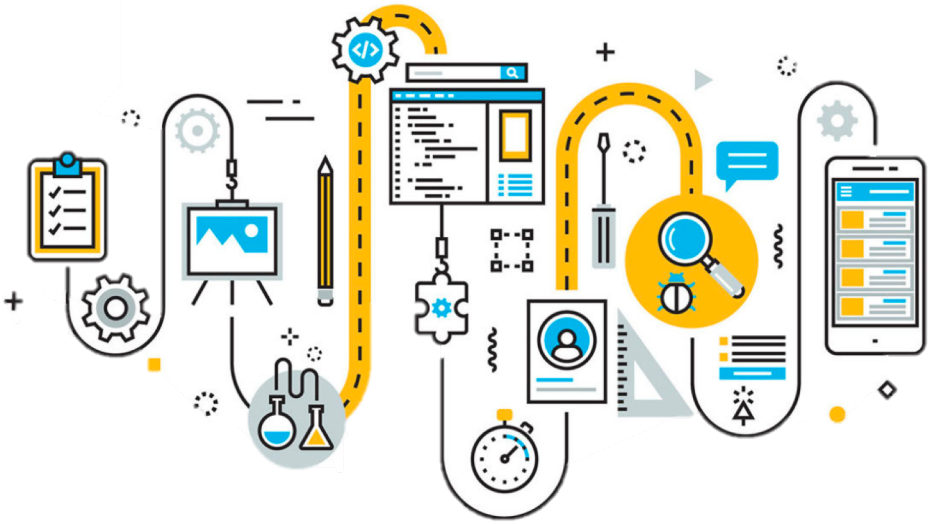
UANL
UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN

FIME
FACULTAD DE INGENIERÍA MECÁNICA Y ELÉCTRICA

Facultad de Ingeniería Mecánica y Eléctrica.
Posgrado de Ingeniería en Sistemas.
Doctorado en Ingeniería de Sistemas.



Portafolio de Evidencias

Análisis y diseño de algoritmos
Segundo semestre.

Profesora: Dra. Satu Elisa Schaeffer.

Alumna: Mayra Cristina Berrones Reyes

30 de Mayo 2020

Exercises of chapter 1

January 29, 2020

Mayra Cristina Berrones Reyes

1 Approximation

Determine experimentally in which situations it is convenient to resort to approximations of the factorial like Equation 2 and Equation 3. That is to say, how much computational time is saved, and at what precision cost. Additionally, what happens if the values ??of e are also approximates. (For example, with Equation 2 or 3 or some other of the various forms that are known to approximate their value) and of π (for which there are numerous approximations) instead of assuming they were constants of fixed precision.

Factorial numbers can be used in combinational probability, to calculate combinations and permutations. Through this, factorials are also often used to calculate probabilities, such as binomial coefficient where ! marks the factorial number, and it is defined as all integer number k $\in \mathbb{Z}$, such as

$$k! = k * (k-1) * (k-2) * ... * 2 * 1 \tag{1}$$

A very useful approximation of the factorial is the Stirling approximation:

$$k! \approx k^k e^{-k} \sqrt{2\pi k} \tag{2}$$

and the improved version by Gosper:

$$k! \approx \sqrt{\pi(2k + \frac{1}{3})} k^k e^{-k} \tag{3}$$

This approximations are used when the factorial number is too big. However, this approximations come with various degrees of error. Equation 4 shows the formula to get the percentage error.

$$E = \left| \frac{Tv - Av}{Tv} \right| * 100\% \tag{4}$$

Where:

E: Percentage error.
Tv: True value
Av: Approximation value

In Python the library math has already a fixed parameter for the value π and e.

```
In [27]:
1 import math
2 from math import pi, e
3 from timeit import timeit
4
5 print (pi, e)
```

```
3.141592653589793 2.718281828459045
```

First we begin our experiment by taking those parameters of π and e from the math library, and using the Stirling approximation.[1] In Table 1 we can se the results. Column N is the number we used to take the value of the factorial. N! is the answer provided by python form the same math library, and the column Stirling is the approximation. Time is the processing time it took to compile, and the error was calculated with Equation 4.

Table 1: Results of the Stirling approximation

N	N!	Stirling	Time	Error
1	1	0.922137009		0.077862991
5	120	118.941305	0.000825167	0.008822459
10	3628800	3598814.56	0.001646757	0.008263183
25	1.55112E+25	1.54596E+25	0.001966	0.003327607
50	3.04141E+64	3.03634E+64	0.002142906	0.001665256
75	2.4809E+109	2.4782E+109	0.002707958	0.001110487
100	9.3326E+157	9.3248E+157	0.002886772	0.000832983
125	1.8827E+209	1.8814E+209	0.003069878	0.000666443
130	6.4669E+219	6.4627E+219	0.003262997	0.000640819

Then in Table 2 we have the results of the Gosper Equation 3.
We see that the bigger the value of N, the approximation gets similar to the true value.

As the next step on our experimentation, we wanted to explore a little more inside the formulas of Stirling and Gosper, since both of them use the π and e

[1] All experimentation mentioned we used a code in python, and it can be consulted on the github

Table 2: Results of the Gosper approximation.

N	N!	Gosper	Time	Error
1	1	0.996		0.003978193
5	120	120.96	0.012824059	-0.008050433
10	3628800	3628681.791	0.013508797	3.25753E-05
25	1.55112E+25	1.551 E+25	0.013875008	1.08842E-05
50	3.04141E+64	3.041 E+64	0.014073133	2.7494E-06
75	2.4809E+109	2.480 E+109	0.014258862	1.22616E-06
100	9.3326E+157	9.332 E+157	0.016538858	6.90896E-07
125	1.8827E+209	1.882 E+209	0.016858816	4.42627E-07
130	6.4669E+219	6.466 E+219	0.017882824	4.09269E-07

values, and they are approximations as well.

Both of this values have formulas to get a better approximation to its real value. For example, Equation 5 for the π value called the Leibniz formula, and Equation 6 for the e value. In Table 3 the column of Value for π and e is the value of n in the equations.

$$\pi = 4 * \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+2} = \frac{\pi}{4} \tag{5}$$

$$e = (1 + \frac{1}{n})^n \tag{6}$$

Table 3: Description of the distribution of the values for π and e.

Value for π and e	π	e
10	3.041839619	2.59374246
100	3.131592904	2.704813829
1000	3.140592654	2.716923932
10000	3.141492654	2.718145927
100000	3.141582654	2.718268237
1000000	3.141591654	2.718280469
10000000	3.141592554	2.718281694
100000000	3.141592644	2.718281798

Now we show the results of the experimentation in Table 4 and 5. Here is where the computational time shows some significance, because in the larger ones, it took more than a minute to compile and give us an answer.

Table 4: Results of the Stirling approximation using the different values of π and e

N	N!	Stirling	Time	Error
1	1	0.922232625		0.077767375
5	120	147.7417932	0.002756834	-0.23118161
10	3628800	3776077.14	0.004431009	-0.040585632
25	1.55112E+25	1.56514E+25	0.009263992	-0.009039953
50	3.04141E+64	3.0439E+64	0.04496026	-0.000817607
75	2.4809E+109	2.4791E+109	0.343361855	0.000737427
100	9.3326E+157	9.3253E+157	2.778558016	0.000783175
125	1.8827E+209	1.8814E+209	25.75114012	0.000660286
130	6.4669E+219	6.4627E+219	251.0401418	0.000639381

Table 5: Results of the Gosper approximation using the different values of π and e.

N	N!	Gosper	Time	Error
1	1	1.527525232	0	-0.527525232
5	120	150.7740198	0.00718689	-0.256450165
10	3628800	3807416.223	0.005645037	-0.049221843
25	1.55112E+25	1.57035E+25	0.012120962	-0.012397832
50	3.04141E+64	3.04896E+64	0.056209087	-0.002484249
75	2.4809E+109	2.4818E+109	0.380222797	-0.000372249
100	9.3326E+157	9.3331E+157	3.328314781	-4.91594E-05
125	1.8827E+209	1.8827E+209	30.81744933	-5.71849E-06
130	6.4669E+219	6.4669E+219	299.2242897	-1.02921E-06

2 Conclusions

At the end of the experimentation, we noticed that the second part of the experiment, when we began to change the parameters of π and e, the computational time started to grow, and comparing the first results where we used the default ones given by python, there is not much difference in accuracy of the approximation on the bigger numbers.

On the smaller numbers, the difference between the first and the second experiment is quite noticeable. This being because the accuracy of the π and e values are not very good.

With this we conclude that the accuracy of the π and e values are very significant on the result of the Stirling and Gosper approximation formulas. Also, the Gosper formula proves to be the more accurate of the two on the bigger numbers.

Exercises of chapter 2

February 5, 2020

Mayra Cristina Berrones Reyes

1 Complexity experiments

To compare the complexity of the best case, worst case, and average case, several sorting algorithms can be executed, giving it as an input, one at the time, all possible permutations of an array.

Performing multiple replicas with each entry to establish a typical execution time (to counteract the noise introduced by the operating system with its scheduling algorithms and other activities that take place on the computer) the permutation with the shortest execution time is the best case , the longest time is the worst case and the average among all these permutations is the average case in experimental terms. By increasing the number of elements in the array, it is possible to estimate the shape of the asymptotic growth curves of better and worse case as well as the average case. Some algorithms win in one aspect while losing in another. Which algorithm is good in what circumstances? In general, no reasonable algorithm will always be better to all the others.

2 Sorting algorithms

A sorting algorithm is made up from a series of instructions that take a list of elements, performs several operations (depending on the algorithm) and has as an output the list but sorted. To select the best suited algorithm for our array, we must compare their performance to see which one is best. The two performance metrics for this type of algorithms are time and space complexity.

In this experiments we will see the behavior of different sorting algorithms through their time complexity.

2.1 Time complexity

Big O notation is useful when analyzing algorithms for efficiency. For example, the time (or the number of steps) it takes to complete a problem of size n. The algorithms we are going to be experimenting with are the ones shown in Table 1.

Table 1: Time complexity of the selected sorting algorithms

Algorithm	$Best - case$	$Worst - case$	$Average - case$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Heap sort	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$
Merge sort	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$
Quick sort	$O(nlog(n))$	$O(n^2)$	$O(nlog(n))$

The idea of this experiment is to measure the time it takes each of this sorting algorithms to sort all the permutations of arrays of different sizes. The permutation with the least amount of processing time will be the best case scenario, the permutation with the bigest processing time will be the worst case scenario, and the average of all of the processing times will be the average case.

In this experiment we use several libraries of python to help us build this arrays faster, such as *random* to build an array with different numbers each iteration, and *itertools* to calculate all the permutations of the array. In Figure 1 we see the two functions to create an array, and a list with all its permutations.

Figure 1: Excerpt of the code used in this experiment

```
In [1]: import itertools
        import random
        from timeit import timeit

In [2]: def crear_lista(num_elementos):
            m = 1
            lista_aleatoria = []
            while m <= num_elementos:
                n = random.randint(1,100)
                lista_aleatoria.append(n)
                m +=1
            return lista_aleatoria

In [3]: def len_lista(lista):
            perm = list(itertools.permutations(lista))
            return perm
```

In every number of elements, we made 100 iterations of arrays with different elements in it. We began with 4 elements, and kept adding one until we reached 9 elements. To see how many permutations had each array, we only needed to know the factorial of the number of elements. This is exemplified in Table 2.

Table 2: Number of permutations for each number of elements on the arrays.

Number of elements	Factorial of number
4	24
5	120
6	720
7	5040
8	40320
9	362880

From Table 3 to 8 we calculated the average of all 100 iterations of the best and worst cases, as well as an average of the average cases in all 6 sorting algorithms. Later, form Figures 2 to 7 we show the best, worst, and average cases in a graphic of each of the algorithms.

Table 3: Description of the average of the 100 iterations in the best, worst and average cases of the **bubble sort algorithm**.

Number of elements	$Best-case$	$Worst-case$	$Average-case$
4	$2.0425x10^{-05}$	$2.1887x10^{-06}$	$5.3192x10^{-06}$
5	$1.6544x10^{-05}$	$2.1362x10^{-06}$	$5.4265x10^{-06}$
6	$3.8385x10^{-05}$	$1.7094x10^{-06}$	$5.7338x10^{-06}$
7	$9.4707x10^{-05}$	$2.0742x10^{-06}$	$7.1020x10^{-06}$
8	$1.7677x10^{-04}$	$2.2077x10^{-06}$	$8.7505x10^{-06}$
9	$1.2330x10^{-03}$	$2.1433x10^{-06}$	$1.0658x10^{-05}$

Table 4: Description of the average of the 100 iterations in the best, worst and average cases of the **selection sort algorithm**.

Number of elements	$Best-case$	$Worst-case$	$Average-case$
4	$4.9520x10^{-06}$	$3.3545x10^{-06}$	$4.0137x10^{-06}$
5	$2.0688x10^{-05}$	$4.0483x10^{-06}$	$5.0757x10^{-06}$
6	$8.5609x10^{-05}$	$4.8876x10^{-06}$	$5.8708x10^{-06}$
7	$4.1342x10^{-04}$	$5.5790x10^{-06}$	$6.8088x10^{-06}$
8	$1.2736x10^{-03}$	$6.3324x10^{-06}$	$7.4544x10^{-06}$
9	$3.8825x10^{-03}$	$7.1430x10^{-06}$	$8.7374x10^{-06}$

Lastly we have the Figures 8, 9, 10 in which we put all the sorting algorithms together and showed them divided by best, worst, and average to see better the contrast between all the algorithms performances.

Figure 2: Graphic description of the behavior of the data colected in this experiment for the bubble sort algorithm

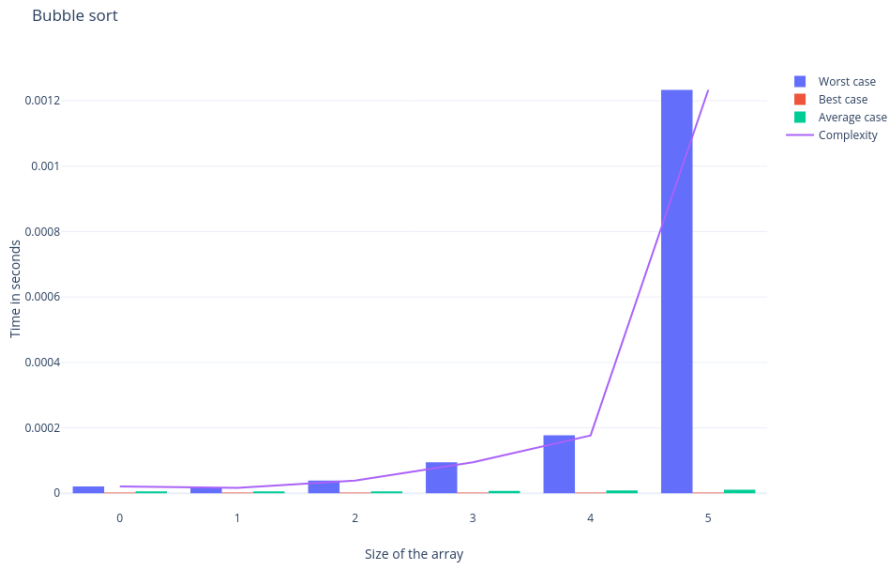


Figure 3: Graphic description of the behavior of the data colected in this experiment for the selection sort algorithm

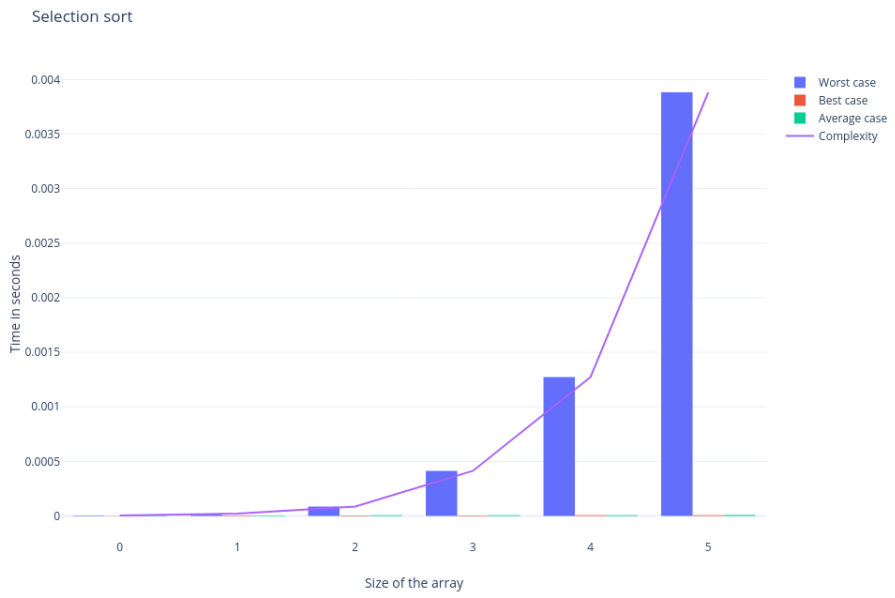


Figure 4: Graphic description of the behavior of the data colected in this experiment for the insertion sort algorithm

