7 is reachable from v_2, but not *vice versa*. However, it is possible to induce several subgraphs that are strongly connected.

If the directed graph in Figure 5.21 is divided to partitions of vertices that are strongly connected $V_1 = \{v_1, v_2, v_3\}, V_2 = \{v_4, v_5, v_6\}, V_3 = \{v_7, v_8, v_9\}$, their induced subgraphs form three strongly connected components.

Definition 5.27 (Strongly connected components (SCC)). If we split directed graph $G = (V, E)$ to vertex partitions $P(V) = \{V_1, V_2, ..., V_n\}$ such that each partition is strongly connected

$$\forall V_i \in P(V) \forall a, b \in V_i \mid aW^G b \wedge bW^G a \tag{5.38}$$

then induced subgraphs $G[V_1], G[V_2], ..., G[V_n]$ represent strongly connected components of the directed graph G.

Definition 5.28 (Strong articulation point). A vertex in a directed graph G whose removal increases the number of strongly connected components in G is called a strong articulation point.

Definition 5.29 (Strong bridge). An edge in a directed graph G whose removal increases the number of strongly connected components in G is called a strong bridge.

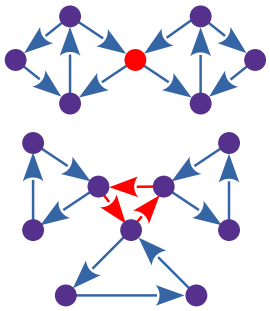


Figure 5.22: Strong articulation point and strong bridge examples.

Detecting strongly connected components in a directed graph is NP-complex, since it requires checking each pair of vertices in the graph. Creating an efficient algorithm that detects strongly connected component presents a challenge.

Weighted graph

Definition 5.30 (Weighted graph). A weighted graph is an ordered triple $G = (V, E, w)$, where V is a set of vertices, E is a set of edges, and w is a weight mapping function

$$w : E \to \mathbb{R} \tag{5.39}$$

Each edge of the weighted graph is mapped to its numeric weight.

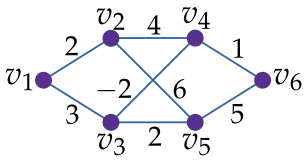


Figure 5.23: Weighted graph example.

Numeric weights are added to graph edges to simulate the properties of real-life scenarios. Most often, the weight of an edge is interpreted as cost. Weight can be interpreted as a route length, cost of traveling between cities, time needed for moving from one place to another, and similar.

Graph representations

To store graphs in computer memory, and to be able to access graph elements from various algorithms, we need meaningful and optimal data structures for it. Most of the graph-related algorithms are based on graph traversals and walks, which is reflected in most of the data structures used for graph storing. Such data structures are made to allow fast graph traversals.

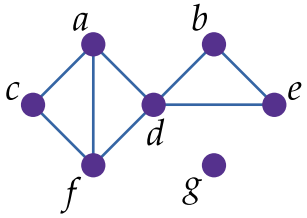


Figure 5.24: Undirected graph example.

Figure 5.24 shows a graph example that will be used to explain data structures suitable for graph storing. There are two major approaches in storing graphs:

First approach is to organize graph storing data structure by the adjacent vertices in the graph.

Adjacency table (*star representation*)

Adjacency table is row-organized, where each table row represents one graph vertex, whose label is stored in the first column. The rest of the columns (second and on) are used to store its adjacent vertices. After finding the row of the currently processed vertex, all adjacent vertices can be found by iterating columns of the same row.

a	c	d	f	
b	d	e		
c	a	f		
d	a	b	e	f
e	b	d		
f	a	c	d	
g				

Table 5.1: Adjacency table example.

Most unoptimized way of searching through the adjacency table would be though sequential iterations. This can be faster if we use some additional way of organizing data, such as hash tables, indexing trees, or simple row and column sorting as in Table 5.1.

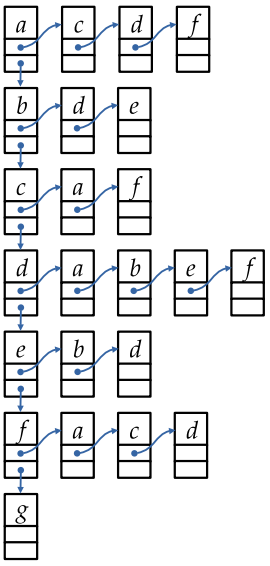


Figure 5.25: Two-dimensional adjacency list example.

In Table 5.1 we can see that the each row is dedicated to its own vertex from the graph in Figure 5.24, whose labels are stored in the first column. Let us say that our algorithm is currently focused on the vertex d. Based on the graph in Figure 5.24, vertices $\{a, b, e, f\}$ are adjacent to the vertex d. The vertex d is stored in the 4th row of Table 5.1, and its label is stored at position $[4, 1]$. All adjacent vertices are sorted and continue from the second column of the same row, so we have adjacent vertices placed as $[4, 2] = a, [4, 3] = b, [4, 4] = e, [4, 5] = f$, which can be accessed by a simple iteration in the algorithm.

Two-dimensional adjacency list

Another structure, very similar to the adjacency table, is using two-dimensional linked lists. Here we have the vertical linked list, used to organized all vertices of the graph. We can move forwards and backwards through this vertical list, and by this to select our focused vertice. Each vertex forms its own horizontal list, which is used to store all adjacent vertices. By moving through the horizontal lists, we iterate through adjacent vertices.

```
G={}
G['a']={'adj':['c','d','f']}
G['b']={'adj':['d','e']}
G['c']={'adj':['a','f']}
G['d']={'adj':['a','b','e','f]}
G['e']={'adj':['b','d']}
G['f']={'adj':['a','c','d']}
G['g']={'adj':[]}
```

Listing 5.1: Adjacency table python code.

Both adjacency table and adjacency list require a stack of pointer pairs for graph traversals. Each pointer pair must contain a pointer to the current vertex, and a pointer to the current adjacent vertex. Since we store adjacency list for each vertex, the second pointer points to the position in the adjacency list. For a walk

$$w = acfdb \tag{5.40}$$

we have the following stack values when using the adjacency table in Table 5.1

Stack position	Pointer pair	Vertex	Adjacency list
4	$[4,3]$	d	b
3	$[6,4]$	f	d
2	$[3,3]$	c	f
1	$[1,2]$	a	c

Definition 5.31 (Adjacency matrix). For a simple graph $G = (V, E)$ having a vertex set $V = \{v_1, v_2, ..., v_n\}$, adjacency matrix A is a square $n \times n$ matrix such that its element A_{ij} is

$$A_{ij} = \begin{cases} 0, & \nexists v_i v_j \in E \\ 1, & \exists v_i v_j \in E \end{cases} \tag{5.41}$$

1 when there is an edge between v_i and v_j, and 0 when there is no edge between these vertices.

The adjacency matrix A is symmetrical for undirected graphs, and asymmetrical for directed graphs. Diagonal values of the adjacency matrix are set to $diag(A) = 0$, unless loops are allowed (which is not the case for simple graphs). For the graph in Figure 5.24, the adjacency matrix is

$$\mathbf{A} = \begin{array}{c} \\ \text{a} \\ \text{b} \\ \text{c} \\ \text{d} \\ \text{e} \\ \text{f} \\ \text{g} \end{array} \begin{array}{c} \begin{array}{ccccccc} \text{a} & \text{b} & \text{c} & \text{d} & \text{e} & \text{f} & \text{g} \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

which is symmetrical since the graph is undirected.

Definition 5.32 (Incidence matrix). For a simple graph $G = (V, E)$ having a vertex set $V = \{v_1, v_2, ..., v_n\}$ and edge set $E = \{e_1, e_2, ..., e_m\}$, incidence matrix B is a $n \times m$ matrix, where rows represent graph vertices and columns represent graph edges. The incidence matrix element B_{ij} is

$$B_{ij} = \begin{cases} 0, & e_j \neq v_i u \wedge e_j \neq u v_i \\ 1, & e_j = v_i u \vee e_j = u v_i \end{cases} \tag{5.42}$$

1 when the vertex v_i is incident to the edge e_j, and 0 when there is no such incidence.

Lemma 5.1 can be applied to incidence matrices

$$\sum B_{ij} = 2m \tag{5.43}$$

meaning that each edge is incident to two distinct vertices of the graph G. For the graph in Figure 5.24 we have the following incidence matrix

$$\mathbf{B} = \begin{array}{c} \\ \text{a} \\ \text{b} \\ \text{c} \\ \text{d} \\ \text{e} \\ \text{f} \\ \text{g} \end{array} \begin{array}{c} \begin{array}{cccccccc} \text{ac} & \text{ad} & \text{af} & \text{bd} & \text{be} & \text{cf} & \text{de} & \text{df} \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

```
G={'a':{'a':0,'b':6,'c':5},
   'b':{'a':6,'b':0,'c':9},
   'c':{'a':5,'b':9,'c':0}}
```

Listing 5.2: Weighted adjacency matrix python code.

Definition 5.33 (Weighted adjacency matrix). For a weighted graph $G = (V, E, w)$ having a vertex set $V = \{v_1, v_2, ..., v_n\}$ and weight mapping function w, weighted adjacency matrix W is a square $n \times n$ matrix such that its element W_{ij} is weight of the edge $w(v_i v_j)$ when there is an edge between v_i and v_j, and 0 when there is no edge between these vertices

$$W_{ij} = \begin{cases} 0, & \nexists v_i v_j \in E \\ w(v_i v_j), & \exists v_i v_j \in E \end{cases} \tag{5.44}$$

The weighted adjacency matrix W has all the properties of the adjacent matrix. For the graph in Figure 5.23, the weighted adjacency matrix is

$$\mathbf{W} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array} \begin{array}{c} \begin{array}{cccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \end{array} \\ \begin{bmatrix} 0 & 2 & 3 & 0 & 0 & 0 \\ 2 & 0 & 0 & 4 & 6 & 0 \\ 3 & 0 & 0 & -2 & 2 & 0 \\ 0 & 4 & -2 & 0 & 0 & 1 \\ 0 & 6 & 2 & 0 & 0 & 5 \\ 0 & 0 & 0 & 1 & 5 & 0 \end{bmatrix} \end{array}$$

which is symmetrical since the weighted graph is undirected.

```
def adj(G,u):
  a=[]
  for v in G[u]:
    if G[u][v]!=0:
      a.append(v)
  return a

def edges(G):
  e={}
  for u in G:
    for v in adj(G,u):
      if not (v,u) in e.keys():
        e[(u,v)]=G[u][v]
  return e
```

Listing 5.3: Adjacency matrix additional functions in python. Adjacency matrices sometimes require small transformations, such as knowing the adjacency vertices and the list of edges.

5.2 *Basic graph traversal algorithms*

For solving our real-life problems we need algorithms. In this case, real-life problems are modelled in graph structure, therefore we need specific algorithms that can work on graphs. We always need to be aware that every problem requires it own specific solution, which means that there is no universal algorithm that can solve every problem. Instead, we group algorithms according to the class of problems they are capable solving. Even so, a small change in the problem we are trying to solve can require algorithm extension.

Graph definition is quite generic, meaning that there are no rules imposed on graph structure and elements, and we are free to model a real-life problem as needed and necessary. For that reason, the focus is switched to algorithms. It is on the algorithm specificity to perform the analysis and compute the solution out of the generic graph structure.

If we are oriented solely to search data in graphs, better solution is to use balanced trees (*AVL tree, RB tree, ...*), which have the necessary constraints and rules for ordering and balancing that make searching efficient and optimized, unlike generic graphs. Also, we need to notice that balanced trees are just a specific form of graphs (directed acyclic graphs).

Solving graph modelled problems includes more than one graph element in many cases. Let us give some examples:

- **Walk oriented problems** - A class of problems where a solving algorithm is required to find a walk (*path or cycle*) as the solution to the problem. For example, navigation problems, where we need to find a walk that has the smallest cost between two places. This cost can be taken as distance, money or time needed for traveling, road speed and quality, and similar. These problems are often perceived as optimization problems, since the solving algorithm must select the most optimal walk from all suitable walks in the input graph.
- **Subgraph isolation problems** - Certain problems require identification and isolation of subgraphs that adhere solution constraints. For example, we want to isolate certain group of people from a social graph that have common characteristics and are connected, being a family, acquaintances or friends. Let us say, a person bought a specific product which we want to market to similar people in his/hers social subgraph.

Solution to all these problems require more general traversals though the graph structure, having simultaneous traversal optimization capabilities. For that reason, we require a hierarchy of traversal algorithms, starting from two basic variants, breadth and depth traversing algorithms.

We start performing graph traversals from a starting vertex. Choosing the starting vertex is arbitrary, meaning that the choice must be such that the traversal is minimal and optimal. For example, in navigation problems we want to start traversals from our traveling departure point. All traversals are performed either in iterations or recursions. Iterative algorithms use lists and queues as their helping data structure, while recursive algorithms use stacks.

Breadth-First-Search (BFS)

The main idea in the Breadth-First-Search (BFS) algorithm is to explore the adjacent vertices first. Then we iterate through the vertices we already explored, and repeat everything with all of their adjacent vertices, and so on. Using iterations is the natural way of doing it, although BFS can be implemented recursively as well. Implementing BFS recursively is meaningless, as it brings an unnecessary complexity to the BFS algorithm.

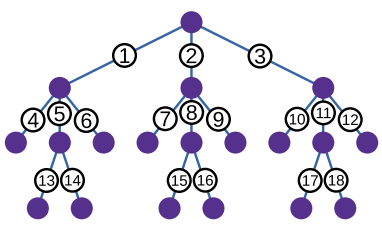


Figure 5.26: BFS concept

Figure 5.26 shows the basic concept of the BFS algorithm. We first explore the first level of adjacent vertices from the *root* vertex, then the second level of adjacent vertices, and so on. A natural data structure that supports this kind of traversal is queue.

```
procedure BFS(G)
    initialize all vertices in G as not visited
    Q ← empty queue
    while there is an unvisited vertex u_0 in G do
        mark u_0 as visited
        Q.enqueue(u_0)
        while Q is not empty do
            u ← Q.dequeue
            process u
            for v in adjacent vertices of u do
                if v is not visited then
                    mark v as visited
                    Q.enqueue(v)
```

Algorithm 5.1: Iterative BFS algorithm.

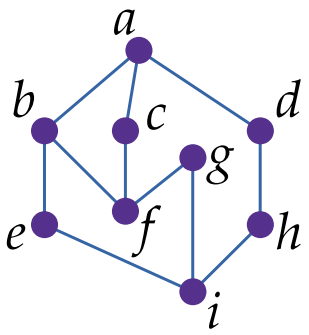


Figure 5.27: Undirected graph example for DFS and BFS algorithms.

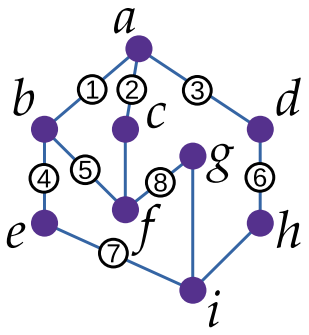


Figure 5.28: Undirected BFS traversal example. a is taken as the first u_0 vertex.

Step	u	queue Q
0		a
1	a	bcd
2	b	$cdef$
3	c	def
4	d	efh
5	e	fhi
6	f	hig
7	h	ig
8	i	g
9	g	

Table 5.2: Iterative BFS while loop steps, and the related queue content.

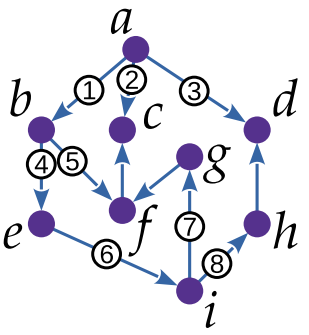


Figure 5.29: Directed BFS traversal example. a is taken as the first u_0 vertex.

The algorithm first initializes the input graph G by marking all vertices as *unvisited*. This means that a vertex in the input graph G must have an auxiliary structure that contains a *visited* flag, which is then used by the BFS algorithm to prevent infinite loop when traversing the graph. The outer *while* loop is there is pick all disconnected vertex partitions by choosing vertices in the alphabetical order. The inner *while* loop is active as long as there are some vertices in the queue Q. For each internally connected vertex partition of the input graph G, this means until all vertices that are reachable from the vertex u_0 are visited. The queue Q is filled by the *for* loop, which iterates through all adjacent vertices of the current vertex u, and adds them to the queue Q. Each vertex added to the queue Q is marked as *visited*, to prevent from adding the same vertex to the queue twice, hence, to prevent infinite loop. This way, our queue Q will contain vertices in a specific order. When looking at the adjacent table graph representation, for example Table 5.1, the *for* loop in the algorithm iterates through the whole row in the adjacency table, which makes it the natural structure to store graphs for the BFS algorithm.

Programming tips: Iterative BFS is programming friendly, as there are only two local loops. We need one queue and *visited* flag for vertices, which could be kept as the integral part of the vertex data structure. Queues are elementary data structures present in all layers of computer systems, and as such can be found in all programming languages, for example `std::queue` in `C++`, `collection.deque` in `python`, `java.util.Queue` interface implementations in `Java`, `dequer` package in `R` and similar. Since the BFS algorithm has simple local loops, for trivial non-functional requirements (no persistence or parallel processing) queue can be simulated with vectors, lists and arrays. If we are not able to extend the graph G data structure to add the *visited*

flag, they can be kept in hash tables, such as `std::unordered_map` in `C++`, dictionaries in `python`, `java.util.HashMap` in `Java`, or environments in `R`, where keys are vertex identifiers and values are visited flags. The other important implementation detail is the graph G representation. All representations that keep adjacent vertices together, such as the adjacent table or the adjacency matrix, are suitable for implementing this variant of the BFS algorithm.

```
def BFS(G):
  vis=[]
  Q = collections.deque()
  for uo in G:
    if uo not in vis:
      Q.append(uo)
      vis.append(uo)
      while Q:
        u=Q.popleft()
        for v in G[u]['adj']:
          if v not in vis:
            vis.append(v)
            Q.append(v)
  return vis
```

Listing 5.4: Iterative BFS python code. The code uses adjacency table graph representation. The returning *vis* list keeps the order of vertices traversal.

Depth-First-Search (DFS)

Rather than exploring each level of adjacent vertices fully as in BFS, DFS focuses on crossing the input graph G in depth, level after level, until there are no *unvisited* adjacent vertices. When returning back to the higher levels closer to the u_0 vertex, we examine if there are some *unvisited* adjacent vertices. If there are, we descend back in depth until there are no *unvisited* adjacent vertices. This is repeated until we return back to the u_0 vertex and there are no *unvisited* adjacent vertices.

Figure 5.30 shows the basic concept of the DFS algorithm. DFS can be implemented in two variants, using stack and iterations, and recursively. We first give the iterative DFS implemented with stack.

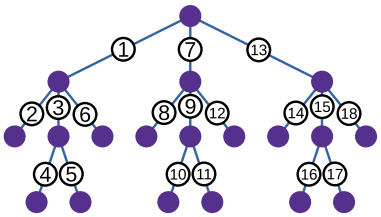


Figure 5.30: DFS concept

procedure DFS(G)
  initialize all vertices in G as **not visited**
  $S \leftarrow$ empty stack
  while there is an unvisited vertex u_0 in G **do**
    S.push(u_0)
    while S is not empty **do**
      $u \leftarrow$ S.pop
      if u is not visited **then**
        mark u as **visited**
        process u
        for v in **reversed list** of adjacent vertices of u **do**
          if v is not visited **then**
            S.push(v)

Algorithm 5.2: Iterative DFS algorithm.

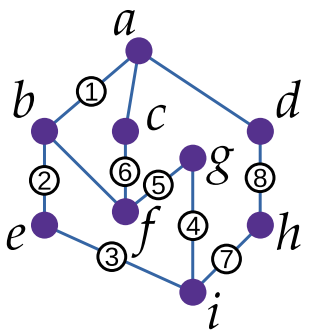


Figure 5.31: Undirected DFS traversal example. a is taken as the first u_0 vertex.

The iterative DFS is similar to the iterative BFS algorithm. One major difference is the data structure used to store vertices that need to be processed next. While a queue is a FIFO data structure, a stack is a LIFO data structure. The difference is the way vertices are organized in these data structures. Since stack is a LIFO data structure, we need to push vertices in the reversed order. For example, for the vertex a adjacent vertices are b, c, d. If we push them on the stack in the reversed order d, c, b, popping them out will be in the right, original order. Another difference is how we mark visited vertices. In the iterative DFS, we mark visited vertices after taking them from the stack, which means that a vertex can be placed on the stack multiple times. Each occurrence of the vertex means a distinct walk that led us from u_0 to that vertex. This only means that we might have some *empty*

Step	u	visited	stack S
0			a
1	a	No	bcd
2	b	No	$efcd$
3	e	No	$ifcd$
4	i	No	$ghfcd$
5	g	No	$fhfdc$
6	f	No	$chfcd$
7	c	No	$hfcd$
8	h	No	$dfcd$
9	d	No	fcd
10	f	Yes	cd
11	c	Yes	d
12	d	Yes	

Table 5.3: Iterative DFS while loop steps, and the related stack content.

iterations of the *while* loop, in cases when we take already visited vertices from the stack. When iterating through the adjacent vertices of the current vertex u, we need to take them reversely. This issue is resolved with the recursive DFS variant, which also relies on a stack to perform recursive calls.

procedure DFS(G)
  initialize all vertices in G as **not visited**
  while there is an unvisited vertex u_0 in G **do**
    DFS_recursive(G, u_0)

procedure DFS_RECURSIVE(G, u)
  mark u as **visited**
  process u
  for v in adjacent vertices of u **do**
    if v is not visited **then**
      DFS_recursive(G, v)

Algorithm 5.3: Recursive DFS algorithm.

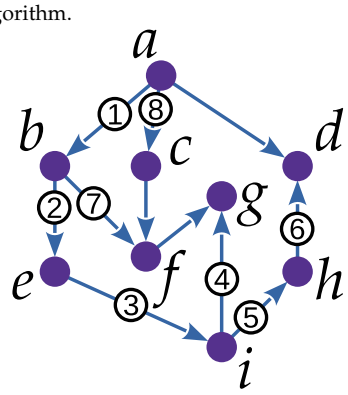


Figure 5.32: Directed DFS traversal example. a is taken as the u_0 vertex.

The recursive DFS is divided in two distinct procedures. The main DFS procedure prepares the input graph G and other supplied data for the recursion. In the main procedure, we first initialize all vertices as *unvisited*, as well as select the top u_0 vertex of each internally connected graph G partition. As in BFS, this also means that a vertex in the input graph G must have an auxiliary structure that contains a *visited* flag. Marking vertices as visited has the same purpose as in BFS, to prevent infinite loop in graphs that have closed walks. After this, we make the first call to the recursive procedure DFS_recursive, using the u_0 vertex. Recursive procedure takes a single vertex u as the input, which can be taken as the top of the subgraph that needs to be traversed by the procedure and all subsequent recursive calls. In the recursive procedure, we first mark the input vertex u as *visited*. Then we use a *for* loop to iterate through all *unvisited* adjacent vertices of the input vertex u. For each *unvisited* adjacent vertex v we initiate a recursive call to the same DFS_recursive procedure, which means further descend into the graph G. We return back from the recursion when the whole subgraph of the input vertex u was traversed and there are no *unvisited* vertices in this subgraph.

Programming tips: For the iterative DFS variant we need a stack implementation such as `std::stack` in `C++`, `collections.deque` in `python`, `java.util.Stack` interface implementations in `Java`, `dequer` package in `R` and similar.

The recursive DFS variant require a bit more insights into the memory management done by the programming language you use. Passing the input graph G between recursive calls should not be done *by value*. Most of the programming languages have object (memory) pointers at disposal, and these should be use to reference the input graph G from all recursive calls. Some difficulties could be faced in the `R` programming language, as most of the variable passing is done by value. Using `R` object-oriented programming (*reference classes* for

```
def DFSIterative(G):
 vis=[]
 S=collections.deque()
 for uo in G:
  if uo not in vis:
   S.append(uo)
   while S:
    u=S.pop()
    if u not in vis:
     vis.append(u)
     for v in reversed(
          G[u]['adj']):
       if v not in vis:
        S.append(v)
 return vis
```

Listing 5.5: Iterative DFS python code. The code uses adjacency table graph representation. The returning *vis* list keeps the order of vertices traversal.

```
def DFS(G):
  vis=[]
  for uo in G:
    if uo not in vis:
      DFS_r(G,uo,vis)
  return vis

def DFS_r(G,u,vis):
  vis.append(u)
  for v in G[u]['adj']:
    if v not in vis:
      DFS_r(G,v,vis)
```

Listing 5.6: Recursive DFS python code. The code uses adjacency table graph representation. The returning *vis* list keeps the order of vertices traversal.

example) or environments, allows you to define a global variable that can be accessed by reference.

As in BFS, all representations that keep adjacent vertices together, such as the adjacent table, are suitable for implementing all variants of the DFS algorithm, since in both we encounter a *for* loop that iterates through the adjacent vertices.

Spanning tree

Spanning trees are an interesting byproduct of the graph traversal algorithms, both BFS and DFS.

Definition 5.34 (Spanning tree). For a **connected** simple graph G having a vertex set V, spanning tree is its **acyclic** subgraph $S = (V, E')$, where $E' \subseteq E$, such that $|E'| = |V| - 1$.