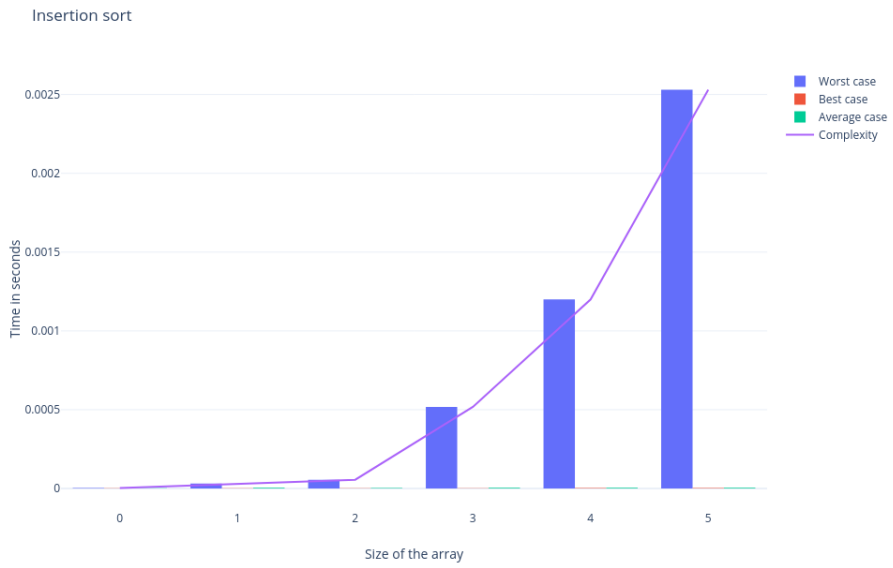


Figure 5: Graphic description of the behavior of the data colected in this experiment for the heap sort algorithm

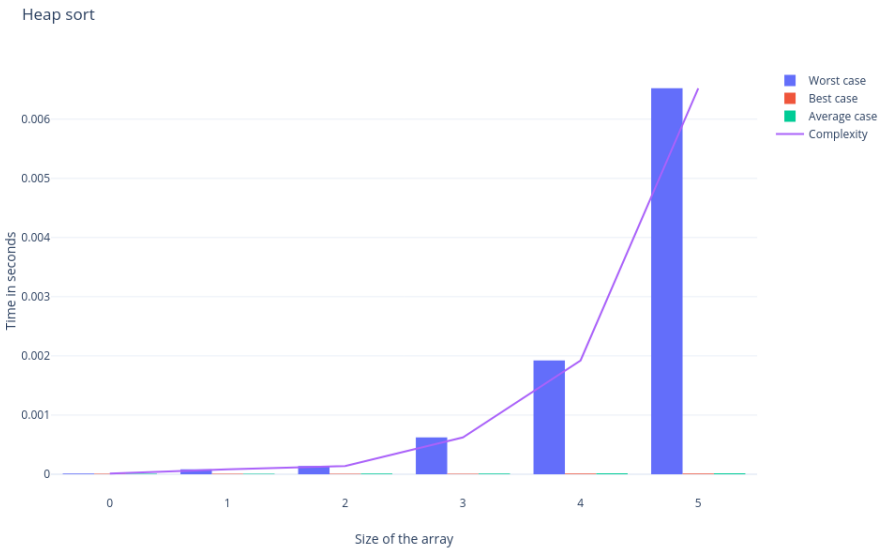


Figure 6: Graphic description of the behavior of the data colected in this experiment for the merge sort algorithm

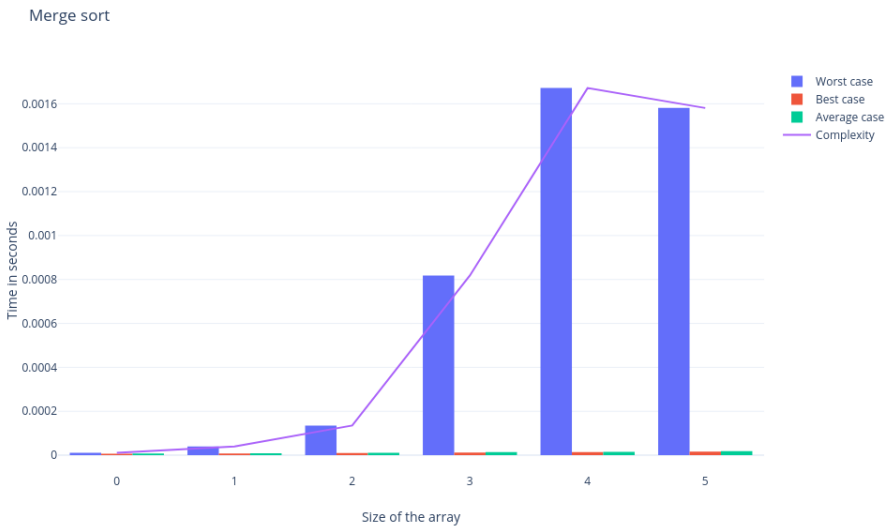


Figure 7: Graphic description of the behavior of the data colected in this experiment for the quick sort algorithm

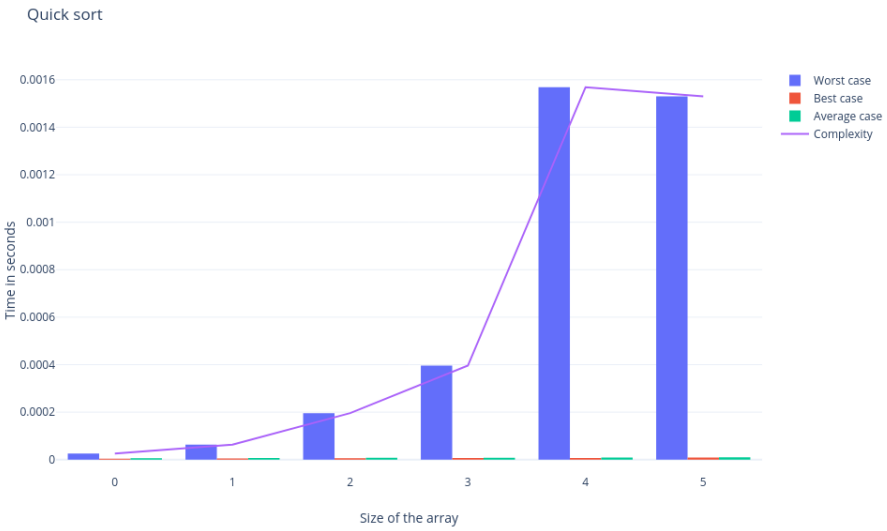


Figure 8: Graphic description of the average of the best cases of all of the sorting algorithms used in this experiments

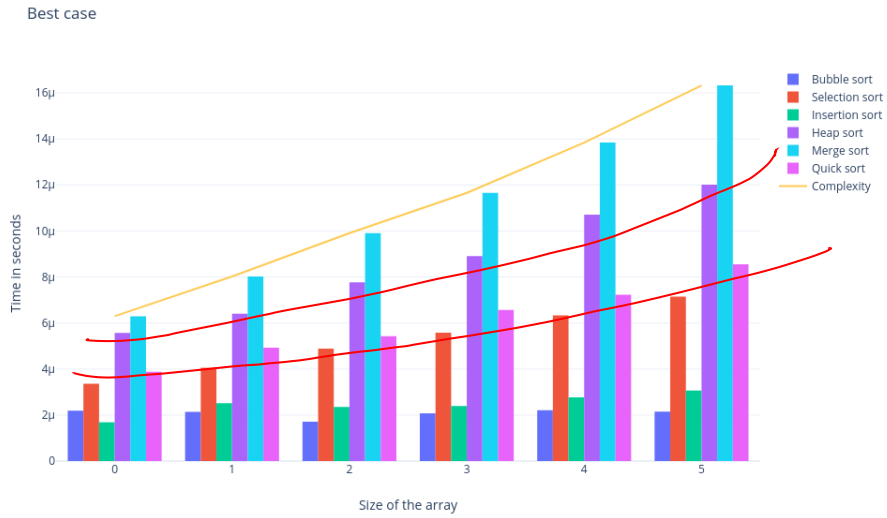


Figure 9: Graphic description of the worst of the best cases of all of the sorting algorithms used in this experiments

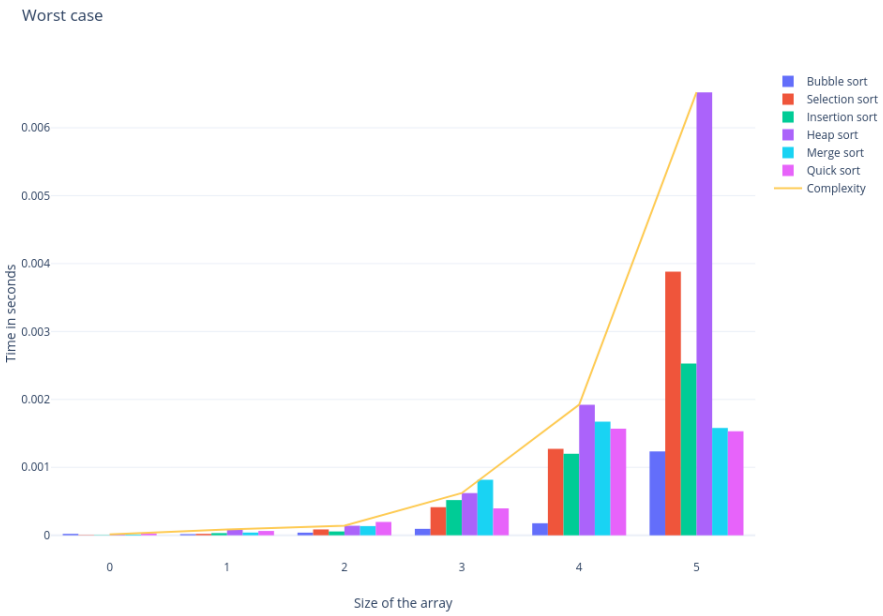


Table 5: Description of the average of the 100 iterations in the best, worst and average cases of the **insertion sort algorithm**.

Number of elements	$Best - case$	$Worst - case$	$Average - case$
4	$3.9959x10^{-06}$	$1.6856x10^{-06}$	$2.5314x10^{-06}$
5	$3.1478x10^{-05}$	$2.5105x10^{-06}$	$4.2121x10^{-06}$
6	$5.6040x10^{-05}$	$2.3532x10^{-06}$	$3.6283x10^{-06}$
7	$5.1801x10^{-04}$	$2.3937x10^{-06}$	$4.3553x10^{-06}$
8	$1.1991x10^{-03}$	$2.7704x10^{-06}$	$4.9595x10^{-06}$
9	$2.5292x10^{-03}$	$3.0589x10^{-06}$	$5.6250x10^{-06}$

Table 6: Description of the average of the 100 iterations in the best, worst and average cases of the **heap sort algorithm**.

Number of elements	$Best - case$	$Worst - case$	$Average - case$
4	$1.2109x10^{-05}$	$5.5647x10^{-06}$	$7.0568x10^{-06}$
5	$8.3866x10^{-05}$	$6.4063x10^{-06}$	$9.0000x10^{-06}$
6	$1.4008x10^{-04}$	$7.7701x10^{-06}$	$1.0264x10^{-05}$
7	$6.2078x10^{-04}$	$8.9073x10^{-06}$	$1.2097x10^{-05}$
8	$1.9230x10^{-03}$	$1.0705x10^{-05}$	$1.4125x10^{-05}$
9	$6.5225x10^{-03}$	$1.2009x10^{-05}$	$1.6323x10^{-05}$

Table 7: Description of the average of the 100 iterations in the best, worst and average cases of the **merge sort algorithm**.

Number of elements	$Best - case$	$Worst - case$	$Average - case$
4	$1.1375x10^{-05}$	$6.2943x10^{-06}$	$7.5241x10^{-06}$
5	$3.9556x10^{-05}$	$8.0180x10^{-06}$	$9.1617x10^{-06}$
6	$1.3530x10^{-04}$	$9.9039x10^{-06}$	$1.1302x10^{-05}$
7	$8.1806x10^{-04}$	$1.1654x10^{-05}$	$1.3662x10^{-05}$
8	$1.6717x10^{-03}$	$1.3838x10^{-05}$	$1.5335x10^{-05}$
9	$1.5808x10^{-03}$	$1.6322x10^{-05}$	$1.8028x10^{-05}$

3 Conclusions

After reading some things about the complexity of time in this algorithms, and the concept of big O, reviewing the results of the tables and figures, they seem to fit in the normal behavior for each type of algorithm.

Reading further into the results, we see that all of the algorithms seem to have a similar performance in the size 4, 5, and 6 of elements on the array, and it is very clear in Figure 9 that the bigger the number of elements in the array,

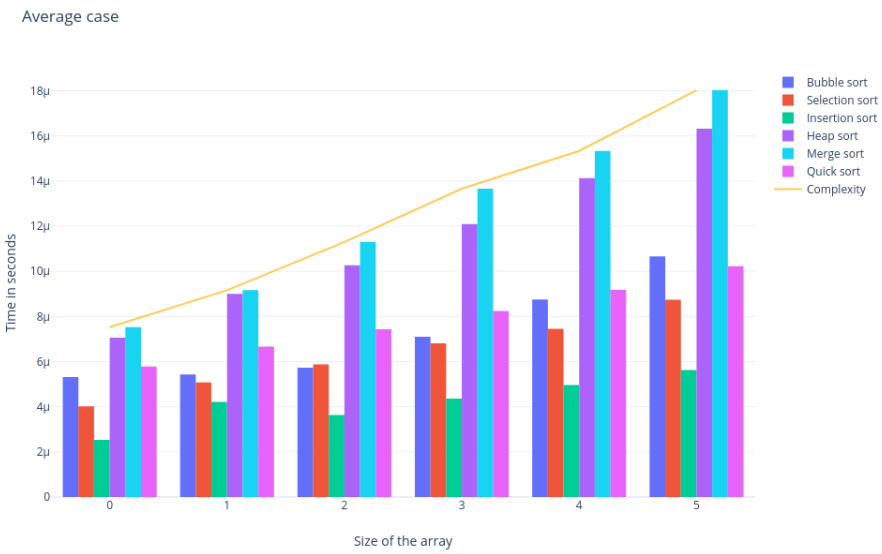
Table 8: Description of the average of the 100 iterations in the best, worst and average cases of the **quick sort algorithm**.

Number of elements	$Best - case$	$Worst - case$	$Average - case$
4	$2.5382x10^{-05}$	$3.8815x10^{-06}$	$5.7858x10^{-06}$
5	$6.3350x10^{-05}$	$4.9305x10^{-06}$	$6.6607x10^{-06}$
6	$1.9596x10^{-04}$	$5.4264x10^{-06}$	$7.4335x10^{-06}$
7	$3.9595x10^{-04}$	$6.5613x10^{-06}$	$8.2383x10^{-06}$
8	$1.5688x10^{-03}$	$7.2289x10^{-06}$	$9.1766x10^{-06}$
9	$1.5300x10^{-03}$	$8.5521x10^{-06}$	$1.0223x10^{-05}$

the complexity increases.

There is a slight change in the Figure 5 in the end, where the complexity lowers. This can be attributed to heap sort algorithms are not stables, same as the quick sort algorithm. This is one of the reasons we need to know the complexity in the best, worse, and average case, so we can decide if its worth to spend it all on a stable but slow algorithm, or if it might be worth the trouble of implementing an algorithm that is not as stable, but has a better performance on average for bigger amount of elements in the array.