Algorithm 5.4: Spanning tree generating DFS.

```
procedure GenerateST(G, root)
    initialize all vertices in G as not visited
    ST ← (V = V(G), E = ∅)
    return GenerateST_recursive(G, ST, root)
procedure GenerateST_recursive(G, ST, u)
    mark u as visited
    for v in adjacent vertices of u do
        if v is not visited then
            add edge uv to the edges E(ST)
            ST ← GenerateST_recursive(G, ST, v)
    return ST
```

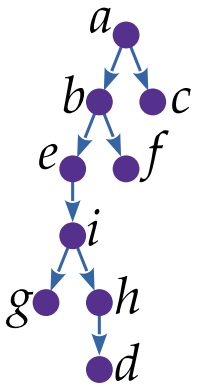


Figure 5.33: Spanning tree for the traversal in Figure 5.32, generated by DFS.

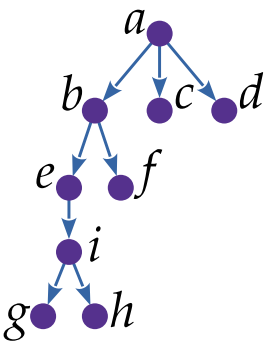


Figure 5.34: Spanning tree for the traversal in Figure 5.29, generated by BFS.

Spanning trees are obtained when traversal algorithms advance to *unvisited* vertices in the input graph G. Each advancement to already *visited* vertex would mean a cycle, which directly contradicts to the acyclic nature of the spanning tree. For example, in the recursive DFS we add an edge to the spanning tree each time before performing a recursive call. This way we transfer edges from the input graph G that form the spanning tree structure.

Spanning trees in Figures 5.33 and 5.34 are both derived from the directed graph in Figure 5.27. We need to notice that DFS and BFS will give distinct spanning trees, which are both correct as long as they are according to definition 5.34.

5.3 *Finding shortest paths*

We often use weighted graphs (definition 5.30) to model structures for real-life problems that include costs, from example money, time, distance, and similar. For example, we can model a city so that streets are edges and street junctions vertices. Weights of edges can represents street lengths. Problem is to find the shortest path between *Marble Arch* and *Piccadilly Circus*. To solve this, we have a number

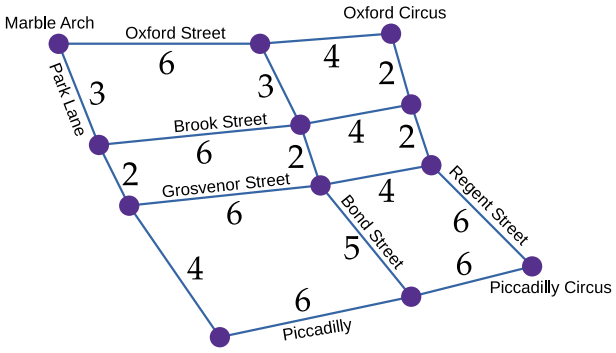


Figure 5.35: Weighted graph of London center.

of algorithms at disposal. Shortest path finding algorithms usually work in two distinct ways:

1. Algorithms that calculate the shortest path between two designated vertices, named **source** and **destination** vertices,
2. All-to-all algorithms that calculate shortest paths between all pairs of vertices in the input weighted graph.

Algorithms that find the shortest path between two designated vertices use vertex labels to set distance of the vertex from the source. To set this label, we extend the weighted graph definition 5.30 to add vertex distance labels. So extended graph is an ordered quadruple $G = (V, E, w, d)$, where d is the mapping function

$$d : V \to \mathbb{R} \tag{5.45}$$

which maps graph vertices to their distances from the source vertex. The distance is the label of the vertex.

Given this graph extension, we divide the shortest path finding algorithms to the following groups:

- **Label-setting algorithms**: Once the distance of a vertex is set, it does not change over the course of the algorithm,
- **Label-correcting algorithms**: Setting and updating vertex distance is allowed over the algorithm course. Distances are updated until the convergence of the algorithm, meaning there are no more distance updates.

Negative weights

A special case is to have an edge $e \in E$ such that the weight of the edge is negative $w(e) < 0$. This poses a problem for the label-setting algorithms, since these algorithms do not update vertex distances. In the example in Figure 5.36 we can choose a for the source vertex, and d for the destination vertex. To calculate the shortest path from a to d we have to take in consideration two distinct paths

$$\begin{aligned} p_1 &= abd \\ p_2 &= abcd \end{aligned} \tag{5.46}$$

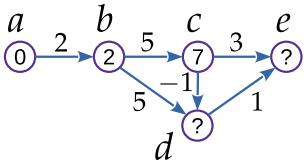


Figure 5.36: Directed graph example having an edge with negative weigth.

Path p_1 has less edges, so we assume that this walk first sets the vertext d distance to $d(d) = 7$. However, distance of the path p_2 is 6, which would require a subsequent update of the vertex d distance. Only by using label-correcting algorithms we can successfully and correctly calculate the distance $d(d)$.

Negative cycles

Although label-correcting algorithms can deal with negative weight edges, having a *negative cycle* in the input graph is not solvable by any known algorithm. Figure 5.37 shows such a graph. Starting, or *setting* distances are $d(b) = 2$ and $d(c) = 7$. However, the negative weight edge $w(cb) = -7$ forces correcting labels of vertices b and c to

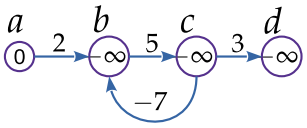


Figure 5.37: Directed graph example comprising a negative cycle.

Iteration	$d(b)$	$d(c)$
1	2	7
2	0	5
3	-2	3
4	-4	1
5	-6	-1
...	...	...
∞	$-\infty$	$-\infty$

which ends in the infinite loop, where label-correcting algorithm tries to reach $d(b) = -\infty$ and $d(c) = -\infty$. The only way to detect a negative cycle is to have a limited number of iterations for reaching distance convergence. If distances don't converge in the limited number of iterations, then there is a good chance we have a negative cycle in the input graph.

Weighted graph transformation

Label-setting algorithms are not capable calculating shortest paths in graphs that have negative weights. However, *label-setting* algorithms are usually faster than *label-correcting* algorithms. When performing all-to-all distance calculations in dense graphs, we must resort to the fastest method possible. Can we transform our negative weighted graphs into graphs that don't have negative weights? At what cost?

We can try to apply a naïve strategy by adding a number to all weights equal to the most negative weight in the graph. Figure 5.38 shows an example of a failed naïve transformation. Before the transformation attempt, in the original graph we have one negatively weighted edge. The shortest path from a to c is $abdc$. If we add $+2$ to all edges of the weighted graph, we get the transformed graph that has completely different shortest path between vertices a and c, which is now either abc or adc.

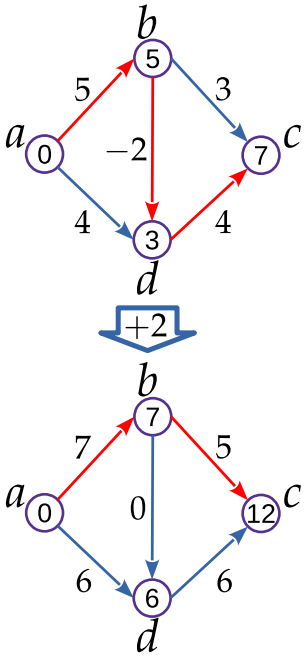


Figure 5.38: Weighted graph naïve transformation attempt.

For an edge uv, we have the following shortest distances

$$d(v) \leq d(u) + w(uv) \tag{5.47}$$

where the shortest distance to the vertex v is not greater than the shortest distance to the vertex u increased by the weight of the edge uv. If we take this slightly differently

$$0 \leq d(u) + w(uv) - d(v) = w'(uv) \tag{5.48}$$

By applying (5.48) to a path $p = v_1v_2...v_k$, we get

$$L'(v_1v_k) = \sum_{i=1}^{k-1} w'(v_iv_{i+1}) = \\ w(v_1v_2) + d(v_1) - d(v_2) + ... + w(v_{k-1}v_k) + d(v_{k-1}) - d(v_k) \tag{5.49}$$

where $L'(v_1v_k)$ is the transformed shortest distance between vertices v_1 and v_k, which becomes

$$L'(v_1v_k) = (\sum_{i=1}^{k-1} w(v_iv_{i+1})) + d(v_1) - d(v_k) = L(v_1v_k) + d(v_1) - d(v_k) \tag{5.50}$$

Finally, the bijective transformation is done as

$$L'(v_iv_k) = L(v_iv_k) + d(v_i) - d(v_k) \\ L(v_iv_k) = L'(v_iv_k) + d(v_k) - d(v_i) \tag{5.51}$$

For the example in Figure 5.38 the transformation is performed as follows

v_i	v_j	$w(v_iv_j)$	$d(v_i)$	$d(v_j)$	$w'(v_iv_j)$
a	b	5	0	5	5+0-5=0
a	d	4	0	3	4+0-3=1
b	d	-2	5	3	-2+5-3=0
b	c	3	5	7	3+5-7=1
d	c	4	3	7	4+3-7=0

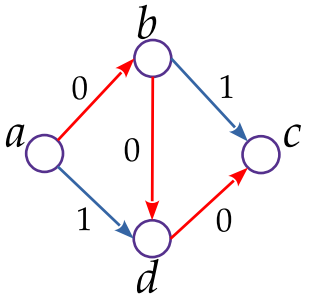


Figure 5.39: Correctly transformed weighted graph from Figure 5.38.

1: Using a *label-correcting* algorithm, determine the shortest distance to every vertex in the input graph G from an arbitrary reference vertex v_r. All vertices in the input graph G must be reachable from the reference vertex v_r.
2: Transform the input graph G into the non-negative weighted graph G' using the bijective transformation given in (5.51).
3: **for** $\forall v_i, v_k \in V(G')$ **do**
4: Find the shortest path between source and destination vertices v_i and v_k in the transformed graph G' using a *label-setting* algorithm. The length of this path is $L'(v_iv_k)$.
5: Use the inverse transformation in (5.51) to get the original length of the shortest path as $L(v_iv_k) = L'(v_iv_k) + d(v_k) - d(v_i)$.

Algorithm 5.5: All-to-all distance calculation using graph transformation.

We perform the initial distance calculations in the first step. Before starting, we need to choose one arbitrary reference vertex v_r such that all other remaining vertices $V(G) \setminus \{v_r\}$ are reachable from it. Then we use a *label-correcting* algorithm to determine distance of all vertices from the reference vertex v_r. In the second step we perform the graph transformation using (5.51) to obtain a non-negative weighted graph G'. In the *for* loop, we iterate through all vertex pairs v_i and v_k to calculate distance between them, using a *label-setting* algorithm, after which we get all-to-all distances in the graph G'. Using inverse transformation (5.51) we get distances in the original weighted graph G.

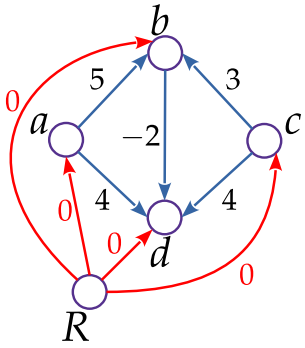


Figure 5.40: Adding an artificial vertex R to calculate initial distances in the graph.

In case there is no reference vertex v_r such that all other vertices are reachable from it, we need to add one artificial vertex incident to all other vertices. This added artificial vertex is then used as the reference vertex for the initial distance calculation in the first step of Algorithm 5.5. Such an example can be seen in Figure 5.40, where we added vertex R and made it adjacent to all other vertices in the graph, which makes all other vertices reachable from R. To make R adjacent to all other vertices, we must add edges having an arbitrary weight, preferably 0.

v_i	v_j	$w(v_iv_j)$	$d(v_i)$	$d(v_j)$	$w'(v_iv_j)$
a	b	5	0	0	5+0-0=5
a	d	4	0	-2	4+0+2=6
b	d	-2	0	-2	-2+0+2=0
c	b	3	0	0	3+0-0=3
c	d	4	0	-2	4+0+2=6

v	$d(v)$
a	0
b	0
c	0
d	-2

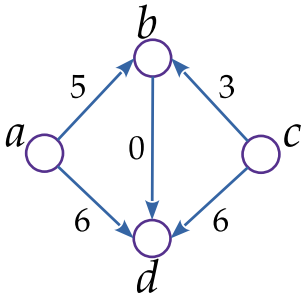


Figure 5.41: Transformed graph from Figure 5.40.

Dijkstra's shortest path algorithm

Dijkstra's algorithm is one of the *label-setting* algorithms, and can be used to calculate the shortest path between two vertices of the input graph G. As all algorithms in this section, Dijkstra's algorithm uses the well-known principle of an intermediate vertex. Any path between the source vertex v_1 and the destination vertex v_n

$$p = v_1v_2...v_{n-1}v_n \tag{5.52}$$

If we take $d(v_i, v_j)$ as a generic distance between two vertices in the path p, shortly as $d(v_j)$ having in mind that this the distance from the vertex v_i, then we surely know that

$$d(v_1, v_n) = d(v_1, v_i) + d(v_i, v_n) \tag{5.53}$$

which is the additive property of the distance. If the path p is found to be the shortest path, then we know that

$$d_{min}(v_1, v_n) = d(v_1, v_n) \tag{5.54}$$

the distance between v_1 and v_n is the minimal distance. By walking through the path p backwards, from v_n to v_1, we can find that the additive property can be applied as

$$\begin{aligned} d_{min}(v_1, v_n) &= d_{min}(v_1, v_{n-1}) + d_{min}(v_{n-1}, v_n) \\ &\dots \\ d_{min}(v_1, v_n) &= d_{min}(v_1, v_{n-2}) + d_{min}(v_{n-2}, v_n) \\ &\dots \\ d_{min}(v_1, v_n) &= d_{min}(v_1, v_2) + d_{min}(v_2, v_n) \end{aligned} \tag{5.55}$$

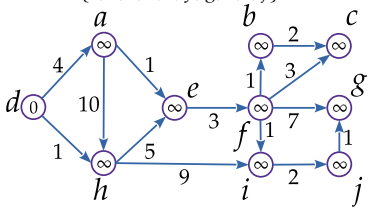
Algorithm 5.6: Dijkstra's shortest path algorithm

```
procedure DIJKSTRA(G, source, destination)
    for each vertex v in V(G) do
        d(v) ← ∞
        predecessor(v) ← null
    d(source) ← 0
    work ← V(G)
    while work in not empty do
        u ← take a vertex from work having minimal d(u)
        if u = destination then
            return
        for vertices v adjacent to u and in work do
            temp ← d(u) + w(uv)
            if temp < d(v) then
                d(v) ← temp
                predecessor(v) ← u
```

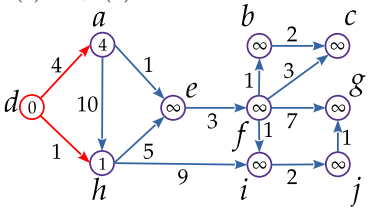
Complexity is $O(|V|^2)$

Let us explain Dijkstra's algorithm through an example. We take $source = d$ for the starting vertex and $destination = g$ for the destination vertex.

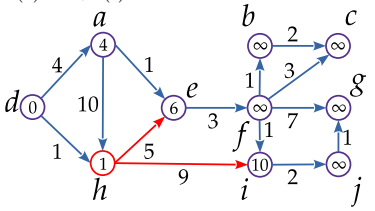
Iteration 0: The input graph G and initialization
$work = \{a, b, c, d, e, f, g, h, i, j\}$



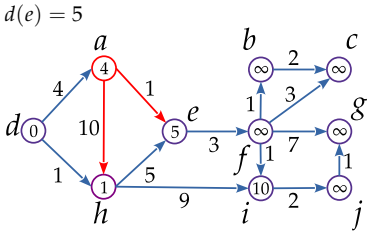
Iteration 1: $u = d$.
$work = \{a, b, c, e, f, g, h, i, j\}$
$d(a) = 4, d(h) = 1$



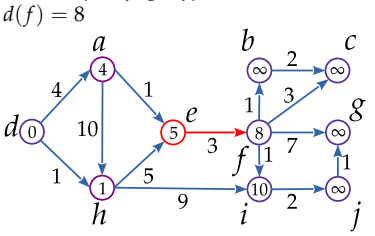
Iteration 2: $u = h$.
$work = \{a, b, c, e, f, g, i, j\}$
$d(e) = 6, d(i) = 10$



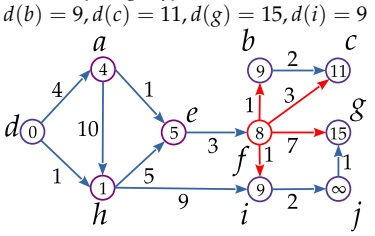
Iteration 3: $u = a$
$work = \{b, c, e, f, g, i, j\}$
$d(e) = 5$



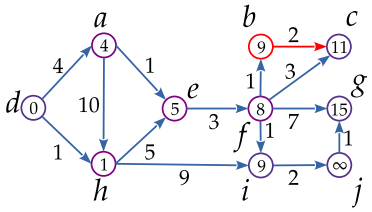
Iteration 4: $u = e$.
$work = \{b, c, f, g, i, j\}$
$d(f) = 8$



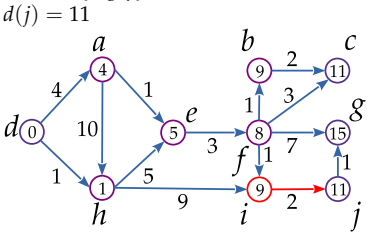
Iteration 5: $u = f$.
$work = \{b, c, g, i, j\}$
$d(b) = 9, d(c) = 11, d(g) = 15, d(i) = 9$



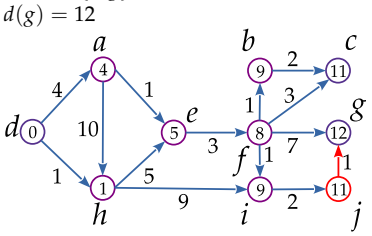
Iteration 6: $u = b$
$work = \{c, g, i, j\}$



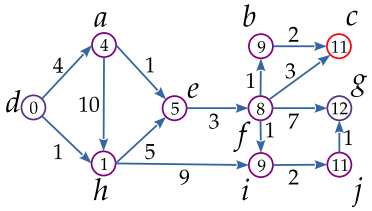
Iteration 7: $u = i$.
$work = \{c, g, j\}$
$d(j) = 11$



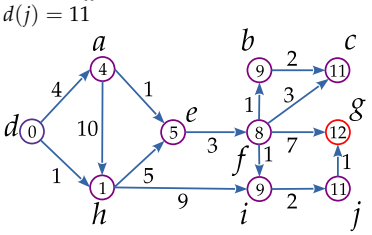
Iteration 8: $u = j$.
$work = \{c, g\}$
$d(g) = 12$



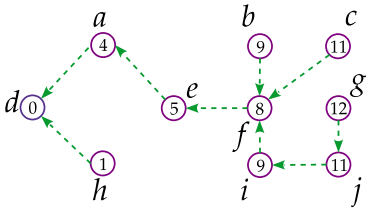
Iteration 9: $u = c$
$work = \{g\}$



Iteration 10: $u = g$.
$work = \{\}$
$d(j) = 11$



Final predecessors in the graph G.



The same example could be written in a form of an iterative table.

iteration current vertex	0	1 d	2 h	3 a	4 e	5 f	6 b	7 i	8 j	9 c	10 g
a	∞	$4/d$									
b	∞	∞	∞	∞	∞	$9/f$					
c	∞	∞	∞	∞	∞	$11/f$					
d	0										
e	∞	∞	$6/h$	$5/a$							
f	∞	∞	∞	∞	$8/e$						
g	∞	∞	∞	∞	∞	$15/f$	$15/f$	$15/f$	$12/j$		
h	∞	$1/d$									
i	∞	∞	$10/h$	$10/h$	$10/h$	$9/f$					
j	∞	∞	∞	∞	∞	∞	∞	$11/i$			

We pass the input graph G, the source vertex, and the destination vertex into the Dijkstra's algorithm. In the initialization part of the algorithm, we set all vertices distance to $d(v) = \infty$ except the source vertex, which is set to $d(source) = 0$, since we want to start with this vertex. We set a working list *work* to the list of all vertices in the input graph G. The *while* loop is executed as long as there are vertices in the working list. At the beginning of each iteration, we retrieve vertex that is currently the closest vertex to the source vertex, which becomes the current working vertex u. If the current working vertex u is the destination vertex, we exit the algorithm. If the destination vertex has not been reached, in the *for* loop we iterate through the adjacent vertices of the current working vertex u and set their labels if they are closer to the source vertex than previously calculated, but we also store the *predecessor* vertex to the current vertex u, which is done to build the shortest path. This way, according to (5.55), we can always reconstruct the shortest path backwards from the destination vertex.

```
def Dijkstra(G,s,d):
  W=edges(G)
  D={}
  for v in G:
    D[v]={'d':sys.maxsize,
          'p':None}
  D[s]['d']=0
  work=list(D.keys())
  while work:
    work=sorted(work,
        key=lambda x:D[x]['d'])
    u=work.pop(0)
    if u==d:
      return D
    for v in adj(G,u):
      temp=D[u]['d']+W[(u,v)]
      if temp<D[v]['d']:
        D[v]={'d':temp,'p':u}
  return D
```

Listing 5.7: Dijkstra python code. The code uses adjacency matrix graph representation.

Backwards reconstruction from the example iterative table between vertices $source = d$ and $destination = g$ gives us the shortest path

$$g \leftarrow j \leftarrow i \leftarrow f \leftarrow e \leftarrow a \leftarrow d \tag{5.56}$$

or

$$p = daefijg \tag{5.57}$$

which can be additionally visually verified in the final example Figure.

```
def RevDijkstra(D,d):
  P=[d]
  while D[d]['p']!=None:
    d=D[d]['p']
    P.append(d)
  return P
```

Listing 5.8: Dijkstra path reconstruction python code. The input parameter D is the *dictionary* returned from the *Dijkstra* function.

Complexity of the Dijkstra's shortest path algorithm as given in Algorithm 5.6 is $O(V^2)$. The reason for this quadratic complexity are two loops. The outer *while* loop must iterate through all vertices of the input graph G. The inner *for* loop must iterate through all adjacent vertices of the currently focused vertex. We can take a complete graph (definition 5.11) as the worst-case scenario, since each vertex has all other vertices as adjacent, which gives precisely V^2 iterations.

Bellman-Ford algorithm

Unlike Dijkstra's shortest path algorithm, Bellman-Ford algorithm is a *label-correcting* algorithm, meaning that the iteration through the input graph G continues until all vertex distances converge, or the iteration limit is reached. If the iteration limit is reached, and vertex distances still did not converge, we must conclude that there is a negative cycle in the input graph G. The total number of iterations of the Bellman-Ford algorithm is set to $|V| * |E|$.

procedure BELLMAN-FORD($G, source$)
  for each vertex v in $V(G)$ **do**
    $d(v) \leftarrow \infty$
    $predecessor(v) \leftarrow null$
  $d(source) \leftarrow 0$
  loop $|V(G)| - 1$ **times**
    for each edge $uv \in E(G)$ **do**
      if $d(u) + w(uv) < d(v)$ **then**
        $d(v) \leftarrow d(u) + w(uv)$
        $predecessor(v) \leftarrow u$
  for each edge $uv \in E(G)$ **do**
    if $d(u) + w(uv) < d(v)$ **then**
      raise exception 'negative cycle has been detected'

Algorithm 5.7: Standard Bellman-Ford shortest path algorithm

Complexity is $O(|V| * |E|)$

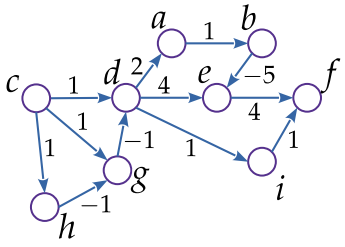


Figure 5.42: Weighted graph example.

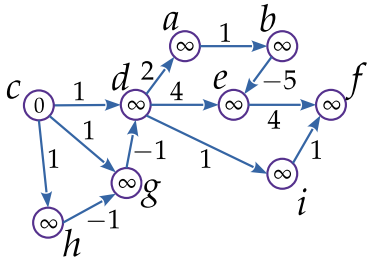
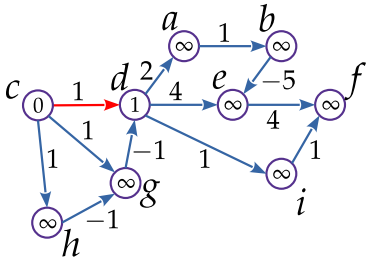
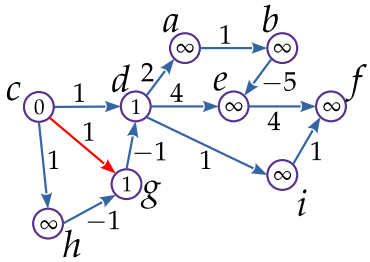
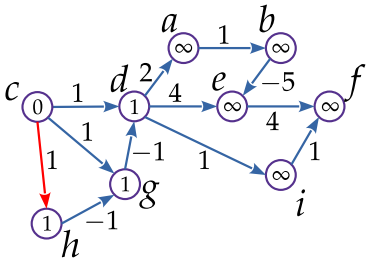
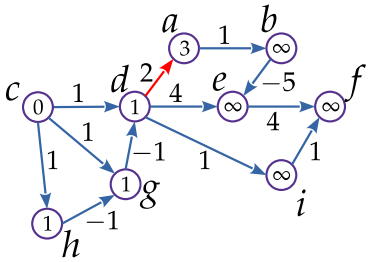
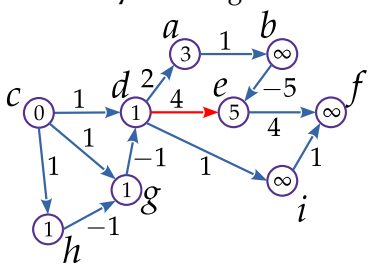
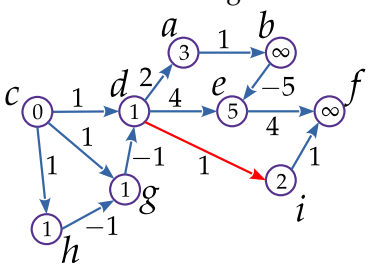
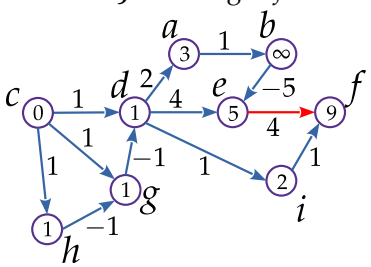
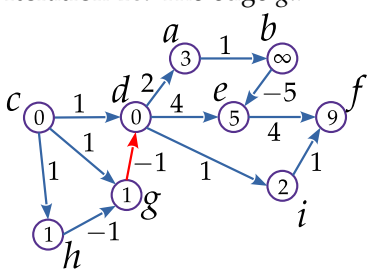
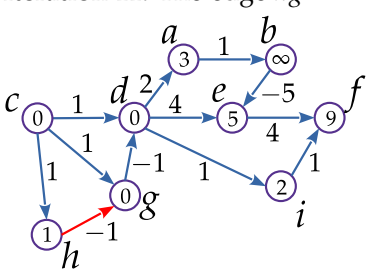
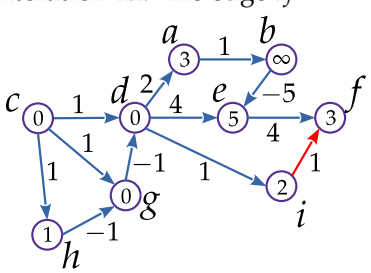
The Bellman-Ford algorithm is clearly slower than the Dijkstra's algorithm, but has the advantage of being able to work with graphs that have negative weights. In the context of the Algorithm 5.5 used to calculate all-to-all distances in the input graph G, the Bellman-Ford algorithm is usually used in the step 1, where we need to calculate initial distances of the original input graph G, and Dijkstra's algorithm is used in the step 4, where we need to find the shortest path between each pair of vertices in the transformed graph G'.

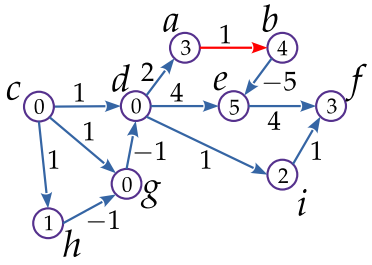
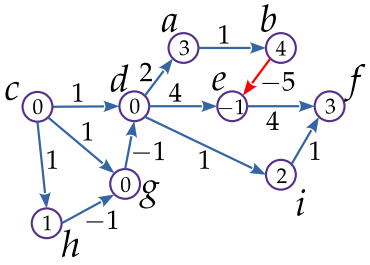
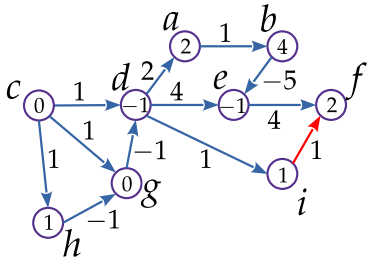
We do not pass the *destination* vertex to the Bellman-Ford algorithm, as it calculates the distance between the *source* vertex and all other vertices in the input graph G. Simply, the Bellman-Ford algorithm does not have a similar exit condition as in the Dijkstra's algorithm.

For the example we use the graph presented in Figure 5.42. We first sort edges of the input graph G to be in some order, such as

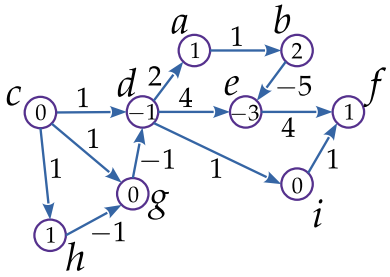
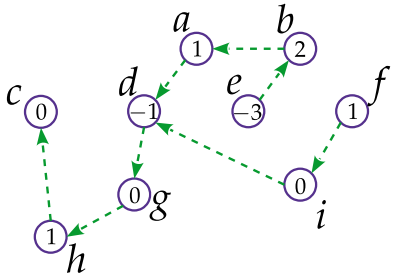
$$E(G) = \{ab, be, cd, cg, ch, da, de, di, ef, gd, hg, if\} \tag{5.58}$$

Using this ordering, we want to calculate the distance from the *source* $= c$ vertex to all other vertices in the graph G.

<table>
<tr><th colspan="3">Outer loop iteration 1</th></tr>
<tr>
<td>Iteration 0: The input graph G and initialization
  </td>
<td>Iterations 1 and 2 are ineffective, since edges ab and be do not update any vertex distances.</td>
<td>Iteration 3: The edge cd.
  </td>
</tr>
<tr>
<td>Iteration 4: The edge cg
  </td>
<td>Iteration 5: The edge ch
  </td>
<td>Iteration 6: The edge da.
  </td>
</tr>
<tr>
<td>Iteration 7: The edge de
  </td>
<td>Iteration 8: The edge di
  </td>
<td>Iteration 9: The edge ef.
  </td>
</tr>
<tr>
<td>Iteration 10: The edge gd
  </td>
<td>Iteration 11: The edge hg
  </td>
<td>Iteration 12: The edge if.
  </td>
</tr>
</table>

<table>
<tr><th colspan="3">Outer loop iteration 2</th></tr>
<tr><td>Iteration 1: The edge ab
  </td><td>Iteration 2: The edge be
  </td><td>We skip a number of the inner edge iterations here.</td></tr>
<tr><td>Iteration 12: The edge if
  </td><td></td><td></td></tr>
</table>

Ultimately, after $|V| - 1$ iterations of the outer loop, we reach the following graph

<table>
<tr><td>The final graph
  </td><td>Predecessors
  </td></tr>
</table>

The same example could be written in a form of an iterative table.

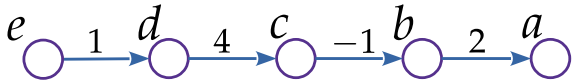
	Iteration					
	0	1		2	3	4
a	∞	3		2	1	
b	∞			4	3	2
c	0					
d	∞	1	0	-1	-1	
e	∞	5		-1	-2	-3
f	∞	9	3	2	1	
g	∞	1	0			
h	∞	1				
i	∞	2		1	0	

The outer loop of the Bellman-Ford algorithm has purpose to limit the number iterations we do for calculating distances. If distances don't converge within this limit, we can assume that we deal with a graph that has a negative cycle. The inner *for* loop iterates through all edges and updates vertex distances. After we finished the $|V| - 1$ iterations of the outer loop, we can test whether distances converged or not. This is done by passing through all edges and testing whether we can still update another vertex distance. If such update is possible, then distances did not converge and we can conclude that there

is a negative cycle in the input graph G.

The main question is, how do we know that $|V| - 1$ iterations of the outer loop is sufficient to reach the convergence. The answer is in the following example.

Exercise 5.3.1. Using Bellman-Ford algorithm, determine distances in the following graph G



having $source = a$. Use the edge ordering as $E(G) = \{ab, bc, cd, de\}$. Count the number of iterations of the outer loop needed to update vertex e.

	Iteration				
	0	1	2	3	4
a	0				
b	∞	2			
c	∞	∞	1		
d	∞	∞	∞	5	
e	∞	∞	∞	∞	6

Table 5.4: Exercise 5.3.1 solution.

Complexity of the Bellman-Ford algorithm depends on two loops. The outer loop depends on the number of vertices $|V|$. The inner *for* loop depends on the number of edges $|E|$. The complexity of the Bellman-Ford algorithm is then $O(|V| * |E|)$. If we take the worst-case scenario, which is again a complete graph (definition 5.11), according to (5.17), the complexity of the Bellman-Ford algorithm in such case is $O(|V|^3)$, which is much higher than the Dijkstra's shortest path algorithm. This is the reason why in some cases we want to use Algorithm 5.5 for calculating all-to-all distances.

```
def BellmanFord(G,s):
 W=edges(G)
 D={}
 for v in G:
   D[v]={'d':sys.maxsize,'p':None}
 D[s]['d']=0
 for i in range(1,len(D)):
  for e in W:
   if D[e[0]]['d']+W[e]<D[e[1]]['d']:
     D[e[1]]={'d':D[e[0]]['d']+W[e],
              'p':e[0]}
 for e in W:
  if D[e[0]]['d']+W[e]<D[e[1]]['d']:
   raise Exception('negative cycle')
 return D
```

Listing 5.9: Bellman-Ford python code. The code uses adjacency matrix graph representation.

Faster variant of the Bellman-Ford algorithm

We already noticed in the example we used to illustrate the standard Bellman-Ford algorithm that we could iterate through some edges without updating vertex distances. These are *wasted computations*, as we iterate through the inner *for* loop without knowing that this specific iteration will be effective. Moreover, if we did not update a specific vertex, this will most certainly cause that its subgraph distances don't need updating as well.

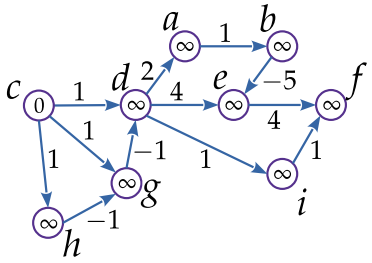
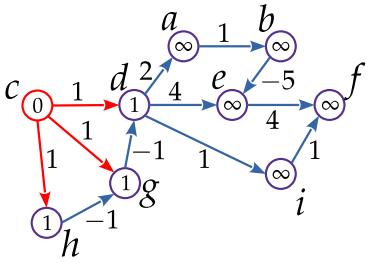
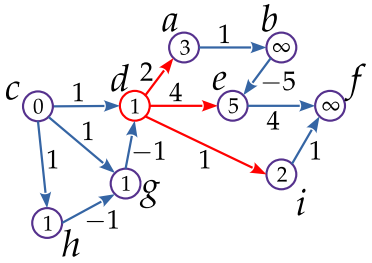
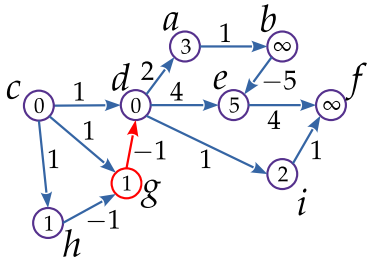
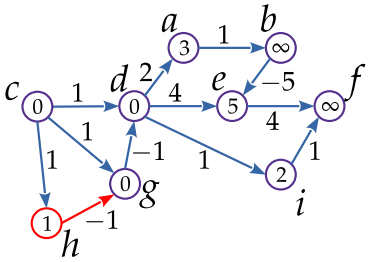
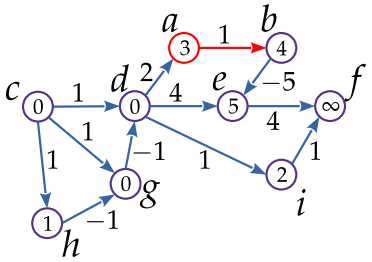
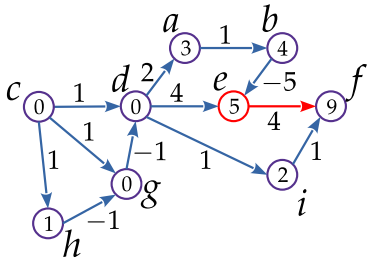
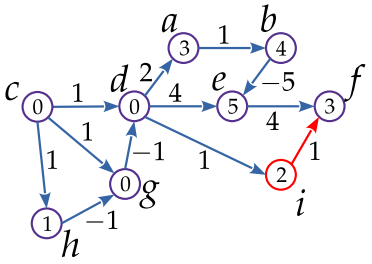
procedure BELLMAN-FORD(G, *source*)
  for each vertex v in $V(G)$ **do**
    $d(v) \leftarrow \infty$
    $predecessor(v) \leftarrow null$
  $d(source) \leftarrow 0$
  Q $\leftarrow$ empty queue
  Q.enqueue(*source*)
  while Q is not empty **do**
    $u \leftarrow Q.dequeue$
    for v in adjacent vertices of u **do**
      if $d(u) + w(uv) < d(v)$ **then**
        $d(v) \leftarrow d(u) + w(uv)$
        $predecessor(v) \leftarrow u$
        if v not in Q **then**
          Q.enqueue(v)

Algorithm 5.8: Faster variant of the Bellman-Ford shortest path algorithm

```
def BellmanFordFast(G,s):
 W=edges(G)
 D={}
 for v in G:
   D[v]={'d':sys.maxsize,'p':None}
 D[s]['d']=0
 Q=collections.deque()
 Q.append(s)
 while Q:
   u=Q.popleft()
   for e in W:
    if D[e[0]]['d']+W[e]<D[e[1]]['d']:
      D[e[1]]={'d':D[e[0]]['d']+W[e],
               'p':e[0]}
       if e[1] not in Q:
         Q.append(e[1])
 for e in W:
  if D[e[0]]['d']+W[e]<D[e[1]]['d']:
   raise Exception('negative cycle')
 return D
```

Listing 5.10: Fast Bellman-Ford python code. The code uses adjacency matrix graph representation.

To circumvent this issue, we can use the approach shown in the BFS algorithm. Basic idea behind this approach is to prune all subgraphs of vertices that have not been updated. We use the queue mechanism to store each vertex that was updated, which most certainly will lead to updating distances in their subgraphs.

<table>
<tr>
<td>Iteration 0: Initialization
After: $Q = \{c\}$
  </td>
<td>Before: $Q = \{c\}$
Iteration 1: $u = c$
After: $Q = \{d, g, h\}$
  </td>
<td>Before: $Q = \{d, g, h\}$
Iteration 2: $u = d$.
After: $Q = \{g, h, a, e, i\}$
  </td>
</tr>
<tr>
<td>Before: $Q = \{g, h, a, e, i\}$
Iteration 3: $u = g$
After: $Q = \{h, a, e, i, d\}$
  </td>
<td>Before: $Q = \{h, a, e, i, d\}$
Iteration 4: $u = h$
After: $Q = \{a, e, i, d, g\}$
  </td>
<td>Before: $Q = \{a, e, i, d, g\}$
Iteration 5: $u = a$.
After: $Q = \{e, i, d, g, b\}$
  </td>
</tr>
<tr>
<td>Before: $Q = \{e, i, d, g, b\}$
Iteration 6: $u = e$
After: $Q = \{i, d, g, b, f\}$
  </td>
<td>Before: $Q = \{i, d, g, b, f\}$
Iteration 7: $u = i$
After: $Q = \{d, g, b, f\}$
  </td>
<td>We continue this until there are vertices in the queue Q. The result must be the same as with standard Bellman-Ford algorithm.</td>
</tr>
</table>

The problem with this faster variant of the Bellman-Ford algorithm arises when we have a negative cycle in the input graph G. In such case, distance updating continues endlessly, which means that the queue Q will never get empty. This can be solved by adding a counter in the algorithm, limited to the same limit as in the standard algorithm variant $|V| * |E|$. After updating $|V| * |E|$ vertex distance, we exit by raising the negative cycle exception.

Warshall-Floyd-Ingerman algorithm

The Warshall-Floyd-Ingerman (WFI) algorithm is a *label-correcting* algorithm capable of calculating all-to-all distances and paths in one execution. The main characteristic of the WFI algorithm is the use of

the intermediate vertices for the calculation. Let us define a graph

$$\begin{aligned} G &= (V, E) \\ V &= \{v_1, v_2, ..., v_n\} \end{aligned} \tag{5.59}$$

Consider having an arbitrary set of vertices, such as $V = \{a, b, c, d, e\}$. This can be mapped to the previous graph definition as

$$v_1 = a, v_2 = b, v_3 = c, v_4 = d, v_5 = e \tag{5.60}$$

In the WFI algorithm we use the *weighted adjacency matrix* graph representation (Definition 5.33), here called the *distance matrix*. In case of our example, this would be for example

$$\mathbf{D} = \begin{array}{c} \\ v_1 = a \\ v_2 = b \\ v_3 = c \\ v_4 = d \\ v_5 = e \end{array} \begin{array}{c} \begin{array}{ccccc} v_1 = a & v_2 = b & v_3 = c & v_4 = d & v_5 = e \end{array} \\ \begin{bmatrix} d_{11} & d_{12} & d_{13} & d_{14} & d_{15} \\ d_{21} & d_{22} & d_{23} & d_{24} & d_{25} \\ d_{31} & d_{32} & d_{33} & d_{34} & d_{35} \\ d_{41} & d_{42} & d_{43} & d_{44} & d_{45} \\ d_{51} & d_{52} & d_{53} & d_{54} & d_{55} \end{bmatrix} \end{array}$$

where d_{ij} is the distance between vertices v_i and v_j. The basic idea of the WFI algorithm is to explore all vertices from v_1 to v_n and see if they are on the shortest path between vertices v_i and v_j. For this reason, we choose an intermediate vertex v_k such that $1 \leq k \leq n$. As with all other algorithms, in case we have

$$d(v_i, v_k) + d(v_k, v_j) < d(v_i, v_j) \tag{5.61}$$

then we are clear that the intermediate vertex v_k is on the shortest path between v_i and v_j. We need to iterate through the vertex set $\forall v_k \in V$ to test all vertices as the intermediate vertex v_k. This can be done using an ordinary *for* loop

for k from 1 to n **do**
  $v_k = V(G)[k]$

Distance calculation can be then generally written recursively as

$$d_{ij}^k = \begin{cases} w_{ij}, & k = 0 \\ min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}), & k \geq 1 \end{cases} \tag{5.62}$$

where d_{ij}^k is the k-th step in the distance calculation between vertices v_i and v_j, done for the intermediate vertex v_k. Reconstructing the shortest path between vertices v_i and v_j is similar as for the other shortest path distance algorithms. As soon as we update distance d_{ij}^k, we need to update the predecessor for the vertex v_j. This can also be expressed through a recursive function

$$\pi_{ij}^0 = \begin{cases} null, & i = j \text{ or } \nexists v_i v_j \in E \\ i, & i \neq j \text{ and } \exists v_i v_j \in E \end{cases} \tag{5.63a}$$

$$\pi_{ij}^k = \begin{cases} \pi_{ij}^{k-1}, & d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{kj}^{k-1} \\ \pi_{kj}^{k-1}, & d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1} \end{cases} \tag{5.63b}$$

defining values that form the *path matrix* Π.

For the example in Figure 5.43 we start with initial distance matrix D^0, where we use $k = 0$ (see (5.62)). All elements of the initial distance matrix D^0 are set to $d^0_{ij} = \infty$ if there is no edge connecting vertices v_i and v_j, otherwise we set $d^0_{ij} = w_{ij}$ and $d^0_{ii} = 0$. The element of the initial path matrix Π^0 are set according to (5.63a). Using mapping from (5.60), we get the following initial distance and path matrices

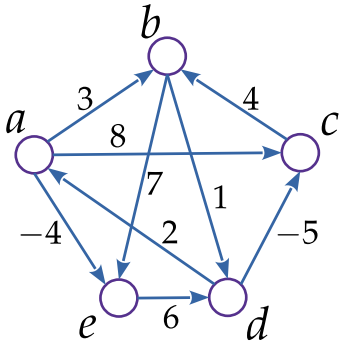


Figure 5.43: Weighted graph example.

$$\mathbf{D^0} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \end{array}, \mathbf{\Pi^0} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} null & 1 & 1 & null & 1 \\ null & null & null & 2 & 2 \\ null & 3 & null & null & null \\ 4 & null & 4 & null & null \\ null & null & null & 5 & null \end{bmatrix} \end{array}$$

For $k = 1$, the value d^0_{ik} resembles the first column and d^0_{kj} the first row of the distance matrix D^0. In this iteration we calculate the intermediate vertex $v_k = a$. We go trough the distance matrix D^0 trying to find the condition in (5.62). In our case se have new values in $d^1_{42} = d^0_{41} + d^0_{12} = 5$, which was $d^0_{42} = \infty$, and $d^1_{45} = d^0_{41} + d^0_{15} = -2$, which was $d^0_{45} = \infty$. We also need to update the path matrix with $\pi^1_{42} = \pi^0_{12} = 1$ and $\pi^1_{45} = \pi^0_{15} = 1$. New distance and path matrices for $k = 1$ are

$$\mathbf{D^1} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5(\infty) & -5 & 0 & -2(\infty) \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \end{array}, \mathbf{\Pi^1} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} null & 1 & 1 & null & 1 \\ null & null & null & 2 & 2 \\ null & 3 & null & null & null \\ 4 & 1(null) & 4 & null & 1(null) \\ null & null & null & 5 & null \end{bmatrix} \end{array}$$

For $k = 2$, the value d^1_{ik} resembles the second column and d^1_{kj} the second row of the distance matrix D^1. In this iteration we calculate the intermediate vertex $v_k = b$. In this case se have new values in $d^2_{14} = d^1_{12} + d^1_{24} = 4$, which was $d^1_{14} = \infty$, $d^2_{34} = d^1_{32} + d^1_{24} = 5$, which was $d^1_{34} = \infty$, and $d^2_{35} = d^1_{32} + d^1_{25} = 11$, which was $d^1_{35} = \infty$. We also need to update the path matrix with $\pi^2_{14} = \pi^1_{24} = 2$, $\pi^2_{34} = \pi^1_{24} = 2$, and $\pi^2_{35} = \pi^1_{25} = 2$. New distance and path matrices for $k = 2$ are

$$\mathbf{D^2} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 3 & 8 & 4(\infty) & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5(\infty) & 11(\infty) \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \end{array}, \mathbf{\Pi^2} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} null & 1 & 1 & 2(null) & 1 \\ null & null & null & 2 & 2 \\ null & 3 & null & 2(null) & 2(null) \\ 4 & 1 & 4 & null & 1 \\ null & null & null & 5 & null \end{bmatrix} \end{array}$$

For $k = 3$, the value d^2_{ik} resembles the third column and d^2_{kj} the third row of the distance matrix D^2. In this iteration we calculate the

intermediate vertex $v_k = c$. All calculations are the same as in the previous steps. New distance and path matrices for $k = 3$ are

$$\mathbf{D^3} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1(5) & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \end{array}, \mathbf{\Pi^3} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} null & 1 & 1 & 2 & 1 \\ null & null & null & 2 & 2 \\ null & 3 & null & 2 & 2 \\ 4 & 3(1) & 4 & null & 1 \\ null & null & null & 5 & null \end{bmatrix} \end{array}$$

For $k = 4$, the value d^3_{ik} resembles the fourth column and d^3_{kj} the fourth row of the distance matrix D^3. In this iteration we calculate the intermediate vertex $v_k = d$. All calculations are the same as in the previous steps. New distance and path matrices for $k = 4$ are

$$\mathbf{D^4} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 3 & -1(8) & 4 & -4 \\ 3(\infty) & 0 & -4(\infty) & 1 & -1(7) \\ 7(\infty) & 4 & 0 & 5 & 3(11) \\ 2 & -1 & -5 & 0 & -2 \\ 8(\infty) & 5(\infty) & 1(\infty) & 6 & 0 \end{bmatrix} \end{array}, \mathbf{\Pi^4} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} null & 1 & 4(1) & 2 & 1 \\ 4(null) & null & 4(null) & 2 & 1(2) \\ 4(null) & 3 & null & 2 & 1(2) \\ 4 & 3 & 4 & null & 1 \\ 4(null) & 3(null) & 4(null) & 5 & null \end{bmatrix} \end{array}$$

For $k = 5$, the value d^4_{ik} resembles the fifth column and d^4_{kj} the fifth row of the distance matrix D^4. In this iteration we calculate the intermediate vertex $v_k = e$. All calculations are the same as in the previous steps. New distance and path matrices for $k = 5$ are

$$\mathbf{D^5} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 1(3) & -3(-1) & 2(4) & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix} \end{array}, \mathbf{\Pi^5} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} null & 3(1) & 4(4) & 5(2) & 1 \\ 4 & null & 4 & 2 & 1 \\ 4 & 3 & null & 2 & 1 \\ 4 & 3 & 4 & null & 1 \\ 4 & 3 & 4 & 5 & null \end{bmatrix} \end{array}$$

Now we want do know the distance and the shortest path between vertices c and e. Getting the distance is relatively easy $d(c, e) = d^5_{35} = 3$. Getting the shortest path from the path matrix Π^5 requires several steps:

k	vertex
$\pi^5_{35} = 1$	$v_5 = e$
$\pi^5_{31} = 4$	$v_1 = a$
$\pi^5_{34} = 2$	$v_4 = d$
$\pi^5_{32} = 3$	$v_2 = b$
$\pi^5_{33} = null$	$v_3 = c$

which makes the shortest path $p_{min} = cbdae$. We repeat the same for vertices d and e. The distance is $d^5_{45} = -2$ and the shortest path is

k	vertex
$\pi^5_{45} = 1$	$v_5 = e$
$\pi^5_{41} = 4$	$v_1 = a$
$\pi^5_{44} = null$	$v_4 = d$

which is $p_{min} = dae$.

```
procedure WFI(W)
    Create initial distance matrix D = D^0 from W
    Create initial path matrix Π = Π^0 from W
    for k from 1 to |V| do
        for i from 1 to |V| do
            for j from 1 to |V| do
                if D[i,k] + D[k,j] < D[i,j] then
                    D[i,j] = D[i,k] + D[k,j]
                    Π[i,j] = Π[k,j]
```

Algorithm 5.9: WFI shortest path algorithm

The WFI algorithm can be used for some deeper analyses of the input graph G:

- When k reaches $|V|$, if there is still an element in the distance matrix $D^{|V|}$ being $d^{|V|}_{ij} = \infty$, this means that the vertex v_j is not reachable from the vertex v_i. This can be used to determine the input graph G *transitive closure*.
- If we create the initial distance matrix D^0 such that diagonal elements are set to $d^0_{ii} = \infty$, the WFI algorithm can detect cycles in the input graph G. After the WFI algorithm finishes, any diagonal element $d^{|V|}_{ii} \neq \infty$ represent the cycle starting and ending in this vertex v_i. However, in this case we know only the distance of the cycle, but not the cycle itself.