Figure 10: Graphic description of the average of the average cases of all of the sorting algorithms used in this experiments



Exercises of chapter 3

February 12, 2020

Mayra Cristina Berrones Reyes

1 Turing machines

Turing machine (TM) was invented by Alan Turing in 1993, and consists of a tape with an infinite length, on which read and write operations can be done, using a certain alphabet, and with the instructions of a transition state table, it solves a certain task.

On this document we demonstrate some examples of the use of a TM on different tasks. First we define the special symbols that show up in every example:

- Σ is our alphabet.
- $\triangleright$ is the start of the tape.
- $-$ means no movement.
- $\sqcup$ is a blank space.
- $\leftarrow$ means move to the left.
- $\rightarrow$ means move to the right.

1.1 Example 1: Multiply a binary number by two

Define a Turing machine that produces a binary number (provided as input) multiplied by two.

In this TM our alphabet is $\Sigma = \{0, 1, \triangleright, \sqcup, \leftarrow, \rightarrow, -\}$ and our transition states are s, q and *stop*.

The transition table is shown in Table 1.

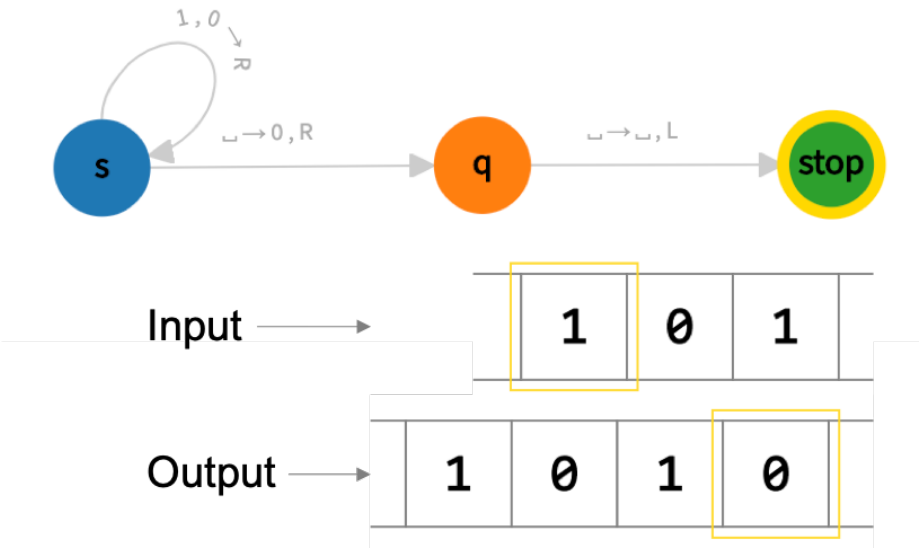
Table 1: Transition table for example 1

State	Transition
s, $\triangleright$	(s, $\triangleright$, $\rightarrow$)
s, 0	(s, 0, $\rightarrow$)
s, 1	(s, 1, $\rightarrow$)
s, $\sqcup$	(q, 0, $\rightarrow$)
q, $\sqcup$	($stop$, $\sqcup$, $-$)

```
input: '101'
blank: ' '
start state: s
table:
  s:
    [1,0]: R
    ' '  : {write: 0, R: q}
  q:
    ' ' : {write: ' ', L: stop}
  stop:


```

Listing 1: TM Visualization code

Figure 1: Diagram of all the transitions and the result of an example.



1.2 Example 2: Contains at least one letter

Given the alphabet $\Sigma = \{a, b, c, \triangleright, \sqcup, \leftarrow, \rightarrow, -\}$, the TM must check that at least one of each letter a, b, c appears on a string. This time we have seven states (*start*, and *case*2 to 7), and two acceptance states *accept* and *reject*.

The transition table is shown in Table 2.

Table 2: Transition table for example 2

State	Transition	State	Transition	State	Transition
$start, \triangleright$	$(start, \triangleright, \rightarrow)$	$case2, a$	$(case2, a, \rightarrow)$	$case3, a$	$(case5, a, \rightarrow)$
$start, a$	$(case2, 0, \rightarrow)$	$case2, b$	$(case5, b, \rightarrow)$	$case3, b$	$(case3, b, \rightarrow)$
$start, b$	$(case3, 1, \rightarrow)$	$case2, c$	$(case6, c, \rightarrow)$	$case3, c$	$(case7, c, \rightarrow)$
$start, c$	$(case4, 0, \rightarrow)$	$case2, \sqcup$	$(reject, \sqcup, -)$	$case3, \sqcup$	$(reject, \sqcup, -)$
$start, \sqcup$	$(reject, \sqcup, -)$				

State	Transition	State	Transition	State	Transition
$case4, a$	$(case6, a, \rightarrow)$	$case5, a$	$(case5, a, \rightarrow)$	$case6, a$	$(case6, a, \rightarrow)$
$case4, b$	$(case7, b, \rightarrow)$	$case5, b$	$(case5, b, \rightarrow)$	$case6, b$	$(accept, b, -)$
$case4, c$	$(case4, c, \rightarrow)$	$case5, c$	$(accept, \sqcup, -)$	$case6, c$	$(case6, c, \rightarrow)$
$case4, \sqcup$	$(reject, \sqcup, -)$	$case5, \sqcup$	$(reject, \sqcup, -)$	$case6, \sqcup$	$(reject, \sqcup, -)$

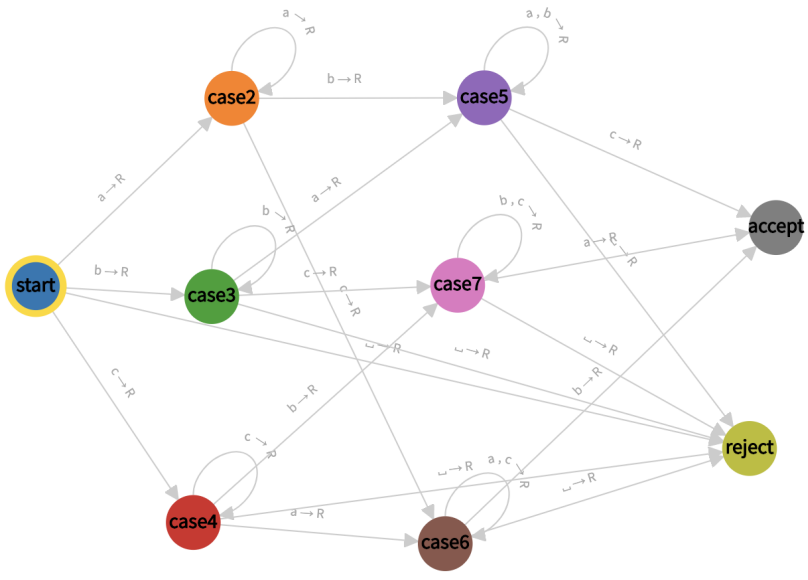
State	Transition				
$case7, a$	$(accept, a, -)$				
$case7, b$	$(case7, b, \rightarrow)$				
$case7, c$	$(case7, c, \rightarrow)$				
$case7, \sqcup$	$(reject, \sqcup, -)$				

```
input: 'ccaacb'
blank: ' '
start state: start
synonyms:
  accept: {R: accept}
  reject: {R: reject}

table:
  start:
    a: {R: case2}
    b: {R: case3}
    c: {R: case4}
    ' ': reject
  case2:
    a: R
    b: {R: case5}
    c: {R: case6}

  case3:
    b: R
    a: {R: case5}
    c: {R: case7}
    ' ': reject
```

```
  case4:
    c: R
    a: {R: case6}
    b: {R: case7}
    ' ': reject

  case5:
    [a,b]: R
    c: accept
    ' ': reject
  case6:
    [a,c]: R
    b: accept
    ' ': reject

  case7:
    [b,c]: R
    a: accept
    ' ': reject

  accept:
  reject:


```

Listing 2: TM Visualization code

Figure 2: Diagram of all the transitions and the result of an example.



The input for this TM is the string *ccaacb*, and the output is *accept*. If we

change that string to *caacc* the output is *reject*.

1.3 Example 3: K[redacted] if a string is a palindrome

Given the alphabet $\Sigma = \{a, b, c, \triangleright, \sqcup, \leftarrow, \rightarrow, -\}$, the TM must take the string and check if is a palindrome or not. This time we have eight states (*start*, and *case*2 to 8), and two acceptance states *accept* and *reject*.

The transition table is shown in Table 3.

Table 3: Transition table for example 3

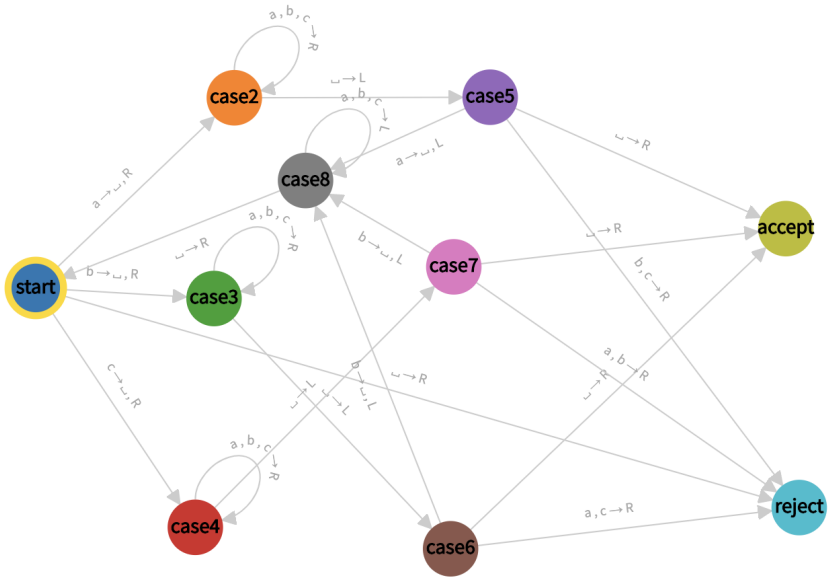
State	Transition	State	Transition	State	Transition
$start, \triangleright$	$(start, \triangleright, \rightarrow)$	$case2, a$	$(case2, a, \rightarrow)$	$case3, a$	$(case3, a, \rightarrow)$
$start, a$	$(case2, \sqcup, \rightarrow)$	$case2, b$	$(case2, b, \rightarrow)$	$case3, b$	$(case3, b, \rightarrow)$
$start, b$	$(case3, \sqcup, \rightarrow)$	$case2, c$	$(case2, c, \rightarrow)$	$case3, c$	$(case3, c, \rightarrow)$
$start, c$	$(case4, \sqcup, \rightarrow)$	$case2, \sqcup$	$(\sqcup, \sqcup, \leftarrow)$	$case3, \sqcup$	$(\sqcup, \sqcup, \leftarrow)$
$start, \sqcup$	$(reject, \sqcup, -)$				
State	**Transition**	**State**	**Transition**	**State**	**Transition**
$case4, a$	$(case4, a, \rightarrow)$	$case5, a$	$(case8, \sqcup, \leftarrow)$	$case6, a$	$(reject, a, \rightarrow)$
$case4, b$	$(case4, b, \rightarrow)$	$case5, b$	$(reject, b, \rightarrow)$	$case6, b$	$(case8, \sqcup, \leftarrow)$
$case4, c$	$(case4, c, \rightarrow)$	$case5, c$	$(reject, c, \rightarrow)$	$case6, c$	$(reject, c, \rightarrow)$
$case4, \sqcup$	$(\sqcup, \sqcup, \leftarrow)$	$case5, \sqcup$	$(accept, \sqcup, -)$	$case6, \sqcup$	$(accept, \sqcup, -)$
State	**Transition**	**State**	**Transition**		
$case7, a$	$(reject, a, \rightarrow)$	$case8, a$	$(case8, a, \leftarrow)$		
$case7, b$	$(reject, b, \rightarrow)$	$case8, b$	$(case8, b, \leftarrow)$		
$case7, c$	$(case8, \sqcup, \leftarrow)$	$case8, c$	$(case8, c, \leftarrow)$		
$case7, \sqcup$	$(accept, \sqcup, -)$	$case8, \sqcup$	$(start, \sqcup, \rightarrow)$		

```
input: 'aacbbcaa'
blank: ' '
start state: start
synonyms:
  accept: {R: accept}
  reject: {R: reject}

table:
  start:
    a: {write: ' ', R: case2}
    b: {write: ' ', R: case3}
    c: {write: ' ', R: case4}
    ' ': reject
  case2:
    [a,b,c]: R
    ' ' : {L: case5}
  case3:
    [a,b,c]: R
    ' ' : {L: case6}
```

```
  case4:
    [a,b,c]: R
    ' ' : {L: case7}
  case5:
    [b,c]: {R: reject}
    a: {write: ' ', L: case8}
    ' ': accept
  case6:
    [a,c]: {R: reject}
    b: {write: ' ', L: case8}
    ' ': accept

  case7:
    [a,b]: {R: reject}
    b: {write: ' ', L: case8}
    ' ': accept
  case8:
    [a,b,c]: L
    ' ' : {R: start}

  accept:
  reject:

```

Listing 3: TM Visualization code

Figure 3: Diagram of all the transitions and the result of an example.



The input for this TM is *aacbbcaa* and the output is *accept*. If we enter as input *abbca* the output is *reject*.

2 Conclusions

As a personal opinion, I think is easier to understand how a TM works if we drafted as a graph first. It also helps to visualize the transition tables and see where it can lead to an error. One of the tricks that I learned at the end of this exercise is to start with a small example, and then ask yourself more difficult questions every time you successfully finish one TM, to see if it still works properly.

3 Reference

To make the graphs we used `https://turingmachine.io/`.

Exercises of chapter 4

February 19, 2020

Mayra Cristina Berrones Reyes

1 Boolean satisfiability problem

This problem, abbreviated SAT, determines if the variables of a boolean formula can be replaced by *TRUE* or *FALSE* values. If the case is *TRUE* the formula is satisfiable. The complementary case in which the value is *FALSE* for all possible assignments, then the formula is unsatisfiable.

SAT was the first problem that was proven to be NP-complete, stated by Cook-Levin theorem. This theorem basically states that if there exists a deterministic polynomial algorithm for solving Boolean satisfiablility, then every problem NP-complete can be solved by a deterministic polynomial algorithm. This sort of question can be equivalent to the P versus NP problem, which has been studied for many years, and to this day it is still considered a popular unsolved problem in computer science [1].

The SAT tool helps us determine the satisfiability of a boolean formula. An algorithm by Davis Putnam Logemann Loveland (DPLL) makes a search through all of the possible interpretations of the problem to prove if the algorithm can find an interpretation that satisfies the problem, or if the algorithm explores all the search space and does not find a solution, in which case the problem is unsatisfiable.

This method uses the Backtracking technique, which is a search method that analysis all possible combinations of the True values of the variables. This, in conjunction with the pruning function, determines if it is posible to arrive to an answer giving certain node in the tree [2].

2 Hamiltonian paths and cycles

A very important problem of graph theory is the search for one cycle or path that pases through every vertex exactly one time. One of the definitions of

hamiltonian cycles is that is a circuit that starts in vertex a and goes through all of the other vertices exactly once and then returns to the starting vertex a. A path, similar to the cycle, starts in vertex a passes through all of the other vertices only once, and finally stops on the final vertex [3].

The problem of finding either a hamiltonian path or cycle is a NP-complete problem, making it highly unlikely to find a polynomial algorithm for solving it. Many studies have tried to give a linear solution to solving a hamiltonian path, giving it certain rules that do not hold up well when we change some of the parameters such as dimensionality. Another way is to use the algorithm of 3SAT in which the graph will be constructed of various parts to represent the variables of each clause that appear in the SAT problem. Then, by solving the SAT problem, we will determine if we can connect all the different paths [4].

Almost all studies agree in certain rules to prove which graphs can not hold a hamiltonian path or cycle, that depends on the number of nodes in the graph, if it has directions or not, etc. All of this, has not been sufficient to prove that not all the hamiltonian cycles can be solved in polynomial time.