5.4 *Finding cycles, blocks and strongly connected components*

Finding cycles

Cycle detection is an important prerequisite for a number of algorithms, for example those that select the *minimum spanning tree* in a graph, to detect Eulerian or Hamiltonian cycles, and similar. The most pragmatic and easiest way to detect cycles in an input graph is to use one of the graph traversal algorithms as basis, DFS or BFS. A cycle appears each time graph traversal algorithm tries to visit an already visited vertex.

Such an approach has one major disadvantage for both directed and undirected graphs. As we traverse through the graph, we mark vertices as visited. If we leave these marks, after returning back from the recursive calls, and descending again through the graph structure, we can encounter a previously marked vertex again, which could lead us to a conclusion that there are cycles in the graph, while in fact there are none. An example can be seen in Figure 5.45. A

Complexity is $O(|V|^3)$

```
def WFI(G):
  D=G
  P=NewAdjMatrix(G)
  for u in G:
    for v in G:
      if D[u][v]: P[u][v]=u
      if u!=v and D[u][v]==0:
        D[u][v]=sys.maxsize
  for k in G:
    for i in G:
      for j in G:
        if D[i][k]+D[k][j]<D[i][j]:
          D[i][j]=D[i][k]+D[k][j]
          P[i][j]=P[k][j]
  return (D,P)
```

Listing 5.11: WFI python code. The code uses adjacency matrix graph representation. Function *NewAdjMatrix* creates a new *null* matrix having the same $n \times n$ dimensions as the original adjacency matrix.

```
def RevWFI(P,s,d):
  PR=[d]
  while P[s][d]!=0:
    d=P[s][d]
    PR.append(d)
  return PR
```

Listing 5.12: Path finding WFI python code. The input parameter is the Π matrix.

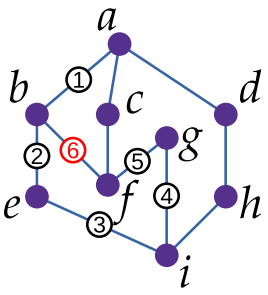


Figure 5.44: Using DFS to detect a cycle in an undirected graph. Recursive call ⑥ tries to visit an already visited vertex, which shows an existence of a cycle in the graph.

simple solution suggested by some scholars is to remove flags as we return back from the recursion. However, in the case of detecting cycles in a disconnected graph, this solution might disrupt our ability to process all graph partitions $P(V)$ (5.35). To solve this, we add a stack as an additional mechanism that will allow us to track only the current path. As we descend down the input graph, we add vertices to the stack, and remove them when returning back from the recursion.

Algorithm 5.10: Cycle finding DFS algorithm.

```
procedure FindCycle(G)
    initialize all vertices in G as not visited
    S ← empty stack
    while there is an unvisited vertex u_0 in G do
        FindCycle_recursive(G, u_0, S)

procedure FindCycle_recursive(G, u, S)
    mark u as visited
    S.push(u)
    for v in adjacent vertices of u do
        if v not in S then
            set predecessor of v to u
            FindCycle_recursive(G, v, S)
        else
            if predecessor of u is not v then
                initialize cycle c ← {}
                repeat
                    retrieve vertex vx backwards from the stack S
                                                    ▷ Do not pop edges
                    add vertex vx to the cycle c
                until vx = v
                report the cycle c
    S.pop()
```

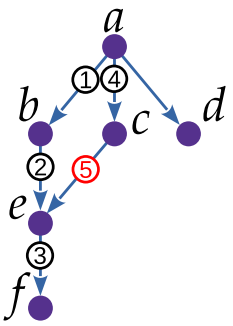


Figure 5.45: An issue with Algorithm 5.10 that could occur in directed graphs that due to the way of marking vertices as visited.

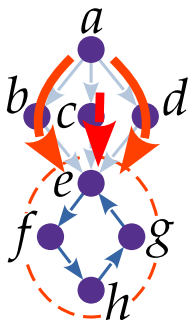


Figure 5.46: Rediscovering the same cycle multiple times.

The solution to track only the current path might result in rediscovering the same cycle multiple times. We can see such an example in Figure 5.46. The same cycle $efhge$ can be found in three distinct paths $abefhge$, $acefhge$, and $adefhge$.

Also, when traversing an undirected graph, we must know which was the last vertex we visited, so we don't take the same edge back to the vertex we came from. For this reason, we must track each vertex predecessor in the path we are traversing.

If we found a cycle, then we can start forming a set of cycle vertices retrieving them from the stack S backwards, until we find the same vertex v for which we detected the cycle. When forming the set of cycle vertices, we do not pop vertices from the stack S, as we want to keep the current path in the stack.

For the example in Figure 5.47, we have the course of execution as given in the following table. Iterations marked red are those in which we detect a cycle, by encountering the same vertex b we already have

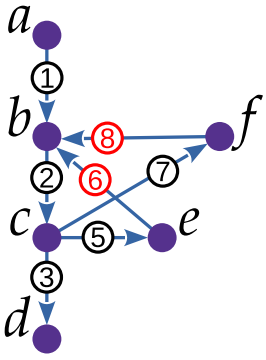


Figure 5.47: Graph for the cycle finding.

in the stack S. Vertices in the stack S marked red are the ones we retrieve backwards to form the detected cycle.

Iteration	1	2	3	4	5	6	7	8
vertex u	a	b	c	d	c	e	c	f
vertex v	b	c	d		e	b	f	b
stack S	a	b a	c b a	d c b a	e c b a	e c b a	f c b a	f c b a
cycle c						e, c, b		f, c, b

Programming tips: Accessing elements backwards from a stack without removing them is available in most of the programming languages. For example, if we use `collections.deque` in python, we can access the stack backwards through the `reversed` function, as shown in Listing 5.13. Other programming languages have similar features.

```
def FindCycles(G):
  visited=[]
  cycs=[]
  S=collections.deque()
  for uo in G:
    if uo not in visited:
      FindCycles_r(G,uo,visited,S,
                   cycs)
  return cycs

def FindCycles_r(G,u,visited,S,cycs):
  visited.append(u)
  S.append(u)
  for v in G[u]['adj']:
    if v not in S:
      G[v]['p']=u
      FindCycles_r(G,v,visited,S,
                   cycs)
    else:
      if v is not G[u]['p']:
        c = []
        for vx in reversed(S):
          c.append(vx)
          if vx is v:
            break
        if c not in cycs:
          cycs.append(c)
  S.pop()
```

Listing 5.13: Algorithm 5.10 python code. Uses the adjacent table graph representation.

Algorithm 5.10 represent a basis for algorithms that can be used to detect *biconnected components* in undirected graphs and *strongly connected components* in directed graphs.

Union-Find method for cycle detection in undirected graphs

Even if we loose all the unnecessary details from Algorithm 5.10, it still remains a DFS (5.3) in the nutshell, which is $O(|V| + |E|)$. We need a simple method for detecting cycles in undirected graphs. Let us say that the edges of the input undirected graph $G = (V, E)$ are coming one by one. At the beginning we have $|V|$ a set of disjoint-set trees, each tree representing its own vertex, as in Figure 5.48. Each disjoint-set tree consists of one or more vertices that are connected together by the graph edges, thus each tree represents one acyclic subgraph of the input undirected graph. All disjoint-set trees combined together are called a **disjoint-set forest**. As we add edges from the graph G, we can encounter two situations:

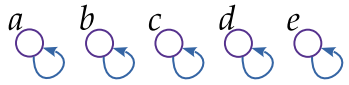
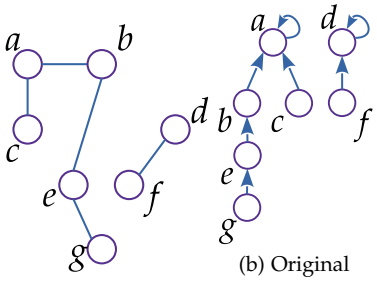
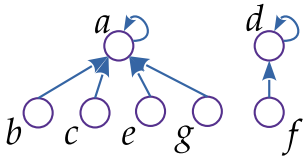


Figure 5.48: A set of disjoint trees, each for one vertex of an undirected graph.



(a) Input graph

(b) Original disjoint-set forest



(c) Compressed disjoint-set forest

Figure 5.49: Graph and disjoint-set forests examples..

1. The new edge is targeting vertices in two distinct disjoint-set trees, which means the edge is not forming a cycle in the graph. We can add this edge by merging targeted disjoint-set trees together.

2. The new edge is targeting vertices in a single set, which means that this edge would form a cycle in the graph. We report this edge as a cycle forming to the calling algorithm. Most of the algorithms that use *UnionFind* usually drop edges that form cycles, which means that there will be no change in the disjoint-set forest for this case.

Additionally, we can forcedly compress disjoint-set trees to depth 1. If we place all vertices of a disjoint-set tree directly under the root, which is used to identify the tree, the number of searches for the root vertex of the disjoint-set tree drops down to 1, which sets the complexity of this method to $O(1)$.

function MAKESET(F, x)
  if x not in F **then**
    $x.parent = x$
    add tree x to the forest F
  return F
function FIND(x)
  while $x \neq x.parent$ **do**
    $x.parent = x.parent.parent$ ▷ The compression
    $x = x.parent$
  return x
procedure UNION(x, y)
  $x = Find(x)$
  $y = Find(y)$
  if $x \neq y$ **then**
    $x.parent = y$

Algorithm 5.11: *UnionFind* algorithm.

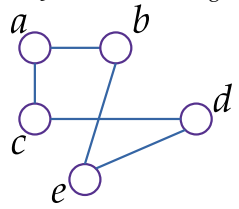
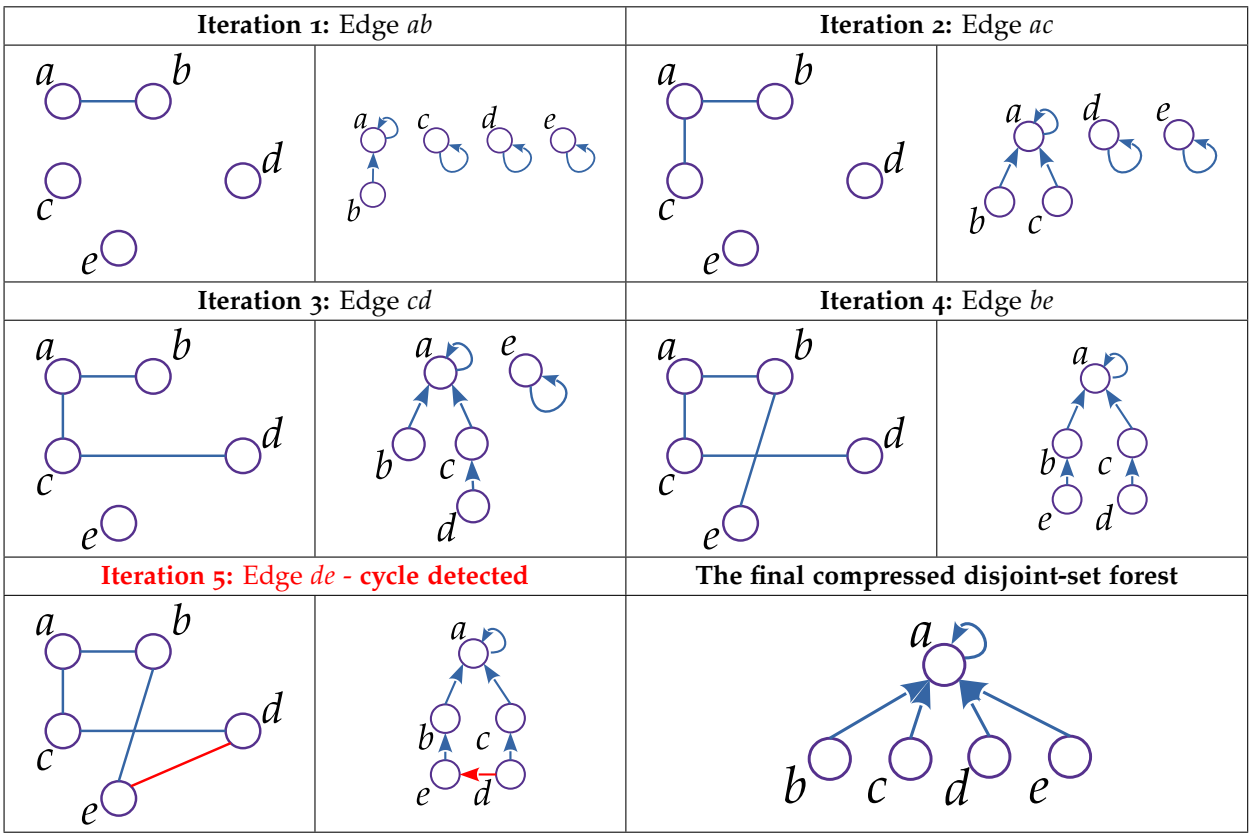


Figure 5.50: Undirected graph example for *UnionFind*.

```
def MakeSet(F,x):
  if x not in F:
    F[x]={'p':x}
  return F

def Find(F,x):
  while F[x]['p']!=x:
    F[x]['p']=F[F[x]['p']]['p']
    x=F[x]['p']
  return x

def Union(F,x,y):
  x=Find(F,x)
  y=Find(F,y)
  F[x]['p']=y
```

Listing 5.14: *UnionFind* python code.



Detecting biconnected components (blocks) in undirected graphs

We already defined the meaning of the articulation point (definition 5.24), bridge (definition 5.25), and a biconnected component (definition 5.26). A biconnected component as a subgraph G' of the **undirected connected** graph G should not have articulation points, meaning that if we remove any of the vertices from $V(G')$, the biconnected component must remain connected. Looking from the graph G perspective, containing biconnected components are connected using articulation points and bridges. The goal of detecting biconnected components in the graph G is to find articulation points, which is a similar problem to finding cycles.

When traversing the graph G using DFS, we are creating a correspondent spanning tree. In such spanning tree, an articulation point is a vertex that has no edge to its predecessors in the tree, but has a returning edge from the successor subtree. The articulation point is the *top* vertex in the spanning tree that forms a cycle in its successor subtree. Such articulation point is taken as the root of the biconnected component.

When we look at Figure 5.51, we can intuitively see c as the single articulation point. This divides the graph into 3 biconnected components. Traversing this graph using DFS gives us a spanning tree shown in Figure 5.52. We can see that there is a subtree

$$G' = (V = \{c, d, e, f\}, E = \{cd, de, ef, fc\}) \tag{5.64}$$

forming a cycle, which has the vertex c as the top vertex in the spanning tree. We can also see that c has no returning edges to any of its predecessors. This makes vertex c an articulation point.

If we enumerate vertices in the spanning tree, we can track the last predecessor of the spanning subtree. We define a graph G as

$$G = (V, E, p) \tag{5.65}$$

where p is the enumeration and predecessor mapping function

$$p : V \to \mathbb{N} \times \mathbb{N} \tag{5.66}$$

We assign each vertex an ordered pair (n, p), where n is the vertex unique number and p is the number of its top predecessor. It must be ensured that

$$\nexists u, v \in V : u \neq v \wedge n(u) = n(v) \tag{5.67}$$

The procedure of detecting biconnected components is similar to detecting cycles. As soon as we detect an edge leading to an already visited vertex of the graph G, we need to update predecessor numbers. A predecessor of a vertex $v \in V$ is defined as

$$p(v) = min(n(v), n(u_1), n(u_2), ..., n(u_k)) \tag{5.68}$$

where $u_1, ..., u_k$ are vertices we are returned to from the subtree of the vertex v. For example in Figure 5.52, if we take $v = d$, its subtree are

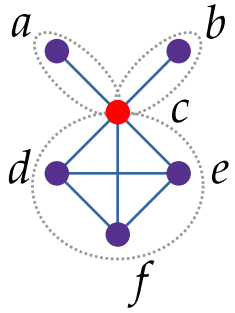


Figure 5.51: An undirected graph with biconnected components.

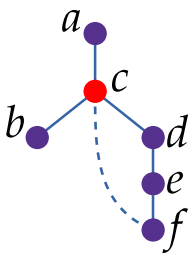


Figure 5.52: A spanning tree from the graph in Figure 5.51. We draw the returning edge fc dashed, since this edge would violate the spanning tree definition.

vertices $\{d, e, f\}$. All these vertices are having a returning edge to the vertex c, which means $u_1 = c$. This results in $p(d) = min(n(d), n(c))$. Equivalently, we have

$$p(v) = min(n(v), p(w_1), p(w_2), ..., p(w_k)) \tag{5.69}$$

where $w_1, ..., w_k$ are vertices in the subtree of the vertex v.

Algorithm 5.12: Block search algorithm.

```
procedure BlockSearch(G)
    initialize all vertices in G as n(v) = 0
    step ← 1
    S ← empty stack
    while there is a vertex u in G, having n(u) = 0 do
        BlockSearch_recursive(G, u, step, S)

procedure BlockSearch_recursive(G, u, step, S)
    p(u) = n(u) = step
    step ← step + 1
    for v in adjacent vertices of u do
        if n(v) = 0 then
            if not edge uv on the stack S then
                S.push(uv)
            BlockSearch_recursive(G, v, step, S)
            if p(v) ≥ n(u) then
                pop edges until uv and form a block
            else
                p(u) = min(p(u), p(v))
        else if S is not empty and vu is not the last element then
            p(u) = min(p(u), n(v))
```

For the graph in Figure 5.51 we have

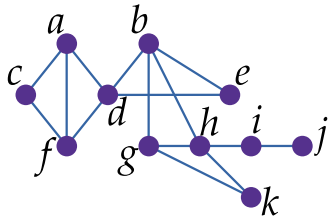
	0	1	2	3	4	5	6	7	8	9	10	11	12
u		a	c	b	c	c	d	e	f	e	d	c	a
v		c	b		b	d	e	f	c	f	e	d	c
	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p
a	0/0	1/1											pop
b	0/0			3/3									
c	0/0		2/2		pop							pop	
d	0/0						4/4				4/2		
e	0/0							5/5		5/2			
f	0/0								6/6 6/2				
stack S		ac	cb ac	cb ac	ac	cd ac	de cd ac	ef de cd ac	ef de cd ac	ef de cd ac	ef de cd ac	ac	

The starting vertex is determined by the *while* loop in the first *BlockSearch* procedure. Usually, this ordering is alphabetical, but this can change and depends on the implementation. We start from the vertex a and descend into the graph. Along with descending, we update vertex numbers, predecessors, and push edges we traverse to the stack S. This occurs until we have no more adjacent vertices, for example when we reached vertices b or f in iterations 3 and 8 respectively. In this case we start returning back from the recursion, which is marked with red iteration numbers. When we encounter a returning edge, such as the edge fc in the iteration 8, we need to update predecessors of the current subtree. For the returning edge fc, we need to update the predecessor of the vertex f to $p(f) = n(c) = 2$. At this point, we also start to return from the recursion, where we update the predecessors of vertices until we reach the vertex c and edge cd. At this point $p(d) \geq n(c)$ (iteration 11) becomes true, which means that we returned back to the articulation point and we need to form a biconnected component by poping edges from the stack S. The first component is made of vertices in the edge set $\{ef, de, cd\}$. As we return from the recursion (at any point), we find that $p(c) \geq n(b)$ (iteration 4) and $p(a) \geq n(c)$ (iteration 12), which gives us two more components from the edge sets $\{ac\}$ and $\{cb\}$. Finally, we got the following biconnected components

$$BC_1 = G[V = \{a, c\}]$$
$$BC_2 = G[V = \{c, b\}]$$
$$BC_3 = G[V = \{c, d, e, f\}]$$

which are induced subgraphs of the input graph G. These biconnected components can be visually verified in Figure 5.51.

Exercise 5.4.1. Using block search algorithm, starting from a, find the biconnected components (blocks) in the following undirected graph.



Detecting strongly connected components (SCC) in directed graphs

We defined strongly connected components in the overview of the graph theory (definition 5.27). As we already mentioned, detecting an SCC is an NP-complex problem as we need to check the mutual reachability for each pair of vertices in the input **directed** graph G. This could be relaxed by knowing that in each cycle of the input directed graph, similar to (5.37), all vertices are mutually reachable. Remember, a cycle is a closed walk having all vertices unique.

```
def BlockSearch(G):
  for v in G:
    G[v]['n']=G[v]['p']=0
  step = 0
  S = collections.deque()
  for u in G:
    if (G[u]['n']==0):
      BlockSearch_r(G,u,step,S)

def BlockSearch_r(G,u,step,S):
  G[u]['p']=G[u]['n']=step
  step=step+1
  for v in G[u]['adj']:
    if (G[v]['n']==0):
      if((u,v) not in S and
         (v,u) not in S):
        S.append((u,v))
      BlockSearch_r(G,v,step,S)
      if(G[v]['p']>=G[u]['n']):
        b=[]
        e=S[-1]
        while(e!=(u,v) and
              e!=(v,u)):
          b.append(S.pop())
          e=S[-1]
        b.append(S.pop())
        print(b)
      else:
        G[u]['p']=min(G[u]['p'],
                      G[v]['p'])
    elif (len(S)>0 and
          (v,u)!=S[-1]):
      G[u]['p']=min(G[u]['p'],
                    G[v]['n'])
```

Listing 5.15: Algorithm 5.12 python code.

Solution is

$$BC_1 = G[V = \{a, c, f, d\}]$$
$$BC_2 = G[V = \{d, b, e\}]$$
$$BC_3 = G[V = \{b, g, h, k\}]$$
$$BC_4 = G[V = \{h, i\}]$$
$$BC_5 = G[V = \{i, j\}]$$

In Figure 5.53, we can observe the following 3 cycles

$$c_1 = acdba$$
$$c_2 = efge$$
$$c_3 = fghf$$

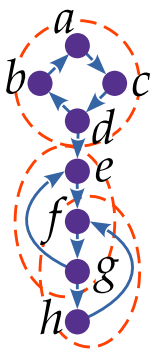


Figure 5.53: Cycles in a directed graph.

Looking within cycles, we can see that all vertices are mutually reachable, which makes each cycle consistent to the SCC definition. Carefully looking at cycles c_2 and c_3, we can see that they overlap by sharing vertices and edges. This way, all vertices in both c_2 and c_3 are mutually reachable, making them both constituting a single SCC.

$$scc_1 = c_1$$
$$scc_2 = c_2 \cup c_3$$

Therefore, unlike biconnected components sharing the same articulation point, two distinct SCC cannot share the same vertex as this makes them an unique SCC. Detecting and capturing cycles is the basis of the SCC detection. In this case we also use the same approach as in Algorithm 5.12. The top node in a subtree that constitutes an SCC is called the SCC root vertex.

Algorithm 5.13: Tarjan's algorithm.

procedure SCCSEARCH(G)
  initialize all vertices in G as $n(v) = 0$
  $step \leftarrow 1$
  $S \leftarrow$ empty stack
  while there is a vertex u in G, having $n(u) = 0$ **do**
    SCCSearch_recursive($G, u, step, S$)

procedure SCCSEARCH_RECURSIVE($G, u, step, S$)
  $p(u) = n(u) = step$
  $step \leftarrow step + 1$
  $S.push(u)$
  for v in adjacent vertices of u **do**
    if $n(v) = 0$ **then**
      $SCCSearch\_recursive(G, v, step, S)$
      $p(u) = min(p(u), p(v))$
    else if $n(v) < n(u)$ and v is on the stack S **then**
      $p(u) = min(p(u), n(v))$
  if $p(u) = n(u)$ **then** ▷ this is the SCC root vertex
    pop vertices from the stack S until u is popped off

```
def SCCSearch(G):
  for v in G:
    G[v]['n']=G[v]['p']=0
  step=0
  S=collections.deque()
  for u in G:
    if (G[u]['n']==0):
      SCCSearch_r(G,u,step,S)

def SCCSearch_r(G,u,step,S):
  G[u]['p']=G[u]['n']=step
  step=step+1
  S.append(u)
  for v in G[u]['adj']:
    if (G[v]['n']==0):
      SCCSearch_r(G,v,step,S)
      G[u]['p']=min(G[u]['p'],
                    G[v]['p'])
    elif (G[v]['n']<G[u]['n']):
      G[u]['p']=min(G[u]['p'],
                    G[v]['n'])
  if (G[u]['p']==G[u]['n']):
    scc=[]
    vx=S[-1]
    while(vx!=u):
      scc.append(S.pop())
      vx=S[-1]
    scc.append(S.pop())
    print(scc)
```

Listing 5.16: Tarjan's algorithm in python code.