3 Conclusions

On this investigation I found the NP vs P theme being mentioned quite repeatedly. I have always found that problem very intimidating because of all of the time and effort people have put in finding an answer. Now, seeing some of the examples used to find simpler solutions to this two problems that are considered to be NP, I can see that maybe the NP label is often used on problems with higher dimensionality, which happens in many other problems. If you increase the variables, parameters and dimensions, all problems will increase in time and computational complexity.

Another thing that I would like to mention, is that for this investigation I read several articles and examples on both subjects. The boolean problem took me more time because I personally have a hard time understanding all of the different symbols used in the formulas used to explain the DLLP algorithms. In the hamiltonian paths I was a bit lost on all of the theorems that I found, because each one is very specific to the type of graph they are using (different shapes, if they are directed, how many nodes, etc). This is why I did not enter in much detail, in the boolean problem because I do not feel comfortable explaining a subject that I do not fully understand, and in the hamiltonian I only explained what is relevant to the complexity problem, because I did not wanted to get side tracked by all the little details of every single graph.

4 Reference

References

[1] Carlos Ansotegui, Felipe M. An introduction to Satisfiability algorithms. Dpto de informatica e Ing. Industrial. 2003

[2] T. Covelli, E. Horquin, M Santillan. SAT una herramienta didactica para el problema de satisfactibilidad.

[3] Notes of Eulerian and hamiltonian paths. `https://www.csd.uoc.gr/~hy583/reviewed_notes/euler.pdf`

[4] J. Bang-Jensen, G. Gutin. On the complexity of hamiltonian path and cycle problems in certain classes of digraphs.

Exercises of chapter 5

February 26, 2020

Mayra Cristina Berrones Reyes.

1 Introduction

In previous excersies we analyzed the computational complexity of some of the more known problems. The efficiency and complexity to solve such problem gives us a clue as to how much time and space it will use to find an answer. There is a general knowledge of problems class $\mathcal{P}$, which are a set of problems that can be solved by a deterministic Turing machine in Polynomial time. Some examples of this problems can be seen in Figure 1. Then we have the $\mathcal{NP}$ problems, which can be described as a set of problems that can be solved by a non deterministic Turing machine in polynomial time [4].

Given this statement we can assume that $\mathcal{P}$ is a subset of $\mathcal{NP}$, because any problem that can be solved by a deterministic machine, can be solved by a non deterministic one in polynomial time. Next we have the $\mathcal{NP}$–complete problems, which are the hardest problems in a $\mathcal{NP}$ set. A $\mathcal{NP}$–complete problem can be verified in polynomial time, but there is no efficient known solution.

If we take this definition of $\mathcal{NP}$–complete problem, it seems impossible to prove that a problem $\mathcal{A}$ is $\mathcal{NP}$–complete. An alternative way of proving this, is to use reductions. In computational complexity theory a polynomial reduction can be computed by a deterministic Turing machine in polynomial time. Using reductions is very important because we can make many transformations between problems to prove their complexity, but to this date there is no reduction of a $\mathcal{NP}$ or $\mathcal{NP}$ complete problem in $\mathcal{P}$.

Now, given the concept of reductions, we can use them to prove that a problem is $\mathcal{NP}$–complete using the properties of transitivity. In [4] we can see some lemmas that helps us prove our problem, as seen below.

A decision problem $\mathcal{B} \in \mathcal{NP}$ is $\mathcal{NP}$–complete if:

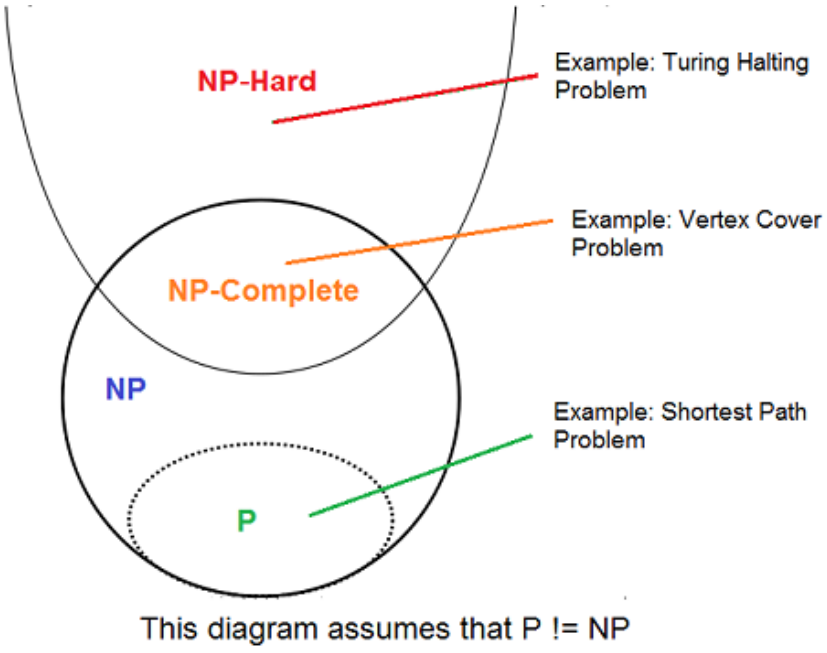


Figure 1: Example of some $\mathcal{P}$ and $\mathcal{NP}$ problems [4]

$$\mathcal{A} \leq_P \mathcal{B} \; \forall \; \mathcal{A} \in \mathcal{NP} \tag{1}$$

If you can solve $\mathcal{B}$ in polynomial time, then every other problem $\mathcal{A}$ in $\mathcal{NP}$ would be solvable in polynomial time. Given the rules of transitivity we have:

- $\mathcal{B} \in \mathcal{NP}$ and
- $\mathcal{A} \leq_P \mathcal{B}$ for some $\mathcal{NP}$ problem $\mathcal{A}$

2 Exercise 3. Hamilton

The HAMILTONCYCLE problem is defined as follows: Given a conected graph $G = (V, E)$, does the graph contain a tour that visits each node $v \in V$ exactly once? The HAMILTONPATH problem is defined as follows: Given connected graph $G = (V, E)$, does this graph contain a path that visits each node $v \in V$ exactly once? In HAMILTONPATH you can select a starting node and then visit all other nodes; you do not have to return to the starting node.

- Prove that HAMILTONCYCLE is $\mathcal{NP}$–complete by reduction from HAMILTONPATH.

This problem was a bit more difficult to understand than the one below (which I did first), because I found several different ways of seeing it. The first one was that given the example below, about the reduction of the Hamitlon cycle, if we rejoin the two vertex v^1, and v^2, we obtain the vertex V and the path gets converted into a Hamilton cycle, proving that if the Hamilton path is $\mathcal{NP}$–complete, then the Hamilton cycle must be $\mathcal{NP}$–complete as well. Another way of viewing it, is if you add another vertex to the graph, and use it to join the begin node and end node of the path. This way as well as the first one, we prove that the Hamilton cycle is $\mathcal{NP}$–complete with the verification of $\mathcal{NP}$–complete we had of the Hamilton path.

- Prove that HAMILTONPATH is $\mathcal{NP}$–complete by reduction from HAMILTONCYCLE.

So, this is the first one that I understood, thanks to an explanation by [3]. Basically what we do here is, we have a Hamiltion cycle $\mathcal{A}$, and we would like it to help us prove that a Hamilton path $\mathcal{B}$ is $\mathcal{NP}$–complete. So first we make sure that $\mathcal{A}$ is a cycle. After we have our verification, we take the node v in which the cycle started, and make it into two nodes, v^1, and v^2; v^1 only has incoming direction, and v^2 only has outgoing direction. Now, after doing this, we have a hamiltonian path, which can be proved to be $\mathcal{NP}$–complete with the verification that we had of the cycle $\mathcal{A}$, that this (cycle $\mathcal{A}$) was $\mathcal{NP}$–complete. With this we show that the answer to the reduced problem is the answer to the original problem, thanks to the rules of transitivity that we discussed earlier.

3 Exercise 5. Parcels and two trucks

A company has two trucks, and must deliver a number of parcels to a number of addresses. They want both drives to be home at the end of the day. This gives the following decision problem.

Instance: Set V locations, for each pair of locations v, $w \in V$, a distance $d(v, w) \in \mathbb{N}$, a starting location $s \in V$, and an integer K.

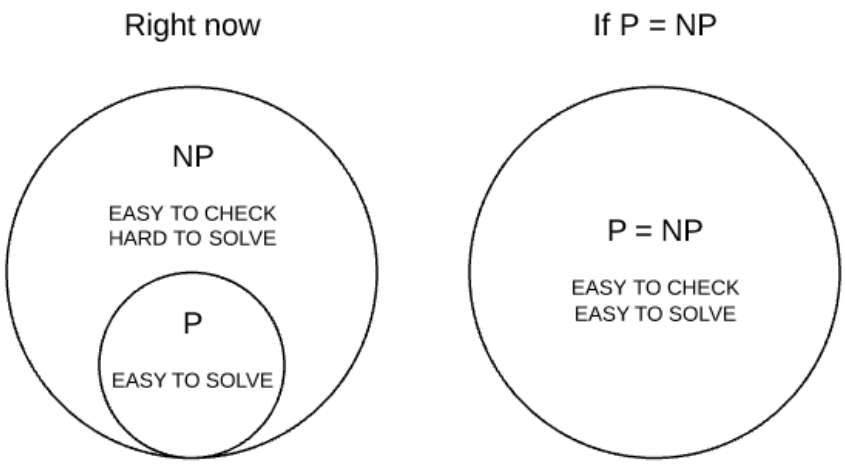
Question: Are there two cycles, that both start in s, such that every location in V is on at least one of the two cycles, and both cycles have length at most K?

Show that this problem is $\mathcal{NP}$–complete.

This is similar to the TSP problem. The only difference is that there are two trucks, and I have a little bit of a hard time trying to form the answer. I think this reduction can be managed as a sub problem type of thing. What I mean by this, is something along the lines of proving that with only one driver, the problem is $\mathcal{NP}$–complete. In that case, I can assume that two drivers is also $\mathcal{NP}$–complete, based on the concepts we stablished for reductions and their rules of transitivity. So in this my graph $\mathcal{A}$ is one cycle that goes trough all the vertex with only one driver. And my graph $\mathcal{B}$ is the one made by the two drivers. Given the implication that $\mathcal{A} \in \mathcal{NP}$–complete, then $\mathcal{B}$ is also $\mathcal{NP}$–complete, since a general problem must be as difficult to solve as one of their subproblems, which is what I understood from the reductions. So, if $\mathcal{A}$ is verified as a TSP problem, then by theory of computational complexity, we know that this problem belongs in the $\mathcal{NP}$–complete class, therefor we can say that $\mathcal{B}$ is also $\mathcal{NP}$–complete.

4 Conclusions

For this exercises, I had to do a lot of research, and the videos by [1, 2, 3] really helped me grasp the concept of reductions. At the end of this work, my concept on this subject is that reductions help analyze the complexity of different problems, and I can see why is important to study them, specially after reading about the $\mathcal{P}$ vs. $\mathcal{NP}$ debate that is going on. If someone can find a reduction that can simplify a $\mathcal{NP}$–complete problem, than maybe they can finally reach the connection to polynomial resolution of all of the different $\mathcal{NP}$–complete problems.



5 Reference

References

[1] Design and Analysis of algorithms. Lecture notes March 1996. `https://www.ics.uci.edu/~eppstein/161/960312.html`

[2] NP complete problems. UHMICSAlgorithms. `https://youtu.be/J5l-crl0LgA`

[3] NP complete problems. MIT OpenCourseWare.`https://youtu.be/G7mqtB6npfE`

[4] K. Seshagiri. Study of NP-complete problems: Polynomial time reductions and solutions. Indian Institute of Information Technology.

Practice 6

March 4, 2020

1 Introduction

Mayra Cristina Berrones Reyes.

As seen in previous exercises, in graph theory we know the Hamiltonian path is a path that visits every node in a graph exactly one time. The hamiltonian cycle is a hamiltonian path that connects the first and last visited nodes. To determine if a graph has a hamiltonian path or cycle, we find a $\mathcal{NP}$–complete problem. There are several algorithms that can solve a hamiltonian path. One of them is known as brute force, in which the algorithms searches for all the possible sequences of the graph. This gives us a $n!$ complexity, and it is obvious that the complexity will rise with the number of nodes in a complete graph.

Then there is another algorithm called backtraking. This technique is based on recursively trying to build a solution one pice at the time. It removes all of the solutions which do not match the constraints of the problem. In the case of the hamiltonian path, that it has to visit every node exactly one time. This algorithm is at most $O(N^K)$. On this work, we are going to try and prove this complexity given some experiments made with the tool Networkx.

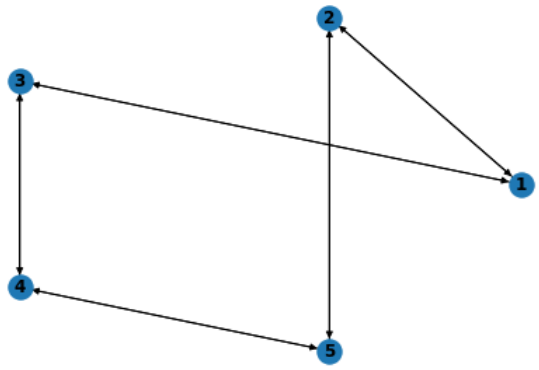
First, we play a little bit with the library, to see how the graphs work, and how we can plot them so we can have a more visually apealing example.

After reading the manual and some of the tutorials provided on their web page, we took some examples of already made algorithms, such as tournament.hamiltonian_path. We were not able to use this code, because it needed to be in a directed graph, and to do that we had to conect the edges manually. So, for the sake os a bigger experimentation, we used other things.

```
In [1]: import networkx as nx
        import matplotlib.pyplot as plt
        from networkx.algorithms import tournament
        %matplotlib inline

        #G = nx.gnm_random_graph(10, 12)
        G=nx.DiGraph()

        G.add_edge(1,2)
        G.add_edge(2,1)
```

```
G.add_edge(1,3)
G.add_edge(3,1)
G.add_edge(3,4)
G.add_edge(4,3)
G.add_edge(4,5)
G.add_edge(5,4)
G.add_edge(2,5)
G.add_edge(5,2)
#nx.draw_circular(G, with_labels=True, font_weight='bold')
#plt.subplot(122)

nx.draw_circular(G, nlist=[range(9, 10), range(5)], with_labels=True, font_weight='bold'
tournament.hamiltonian_path(G)
```

Out[1]: [2, 3, 1, 5, 4]



In order to be able to use the hamiltonian algorithm, we had to find a way to work with an undirected graph, so we searched and found this piece of code in which it takes a graph, and finds a path. Later we installed a library to mesure the time it takes to compute this process.

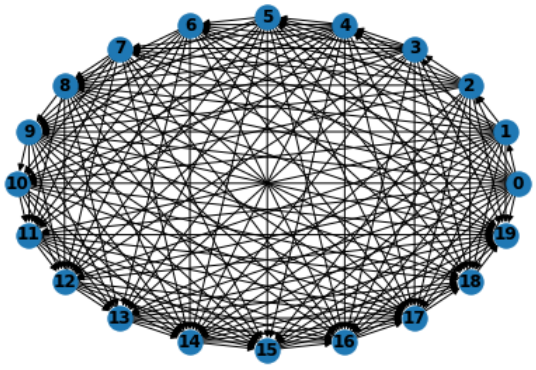
In [5]:
```
#code by https://gist.github.com/mikkelam/ab7966e7ab1c441f947b

import networkx as nx
import time
def hamilton(G):
    F = [(G,[list(G.nodes())[0]])]
    n = G.number_of_nodes()
    while F:
        graph,path = F.pop()
        confs = []
        for node in graph.neighbors(path[-1]):
            conf_p = path[:]
            conf_p.append(node)
```

```
                conf_g = nx.Graph(graph)
                conf_g.remove_node(path[-1])
                confs.append((conf_g,conf_p))
            for g,p in confs:
                if len(p)==n:
                    return "True"
                else:
                    F.append((g,p))
        return "False"
```

Next we have the different graphs. It is important to note that all the experiments and time reported are from experiments with a 100 nodes. The graphics shown bellow each one is a representation of 20 nodes only. This was made like this only to appreciate the number of edges that go away when we change the parameter k.

This parameter k helps us control the number of edges going out of each node. According to the complexity problem, we should have a similar behavior.