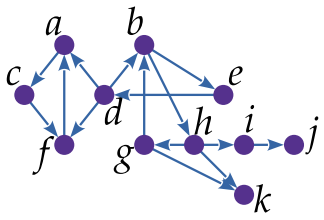
If we apply Tarjan's algorithm to the example in Figure 5.53, we get the execution course given in the next table. Iterations marked red are those occurring when we return from the recursion. Also, vertices marked red in the stack S row are those being popped off by the algorithm.

u v	0 n/p	1 a c n/p	2 c d n/p	3 d b n/p	4 b a n/p	5 d b n/p	6 d e n/p	7 e f n/p	8 f g n/p	9 g e n/p	10 g h n/p	11 h f n/p	12 g h n/p	13 f g n/p	14 e f n/p	15 d e n/p	16 c d n/p	17 a c n/p
a	0/0	1/1																pop
b	0/0				4/4 4/1													
c	0/0		2/2														2/1	
d	0/0			3/3		3/1												
e	0/0							5/5							pop			
f	0/0								6/6					6/5				
g	0/0									7/7 7/5								
h	0/0											8/8 8/6						
stack S		a	c a	d c a	b d c a	b d c a	b d c a	e b d c a	f e b d c a	g f e b d c a	g f e b d c a	h g f e b d c a	h g f e b d c a	h g f e b d c a	h g f e b d c a	b d c a	b d c a	b d c a

From which vertex we start determines the *while* loop in the first *SCCSearch* procedure. Usually this ordering is alphabetical, but this can change and depends on the implementation. We start from the vertex a and descend into the graph. The first significant situation is to encounter a returning edge, which ends up in adjusting the current vertex u predecessor value. This can be seen in iterations 4, 9, and 11. Another situation is returning from the recursion, where the predecessor of the vertex v is smaller than the predecessor of the current vertex u, which also leads to the predecessor adjustment. This can be seen in iterations 5, 13, and 16. Eventually, we pop off the stack S the following SCCs

$$scc_1 = G[V = \{h, g, f, e\}]$$
$$scc_2 = G[V = \{b, d, c, a\}]$$

Exercise 5.4.2. Using Tarjan's algorithm, starting from a, find the SCCs in the following directed graph.



Solution is

$$SCC_1 = G[V = \{a, f, c\}]$$
$$SCC_2 = G[V = \{k\}]$$
$$SCC_3 = G[V = \{j\}]$$
$$SCC_4 = G[V = \{i\}]$$
$$SCC_5 = G[V = \{g, h, d, e, b\}]$$

5.5 Minimal spanning trees (MST)

We already defined what spanning trees are (definition 5.34). Based on this definition, a graph G can have multiple spanning trees, which depend on the traversal order of the used algorithm. That is the reason why DFS and BFS produce two distinct spanning trees from the same input graph G, which is irrelevant as long as we adhere to the spanning tree definition. For a weighted graph $G = (V, E, w)$ we can define a set of spanning trees as

$$\mathcal{ST}(G) = \{G'_i = (V, E_i, w) | E_i \subseteq E(G)\} \tag{5.70}$$

where each G'_i is a spanning tree. Finding a minimal spanning tree is an optimization function

$$MST(G) = \underset{G'_i \in \mathcal{ST}(G)}{\arg\min} \sum_{e \in E_i(G'_i)} w(e) \tag{5.71}$$

According to this optimization function, we search for a spanning tree that has the minimal sum of edge weights of all possible spanning trees. Since the spanning tree is a byproduct of using a graph traversal algorithm on the input graph G, solution of the optimization function can be obtained by extensions of graph traversal algorithms.

All three algorithms presented hereby are in the class of **greedy algorithms**.

Kruskal's algorithm

procedure KRUSKAL(G)
  $MST \leftarrow (V = V(G), E = \emptyset)$
  sort edges $E(G)$ ascending by weights
  for $e_i \in E(G)$ **do**
    if $|E(MST)| < |V(G)| - 1$ **then**
      if no cycle in $G' = (V(MST), E(MST) \cup \{e_i\})$ **then**
        add e_i to $E(MST)$
  return MST

Algorithm 5.14: Kruskal's algorithm.
Complexity is $O(|V|^2 \log_2 |V|)$

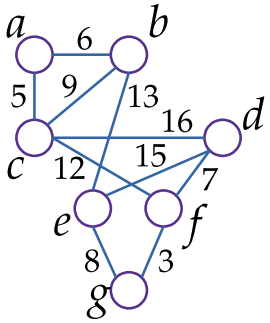


Figure 5.54: An input graph example for MST algorithms.

First, we initialize the output MST as a graph having all vertices as in the input graph G and empty set of edges. We plan to add MST edges as part of the algorithm. Then we sort edges of the input graph G ascending by weights. This sorting is the most limiting part of the Kruskal's algorithm. Once edges are sorted, we iterate through them and add only those that do not form cycles in the output MST. Since we only need to detect whether the edge e_i forms a cycle or not, resulting in a *boolean* output, we can use several options. The fastest option is to use *UnionFind* method for undirected graphs. The compressed variant of the *UnionFind* gives the complexity of $O(1)$ for detecting the cycle. Another option (for directed graphs for example) is to use Algorithm 5.10, which has somewhat higher complexity.

Let us see how Kruskal's algorithm works on the example in Figure 5.54, having the following adjacency matrix

$$\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 9 & 0 & 13 & 0 & 0 \\ 5 & 9 & 0 & 16 & 0 & 12 & 0 \\ 0 & 0 & 16 & 0 & 15 & 7 & 0 \\ 0 & 13 & 0 & 15 & 0 & 0 & 8 \\ 0 & 0 & 12 & 7 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 8 & 3 & 0 \end{bmatrix} \end{array}$$

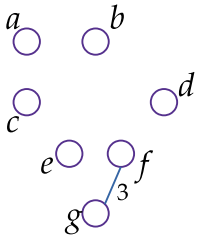
After the sorting of the edges, we have the following list

$$fg = 3, ac = 5, ab = 6, df = 7, eg = 8, bc = 9,$$
$$cf = 12, be = 13, de = 15, cd = 16$$

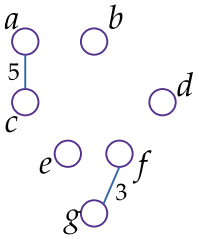
```
def KruskalUndirected(G):
  MST=NewAdjMatrix(G)
  E=edges(G)
  E=dict(sorted(E.items(),
          key=lambda x:x[1]))
  S={}
  for u in G: S=MakeSet(S,u)
  for e in E:
    if Find(S,e[0])!=Find(S,e[1]):
      Union(S,e[0],e[1])
      MST[e[0]][e[1]]=E[e]
      MST[e[1]][e[0]]=E[e]
  return MST
```

Listing 5.17: Kruskal's algorithm python code suitable for undirected graphs. The code uses adjacency matrix representation and *UnionFind* for cycle detection.

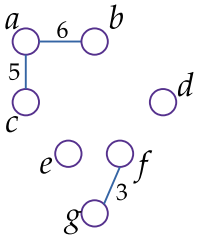
Iteration 1: Edge $fg = 3$



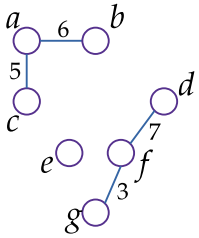
Iteration 2: Edge $ac = 5$



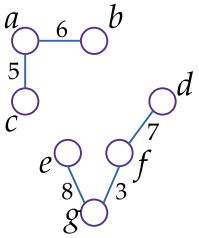
Iteration 3: Edge $ab = 6$



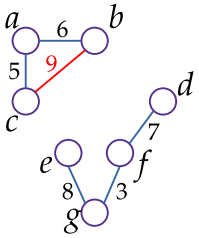
Iteration 4: Edge $df = 7$



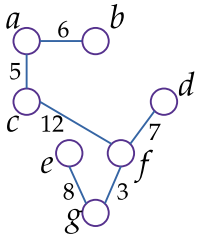
Iteration 5: Edge $eg = 8$



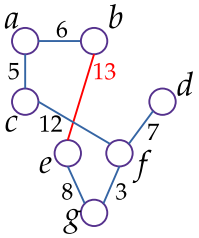
Iteration 6: Edge $bc = 9$



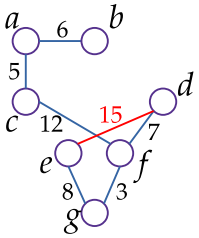
Iteration 7: Edge $cf = 12$



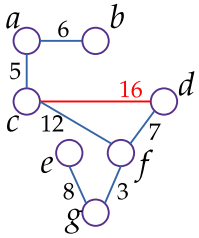
Iteration 8: Edge $be = 13$



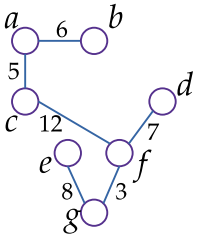
Iteration 9: Edge $de = 15$



Iteration 10: Edge $cd = 16$



Final MST



Final MST adjacency matrix

	a	b	c	d	e	f	g
a	0	6	5	0	0	0	0
b	6	0	0	0	0	0	0
c	5	0	0	0	0	12	0
d	0	0	0	0	0	7	0
e	0	0	0	0	0	0	8
f	0	0	12	7	0	0	3
g	0	0	0	0	8	3	0

The complexity of the Kruskal's algorithm depends on the complexity of the sorting and cycle detection methods. For example,

quicksort complexity is $O(|E| \log_2 |E|)$, which for $|E| < |V|^2$ becomes $O(|V|^2 log_2 |V|)$. Using *UnionFind* method, which has the complexity $O(1)$, the total complexity for the total of $|V| - 1$ edges becomes $O(|V|)$. Adding an edge into the adjacency matrix is also of complexity $O(1)$, while doing the same addition to the adjacency list would take $O(|V|)$, mostly for checking whether a specific edge is already in the graph. The *loop* that detects cycles and adds new edges to the MST is $O(|E|)$, which for the complete graph is asymptotically $O(|V|^2)$. For the fastest variant of the Kruskal's algorithm we can get the complexity $O(|V|^2 \log_2 |V|) + O(|V|^2)$, which becomes $O(|V|^2 \log_2 |V|)$. This confirms that for the undirected variant of the Kruskal's algorithm, the most of the complexity is contributed by the sorting algorithm.

Kruskal's algorithm is **intended to be used on undirected graphs**. Due to the edges sorting, Kruskal's algorithm might not select the absolute **minimal** spanning tree for a directed graph, according to (5.71).

Dijkstra's minimal spanning tree algorithm

The main disadvantage of the Kruskal's algorithm is the need to sort edges. This significantly rises the complexity of the Kruskal's algorithm. Dijkstra's algorithm[1] also iterates through all edges and has a specific decision logic which selects edges to include in the output MST, without need to pre-sort edges.

[1] EW Dijkstra. Some theorems on spanning subtrees of a graph. *Indag. math*, 22(2):196–199, 1960

Algorithm 5.15: Dijkstra's MST algorithm.

procedure DIJKSTRAMST(G)
  $MST \leftarrow (V = V(G), E = \emptyset)$
  for $e_i \in E(G)$ **do**
    add e_i to $E(MST)$
    if there is a cycle in MST **then**
      remove the maximal weight edge from the cycle
  return MST

As with other MST creating algorithms, we need to add $|V| - 1$ edges to form a spanning tree for an input graph G, in which all vertices of the spanning tree are connected. In case when the input graph G has more edges than needed to form the spanning tree $|E| > |V| - 1$, we need to prune some edges to form the spanning tree. In the process of adding edges to the spanning tree, this will be manifested as forming cycles in the tree. According to Dijkstra, if we remove one edge from the cycle, all vertices in the cycle are still in the spanning tree. Therefore, we choose to remove the maximal weighted edge from the cycle, a this will make the remaining spanning tree minimal.

Programming tips: Using *Union-Find* method for this algorithm is not suitable, since we need to examine weights of edges from the formed cycle. For this case, we need to use a DFS extended algorithm, such as Algorithm 5.10. In this case, we already know the

Complexity is $O(|E||V|)$

```
def DijkstraMSTUndirected(G):
  MST=NewAdjMatrix(G)
  E=edges(G)
  for e in E:
    MST[e[0]][e[1]]=E[e]
    MST[e[1]][e[0]]=E[e]
    c=FindCycles(MatrixToTable(MST))
    if len(c)>0:
      c=c[0]
      cp=[(c[-1],c[0])]
      for i in range(len(c)-1):
        cp.append((c[i],c[i+1]))
      E2={}
      for cpi in cp:
        E2=E2|dict(filter(
            lambda x:x[0][0] in cpi
            and x[0][1] in cpi,
            E.items()))
      me=max(E2.items(),
          key=lambda x:x[1])
      if me:
        e2=me[0]
        MST[e2[0]][e2[1]]=0
        MST[e2[1]][e2[0]]=0
  return MST
```

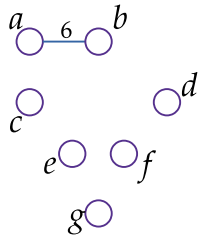
Listing 5.18: Dijkstra's MST algorithm python code. The code uses adjacency matrix representation.

u_0 vertex that can be used as the *root* for the extended DFS algorithm, which is one of the vertices of the newly added edge e_i.

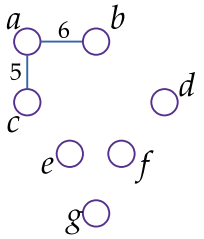
If our edges for the example in Figure 5.54 are ordered as

$$ab, ac, bc, be, cd, cf, de, df, eg, fg$$

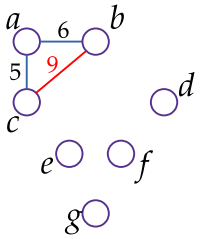
Iteration 1: Edge $ab = 6$



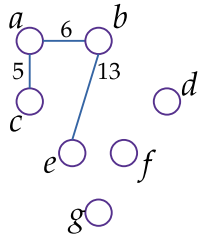
Iteration 2: Edge $ac = 5$



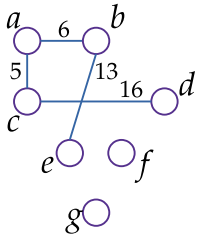
Iteration 3: Edge $bc = 9$, Cycle, Remove bc



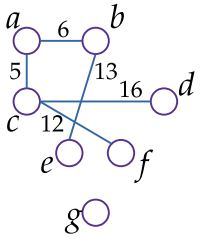
Iteration 4: Edge $be = 13$



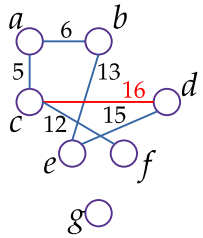
Iteration 5: Edge $cd = 16$



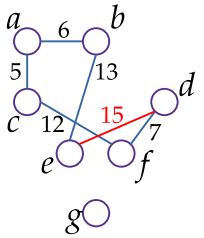
Iteration 6: Edge $cf = 12$



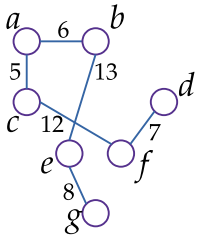
Iteration 7: Edge $de = 15$, Cycle, Remove cd



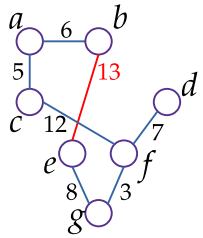
Iteration 8: Edge $df = 7$, Cycle, Remove de



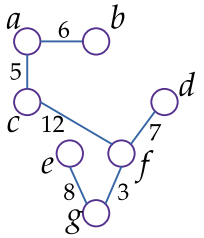
Iteration 9: Edge $eg = 8$



Iteration 10: Edge $fg = 3$, Cycle, Remove be



Final MST



Final MST adjacency matrix

	a	b	c	d	e	f	g
a	0	6	5	0	0	0	0
b	6	0	0	0	0	0	0
c	5	0	0	0	0	12	0
d	0	0	0	0	0	7	0
e	0	0	0	0	0	0	8
f	0	0	12	7	0	0	3
g	0	0	0	0	8	3	0

When it comes to the complexity of Dijkstra's MST algorithm, we have a general *loop* that iterates through all edges, which is $O(|E|)$. The complexity of updating the adjacency matrix is $O(1)$. The complexity of the cycle finding DFS is $O(|V| + |E|)$, which becomes $O(2|V|)$ for a spanning tree. The final complexity is $O(|E|)O(2|V|)$, which becomes $O(|E||V|)$.

Prim's algorithm

The original idea was developed by a Czech mathematician Vojtěch Jarník in 1930. The idea was rediscovered and further developed into an algorithm by both Robert C. Prim in 1957 and Edsger W. Dijkstra in 1959. The algorithm is sometimes called as Dijkstra-Jarník-Prim (DJP) algorithm. The algorithm is applicable to undirected graphs. The algorithm uses the fact that we need $|V| - 1$ edges to create a connected spanning tree and the very definition of the spanning tree. Let us define an input graph as

$$G = (V = \{v_1, v_2, ..., v_n\}, E) \tag{5.72}$$

If we take that a single vertex graph $K_i = (V = \{v_i\}, E = \emptyset)$ is a spanning tree, then we can separate the input graph G into n elementary spanning trees

$$ST(G) = \{ST_i | 1 \leq i \leq n, ST_i = (V = \{v_i\}, E = \emptyset), v_i \in V(G)\} \tag{5.73}$$

To retain the spanning tree definition we can only add a new edge e_n between two distinct spanning trees. By doing so, we merge two spanning trees into a single one

$$ST_i \cup ST_j = (V(ST_i) \cup V(ST_j), E(ST_i) \cup E(ST_j) \cup \{e_n\}) \tag{5.74}$$

The basic idea of the Prim's algorithm is to have one growing spanning tree $ST_g \in ST(G)$ and a set of disconnected elementary spanning trees

$$ST_r = ST(G) \setminus \{ST_g\} \tag{5.75}$$

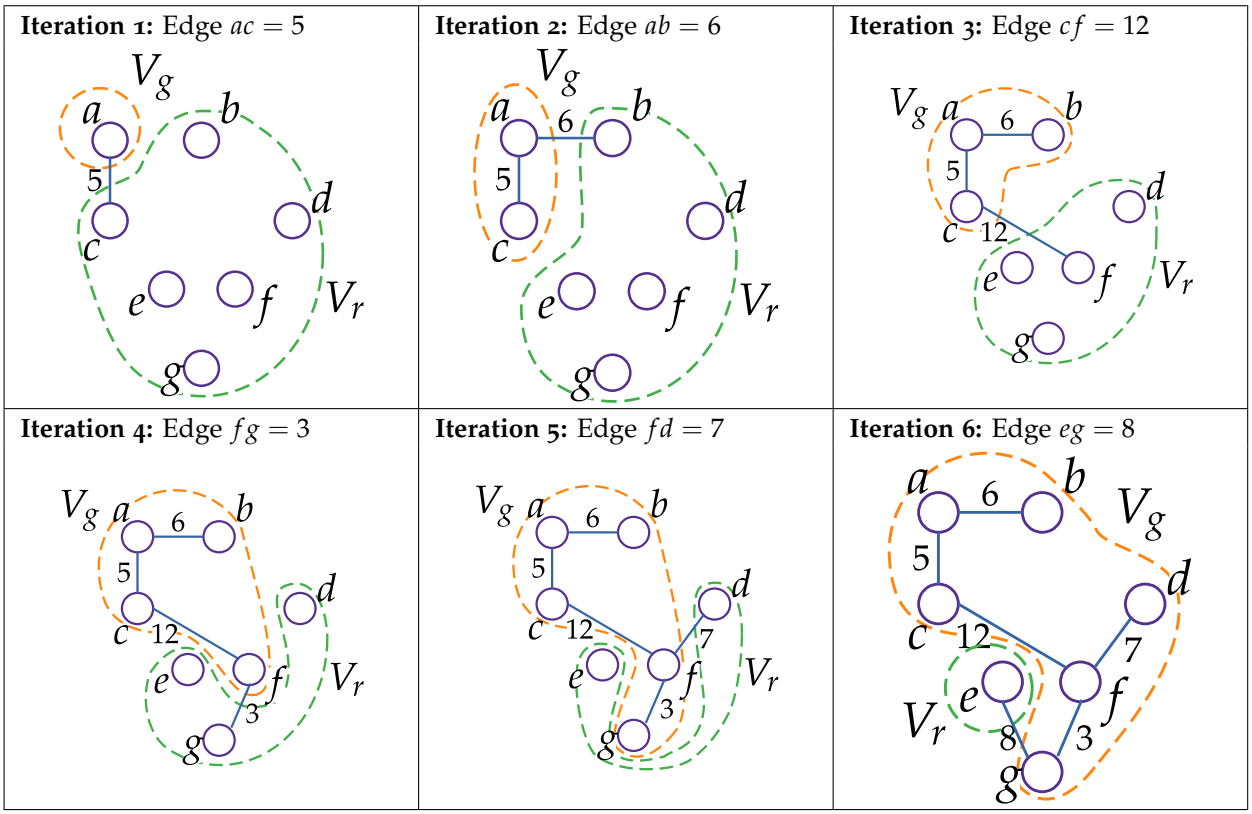
which is de facto remainder of unconnected vertices from the input graph G. We start by having in ST_g a single arbitrary elementary spanning tree in $ST(G)$. Adding a new edge e_n can be only between ST_g and an element of ST_r. We continue to add edges ($|V| - 1$ edges), until $ST_r = \emptyset$. The only way to produce MST is to keep choosing only minimal weighted edges.

Algorithm 5.16: Prim's algorithm.

```
procedure PRIM(G, v_s)
    MST ← (V_g = {v_s}, E = ∅)
    V_r ← V(G) \ {v_s}
    while V_r ≠ ∅ do
        choose minimal weighted edge e_n = uv ∈ E(G) such that
                                    u ∈ V(MST) and v ∈ V_r
        add v to V_g(MST)
        remove v from V_r
        add e_n = uv to E(MST)
    return MST
```

Fastest complexity is $O(|E| + |V| \log_2 |V|)$

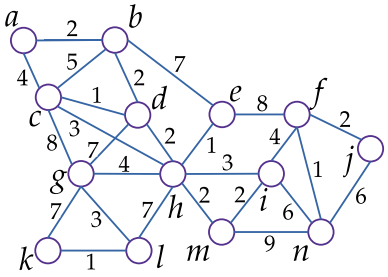
For the example in Figure 5.54, we have the course of execution as in the following table.



As seen in the previous table, we start with vertex a. So, our initial spanning tree ST_g is the elementary spanning tree consisting of only vertex a, or $V_g = \{a\}$. The remainder in this case is $V_r = V(ST_r) = \{b, c, d, e, f, g\}$. First edge we select is the minimally weighted edge that connects V_g and V_r, which is $ac = 5$. We move vertex c from V_r to V_g. The next minimal edge is $ab = 6$. Again, we move vertex b from V_r to V_g. This is repeated until the set V_r becomes empty. Since we had $|V(G)| = 7$, we finished creating our MST in 6 steps.

Complexity assessment of Prim's algorithm is not as simple as $O(|V|)$. We need to include choosing the minimally weighted edge between two sets of vertices. A naïve implementation that iterates through all vertices requires $O(|V|^2)$. Using organizing heaps and trees (for example Fibonacci heap), we can lower this search to $O(|V| \log_2 |V|)$. The solution in listing 5.19 is $O(|V|^2)$, while the fastest solution is $O(|E| + |V| \log_2 |V|)$.

Exercise 5.5.1. Find the MST in the following undirected graph using Kruskal's, Dijkstra, and Prim's algorithm.




```
def Prim(G,vs):
  MST=NewAdjMatrix(G)
  Vg=[vs]
  Vr=list(G.keys())
  Vr.remove(vs)
  while Vr:
    me=None
    for u in Vg:
      for v in Vr:
        if G[u][v]!=0 and
           (not me or me[2]>G[u][v]):
          me=(u,v,G[u][v])
    if me:
      Vg.append(me[1])
      Vr.remove(me[1])
      MST[me[0]][me[1]]=G[me[0]][me[1]]
      MST[me[1]][me[0]]=G[me[0]][me[1]]
  return MST
```

Listing 5.19: Prim's algorithm python code. The code uses adjacency matrix representation.

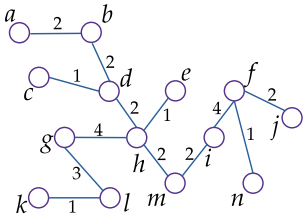


Figure 5.55: Solution of the Exercise 5.5.1.

5.6 Eulerian graphs and circuits

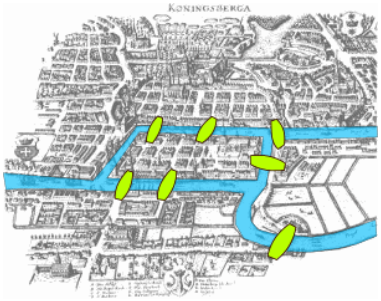


Figure 5.56: Seven bridges of Königsberg (Kaliningrad today).

The birth of the graph theory is associated with Euler's solution to the well-known seven bridges of Königsberg problem. Although, G.W. Leibniz was the first to propose the so-called *positional geometry*, L. Euler was in fact the first who postulated it on the problem of seven bridges. The problem was to find a route through the city by crossing each bridge only once. Although, Euler did not solve the problem, he postulated some theoretical considerations that help us solving similar problems, such as the *Chinese postman problem*. Here we bring a modern graph theory view on the Euler's solution of the seven bridges of Königsberg problem.

Definition 5.35 (Eulerian trail). Eulerian trail is a trail (definition 5.16) that includes all graph edges.

Definition 5.36 (Eulerian circuit). Eulerian circuit (also called *Eulerian circuit*) is a closed Eulerian trail.

Definition 5.37 (Eulerian graph). A connected graph that contains an Eulerian circuit is called Eulerian graph.

Theorem 5.1. A connected undirected graph has an Eulerian circuit only if all of his vertices have even degrees.