In [22]:
```
from itertools import combinations
from random import randint, random
G2 = nx.DiGraph()
node = 20
G2.add_nodes_from(range(node))
k = 0.9
for i in range(node):
    for j in range(i, node):
        if i != j and random() < k:
            G2.add_edge(i, j)
nx.draw_circular(G2, with_labels=True, font_weight='bold')
```

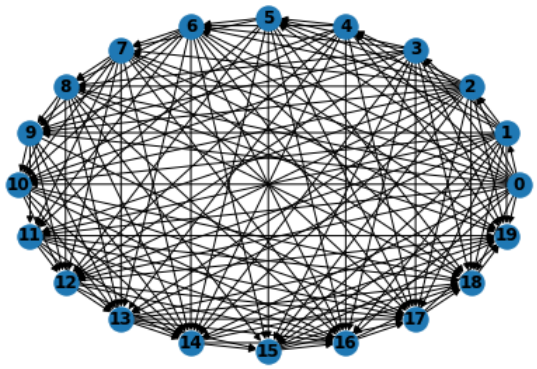


In [49]:
```
import time
start_time = time.time()
```

```
        #k = 0.9
        hamilton(G2)
        print( (time.time() - start_time))
```

```
34.48089385032654
```

```
In [23]: from itertools import combinations
         from random import randint, random
         G2 = nx.DiGraph()
         node = 20
         G2.add_nodes_from(range(node))
         k = 0.8
         for i in range(node):
             for j in range(i, node):
                 if i != j and random() < k:
                     G2.add_edge(i, j)
         nx.draw_circular(G2, with_labels=True, font_weight='bold')
```

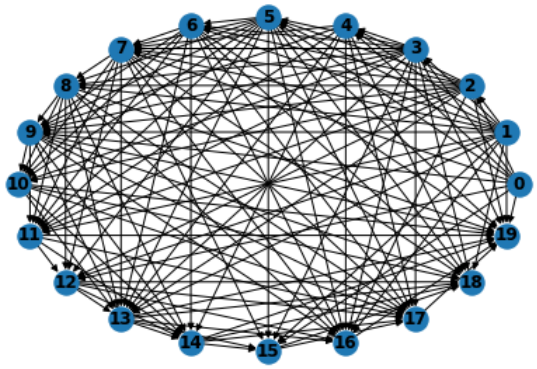



```
In [51]: start_time = time.time()
         #k = 0.8
         hamilton(G2)
         print( (time.time() - start_time))
```

```
28.018199920654297
```

```
In [24]: from itertools import combinations
         from random import randint, random
         G2 = nx.DiGraph()
         node = 20
         G2.add_nodes_from(range(node))
         k = 0.7
```

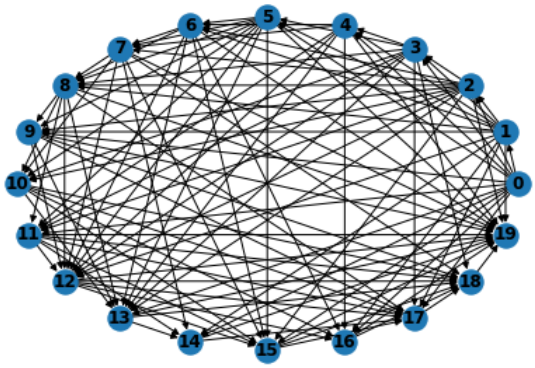
```
        for i in range(node):
            for j in range(i, node):
                if i != j and random() < k:
                    G2.add_edge(i, j)
        nx.draw_circular(G2, with_labels=True, font_weight='bold')
```



In [53]:
```
start_time = time.time()
#k = 0.7
hamilton(G2)
print( (time.time() - start_time))
```

```
19.465803146362305
```

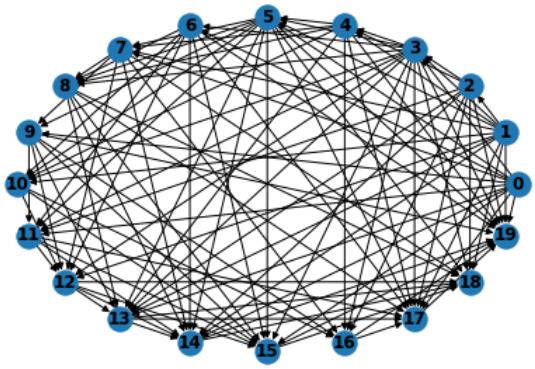
In [25]:
```
from itertools import combinations
from random import randint, random
G2 = nx.DiGraph()
node = 20
G2.add_nodes_from(range(node))
k = 0.6
for i in range(node):
    for j in range(i, node):
        if i != j and random() < k:
            G2.add_edge(i, j)
nx.draw_circular(G2, with_labels=True, font_weight='bold')
```



In [55]:
```
start_time = time.time()
#k = 0.6
hamilton(G2)
print( (time.time() - start_time))
```

```
14.404574871063232
```

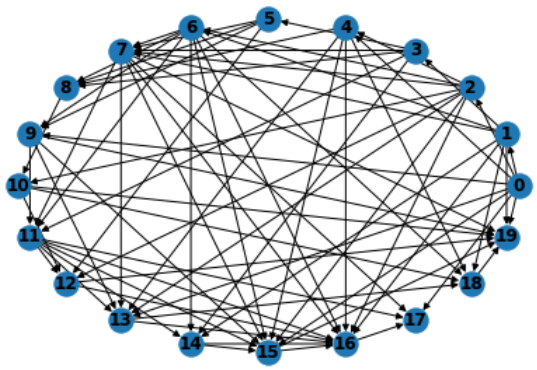
In [26]:
```
from itertools import combinations
from random import randint, random
G2 = nx.DiGraph()
node = 20
G2.add_nodes_from(range(node))
k = 0.5
for i in range(node):
    for j in range(i, node):
        if i != j and random() < k:
            G2.add_edge(i, j)
nx.draw_circular(G2, with_labels=True, font_weight='bold')
```



```
In [58]: start_time = time.time()
         #k = 0.5
         hamilton(G2)
         print( (time.time() - start_time))
```

```
10.570278882980347
```

```
In [27]: from itertools import combinations
         from random import randint, random
         G2 = nx.DiGraph()
         node = 20
         G2.add_nodes_from(range(node))
         k = 0.4
         for i in range(node):
             for j in range(i, node):
                 if i != j and random() < k:
                     G2.add_edge(i, j)
         nx.draw_circular(G2, with_labels=True, font_weight='bold')
```

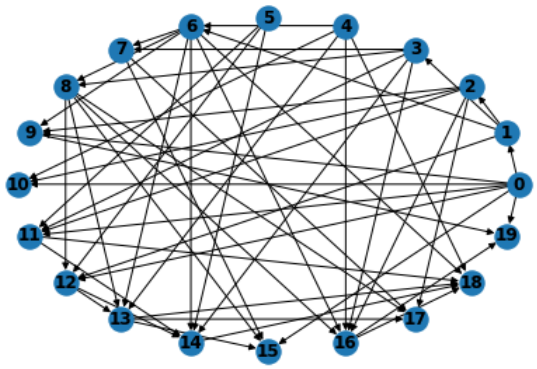



```
In [6]: start_time = time.time()
        #k = 0.4
        hamilton(G2)
        print(hamilton(G2), (time.time() - start_time))
```

```
True 12.908562898635864
```

```
In [29]: from itertools import combinations
         from random import randint, random
```

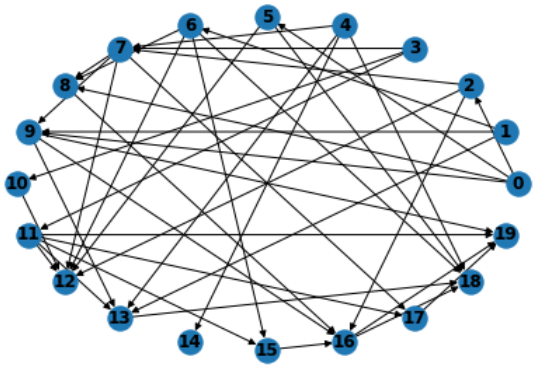
```
G2 = nx.DiGraph()
node = 20
G2.add_nodes_from(range(node))
k = 0.3
for i in range(node):
    for j in range(i, node):
        if i != j and random() < k:
            G2.add_edge(i, j)
nx.draw_circular(G2, with_labels=True, font_weight='bold')
```



In [63]:
```
start_time = time.time()
#k = 0.3
hamilton(G2)
print(hamilton(G2), (time.time() - start_time))
```

```
True 8.270284175872803
```

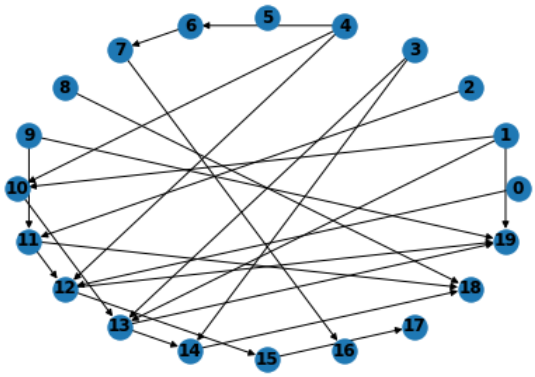
In [32]:
```
from itertools import combinations
from random import randint, random
G2 = nx.DiGraph()
node = 20
G2.add_nodes_from(range(node))
k = 0.2
for i in range(node):
    for j in range(i, node):
        if i != j and random() < k:
            G2.add_edge(i, j)
nx.draw_circular(G2, with_labels=True, font_weight='bold')
```



In [76]:
```
start_time = time.time()
#k = 0.2
hamilton(G2)
print(hamilton(G2))
print( (time.time() - start_time))
```

```
True
4.2513508796691895
```

In [39]:
```
from itertools import combinations
from random import randint, random
G2 = nx.DiGraph()
node = 20
G2.add_nodes_from(range(node))
k = 0.1
for i in range(node):
    for j in range(i, node):
        if i != j and random() < k:
            G2.add_edge(i, j)
nx.draw_circular(G2, with_labels=True, font_weight='bold')
```



```
In [40]: start_time = time.time()
         #k = 0.1
         hamilton(G2)
         print(hamilton(G2))
         print( (time.time() - start_time))

False
0.0604097843170166
```

2 Results and conclusions

As we can appreciate in the graphic above, the expected results and the results we had in this experimentation are not so different. We can conclude that, given the algorithm we choose to search for paths in our graph, it is reasonable that the more edges the graph has, the computational time will grow, this can be blamed on the way it searches for all possible paths that match our restrictions. If we shorten the list of edges, the time decreases. But as seen in the experimentation, too low of a value K, and there will not be enough edges to make a connection between all the nodes, making it impossible to form a hamiltonian path.

6 pts; correcciones indicadas durante la sesión

Practice7

March 10, 2020

Mayra Cristina Berrones Reyes

1 Introduction

1.1 Arrays vs lists

Arrays and lists often are confused on any programming language. In our case, we searched for the concept of arrays and lists inside a python language. In python, this two structures have similar functions, as in both of them store data, but the main difference can be seen on their structures and how the data can be accessed.

A list in python is a data structure that holds a collection of items. They are declared with enclosed brackets like [*item1*], and they appear in a specific order, that enables us to use an index to access the information. Lists are mutable, which mean they can change, add, or remove items after we create de list. The elements inside a list do not require to be unique, and a list can hold several different types of data inside (integers, strings, objects, etc.).

An array is also a data structure that stores a collection of items, they can be mutable, ordered, enclosed in brackets and have non unique items. The first thing that separates lists form arrays is that arrays can not hold different types of data.

Another difference is that arrays need to be declared in python, and lists do not. Python needs to import the library *Numpy* to be able to use an array. In Table 1 we can see the main differences between arrays and lists.

Taking into consideration the pros and cons of using arrays or lists, we demonstrate two problems to see if this features are correct.

Table 1: Main differences of lists and arrays in python.

Lists	**Arrays**
• Lists can be can be flexible and hold arbitrary data.	• Arrays need to be imported from other libraries.
• They are a part of the python syntaxes so they do not need to be declared first.	• Arrays can not be resized, it needs to be copied to another larger array.
• Can be resized quickly in a time efficient manner. Python initializes some elements in the list at the initialization.	• Arrays can only store items with values of uniform data types.
• Lists can hold different types of data.	• They are specially optimized for arithmetic computations.
• Mathematical functions can not be directly applied to all the list, it needs to be applied to each item individually.	• Since arrays stay the size that they were initialized with, they are compact
• They consume more memory as they are allocated a few extra elements to allow for quicker appending.	• In the array you can find an item easily by their index.
• To search for an item, you need to start from the first element, and go through all of the other items until you reach the one you want.	• Operations like delete and insertion take a lot of computational time.
• Delete and insertion are easy.	

1.2 Palindrome

For this first example we have the problem of a palindrome. Remembering the differences we described in Table 1 we have that arrays are more well suited for this type of problem, since we can search and compare the contents of the first and last item on the array using the the index numbers, as we can see in Figure ??. Then in Figure ?? we can see that we will have to pass the list several times to check between the first and the last items on the list, since each element is pointing to the direction were the next item is stored (it is not a fixed parameter).

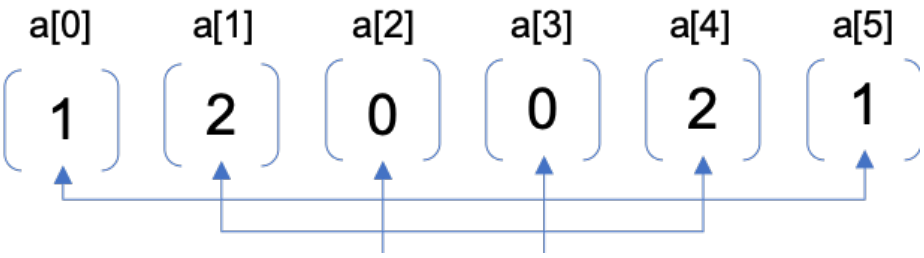


Figure 1: Palindrome example for an array

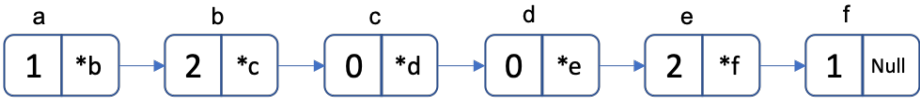


Figure 2: Palindrome example of a list

1.3 Delete all the duplicated values

Now we have another example in which we need to remove the duplicated values of an array. In this case, the array will not do so well, since we have to move all of the elements so that there is not a null value inside the array, and we can safely remove the last items that are null, as we can see on Figure ??. On the other hand, we have the list, that on this case perform better than the arrays, because we can easily bypass the elements to remove, and have the remaining items point only to the unique values. This example can be seen in Figure ??.

Figure 3: Example for deleting duplicated values of an array

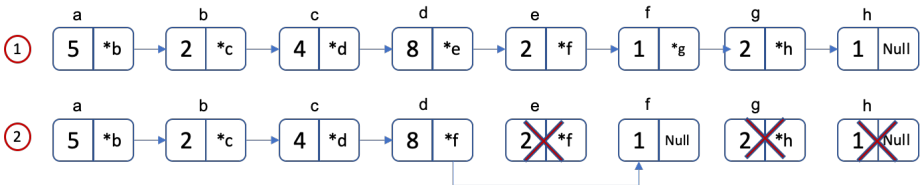


Figure 4: Example to delete duplicated values on a list

2 Conclusions

It took me a while to grasp the concept of this practice, because I am very used to handle python, and every time I saw something referenced as a list or an array, I thought they meant the same. Now, after reading more about the rules, I figured that I have been using lists too liberally, and that it may not seem very significant the computational time, because despite their drawbacks, both structures perform very fast, but I realize now that some of the problems I had in the past, programing simple things that resulted in errors or leaked memory could have been because I was not using the proper structure to store my data.

6 pts, correcciones indicadas por videoconferencia

Practice 8

Mayra Cristina Berrones Reyes

March 18, 2020

1 Introduction

The subject of this practice is Red-Black trees. But before we enter in detail on this type of trees, let us see some basic concepts first.

1.1 Binary trees

A binary tree is a data structure whose elements have at most two children, and they are typically named right and left child. The difference between this data structure and arrays, lists, queues or stacks, is that trees are hierarchical structures. They can be used if you want to store informations that is normally in a hierarchy form, such as a file system for a computer. The top node is called root of the tree. The nodes directly under it are called its children. When this nodes have more nodes below they are called parent nodes.

The main applications for binary trees can be [1]:

- Manipulate data that is ordered in a hierarchy form.
- The way is built, makes it easier to search for an element.
- It gives you a better manipulation of sorted lists of data.
- It can be used in router algorithms.

1.2 Balanced binary tree

A binary tree is balanced if the height of the tree is $\mathcal{O}(\log n)$ where n is the number of nodes. They are different forms in which a binary tree can obtain this balance. For Example, AVL tree maintains $\mathcal{O}(\log n)$ height by making sure that the difference between heights of left and right subtrees is no more than 1. Red-Black trees maintain the same property of height by making sure that the number of black nodes on every root to leaf paths are same and there are no adjacent red nodes.

2 Red-Black trees

A Red-Black tree is a self balancing binary search, in which the node contains extra information about the color of the node, that is either red or black. A Red-Black tree must satisfy the following properties [3]:

1. **Red-Black property:** Every node is colored either red or black.
2. **Root property:** The root is black.
3. **Leaf property:** Every leaf (NIL) is black.
4. **Red property:** If a red node has children then, the children are always black.
5. **Depth property:** For each node, any simple path from this node to any of its descendant leafs has the same black depth (the number of black nodes).

For the first and second properties, they are really self explicatory. The nodes can only be colored red or black, and as a rule, the root node must be black. In the introduction we mentioned which one is the root node.

Now we move to the third property, which says that every leaf is black. By leaf we mean a node that has no children. In this type of tree, when a node has no children, we put two nodes with null leaves, and this two nodes must be colored black.

The fourth property states that if a red node has children, then the children must always be black. With this property we make sure that red nodes are not adjacent to one another, which is important in order to comply with the fifth property that says for each node, any simple path from this node to any of its descendant leafs has the same black depth, which means it has the same number of black nodes on its way up. It is important to remember that this is a self balancing tree, so this limitation on property 5 helps to ensure that any path from root to leaf is balanced from all the other leaves. If this properties are not meet, then we need to apply certain steps to ensure that they do.

This steps or operations can be divided as follows [2]:

1. Rotating the sub-tree in a Red-Black tree
2. Left-right and right-left rotate.
3. Inserting an element into a Red-Black tree.

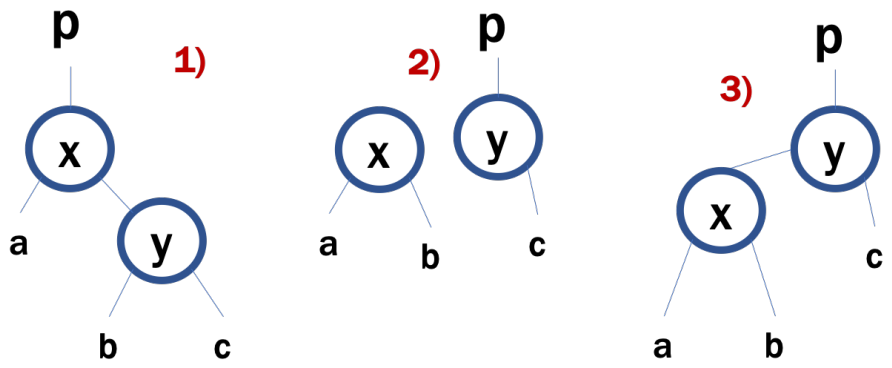


Figure 1: Example of rotating left sub-tree in a Red-Black tree

The first operation is used to maintain the properties of the tree when we use the algorithms of insertion or deletion and the properties shown above are not meet.

In Figure 1 we se an example of the algorithm used to rotate to left. To do this we have the original tree in example 1). Then we see that y has a left node. We give this node as a right child to x and move y upwards to be connected with the root node. Finally in example 3) we connect the node x to be the left child of node y. In case of right rotation is the same, but from example 3) to 1), and instead of conecting the left child of the moving node, is the right one, as we can see in Figure 2

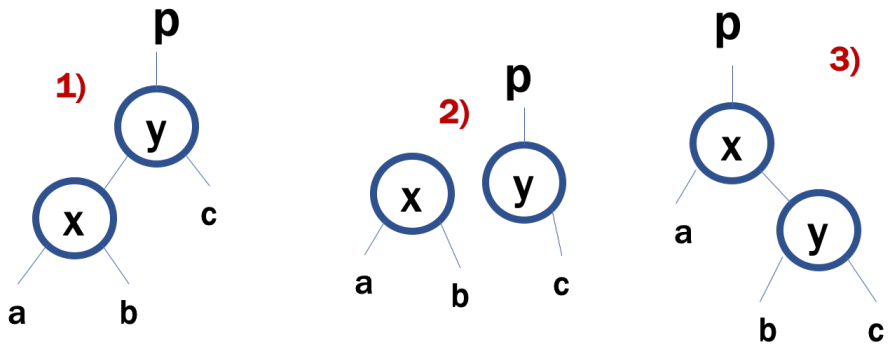


Figure 2: Example of rotating right sub-tree in a Red-Black tree

The left-right and right-left rotate are more or less the same as the first one, but in this case the arrangement of nodes are moved first to the left and then

to the right, and vice versa as we can see in Figure 3. The example of right-left is similar to Figure 3, but we begin with right rotation.

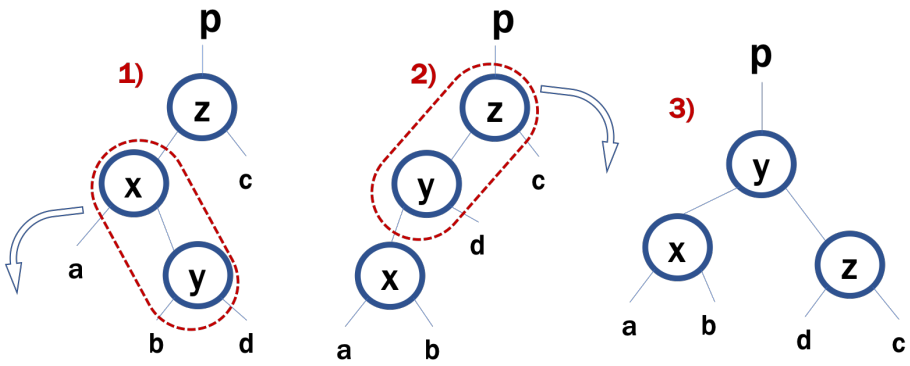


Figure 3: Example of rotate left-right

Then we have the third operation, which is inserting a new node. In this case another rule we have is that the new node that we insert must be red. After we insert this node, if the tree does not match the properties, we can either recolor nodes, or make rotations. To delete a node is more or less the same. First we find the node we want to eliminate, we take it out, and then we make operations to balance the tree.

3 Example

With the help of the code of the tutorial in Programiz [1] we were able to make an example of a Red-Black tree. We go inserting one node at the time in the order of *3, 1, 5, 7, 6, 8, 9, 10*. In Figure 4 we see the results on the console. When there is an indentation in the lines, it means the following nodes are the children of the node above.

To be able to understand it better and the steps it took to arrive to the final tree, we used a tool to build Red-Black trees [2] that we can find online. In each caption we explained the steps and operations made to ensure that the properties of the Red-Black tree were fulfilled.

[1] https://www.programiz.com/dsa/red-black-tree
[2] https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

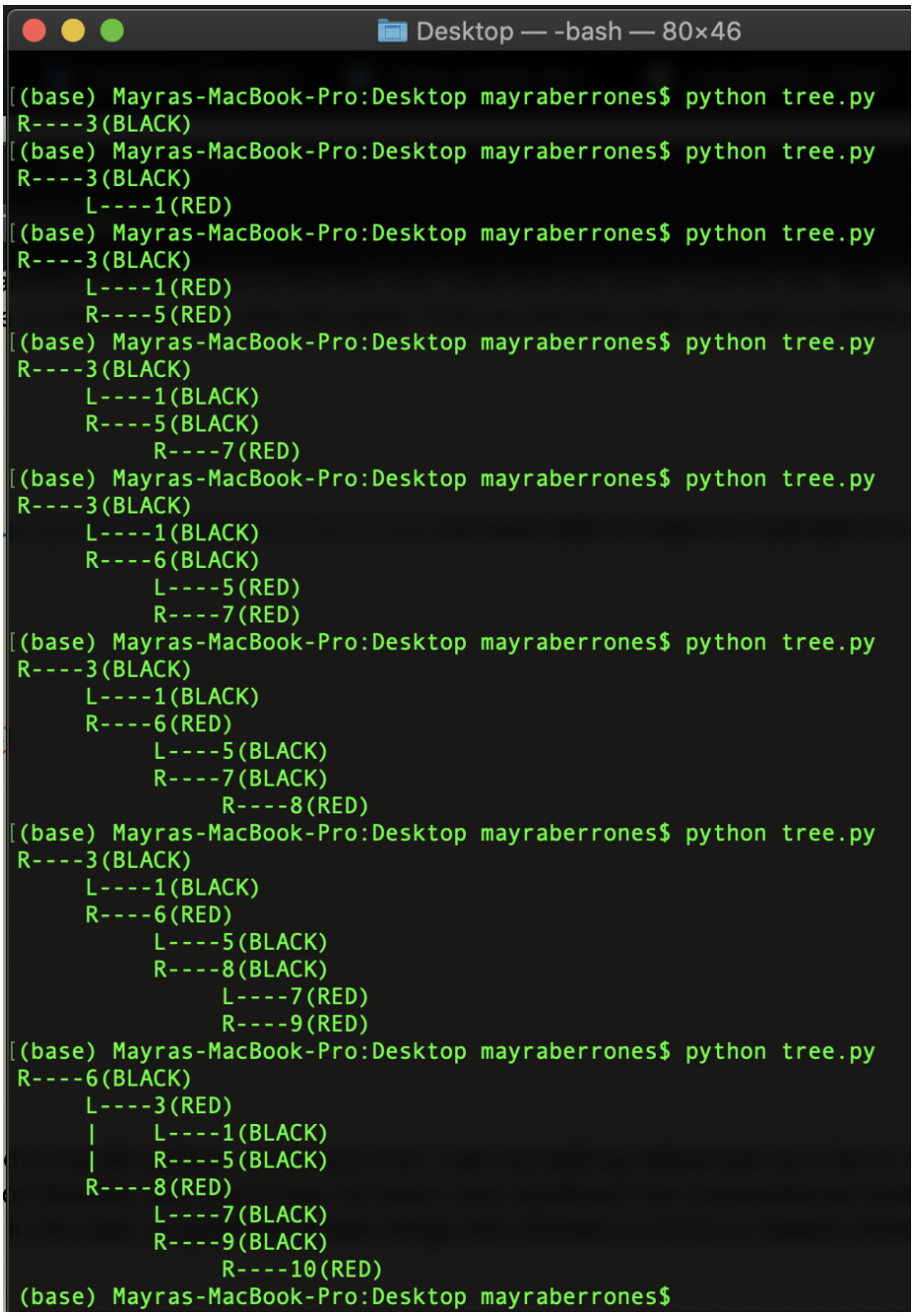


Figure 4: Output of the code to make Red-Black trees

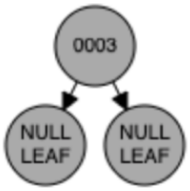


Figure 5: Here we see the first node as the parent node, and its two null nodes. This tree satisfies all the properties.

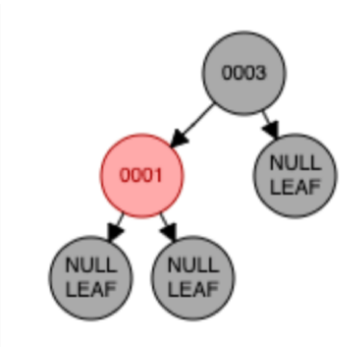


Figure 6: Now we add a red node with value 1. As is smaller than 3, it goes to the left. This does not need further manipulation.

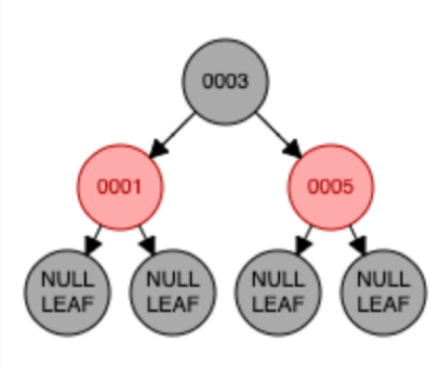


Figure 7: Now we add a red node with value 5. As is bigger than 3, it goes to the right. This does not need further manipulation.

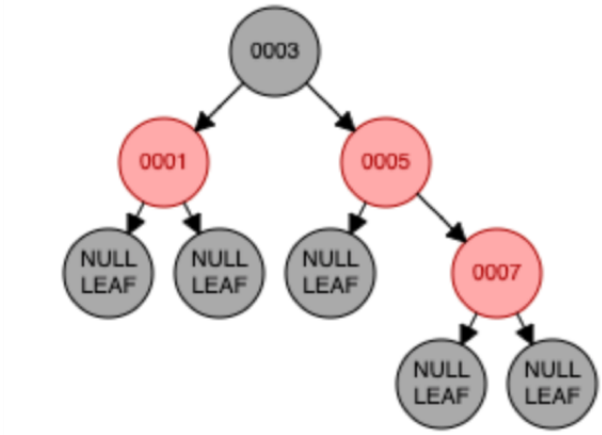


Figure 8: We add a red node with the value of 7. As is bigger than 3, it goes to the right, and it is also bigger than 5 so it goes to the right. Here we have a violation of the properties, since two red nodes can not be adjacent to one another.

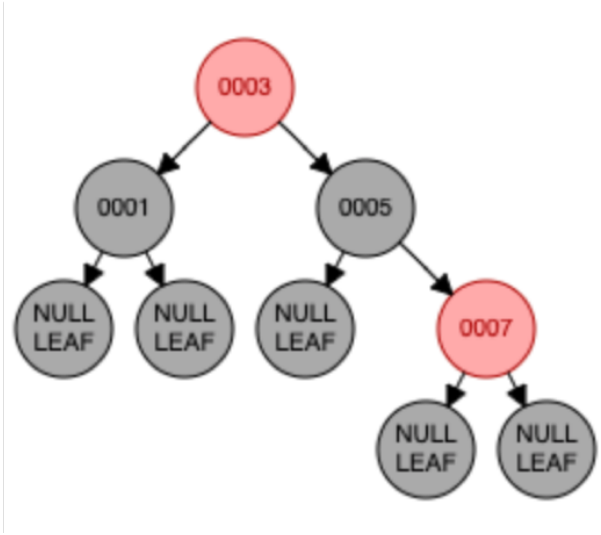


Figure 9: To fix the problem in tree 8, we change the color of the parent node 5, and the uncle node 1. With this result we violate another property, in which the root node has to be black.

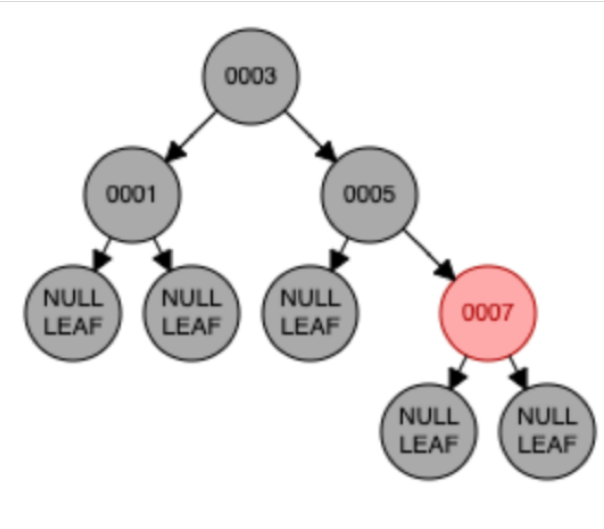


Figure 10: We change the color of the root node, and now we have a tree that meets all the properties.