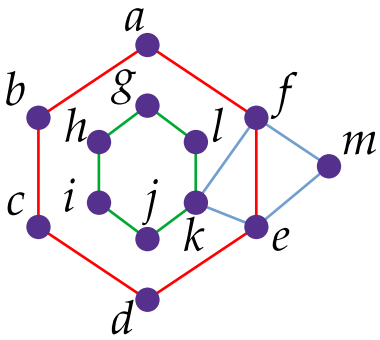


Figure 5.57: Euler cycle and graph.

Proof: Proving the previous theorem is trivial. If we walk around the Eulerian graph, we always use 2 distinct incident edges to visit one vertex, one to enter the vertex and one to exit it. Based on the definition of the Eulerian circuit, we can visit one vertex multiple times, which means that the vertex must have $number\ of\ visits * 2$ incident edges.

An example in Figure 5.57 shows a graph whose vertices have even degrees. According to previous definitions, this is an Eulerian graph containing an Eulerian circuit. We can notice that the Eulerian circuit in this example consists of multiple non-Eulerian circuits

$$\begin{aligned} C_0 &= abcdefa \\ C_1 &= emfke \\ C_2 &= klghijk \end{aligned} \tag{5.76}$$

Theorem 5.2. In a connected Eulerian graph, the Eulerian circuit is an union of non-Eulerian circuits

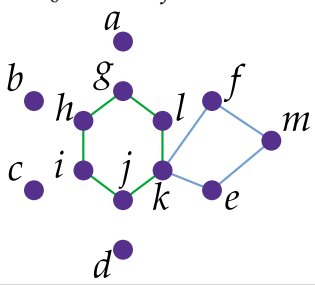
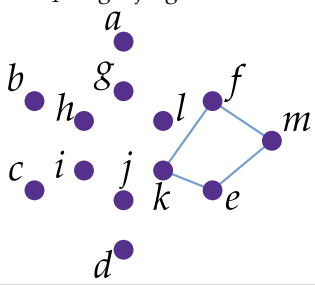
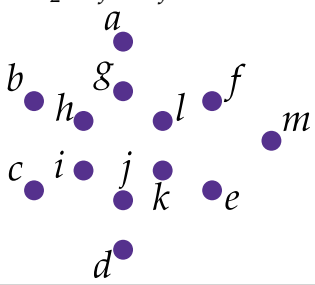
$$\begin{aligned} \mathcal{C} &= \{C_1, C_2, ..., C_n\} \\ E_C &= C_1 \cup C_2 \cup ... \cup C_n \end{aligned} \tag{5.77}$$

where distinct non-Eulerian circuits $C_1, C_2, ..., C_n$ can share only common vertices

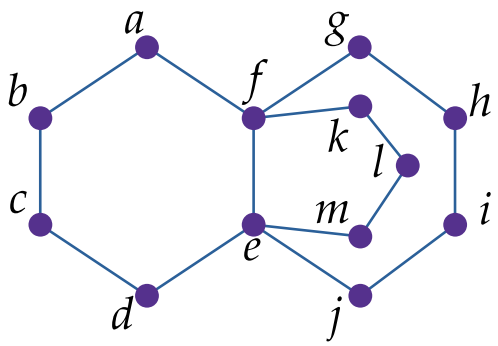
$$\begin{aligned} \forall C_i, C_j \in \mathcal{C} \forall v_k v_l \in C_i | C_i \neq C_j, v_k v_l \notin C_j \\ \forall C_i, C_j \in \mathcal{C} \exists v \in C_i | C_i \neq C_j, v \in C_j \end{aligned} \tag{5.78}$$

Proof: Let us first find a non-Eulerian circuit C_0 in the Eulerian graph G. If we remove edges of the cycle C_0 from the Eulerian graph, due to Theorem 5.1, we removed an even number of edges from vertices in the cycle C_0. Since the Eulerian graph must be connected, vertices from the removed cycle C_0 must be shared with some other non-Eulerian circuits. These shared vertices must have even number of incident edges even after C_0 removal, which means these incident edges are part of other cycles. After this, we pick another non-Eulerian circuit C_1, and remove its edges from the Eulerian graph G. We should have the same situation we had after the removal of C_0. This picking of non-Eulerian circuits are removal of their edges continues until we can either form a new cycle C_i or we removed all edges from the Eulerian graph G. If we cannot form another cycle from the remainder of the edges in G, then some of the vertices had odd number of incident edges, which directly contradicts to the Euler graph definition.

For the example in Figure 5.57 we have the following 3 cycles

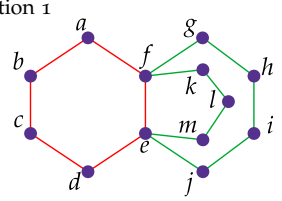
Iteration 1: After removal of the cycle $C_0 = abcdefa$	**Iteration 2:** After removal of the cycle $C_1 = ghijklg$	**Iteration 3:** After removal of the cycle $C_2 = fkemf$
		

Exercise 5.6.1. Is the following graph is Eulerian graph?

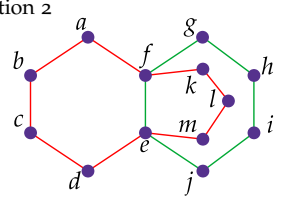


Try to find all non-Eulerian circuits in the graph.

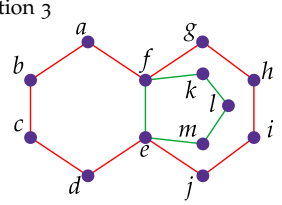
Solution 1



Solution 2



Solution 3



If we try to observe Eulerian circuits in directed graphs, we can find several distinct theorems and definitions. Here we focus on the following simple definition

Definition 5.38 (Eulerian circuit in directed graphs). A directed graph has an Eulerian circuit only if it is **strongly connected** and all of his **vertices have the same indegree and outdegree**, $indeg(v) = outdeg(v)$.

Based of the Theorem 5.2 we can come up with a simple algorithm,

which can be used to detect whether an input graph has an Eulerian circuit or not.

```
procedure IsEulerianCircuit(G, u)
    while true do
        u_0 ← u
        while there is an edge uv in G do
            add uv to the circuit
            remove uv from the graph G
            u ← v
        if u ≠ u_0 then
            return false
        if there are edges in G then
            pick a vertex u that has incident edges
        else
            return true
```

Algorithm 5.17: Detecting Eulerian circuit.

The complexity is $O(|E| + |V|)$

```
def IsEulerianCircuit(G,u):
  hasMore=True
  while hasMore:
    uo=u
    while G[u]['adj']:
      v=G[u]['adj'][0]
      G[u]['adj'].remove(v)
      G[v]['adj'].remove(u)
      u=v
    if uo is not u:
      return False
    hasMore=False
    for u in G:
      if G[u]['adj']:
        hasMore=True
        break
  return True
```

Listing 5.20: Detecting Eulerian circuit python code. The code uses adjacency table representation.

Hierholzer's algorithm

By carefully looking at non-Eulerian circuits in (5.76), we can see that C_2 fits into C_1 as follows

$$emfklghijke \tag{5.79}$$

and then this cycle fits into C_0 giving us the final union of all non-Eulerian circuits and by that the Eulerian circuit (5.77)

$$E_C = abcdemfklghijkefa \tag{5.80}$$

This is consistent with Theorem 5.2, where cycles can share at least one common vertex. Using a stack data structure, we can use this feature to obtain the Eulerian circuit from the input Eulerian graph G.

```
procedure Hierholzer(G, u)
    s ← new empty stack
    cycle ← ∅
    s.push(u)
    while s not empty do
        u ← last element on the stack s
        if u has adjacent vertices then
            v ← one of the adjacent vertices of u
            s.push(v)
            remove edge uv from G
        else
            e ← edge uv from last two vertices on the stack
            add e to cycle
            s.pop()
    return cycle
```

Algorithm 5.18: Hierholzer's algorithm.

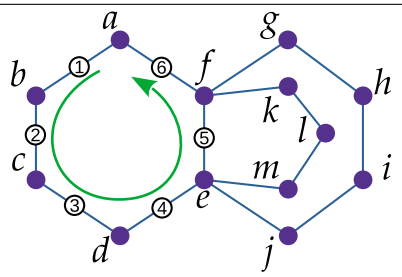
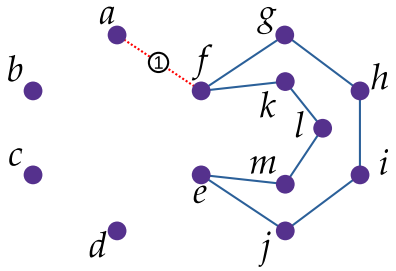
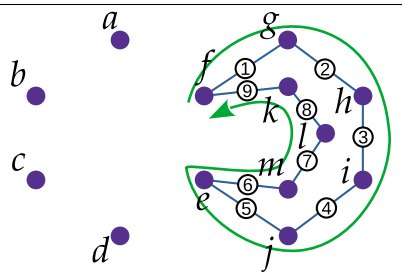
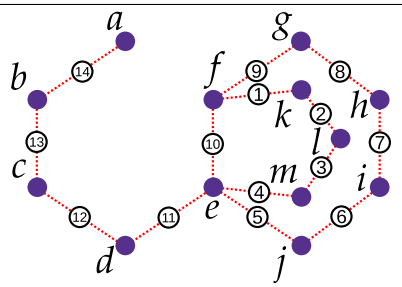
The complexity is $O(|E|)$

```
def Hierholzer(G,u):
  s=deque()
  cycle=[]
  s.append(u)
  while s:
    u=s[-1]
    if len(G[u]['adj'])>0:
      v=G[u]['adj'][0]
      s.append(v)
      G[u]['adj'].remove(v)
      G[v]['adj'].remove(u)
    else:
      if len(s)>1:
        cycle.append(
            (s[-1],s[-2]))
        s.pop()
  return cycle
```

Listing 5.21: Hierholzer's algorithm python code. The code uses adjacency table representation.

In the Hierholzer's algorithm there is a constant exchange of pushing and popping phases. In the pushing phase, we are *consuming* edges and pushing vertices on our walk, until we encounter the last vertex in the current non-Eulerian circuit, meaning the one that has no edges anymore. Remember, when *consuming* edges, we removed them from the graph G. After this, we start the popping phase, where we move backwards through the stack and reconstruct edges, until we reach one vertex that still has incident edges left. This vertex is a common vertex between two non-Eulerian circuit. At this point, Hierholzer's algorithm switches back to the pushing phase, and starts filling the stack with new non-Eulerian circuit.

Based on the example in Exercise 5.6.1, we give the following course of execution.

Phase 1: Pushing phase	
We start by pushing vertices to the stack S, until the whole non-Eulerian circuit $abcdefa$ was traversed. When we reach the starting vertex a, we are left without edges, so we need to switch to the popping phase. $S = abcdefa$	
Phase 2: Popping phase	
$S = abcdefa$ We reconstruct edges backwards from the stack S. The first edge we get is fa. Then we pop the last vertex from the stack S. $cycle = fa$ $S = abcdef$ At this moment, we reached vertex f, which still has some incident edges. Continuing with the pushing phase.	
Phase 3: Pushing phase	
Again pushing vertices to the stack S, until we reach vertex f, which has no more incident edges. Continuing with the popping phase. $S = abcdefghijemlkf$	
Phase 4: Popping phase	
$S = abcdefghijemlkf$ We reconstruct edges backwards from the stack S. Then we pop the last vertex from the stack S. This continues until we pop the last vertex a from the stack S. Algorithm ends here. $cycle = fa, kf, lk, ml, em, je, ij, hi, gh, fg, ef, de, cd, bc, ab$	

Fleury's algorithm

The Fleury's algorithm uses the following lemma as the basis for picking up the best walk through the Eulerian circuit.

Lemma 5.2. Any trail in an Eulerian graph ends up in the vertex that has only one incident edge, which is the same as the starting vertex.

Proof: Once we begin forming a trail from a starting vertex in the Eulerian circuit, by removing the first edge from the graph G, we are making this starting vertex as the only one that has only one incident edge, which is going to be the last ingoing edge in the trail. Hence, the starting vertex becomes the ending vertex of the Eulerian circuit.

The complexity is $O(|E|^2)$

```
function FLEURY(G, u)
    cycle ← ∅
    while there are edges in G do
        pick an edge uv from G prioritizing
              non-bridge edges over bridge edges
        add uv to the cycle
        remove uv from the graph G
        u ← v
    return cycle
```

Algorithm 5.19: Fleury's algorithm.

The main idea of the Fleury's algorithm is to build an Eulerian circuit by prioritizing non-bridge edges over the bridge edges. According to Lemma 5.2, the ending vertex of the Eulerian circuit is determined at the moment when we remove the first edge from the Eulerian graph. We have a such example in Figure 5.58, where we can see what is left of the Eulerian graph after we removed the first edge. The green arrow marks an approximate Eulerian circuit walk. The only two vertices that have the odd degree are the next vertex in the walk (marked light green) and the last vertex of the Eulerian circuit (marked red), which is according to Lemma 5.2. As we remove edges from the graph, more edges can become bridges (definition 5.25). Detecting whether an edge is a bridge or not can be done by detecting biconnected-components or SCC's (see Sections 5.4 and 5.4). Any edge that connects two biconnected-components or SCCs can be taken as a bridge. However, this check is quite expensive, since this is constantly changed by removing edges from the graph. After we removed the first edge, we got a number of bridges in the graph, marked as orange in Figure 5.58.

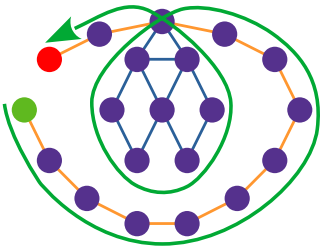


Figure 5.58: Eulerian circuit example.

```
def Fleury(G,u):
  cycle=[]
  blocks=BlockSearch(G)
  while G[u]['adj']:
    for v in G[u]['adj']:
      exit=False
      for b in blocks:
        if len(b)>1 and
            ((u,v) in b or
            (v,u) in b):
          exit=True
          break
        if exit: break
    G[u]['adj'].remove(v)
    G[v]['adj'].remove(u)
    blocks=BlockSearch(G)
    cycle.append((u,v))
    u=v
  return cycle
```

Listing 5.22: Fleury's algorithm python code using block search for bridge detection. The code uses adjacency table representation.

Avoiding bridges when possible, or prioritizing non-bridge edges, allows us not to end in the last vertex of the Eulerian circuit too soon, before exploring all non-Eulerian circuits. The idea is to use the block search (Algorithm 5.12) or Tarjan's algorithm (Algorithm 5.13) each time we remove an edge. The complexity of the Fleury's algorithm is $O(|E|)$. When we add a bridge detecting algorithm, this rises to $O(|E|^2)$. However, some improvements can be made when coding the bridge detecting algorithm variants. We could memorize the

spanning tree that was used by the bridge detecting algorithm, and remove edges directly from the spanning tree, which is a dynamic way of connectivity handling[2]. Such approach lowers the complexity of the Fleury's algorithm to $O(|E| * log|E|^3 * log(log|E|))$, which is still slower than other algorithms such as Hierholzer's algorithm.

[2] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 343–350, 2000

Chinese postman problem

As we found out in the seven bridges of Königsberg problem, we can model any settlement in form of a weighted graph. We have a similar example in Figure 5.35. Edges of the weighted graph represent streets of the modelled settlement, while vertices represent street junctions. Weights of the edges are the cost that takes to pass this street.

A postman needs to deliver mail by traverse all streets (edges) of the settlement, beginning from a specific intersection (vertex), where the postal office is located. An example of the settlement model can be seen in Figure 5.59. The postman must traverse streets the most efficiently as possible, regrading the costs of the streets defined by edges weights.

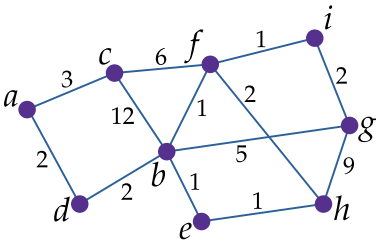


Figure 5.59: Chinese postman problem example.

As we learned from the Euler's solution of the seven bridges of Königsberg problem, the solution of the Chinese postman problem is in finding an Eulerian circuit in the graph that models the settlement. By looking at the graph in Figure 5.59, we can notice that this is not an Eulerian graph, as some of the vertices have odd degrees. We can solve this problem by allowing the postman to traverse **some** of the streets multiple times. The real question is: which ones and how is this affecting the effort minimality requirement.

Graph Eulerization

By adding edges to the graph, we can transform a connected graph to an Eulerian graph, containing an Eulerian circuit. The graph on

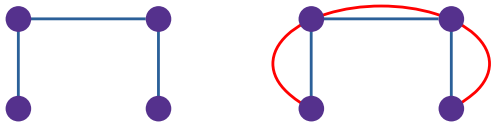


Figure 5.60: A simple graph Eulerization example.

the left side in Figure 5.60 has two vertices with $deg(v) = 1$, which is not according to Theorem 5.1. By adding edges, as on the right side in Figure 5.60 (marked red), the graph becomes an Eulerian graph, having all vertices of even degree.

The main idea of the graph Eulerization is to multiply some edges so that all vertices have the even degree. Due to the handshaking lemma (lemma 5.1), we know that in a connected graph there could be an even number of vertices that have the odd degree. By performing an optimization, we need to find pairs of vertices having the odd degree that are minimally distant from each other.

Algorithm 5.20: Graph Eulerization.

1: Detect all vertices having an odd degree $ODD \subseteq V(G)$.
2: Calculate distances all-to-all between ODD vertices. You can use WFI (algorithm 5.9) for this purpose. We define the mapping set of shortest paths between pairs of vertices in ODD as

$$\forall v_i, v_j \in ODD : ODDPaths(v_i, v_j) = v_i v_{i+1} ... v_{j-1} v_j$$

3: Construct a complete bipartite graph

$$X = ODD, Y = ODD$$
$$H = (X \cup Y, E')$$

such that weights of the bipartite graph edges are as:

$$w(v_{ii}) = \infty,$$
$$w(v_{ij}) = w_{ij}$$

where w_{ij} is the shortest distance between vertices $v_i, v_j \in ODD$, calculated in the step 2.
4: In the complete bipartite graph H find the optimal assignments M between vertices in X and Y, such that

$$M \subseteq X \times Y \Rightarrow \min \sum_{(v_i, v_j) \in M} w(v_i v_j)$$

5: **for** $(v_i, v_j) \in M$ such that $deg(v_i)$ is still odd **do**
6: **for** $v_r v_s \in ODDPaths(v_i, v_j)$ **do**
7: Add the edge $v_r v_s$ to the edges $E(G)$

```
def Eulerization(G):
  ODD=[]
  for u in G:
    t=list(filter(lambda x:
            G[u][x]>0,G[u]))
    if len(t)%2!=0: ODD.append(u)
  (D,P)=WFI(copy.deepcopy(G))
  BG=Permutate(ODD.copy())
  M={'r':sys.maxsize}
  for combo in BG:
    tot=0
    for t in combo:
      tot=tot+D[t[0]][t[1]]
    if tot<M['r']:
      M={'r':tot,'p':{}}
      for t in combo:
        M['p'][t]=RevWFI(P,t[0],t[1])
  G=MatrixToTable(G)
  for t in M['p']:
    p=M['p'][t]
    while len(p)>=2:
      G[p[0]]['adj'].append(p[1])
      G[p[1]]['adj'].append(p[0])
      del p[0]
  return G
```

Listing 5.23: Eulerization python code. The code uses adjacency matrix representation, and returns adjacency table representation.

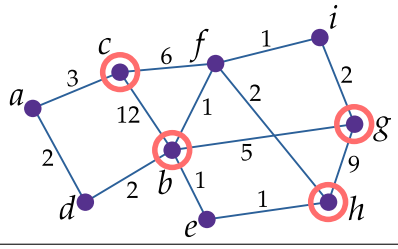
Function *Permutate* is used to create a list of tuples (ordered pairs) that represents all bipartite graph vertex combinations. The optimization then selects the combination that has the minimal sum of weights.

Based on the example in Figure 5.59, we give the following Eulerization procedure.

Step 1: Detecting the odd degree vertices

We start by detecting all vertices having an odd degree. We have the following vertices

$$ODD = \{b, c, g, h\}$$



Step 2: Using WFI, calculate shortest paths and distances in ODD

We get the following shortest distances

$$w(bc) = D^9_{bc} = 7, w(bg) = D^9_{bg} = 4$$
$$w(bh) = D^9_{bh} = 2, w(cg) = D^9_{cg} = 9$$
$$w(ch) = D^9_{ch} = 8, w(gh) = D^9_{gh} = 5$$

and shortest paths

$$ODDPaths(b, c) = bdac$$

$\mathbf{D^0} =$

	a	b	c	d	e	f	g	h	i
a	0	0	3	2	0	0	0	0	0
b	0	0	12	2	1	1	5	0	0
c	3	12	0	0	0	6	0	0	0
d	2	2	0	0	0	0	0	0	0
e	0	1	0	0	0	0	0	1	0
f	0	1	6	0	0	0	0	2	1
g	0	5	0	0	0	0	0	9	2
h	0	0	0	0	1	2	9	0	0
i	0	0	0	0	0	1	2	0	0

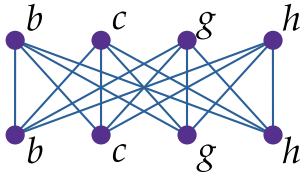
$$ODDPaths(b, g) = bfig$$
$$ODDPaths(b, h) = beh$$
$$ODDPaths(c, g) = cfig$$
$$ODDPaths(c, h) = cfh$$
$$ODDPaths(g, h) = gifh$$

$\mathbf{D^9} =$

	a	b	c	d	e	f	g	h	i
a	0	4	3	2	5	5	8	6	6
b	4	0	7	2	1	1	4	2	2
c	3	7	0	5	8	6	9	8	7
d	2	2	5	0	3	3	6	4	4
e	5	1	8	3	0	2	5	1	3
f	5	1	6	3	2	0	3	2	1
g	8	4	9	6	5	3	0	5	2
h	6	2	8	4	1	2	5	0	3
i	6	2	7	4	3	1	2	3	0

Step 3 and 4: Generate the bipartite graph H and find the optimal assignments M

In this step we need to optimize the pairs of vertices of the complete bipartite graph H.
$M_1 = \{(b, c), (g, h)\}, w(bc) + w(gh) = 7 + 5 = 12$
$M_2 = \{(b, g), (c, h)\}, w(bg) + w(ch) = 4 + 8 = 12$
$M_3 = \{(b, h), (c, g)\}, w(bh) + w(cg) = 2 + 9 = 11$
The optimal assignment is M_3, which is also giving the shortest Eulerian circuit in the final Eulerized graph.



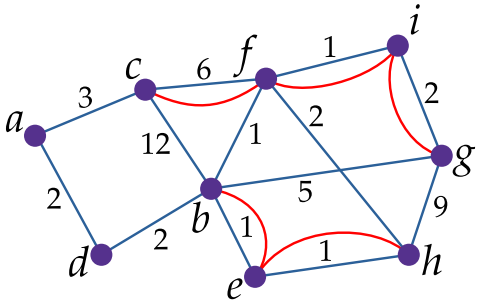
b			c			g			h		
c	g	h	b	g	h	b	c	h	b	c	g
7	4	2	7	9	8	4	9	5	2	8	5

Step 5: Add edges from the shortest paths $ODDPaths$, according to the optimal assignments M

We iterate through M_3 to find the shortest paths edges.

1. $(b, h) \in M_3 : ODDPaths(b, h) = beh$. Edges be and eh are added to the graph G.
2. $(c, g) \in M_3 : ODDPaths(c, g) = cfig$. Edges cf, fi, and ig are added to the graph G.

Finally, we get the Eulerized graph as seen on the right side in which all vertices have even degrees.



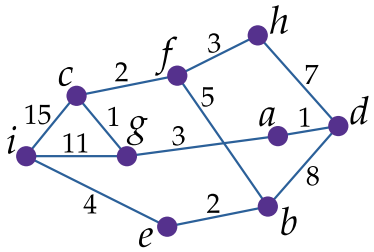
Once we get the Eulerized graph, we can apply either Hierholzer's or Fleury's algorithm to find the Eulerian circuit, knowing the start-

ing vertex. If we take b as the starting vertex in the previous example, we get

$$bdacbehebghfcfigifb$$

which is one of the possible Eulerian circuits in the Eulerized graph, and is also the solution for the Chinese postman problem for the settlement model in Figure 5.59.

Exercise 5.6.2. Solve the Chinese postman problem for the following graph model. The postman starts and ends at the intersection g.



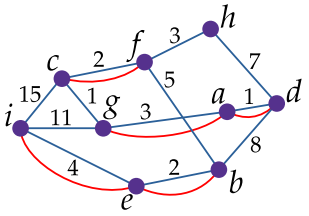


Figure 5.61: Eulerized graph for the Exercise 5.6.2.

5.7 Hamiltonian graphs and cycles

Hamiltonian paths and cycles are named after W.R.Hamilton, an Irish mathematician. We start with several definitions

Definition 5.39 (**Hamiltonian path**). Hamiltonian path is a path (definition 5.17) that includes all graph vertices.

Definition 5.40 (**Hamiltonian cycle**). Hamiltonian cycle is a closed Hamiltonian path.

Definition 5.41 (**Hamiltonian graph**). A connected graph that contains an Hamiltonian cycle is called Hamiltonian graph.

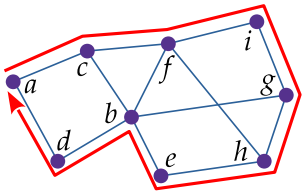


Figure 5.62: Hamiltonian graph and cycle example.