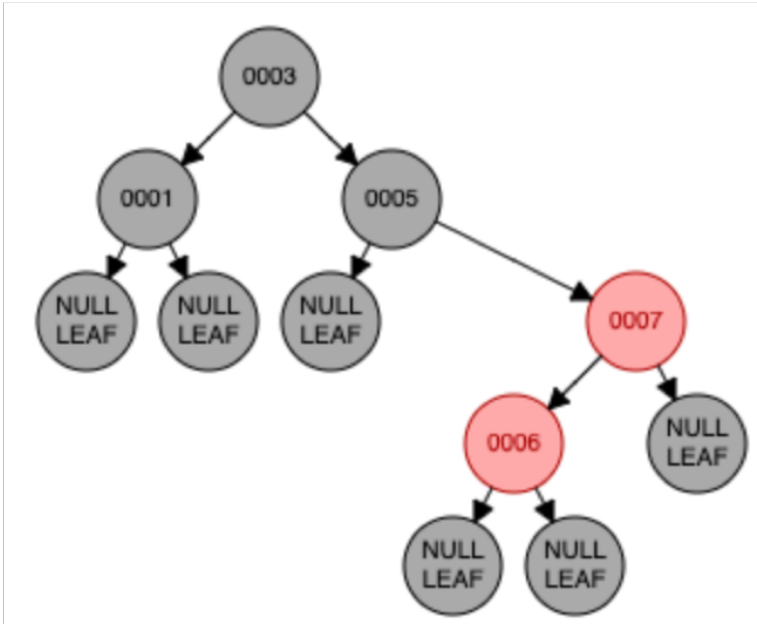


Figure 11: Now we add another red node with the value of 6. It is bigger than 3 and 5, so it moves to the right, and in node 7, it moves to the left. Here we violate the property that two red nodes can not be adjacent to one another. We can not simply change the color of the node, because we would be infringing on the depth property. So we need to make a right-left operation.

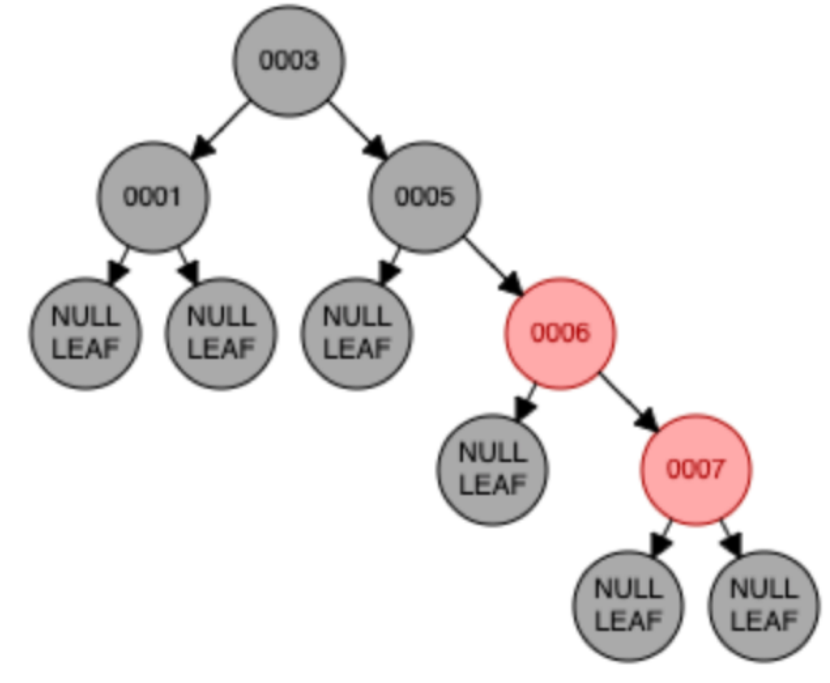


Figure 12: Now that we made the right rotation with node 7, we need to do the left rotation with the node 5

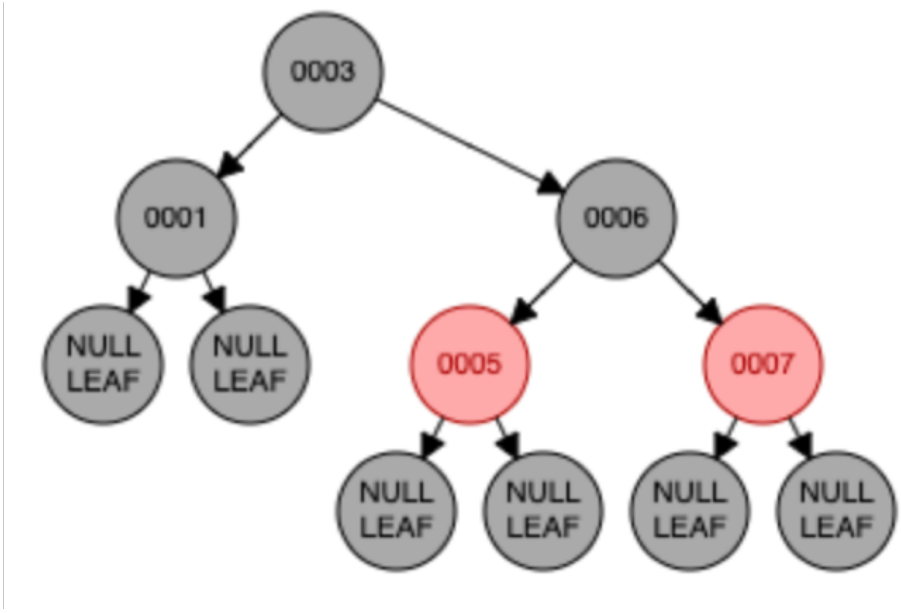


Figure 13: To finish this operation, we need to change the color of the node 6 to black and the node 5 to red. Now we have a tree that meets all the properties.

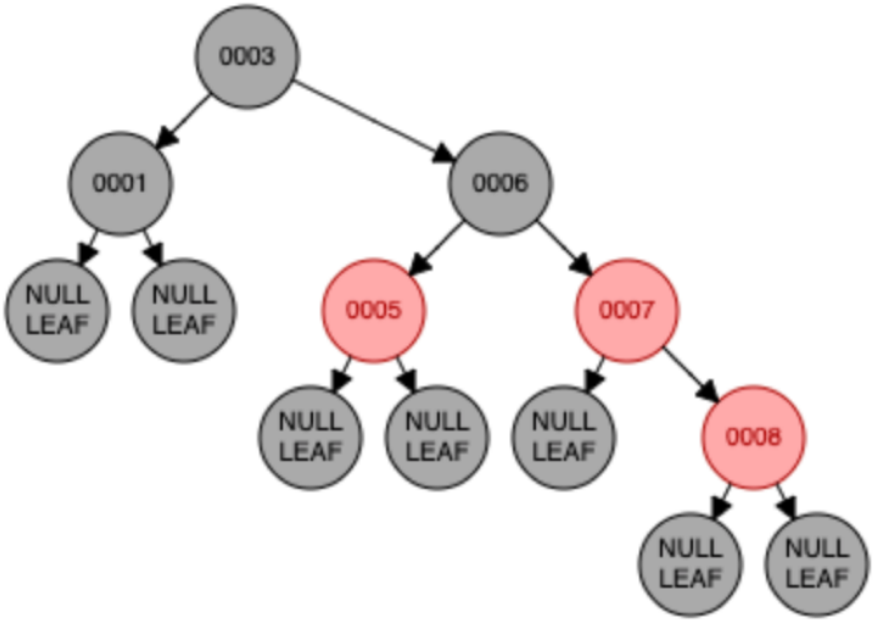


Figure 14: We insert a red node with the value 8. Since it is bigger than 3, 6 and 7, it moves to the right. This tree violates the property of two red nodes, so we need to change the color of the parent and the uncle of node 8.

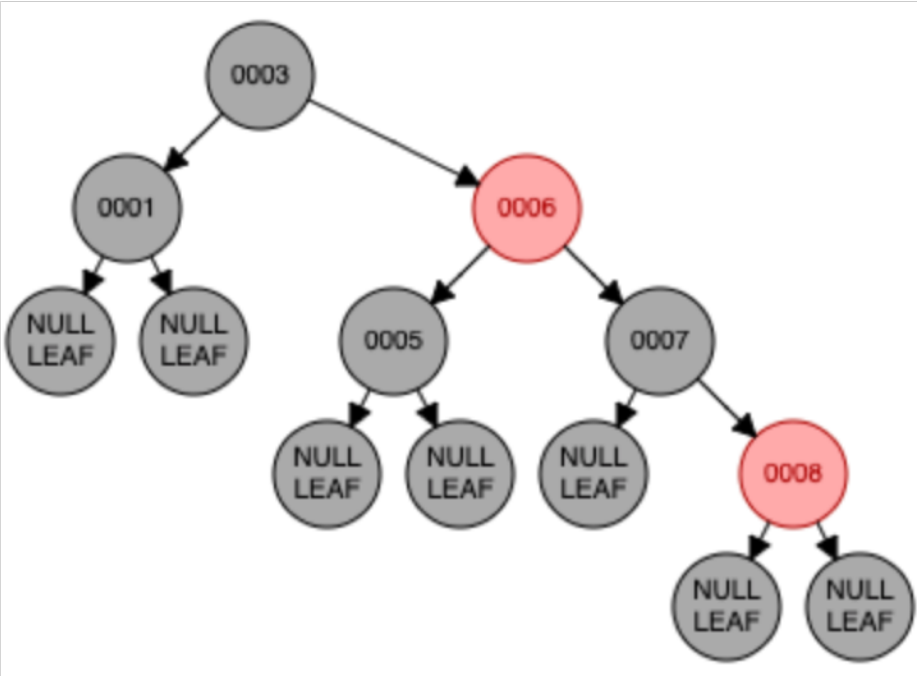


Figure 15: Now that we fixed the colors, all properties are meet.

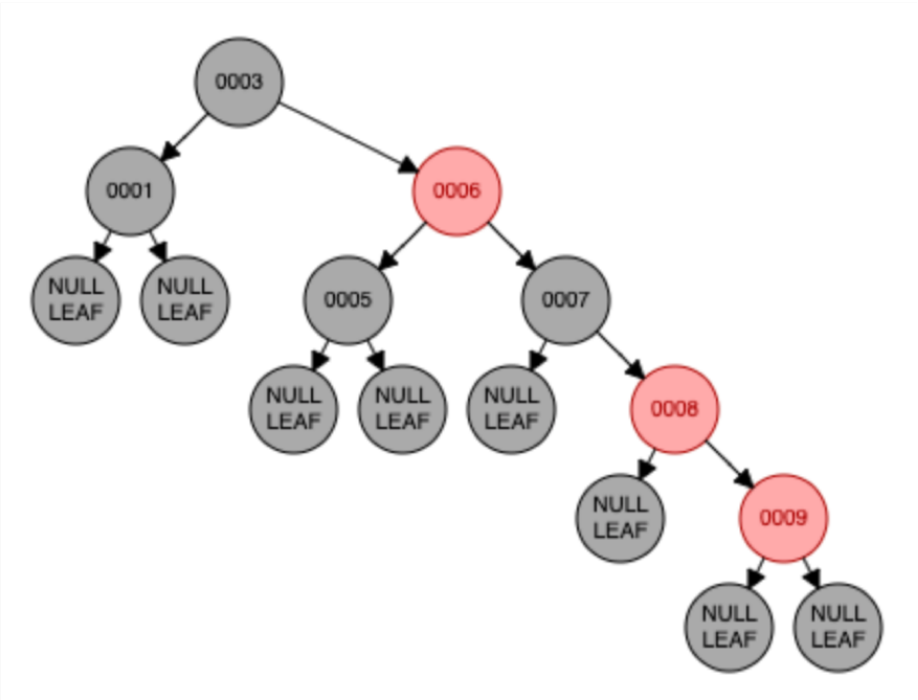


Figure 16: We add a red node with the value 9. It is bigger than 3, 6, 7, and 8, so it moves to the right. This tree violates the rule of two red nodes. Changing colors will still be infringing properties, so first, we make a left rotation.

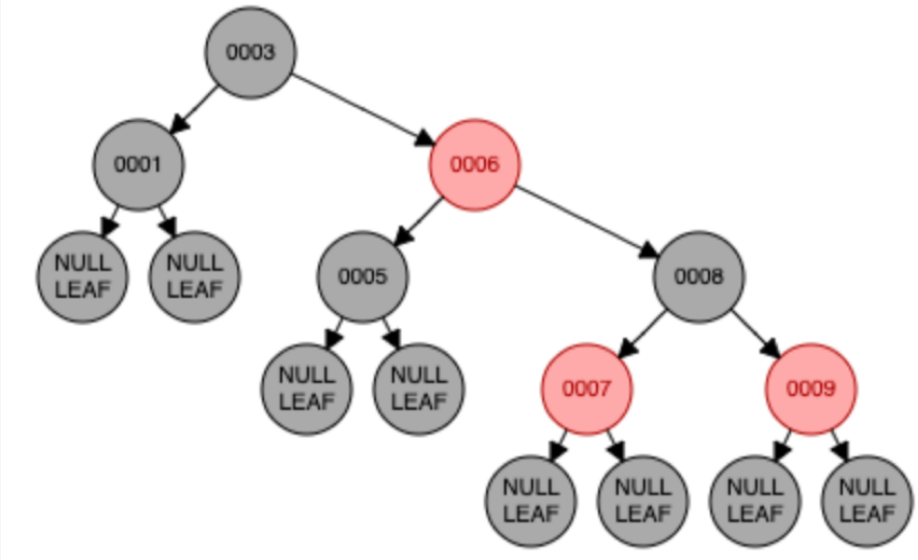


Figure 17: After the rotation, we can now change the color of node 8 to black and node 7 to red.

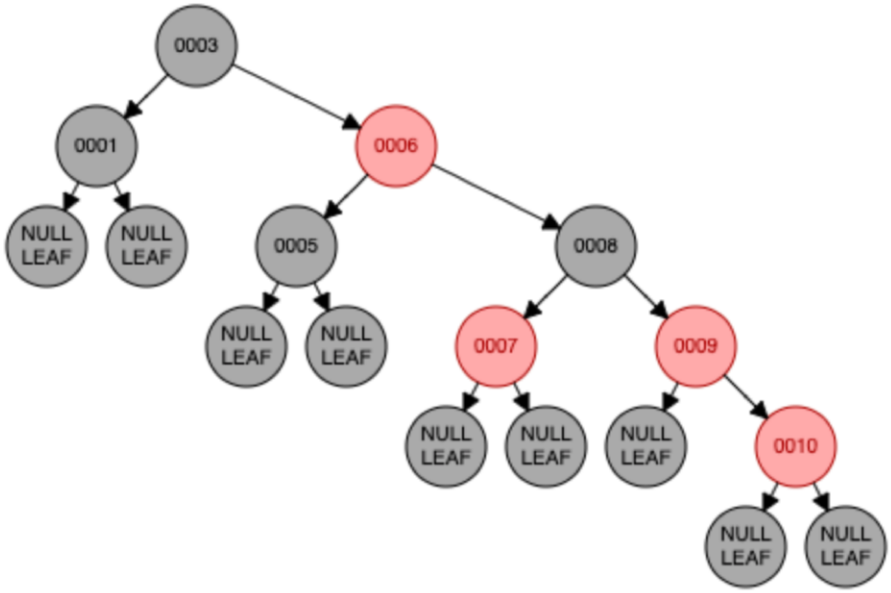


Figure 18: We now add a red node with the value 10. This tree violates the rule of two red nodes, so we change the color of the parent node 9 and the uncle node 7.

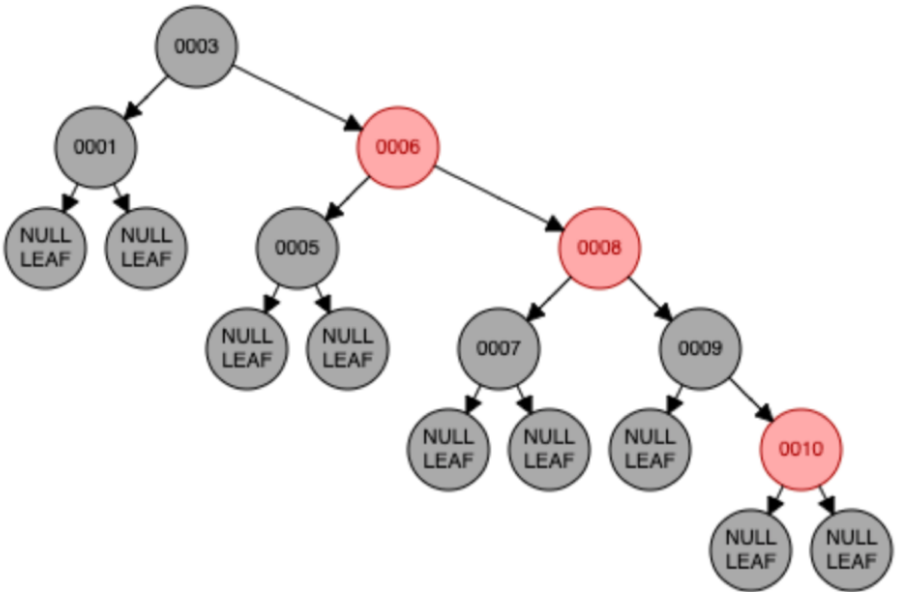


Figure 19: Now that we switched the colors, we are still violating the rule of two red nodes, and changing colors again will infringe in other properties, so we make a left rotation of node 6.