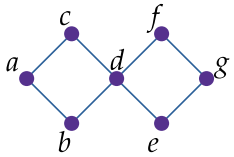


Figure 5.63: A non-Hamiltonian graph.

To claim that a graph G has a Hamiltonian cycle, we need to traverse the whole graph by visiting each vertex only once, and end in the starting vertex. For the example in Figure 5.62, we can see the following Hamiltonian cycle

$$acfighebda$$

where only the starting and the ending is repeated. However, when we look at the graph in Figure 5.63, we can see that the graph cannot be traversed in a way that we visit each vertex only once, since we need to visit vertex d at least twice in every possible walk when trying to find a path. Hence, for the graph in Figure 5.63 we cannot find any path, which automatically means there is no Hamiltonian cycle as well.

Hamiltonian graphs and cycles are useful when trying to solve the *traveling salesman problem* (TSP). Contrary to the Chinese postman problem, in the traveling salesman, we focus on visiting each vertex only once. Vertices in this case are the places which the salesman needs to visit, while edges are possible routes between places. Edges weights represent the cost of the route. The salesman must visit all the places, simultaneously minimizing the total effort for traveling. Solving the traveling salesman problem means finding a route through the graph, such that we visit each vertex only once, while the sum of weights for edges we take is the minimal possible. The definition of the traveling salesman problem is a perfect fit for Hamiltonian graphs and cycles.

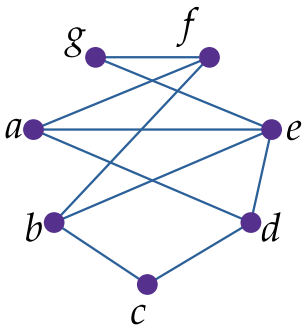


Figure 5.64: Hamiltonian graph example.

Finding a Hamiltonian cycle is a NP-complex problem. We could solve this problem by using a modified DFS algorithm. This way, we can try all vertex combinations, which is of complexity $O(|V|^2)$. For some specific cases, we can use the *Bondy–Chvátal* theorem.

Bondy–Chvátal theorem

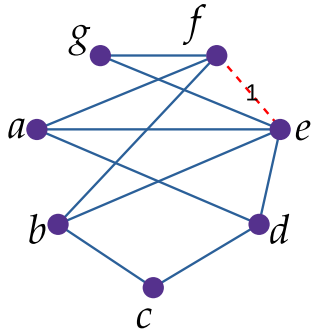
We start by creating a closure of an input graph G. The closure G' of the graph G is created by the following steps

Algorithm 5.21: Creating a graph closure.

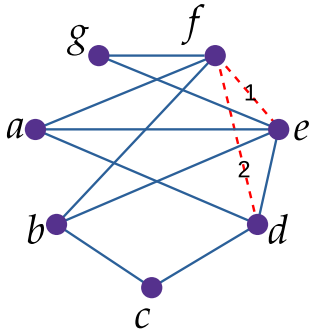
$G' \leftarrow G$
$i \leftarrow 1$
while there are vertices u and v, such that $deg(u) + deg(v) \geq |V(G)|$ **do**
    add the edge uv to the edges $E(G')$, mark it as the i-th edge
    $i \leftarrow i + 1$

For the graph in Figure 5.64, we give the following steps for creating the closure G'.

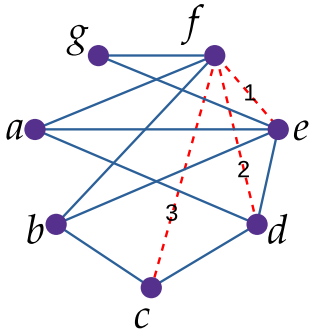
Iteration 1: $deg(f) + deg(e) = 7$



Iteration 2: $deg(f) + deg(d) = 7$

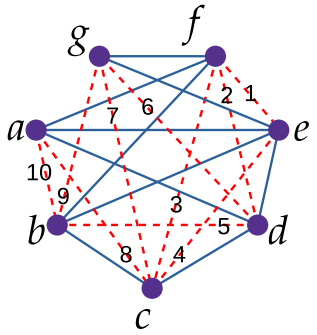


Iteration 3: $deg(f) + deg(c) = 7$

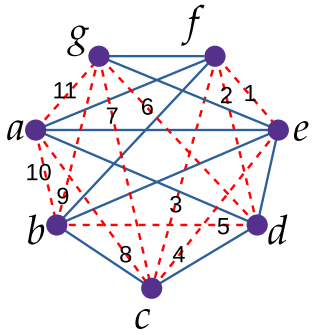


We continue to add edges for pairs of vertices whose degrees comply the requirement $deg(u) + deg(v) \geq |V|$.

Iteration 10: $deg(a) + deg(b) = 9$



Iteration 11: $deg(a) + deg(g) = 10$. This is also the last step and the closure G'.



The only pair of vertices we can start with are f and e, as they are the only ones that combined have sufficient incident edges to comply $deg(f) + deg(e) \geq 7$. After we add the first edge fe to the closure G', we got more vertice pairs where we can add new edges. We mark the order of edges we add to the closure G'. This addition of edges continues until we reach a complete graph, which is the closure G'.

A general recommendation is to add outer closure edges as either the first ones or the last ones. In the previous closure example, we added edges ab and ag as the last ones. This gives us a bit extra space when finding the Hamiltonian cycle, which will be detailed in the following definitions and algorithms.

Theorem 5.3 (Bondy–Chvátal). A simple undirected graph G (definition 5.8) is Hamiltonian only iff its closure G' is Hamiltonian.

Proof 1: If a simple undirected graph G is Hamiltonian, then there is a subset of edges $E_1 \subseteq E(G)$ that make the Hamiltonian cycle C_H in it. The closure G' of the Hamiltonian graph G is created by adding new edges (see algorithm 5.21). We have $E_1 \subseteq E(G) \subseteq E(G')$, which means that the original Hamiltonian cycle C_H is contained in the closure G' as well, which makes the closure G' a Hamiltonian graph.

Proof 2: We have a sequence of graphs between the graph G and its closure G'. Starting with $G = G_0$, we create a new graph each time we add an edge uv to it. This means that the graph G_{k-1} has one edge less than the graph G_k. The sequence of graphs is

$$G = G_0, G_1, G_2, ..., G_{k-1}, G_k, ..., G' \tag{5.81}$$

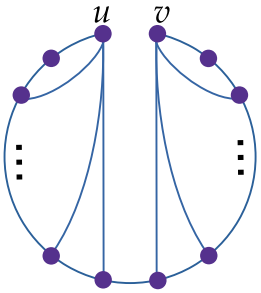
Let us say that $|V| = n$. In Figures 5.65, we can see two distinct situations that can occur in the step $k-1$. In both Figures we are missing only the edge uv to have the complete Hamiltonian cycle C_H. This means, by adding the edge to the graph G_{k-1}, visible in Figures 5.65 we are constructing the graph G_k, for which we are certain that the Hamiltonian cycle exists. By proving that the Hamiltonian cycle exists in the graph G_{k-1}, we prove the Bondy–Chvátal theorem.

In Figure 5.65a vertices u and v together are adjacent to all other vertices in the graph G_{k-1}. By deducting u and v from the total sum of adjacent vertices, we get $deg(u) + deg(v) = n - 2$. However, this contradicts with the requirement for the edge uv addition: $deg(u) + deg(v) \geq n$. In such case we cannot add the edge uv, hence, the graph G_k cannot be created and the Hamiltonian cycle does not exist in the graph G_{k-1}.

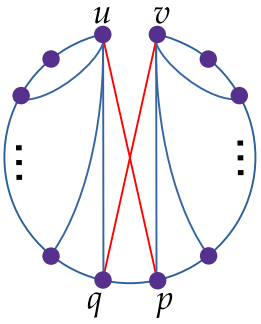
Otherwise, if the graph G_{k-1} satisfies the requirement $deg(u) + deg(v) = n$, as in Figure 5.65b, this means that vertices u and v have exactly two common adjacent vertices p and q. By the *Dirichlet principle*, it means that there is a cycle made of

$$C_H - pq + up + vq \tag{5.82}$$

edges, which is a Hamiltonian cycle. This means that due to the condition for the edge addition, we cannot create the Hamiltonian graph G_k if its predecessor G_{k-1} was not Hamiltonian. By reverse induction, this can be applied back to the graphs G_0 and G_1, which proves the Bondy–Chvátal theorem.



(a) $deg(u) + deg(v) = n - 2$



(b) $deg(u) + deg(v) = n$

Figure 5.65: Graph G_{k-1} when creating the closure G'.

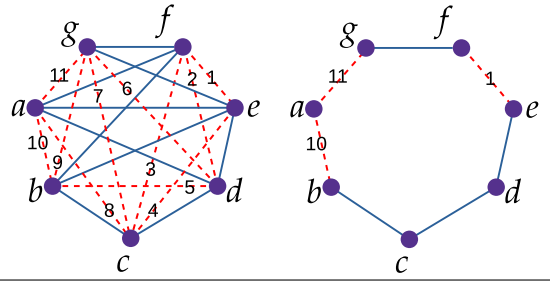
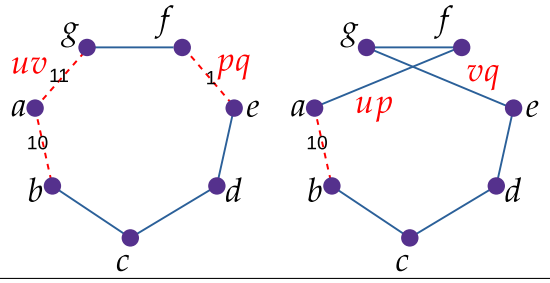
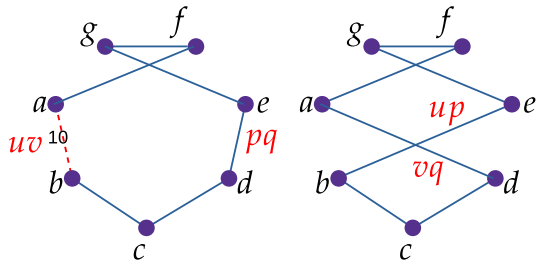
Identifying a Hamiltonian cycle in a Graph G could be a difficult task. Instead, we create the closure G' from the graph G such that has an easy identifiable Hamiltonian cycle. Then we use the Bondy–Chvátal theorem principles to move backwards from the closure G', to identify the Hamiltonian cycle in the original graph G. For the purpose of moving backwards through the creation of the closure G', we marked each added edge uv with ordinal. By replacing an added edge uv having the highest ordinal we effectively move backwards through the creation of the closure G', until we do not

have added edges by the closure creation procedure, which means we reached the original graph G.

Algorithm 5.22: Identifying the Hamiltonian cycle in the graph G by going backwards from its closure G'.

```
1: G_c ←Hamiltonian cycle in the closure G'
2: while there is added edge marked with ordinal in the graph G_c
   do
3:     pick an added edge uv from G_c having the highest ordinal
       i_max
4:     pick an edge pq ≠ uv from G_c such that edges up and vq
       are from G', crossed, and either from the original graph G or
       having ordinal less than i_max
5:     remove edges uv and pq from the graph G_c
6:     add edges up and vq to the graph G_c
```

In the following table, we give the steps for the example in Figure 5.64.

Step 1: Detect the Hamiltonian cycle in the closure G'	
We start by detecting the Hamiltonian cycle in the closure G'.	
Step 2: Replace edges ag and fe.	
We find ag as the edge having the highest ordinal $i_{max} = 11$, which is identified as the edge uv. The other edge fe is taken as the edge pq. We take crossed edges from the G', which are both from the original graph G.	
Step 3: Replace edges ab and ed.	
We find ab as the edge having the highest ordinal $i_{max} = 10$, which is identified as the edge uv. The other edge ed is taken as the edge pq. Taking the edge ed is convinient, as we have crossed edges ad and be from the original graph G. After this step, we have no more added edges, which means we moved backwards to the graph $G = G_0$. The result is the Hamiltonian cycle in the original graph G.	

The previous recommendation to hold on adding outer edges so they could have the highest ordinals now has the purpose. As we identify the outer cycle in the closure G' as the Hamiltonian cycle, as seen in Step 1 of the previous table, having many added edges with high ordinals in this initial Hamiltonian cycle places us closer to the closure G', which means we can take more steps before reaching G_0.

Traveling salesman problem

As already mentioned, solving the traveling salesman problem means finding a route through the graph, such that we visit each vertex only once, while the sum of weights for edges we take is the minimal possible. This route is most certainly a Hamiltonian cycle, since we need to visit each vertex only once. Since we can have more than one Hamiltonian cycle in the graph, the need for optimizing the route selects only one Hamiltonian cycle as the optimal one.

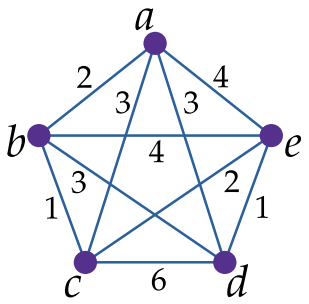
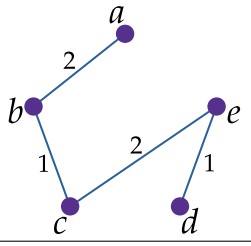
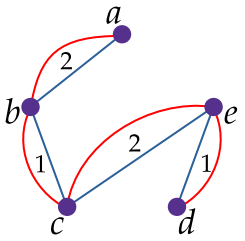
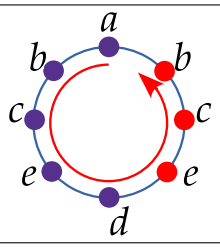
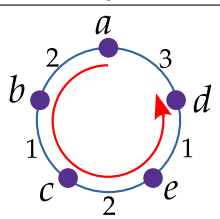


Figure 5.66: TSP graph example.

Algorithm 5.23: 2-MST heuristic.

1: Construct the minimal spanning tree G_{MST} from the graph G.
2: Eulerize the minimal spanning tree G_{MST} by doubling all edges.
3: Find the Eulerian circuit C_E in Eulerized G_{MST}.
4: Turn the Eulerian circuit C_E into a Hamiltonian cycle C_H by skipping already visited vertices and jumping to the first unvisited.

Step 4 can be done using DFS. As soon as we return from the recursion, we skip calls until we find the first unvisited vertex. In the following table, we give the steps for the example in Figure 5.66.

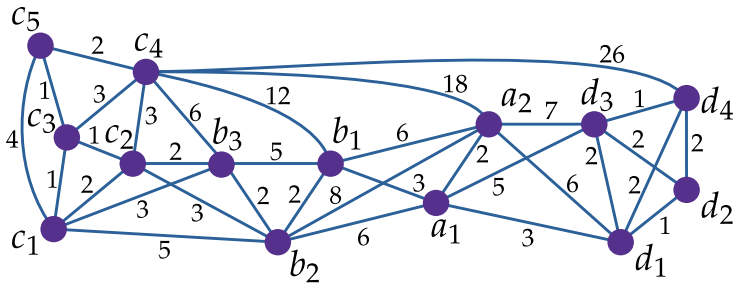
Step 1: Construct the minimal spanning tree G_{MST}	
We start by constructing the minimal spanning tree G_{MST} from the graph G. For this purpose we can use any of the MST algorithms.	
Step 2: Eluerize the minimal spanning tree G_{MST}.	
Eulerize the minimal spanning tree by doubling all edges.	
Step 3: Find the Eulerian circuit C_E.	
Use any of the Eulerian circuit finding algorithms. Respect the starting vertex of the salesman. Notice that some of the vertices (marked red) are repeating in the Eulerian circuit. These vertices are not according to the definition of the Hamiltonian cycle.	
Step 4: Convert the Eulerian circuit C_E to the Hamiltonian cycle C_H.	
Remove all repeating vertices in the Eulerian circuit C_E. Connect the first two non-repeating vertices by using their edge (we need its weight) from the input graph G. In this example we removed repeating vertices e, c, and b, and connected d and a directly using its original edge of weight 3.	

The final result for the example in Figure 5.66 is

$$C_H = abced$$

of total effort $d_H = 9$.

Exercise 5.7.1. Solve the traveling salesman problem using 2-MST heuristic for the following graph model. The salesman starts and ends at the place d_4.



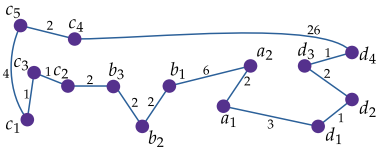


Figure 5.67: Hamiltonian cycle for the Exercise 5.7.1.

```
def TwoMSTTSP(G,s):
  Gmst=DijkstraMSTUndirected(copy.deepcopy(G))
  GEmst=Eulerization(Gmst)
  Ce=Hierholzer(GEmst,s)
  v=[s]
  for e in Ce:
    if e[1] not in v and G[v[-1]][e[1]]>0:
      v.append(e[1])
  Ch=[]
  if len(v)>=2:
    for i in range(len(v)-1):
      Ch.append(((v[i],v[i+1]),G[v[i]][v[i+1]]))
    Ch.append(((v[-1],v[0]),G[v[-1]][v[0]]))
  return Ch
```

Listing 5.24: 2-MST heuristic python code. The code uses adjacency matrix representation, and returns the resulting Hamiltonian cycle.

5.8 Flow networks

Networks can be modelled as graphs, where vertices are junctions, intersections, places, or stations, and edges are connecting conduits, pipes, or tracks that support a flow of material, tokens, or items. Graph modelled networks, or **flow networks**, are conceptual models for many real-world examples, such as electrical circuits, traffic networks (roads, railways, air), commuting networks, telecommunication networks, distribution networks, and similar. Several vertices in the flow network can be taken as the flow sources (generators), which are responsible for generating flow in the network. The generated flow can be continuous or discrete. Examples of continuous flows include fluid or solid material flows (m^3/s), electrical currents (A), and similar. For discrete flows, we always have a number of items per time frame, such as persons per hour in commuting networks, vehicles per hours in traffic models, GB/s in telecommunication networks (remember, we are exchanging discrete packets in TCP/IP networks), and similar. Besides source vertices, we need to have destination (consumer) vertices in the flow network, which are responsible for eliminating flow from the network.

Definition 5.42 (Flow network). A **flow network** is a connected directed graph without loops and reverse edges

$$G = (V, E, c, f, s, d) \tag{5.83}$$

where c is the capacity mapping function that assign a capacity value to each edge of the network, f is the flow mapping function that assigns a flow value to each edge of the network, $s \in V$ is the source (generator) vertex, and $d \in V$ is the destination (consuming) vertex, such that $s \neq d$. Not having reverse edges means

$$\forall uv \in E \nexists vu \in E \tag{5.84}$$

The capacity mapping function can be defined as

$$c : E \to \mathbb{R}^+ \tag{5.85}$$

and is used to map a positive capacity value to each edge of the flow network G. Knowing that edges of the flow network represent connections, conduits, pipes, tracks, roads, we must interpret the capacity value in accordance to the context of the flow network. In electrical circuit, this capacity would be amperes, in roads cars per hour, in gas distribution network m^3/s, and so on.

The flow mapping function can be defined as

$$f : E \to \mathbb{R}^+ \tag{5.86}$$

and is used to map a positive flow value to each edge of the flow network G. We define a helping edge mapping function as

$$V_i, V_j \subseteq V$$
$$E(V_i, V_j) = \{uv : uv \in E, u \in V_i, v \in V_j\} \subseteq E \tag{5.87}$$

.

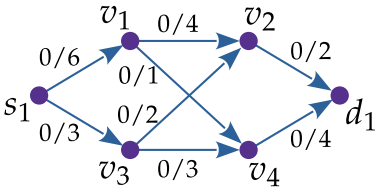


Figure 5.68: A flow network example. Labels of the network between vertices u and v show $f(u,v)/c(u,v)$ or 'flow / capacity' values.

The flow f has the following properties:

- **Capacity constraint**:

$$\forall uv \in E : 0 \leq f(uv) \leq c(uv)$$

  or the flow between adjacent vertices u and v cannot exceeded the capacity of the edge uv between them,

- **Flow preservation**:

$$\forall u \in V \setminus \{s, d\} : \sum_{v_i u \in E(V,u)} f(v_i u) = \sum_{u v_j \in E(u,V)} f(u v_j)$$

  which means that each vertex, not being the source or destination, must have total flow entering the vertex equal to the total flow exiting the vertex. Obviously, source vertices will have only exiting flows, while destination vertices will have only entering flows.

It is essential that the destination vertex d is reachable from the source vertex s, otherwise the flow network is meaningless, as there could not be any flow between source and destination vertices.

An example of flow network can be seen in Figure 5.68, where $s = s_1$, $d = d_1$, all flows are set to 0, and capacities are

$$c(s_1 v_1) = 6, c(s_1 v_3) = 3, c(v_1 v_2) = 4, ..., c(v_4 d_1) = 4$$

Due to the capacity constraint, and capacity mapping function definition, a flow cannot be established between non-adjacent vertices, as there is no capacity between them.

Definition 5.43 (Flow strength). A **flow strength** of a flow network is defined as

$$|f| = \sum_{s v_i \in E(s,V)} f(s v_i) - \sum_{v_j s \in E(V,s)} f(v_j s) \tag{5.88}$$

where s is the source vertex of the flow network. The flow strength is the net sum of the source vertex flow, which should be positive, since s should be generating some positive flow into the network.

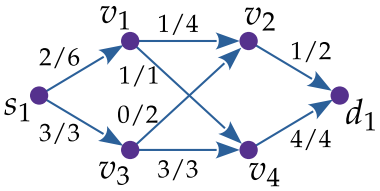


Figure 5.69: A flow network example with flows.

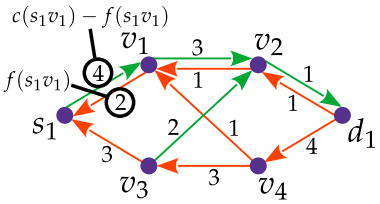


Figure 5.70: The residual network for the example in Figure 5.69. Labels mark the residual capacity.

Ford-Fulkerson algorithm

Definition 5.44 (Residual network). Given the flow network G, we can construct a graph $G_f = (V, E_f, c_f, f_f, s, d)$ that represents a **residual network**, where E_f are the edges of the residual network, which are not the same as the edges of the flow network E, c_f is the residual capacity mapping function, and f_f is the flow of the residual network.

The edges of the residual network G_f are constructed from the edges of the flow network G induced by the flow f and capacity c. The constructed edges can be defined as *forward edges*

$$E_{f_1} = \{uv : uv \in E, c(uv) - f(uv) > 0\} \tag{5.89}$$

and *reverse edges*

$$E_{f_2} = \{vu : uv \in E, f(uv) > 0\} \tag{5.90}$$

where the residual edges are the union of forward and reverse edges

$$E_f = E_{f_1} \cup E_{f_2} \tag{5.91}$$

The residual capacity mapping is defined as

$$c_f(uv) = \begin{cases} c(uv) - f(uv), & \exists uv \in E, c(uv) - f(uv) > 0 \\ f(vu), & \exists vu \in E, f(vu) > 0 \\ 0, & otherwise \end{cases} \tag{5.92}$$

Figure 5.69 shows a flow network G with some flow. Edge s_1v_1 has the flow $f(s_1v_1) = 2$ and the capacity $c(s_1v_1) = 6$. We transform this flow network into a residual network shown in Figure 5.70. In the residual network, according to Definition 5.44 and (5.91)(5.92) we get one forward edge s_1v_1 having the residual capacity $c_f(s_1v_1) = c(s_1v_1) - f(s_1v_1) = 6 - 2 = 4$ and one reverse edge v_1s_1 having the residual capacity $c_f(v_1s_1) = f(s_1v_1) = 2$.

Similar to the flow network G, the residual network G_f can have its own flows, defined by the flow mapping function f_f. If we have a flow $f(uv)$ between vertices u and v in the flow network G, and flow $f_f(uv)$ in the residual network G_f, we define the **flow augmentation** as

$$f \uparrow f_f(uv) = \begin{cases} f(uv) + f_f(uv) - f_f(vu), & \exists uv \in E \\ 0, & otherwise \end{cases} \tag{5.93}$$

Knowing that each edge $uv \in E_f$ has some residual capacity $c_f(uv)$, any path we can find from s to d in the residual network G_f supports additional flow f_f, which can be augmented to the flow f in the flow network G. Which brings us to the following lemma.

Lemma 5.3 (Flow augmentation strength). Given the flow network G and its flow induced residual network G_f, then the flow augmentation defined in (5.93) is having the strength $|f \uparrow f_f| = |f| + |f_f|$.

Proof: In the first part, we prove that the flow augmentation holds the **capacity constraint** property. This means that if we add the flow f_f to the flow f, it will not exceed the capacity c. We intuitively know that this holds, since the residual network G_f is created from the flow network G induced by the flow f.

$$\begin{aligned} (f \uparrow f_f)(uv) &= f(uv) + f_f(uv) - f_f(vu) \\ &\leq f(uv) + f_f(uv) \\ &\leq f(uv) + c_f(uv) \\ &= f(uv) + c(uv) - f(uv) \\ &= c(uv) \end{aligned}$$

which is finally

$$(f \uparrow f_f)(uv) \leq c(uv) \tag{5.94}$$

Then, we move on to proving that the flow augmenting preserves the **flow preservation** property. Which mean, even if we add the flow f_f to the flow f, each vertex except s and d will have the input flow equal to the output flow. To prove the flow augmentation we use the basic flow preservation expression $\sum_{v_iu \in E(V,u)} f(v_iu) = \sum_{uv_j \in E(u,V)} f(uv_j)$.

$$\begin{aligned}
&\forall u \in V \setminus \{s,d\} : \\
\sum_{uv_i \in E(u,V)} (f \uparrow f_f)(uv_i) &= \sum_{uv_i \in E(u,V)} (f(uv_i) + f_f(uv_i) - f_f(v_iu)) \\
&= \sum_{uv_i \in E(u,V)} f(uv_i) + \sum_{uv_i \in E(u,V)} f_f(uv_i) - \sum_{uv_i \in E(u,V)} f_f(v_iu) \\
&= \sum_{v_ju \in E(V,u)} f(v_ju) + \sum_{v_ju \in E(V,u)} f_f(v_ju) - \sum_{v_ju \in E(V,u)} f_f(uv_j) \\
= \sum_{v_ju \in E(V,u)} (f(v_ju) + f_f(v_ju) - f_f(uv_j)) &= \sum_{v_ju \in E(V,u)} (f \uparrow f_f)(v_ju)
\end{aligned}$$

which is finally

$$\sum_{uv_i \in E(u,V)} (f \uparrow f_f)(uv_i) = \sum_{v_ju \in E(V,u)} (f \uparrow f_f)(v_ju) \tag{5.95}$$

Given the previous source vertex s incident edges and adjacent vertices observation, we induce the strength of the augmented flow as

$$\begin{aligned}
|f \uparrow f_f| = \sum_{sv_i \in E(s,V)} (f \uparrow f_f)(sv_i) &- \sum_{v_js \in E(V,s)} (f \uparrow f_f)(v_js) \\
&= \sum_{sv_i \in E(s,V)} (f(sv_i) + f_f(sv_i) - f_f(v_is)) \\
&- \sum_{v_js \in E(V,s)} (f(v_js) + f_f(v_js) - f_f(sv_j)) \\
&= \sum_{sv_i \in E(s,V)} f(sv_i) - \sum_{v_js \in E(V,s)} f(v_js) \\
&+ (\sum_{sv_i \in E(s,V)} f_f(sv_i) - \sum_{sv_i \in E(s,V)} f_f(v_is)) \\
&- (\sum_{v_js \in E(V,s)} f_f(v_js) - \sum_{v_js \in E(V,s)} f_f(sv_j)) \\
&= |f| + |f_f|
\end{aligned} \tag{5.96}$$

where the last summation on the flow of the residual network f_f can be interpreted as

$$(\text{outgoing} - \text{reverse outgoing}) - (\text{incoming} - \text{reverse incoming})$$

which is the strength of the flow in the residual network. This proves the lemma and connects the flow network G, its flow induced residual network G_f, and the flow augmentation between these two networks.

So, how do we choose flows in the residual network G_f that can be augmented to the flow network G without breaking the capacity constraint and flow preservation?

Definition 5.45 (Augmenting path). Given the flow network G and its flow induced residual network G_f, any path p from s to d in the residual network G_f is called an **augmenting path**.

If we look at Figure 5.70, we can see a single augmenting path

$$p = s_1 v_1 v_2 d_1$$

Residual capacities on the path p are

$$c_f(s_1 v_1) = 4, c_f(v_1 v_2) = 3, c_f(v_2 d_1) = 1$$

We can calculate the residual capacity of the augmenting path p as

$$c_f(p) = min_{\forall uv \in p} c_f(uv) \tag{5.97}$$

Lemma 5.4 (Augmenting path flow). Given the residual network G_f and the augmenting path p, we can define the flow for the augmenting path as a function

$$f_{f_p} : E_f \to \mathbb{R}^+ \tag{5.98}$$

which can be calculated as

$$f_{f_p}(uv) = \begin{cases} c_f(p), & \exists uv \in p \\ 0, & otherwise \end{cases} \tag{5.99}$$

The augmenting path flow must have the same value for each edge in the path p, and it must be the minimal residual capacity on the path $c_f(p)$.

$$\forall uv \in p : f_{f_p}(uv) = c_f(p) \tag{5.100}$$

This way we ensure both capacity constraint and flow preservation properties on the residual network G_f. If we take that the augmenting path flow f_{f_p} is the only flow in the residual network G_f, then we have

$$|f_{f_p}| = c_f(p) > 0 \tag{5.101}$$

Once we ensured that, Lemma 5.3 allows us to augment the flow f_{f_p} to the flow network G.

Corollary 5.1 (Network flow augmentation). Given the flow network G, its flow induced residual network G_f, p as an augmenting path in G_f, and f_{f_p} as the flow through the augmenting path p, we can augment the flow f by f_{f_p} as

$$|f \uparrow f_{f_p}| = |f| + |f_{f_p}| > |f| \tag{5.102}$$

The proof of the previous corollary is given through Lemmas 5.3 and 5.4.

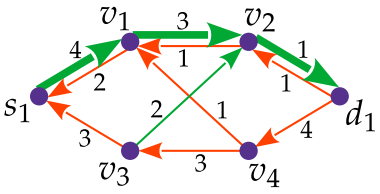


Figure 5.71: An example of the augmented path in the residual network.

Flow network cuts
We can split any flow network G in two disjunct partitions by dividing the network into two subset of vertices

$$S \subset V, D \subset V \tag{5.103}$$

such that $s \in S$ and $d \in D$. Since partitions must be disjunct, the following must hold

$$S \cap D = \emptyset, S \cup D = V \tag{5.104}$$

An example of the network cut can be seen in Figure 5.72. The partitions of the network are $S = \{s_1, v_1, v_3\}$ and $D = \{v_2, v_4, v_5, d_1\}$. Also, the network cut edges are $E(S,D) = \{v_1v_2, v_1v_4, v_3v_2, v_3v_4\}$ and $E(D,S) = \{v_5v_3\}$.

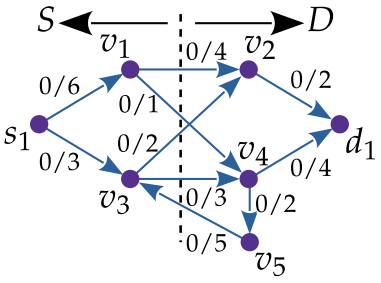


Figure 5.72: A flow network cut example.

We are interested to explore some of the network cut properties. The **network cut capacity** is calculated as

$$c(S,D) = \sum_{uv \in E(S,D)} c(uv) \tag{5.105}$$

We should notice that $c(S,D)$ does not include capacities of the reverse cut edges, directed from D to S, such as v_5v_3. The reason for that is revealed in the following text. Next, we defined the **network cut flow** as

$$f(S,D) = \sum_{uv \in E(S,D)} f(uv) - \sum_{vu \in E(D,S)} f(vu) \tag{5.106}$$

Lemma 5.5 (Network cut flow strength). Any network cut S, D has the flow equal to the network flow strength $f(S,D) = |f|$.

Proof: By using the flow preservation property $f(V \setminus \{s,d\}, V) = 0$ and knowing that $V = S \cup D$ and $S \cap D = \emptyset$, we can extend the flow strength as

$$\begin{aligned} |f| = f(s,V) &= f(s,V) + f(S \setminus \{s\}, V) \\ &= f(S,V) = f(S,V) - f(S,S) \\ &= f(S, V \setminus S) = f(S,D) \end{aligned} \tag{5.107}$$

By examining the example in Figure 5.73 we can see that the previous Lemma holds on a simple network case.

Definition 5.46 (Minimal cut). A network cut S, D that has the minimal capacity $c(S,D)$ is called a **minimal cut**. It is obtained by solving the optimization expression

$$c_m(S,D) = \min_{S,D} c(S,D) \tag{5.108}$$

Corollary 5.2 (Network cut capacity constraint). The capacity constraint can be applied to any network cut, which means that for any network cut S, D we have $0 \leq f(S,D) = |f| \leq c(S,D)$.

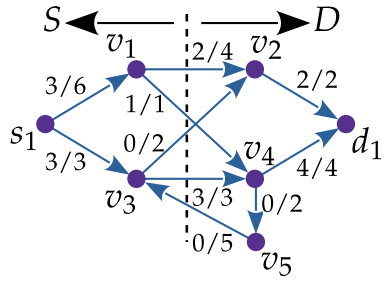


Figure 5.73: Flows added to the example in Figure 5.68.

In Figure 5.73 we have the following network cut flow

$$\begin{aligned} f(S,D) = f(v_1v_2) + f(v_1v_4) \\ + f(v_3v_4) = 6 \end{aligned}$$

the network cut capacity is

$$\begin{aligned} c(S,D) = c(v_1v_2) + c(v_1v_4) \\ + c(v_3v_2) + c(v_3v_4) = 10 \end{aligned}$$

and the flow strength

$$|f| = f(s_1v_1) + f(s_1v_3) = 6$$

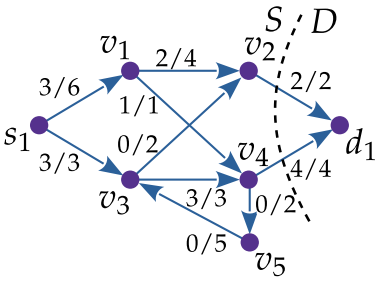


Figure 5.74: Minimal cut for the example in Figure 5.68, where $D = \{d_1\}$ and $c_m(S,D) = 6$, which is the same as the previously calculated flow strength.

Proof: Using (5.106) and Lemma 5.5 we can expand the network cut flow

$$\begin{aligned} |f| = f(S,D) &= \sum_{uv \in E(S,D)} f(uv) - \sum_{vu \in E(D,S)} f(vu) \\ &\leq \sum_{uv \in E(S,D)} f(uv) \\ &\leq \sum_{uv \in E(S,D)} c(uv) \\ &= c(S,D) \end{aligned} \tag{5.109}$$

Theorem 5.4 (Max-flow min-cut theorem). Given the flow network G having flow f, source s, and destination d, all of the following conditions hold true:

1. f is the maximum flow on G,
2. there is no more augmenting paths in the residual network G_f,
3. there is a cut S, D where $|f| = c(S,D)$.

Proof: $1 \rightarrow 2$. Let us suppose that there is an augmenting path p in the residual network G_f. This means that there is a flow f_{f_p} in the augmenting path p. According to the Corollary 5.1 we can augment the flow f by f_{f_p}, and get $|f \uparrow f_{f_p}| > |f|$, which is directly contradicting the condition 1.

$2 \rightarrow 3$. If there is no augmenting path p in the residual network G_f then there is no path between s and d. This means that there must be a network cut S, D where there are no edges between sets of vertices S and D in the residual network G_f, which means that due to (5.89) we will have

$$\forall uv \in E(S,D) \nexists uv \in E_f \Rightarrow c_f(uv) = 0, c(uv) = f(uv) \tag{5.110}$$

Due to Lemma 5.5 and Corollary 5.2, for the network cut S, D we have

$$|f| = f(S,D) = c(S,D) \tag{5.111}$$

which is exactly the condition 3.

$3 \rightarrow 1$. Corollary 5.2 is the direct proof, saying that $|f| \leq c(S,D)$, where $|f| = c(S,D)$ is the maximal flow.

Algorithm 5.24: Ford-Fulkerson algorithm.

```
procedure FordFulkerson(G)
    for each edge uv in E(G) do
        f(uv) ← 0
    calculate the residual network G_f                ▷ Def. 5.44
    while there is an augmenting path p in G_f do
        calculate f_{f_p}                             ▷ (5.99)
        perform the flow augmentation (f ↑ f_{f_p})   ▷ (5.93)
        calculate the residual network G_f            ▷ Def. 5.44
```

The complexity is $O(f_m * |E|)$

Ford-Fulkerson algorithm is a simple algorithm that uses the previous lemmas and corollaries to determine the maximal flow in the input network. In each step we calculate (or update) the residual network G_f. Then we try to find an augmenting path p in this residual network. If the augmenting path p has been found, we can calculate the augmenting path flow f_{f_p} (5.99), which has the strength (5.101) equal to the minimal residual capacity (5.97) along the path p. So calculated augmenting path flow f_{f_p} is then augmented to the original flow $(f \uparrow f_{f_p})$ (5.93). Then we calculate (or update) the residual network G_f, repeat finding a new augmenting path p, and so on. These steps are repeated as long as we can find another augmenting path p in the residual network G_f.

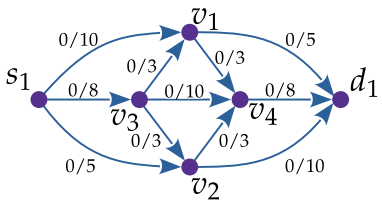
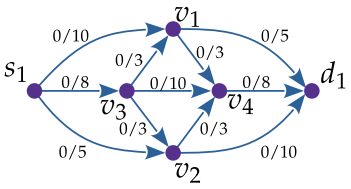
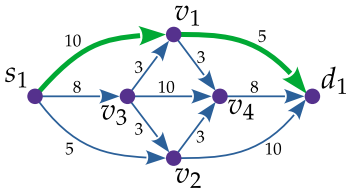
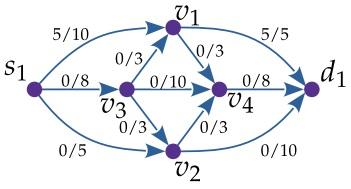
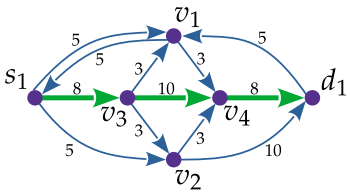
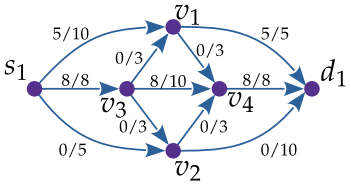
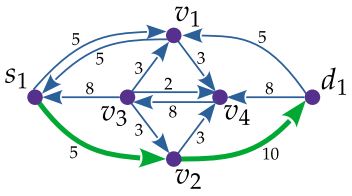
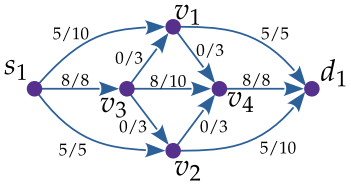
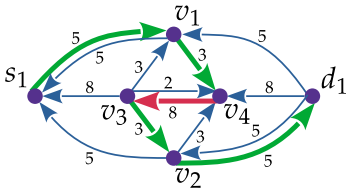
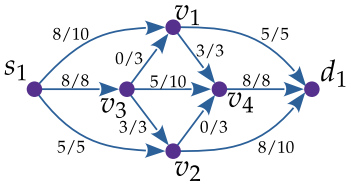
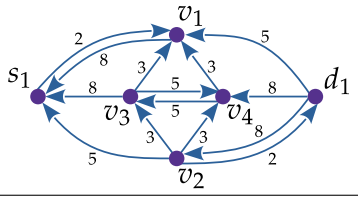


Figure 5.75: A network example.

Step	G	G_f
1: The initial residual network has the residual capacities equal to the original flow network capacities. We choose the augmenting path $p = s_1v_1d_1$, which has the minimal residual capacity $c_f(p) = 5$. The augmenting path flow strength is therefore $\|f_{f_p}\| = 5$.		
2: We choose the augmenting path $p = s_1v_3v_4d_1$, which has the minimal residual capacity $c_f(p) = 8$. The augmenting path flow strength is therefore $\|f_{f_p}\| = 8$.		
3: We choose the augmenting path $p = s_1v_2d_1$, which has the minimal residual capacity $c_f(p) = 5$. The augmenting path flow strength is therefore $\|f_{f_p}\| = 5$.		
4: We choose the augmenting path $p = s_1v_1v_4v_3v_2d_1$, which has the minimal residual capacity $c_f(p) = 3$. The augmenting path flow strength is therefore $\|f_{f_p}\| = 3$. Notice that we used reversed edge v_4v_3 from the residual network G_f.		
5: We have no more available augmenting paths. The final maximal flow is $\|f\| = 21$.		

In the step 4 we had only one possible augmenting path p, which uses reverse path v_4v_3. This is legitimate, as it reduces the original flow $f(v_3v_4)$ after the flow augmentation $(f \uparrow f_{f_p})$ (5.93). This is also the main reason why reverse edges are created when creating residual networks. However, using reverse edges can be avoided by careful picking new augmenting paths.

Exercise 5.8.1. Using Ford-Fulkerson algorithm, find the sequence of augmenting paths and their minimal residual capacities, which combined make the maximal flow $|f| = 21$ for the flow network given in Figure 5.75, **without using reverse edges** in residual networks.

Solution is the following sequence of augmenting paths

$$p_1 = s_1v_1d_1, c_f(p_1) = 5$$
$$p_2 = s_1v_1v_4d_1, c_f(p_2) = 3$$
$$p_3 = s_1v_3v_4d_1, c_f(p_3) = 5$$
$$p_4 = s_1v_3v_2d_1, c_f(p_4) = 3$$
$$p_5 = s_1v_2d_1, c_f(p_5) = 5$$

The complexity of the Ford-Fulkerson algorithm can be expressed as $T(\text{FF}) = T(\text{init}) + f_m * T(\text{augmenting path})$, where f_m is the number of iterations of the while loop, which means the number of flow augmentations. In the initialization we create a new residual network, which is of complexity $\Theta(|E|)$. Finding augmenting paths can be done either by BFS or DFS, both having the complexity $O(|V| + |E|)$. For $|E| \geq |V| - 1$, the complexity of finding a new augmenting path becomes $O(|E|)$. Finally, the complexity of the Ford-Fulkerson algorithm is $T(\text{FF}) = \Theta(|E|) + f_m * O(|E|)$.

Edmonds-Karp algorithm

The Edmonds-Karp algorithm is an extension of the Ford-Fulkerson algorithm. The major distinction is the way how augmenting paths are found. In the Edmonds-Karp algorithm, BFS is used to find the shortest possible augmenting path, with respect to the total number of edges. BFS is applied to the current residual network, without taking in consideration edges weights. It means, we perform BFS on the weightless residual network to find physically the shortest path from s to d.

The complexity of each iteration in Ford-Fulkerson algorithm is $O(|E|)$. On top of that we need to add the BFS complexity for finding the shortest augmenting paths, we get $O(|V| * |E|^2)$ for Edmonds-Karp algorithm.

Programming tips: To implement the Edmonds-Karp algorithm in Listing 5.25, we need to implement the BFS path searching function `BFSPath`. This function needs to return the shortest path from s to d in form of a list.

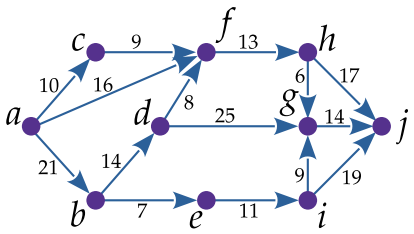
```
def EdmondsKarp(G,s,d):
  G={'C':G, 'F':NewAdjMatrix(G),
     'Gf':NewAdjMatrix(G,True)}
  p=BFSPath(MatrixToTable(G['Gf']),s,d)
  while len(p)>0:
    cfp=None
    for i in range(0,len(p)-1):
      u=p[i]
      v=p[i+1]
      if cfp is None or
          cfp>G['Gf'][u][v]:
        cfp=G['Gf'][u][v]
    for i in range(0,len(p)-1):
      u=p[i]
      v=p[i+1]
      f=G['F'][u][v]=G['F'][u][v]+cfp
      G['Gf'][u][v]=G['C'][u][v]-f
      G['Gf'][v][u]=f
    p=BFSPath(MatrixToTable(G['Gf']),s,d)
  fmax=0
  for v in G['C']:
    fmax=fmax+G['F'][s][v]
    fmax=fmax-G['F'][v][s]
  return (fmax,G['F'])
```

Listing 5.25: Edmonds-Karp algorithm python code.

```
def BFS(G,s,d):
  vis={}
  Q=collections.deque()
  Q.append(s)
  vis[s]={'p':None}
  while Q:
    u=Q.popleft()
    for v in G[u]['adj']:
      if v not in vis:
        vis[v]={'p':u}
        Q.append(v)
      if v is d: return vis
  return vis

def BFSPath(G,s,d):
  vis=BFS(G,s,d)
  if d not in vis: return []
  v=vis[d]
  path=[d]
  while v['p'] is not None:
    path.insert(0,v['p'])
    v=vis[v['p']]
    if path[0] is not s: return []
  return path
```

Listing 5.26: BFS for Edmonds-Karp algorithm.

Exercise 5.8.2. Find the maximal flow for the following flow network using Edmonds-Karp algorithm.



Solution is $|f_{max}| = 34$.

Specific cases

Multiple sources and destinations

Although the previous theory regarding flow networks assumes one source s and one destination d, the whole flow network mathematical formalism could be extended to support multiple sources and destinations.

Looking at Figure 5.76 we can identify multiple sources $\mathcal{S} = \{a_1, a_2, a_3\}$ and multiple destinations $\mathcal{D} = \{j_1, j_2\}$. The most simple solution is use the following transformation. To transform a multiple sources and destinations flow network, we add one *super-source* s and one *super-destination* d to the network. Then we connect the super-source s to all sources from the set $\mathcal{S}$ through edges having ∞ capacity, directed from the super-source s to each source in $\mathcal{S}$. Additionally, we connect all destinations from the set $\mathcal{D}$ to the super-destination d through edges having ∞ capacity, directed from destinations in $\mathcal{D}$ to the super-destination d. The previously described transformation turns a multiple sources and destinations flow network into a single source and destination flow network. This way, we can use all previously described algorithms and mathematical formalisms.

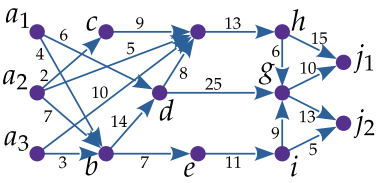


Figure 5.76: Multiple sources and destinations network example.

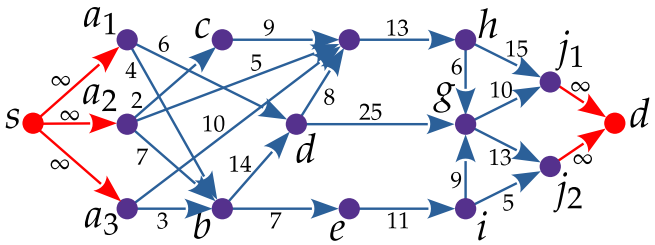


Figure 5.77: Multiple sources and destinations transformation.

Counter edges

Having counter edges in flow networks can pose problems when creating residual networks. Edges we create in Definition 5.44 for counter edges would result in more complex residual capacities than in (5.92).

In Figure 5.78 we can see a counter edges between vertices v_3 and v_4. This can be transformed by inserting a dummy vertex in one of

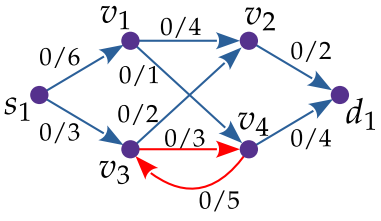


Figure 5.78: Counter edges example.

the counter edges. By adding vertex v_4', we have broken the counter

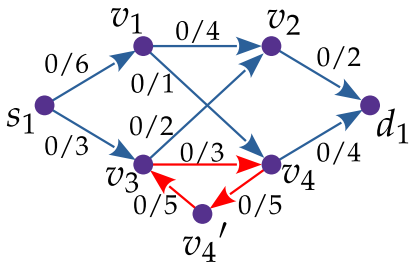


Figure 5.79: Counter edges transformation.

edge and creation of the residual network can be done according to Definition 5.44. When examining the final flows, we need to be aware that flows coming from and going to the inserted vertex (v_4') are being redirected from another vertex.

Bibliography

Geir Agnarsson and Raymond Greenlaw. *Graph theory: Modeling, applications, and algorithms*. Prentice-Hall, Inc., 2006.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd-edition*. MIT Press and McGraw-Hill, 2009.

Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. *Computational Geometry*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-77974-2.

EW Dijkstra. Some theorems on spanning subtrees of a graph. *Indag. math*, 22(2):196–199, 1960.

Adam Drozdek. *Data Structures and algorithms in C++*. Nelson Education, 2012.

Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *International colloquium on automata, languages, and programming*, pages 943–955. Springer, 2003.

Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Annual Symposium on Combinatorial Pattern Matching*, pages 181–192. Springer, 2001.

V. Klee and G. Minty. How good is the simplex algorithm? In *Inequalities III - Proceedings of the Third Symposium on Inequalities*, pages 159–175, University of California, Los Angeles, USA, 1972. Academic Press.

Donald E Knuth. *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014.

Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher Strong, Clark Barrett, and Mykel J. Kochenderfer. Algorithms for Verifying Deep Neural Networks. *Foundations and Trends® in Optimization*, 4(3-4):244–404, February 2021. ISSN 2167-3888, 2167-3918. DOI: 10.1561/2400000035. URL https://www.nowpublishers.com/article/Details/OPT-035. Publisher: Now Publishers, Inc.

Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.

Donald R Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6): 668–676, 1990.

Robert Sedgewick. *Algorithms in Java, Parts 1-4*. Addison-Wesley Professional, 2002.

Keith Sklower. A tree-based packet routing table for berkeley unix. In *USENIX Winter*, volume 1991, pages 93–99. Citeseer, 1991.

Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM*, 51(3):385–463, May 2004. ISSN 0004-5411. DOI: 10.1145/990308.990310. URL https://doi.org/10.1145/990308.990310.

Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 343–350, 2000.

Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

J. v. Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, December 1928. ISSN 1432-1807. DOI: 10.1007/BF01448847. URL https://doi.org/10.1007/BF01448847.

Roman Vershynin. Beyond Hirsch Conjecture: Walks on Random Polytopes and Smoothed Complexity of the Simplex Method. *SIAM Journal on Computing*, 39(2):646–678, January 2009. ISSN 0097-5397. DOI: 10.1137/070683386. URL https://locus.siam.org/doi/abs/10.1137/070683386. Publisher: Society for Industrial and Applied Mathematics.

Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973.

Index