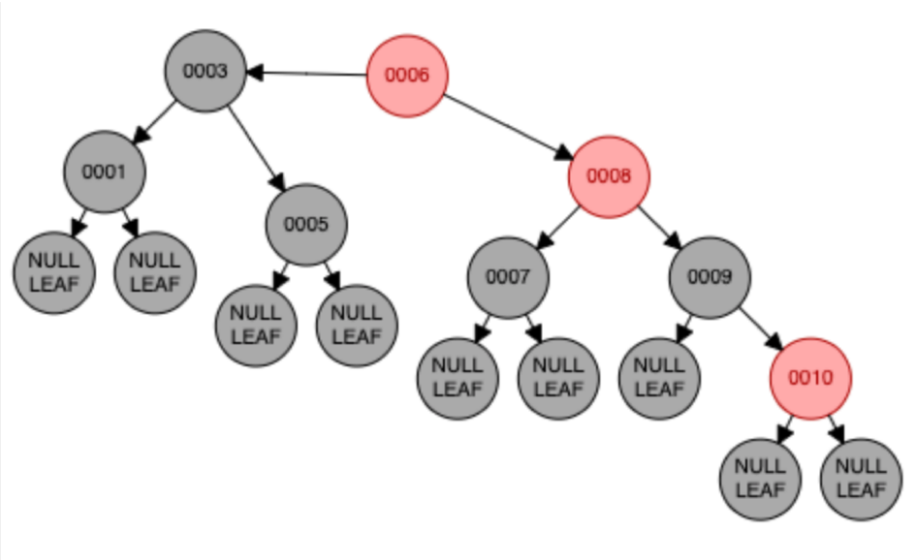


Figure 20: After the rotation, the node 6 is now violating the rule of the root node needing to be black.

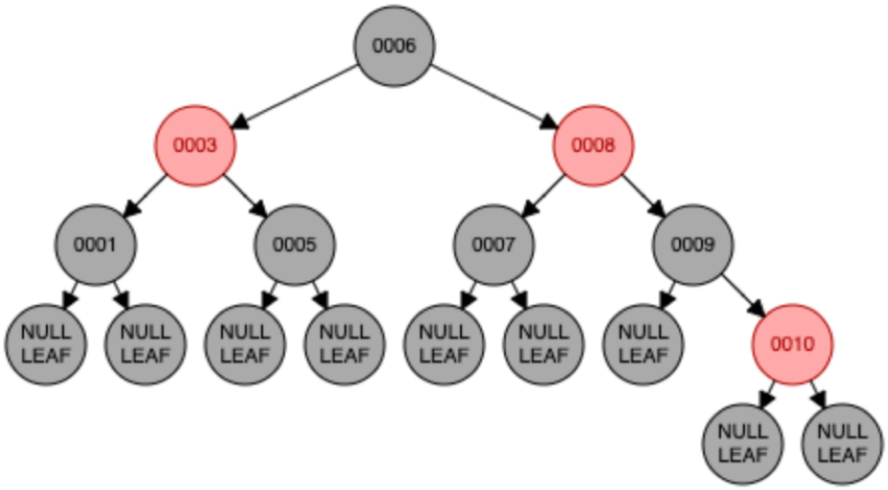


Figure 21: We change the color of the root node 6 to black, and the node 3 to red and we now have a tree that meets all the properties. Now, if we compare this last tree with the last result of the console in Figure 4 we see that they match, concluding this experiment.

4 Conclusions

I had a bit of trouble understanding the rules of this tree, mainly because I did not understand at first the use of the red and black nodes, but after learning the properties and the different operations to balance the tree, I understood the function of the red and black, and it was actually fun to play with the different options to move the nodes. It helped a lot to be able to visualize it on the web page mentioned in the example.

This is my first time working with trees, so it was really interesting to find out how they worked, and they seemed a better choice to structure your data than a queue or a stack, if the goal is to find an element more efficiently.

5 Reference

References

[1] Binary tree. https://www.geeksforgeeks.org/binary-tree-set-3-types-of-binary-tree/

[2] Data structures and algorithms. https://www.cs.auckland.ac.nz/software/AlgAnim/red_black_op.html

[3] Red-Black tree. https://www.programiz.com/dsa/red-black-tree

5 pts

Practice 9

Mayra Cristina Berrones Reyes

March 24, 2020

1 Introduction

The subject of this practice is binomial heaps. Binomial heap is an extension of binary heap, and the main application of binary heap is as implement priority queue. So before we enter the subject of binomial heap, we are going to explain this two concepts first.

1.1 Priority queue

A priority queue is a type of data structure like a queue or a stack, with the added element of having a priority associated with each element. Every priority queue has to have the following properties:

- Every item is associated with a priority.
- An element with high priority is going to be dequeued before an element with low priority.
- If two elements have the same priority, they are served in the order they came in.

This type of queues can be solved with a linked list, but it is only efficient when the elements inside the queue do not need to be moved. Generally, priority queues are solved with the implementation of **binary heaps**.

1.2 Binary heap

A binary heap is like a binary tree but with the following properties:

- It is a complete tree.This property makes it so that this type of tree can be stored in an array.

- This binary heap can be either min or max heap. In a min heap the key root must be the minimum value among all the keys inside the tree. In the max heap is the opposite.

Binary heaps are can be efficient in the implementation of priority queues because they support several operations in such as insert, delete, extract max, decrease key. Binomial heap and fibonacci heap are some variations of binary tree.

2 Binomial heap

Binomial heap is an extension of binary heap, and in this case it provides a faster union operation, but it also provides all the other operations mentioned in binary heap. A binomial tree of order 0 has only 1 node. In Figure 1 we see the order of the different types of trees. Following this example, a binomial tree of order k can be constructed by taking 2 binomial trees of order $k-1$ and making one of them the left-most child of the other. This order k binomial tree has the following properties:

- It has exactly 2^k nodes.
- It has a depth of k
- The root has a degree of k and children of the root are themselves Binomial Trees with order k-1, k-2,.. 0 form left to right.

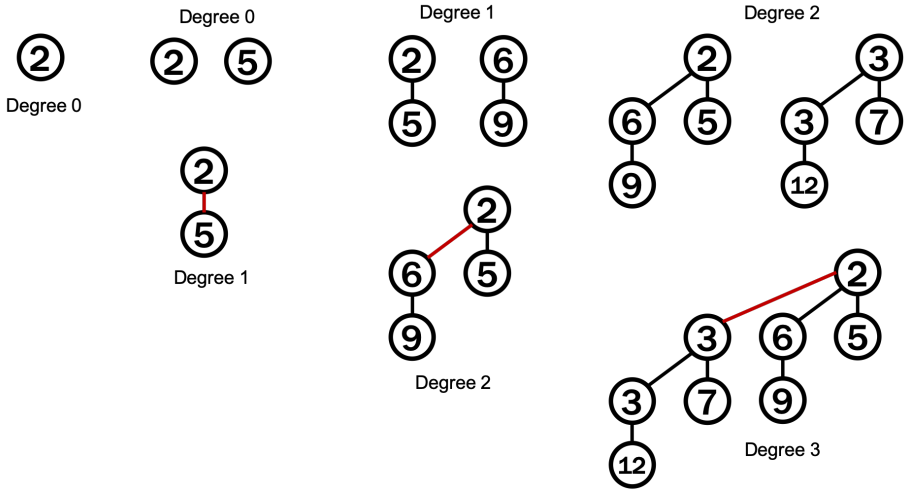


Figure 1: Example of degree level of a binomial heap after we merge trees of the same degree.

Now, as we mentioned before, binomial heap has several operations. First we start with the operation of inserting a new node. The insert operation creates a new heap with the inserted element, which can then be combined using the union operation. In Figure 2 we insert a node with the key 4. Since we already have a node with order 1, we can merge those two together with the union operation. As a result we have a heap with order 1.

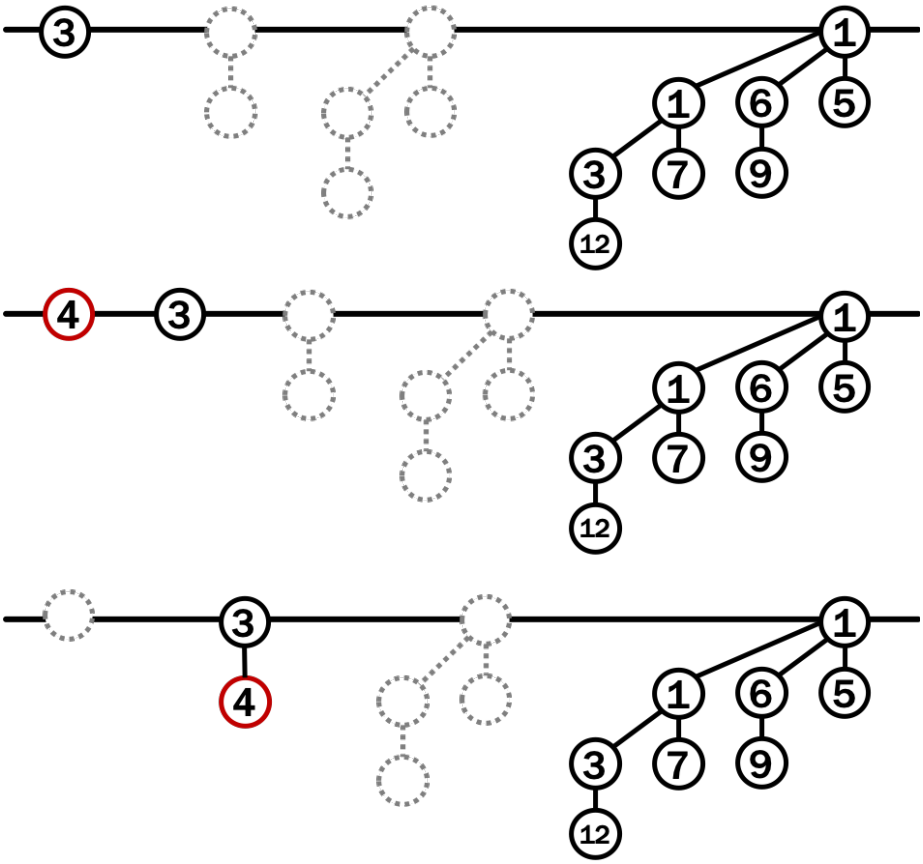


Figure 2: Example of the insert and union operation on heaps. We represent the not used orders with the faded heaps with no keys.

Another operation is the decreasing of a key in a node. As we mentioned in the introduction, this trees are arranged in a way that the smallest key node is at the top (in the case of min tree), so if we decrease the key of a node, depending on the father node of this decreased node, we may need to *"bubble"* up this node until the parent node is smaller than it. To see it in an example, we see in Figure 3, in the first step we take the node with the key 12 and change it to 2. Now, the parent to this node has a key value of 3, so it is necessary to move it up one space, replacing the node 3 with the node 2. Now we see that the new parent is key 1. Since this is smaller than 2, we leave the node in that place.

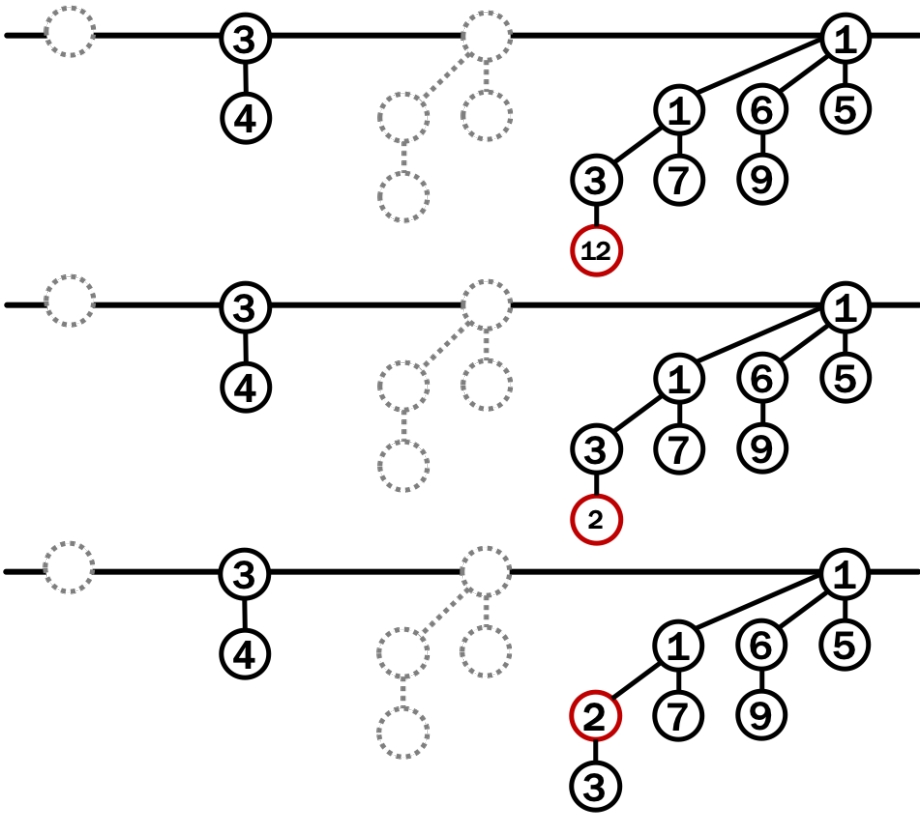


Figure 3: Example of the insert and union operation on heaps. We represent the not used orders with the faded heaps with no keys.

Then we have the find minimum and extract minimum. In this case, since we are working with a min heap, we only need to compare the values of the top node of each heap. In Figure 4 this top nodes are represented with the ones that are on the black line. In this case, the node with the minimum value is 1, so we take it out.

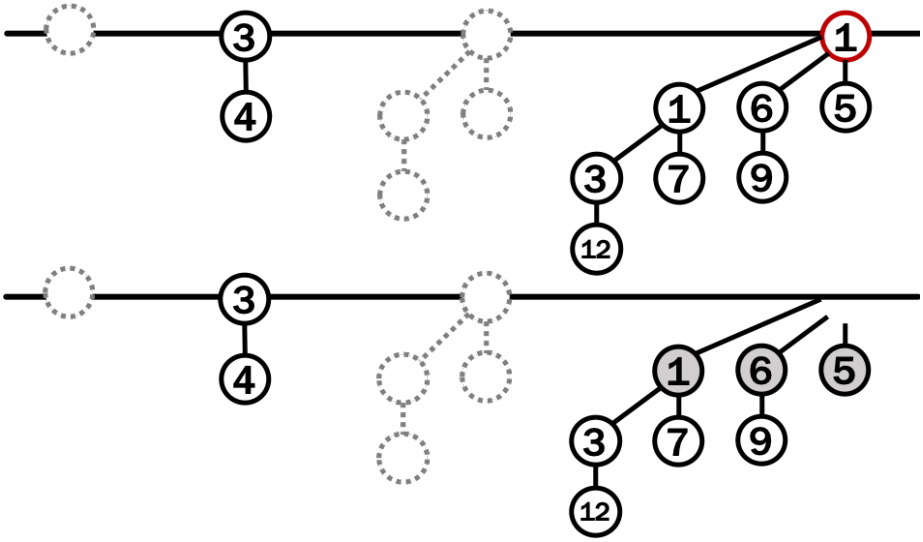


Figure 4: Example of finding the minimum node, and deleting it.

Then, on Figure 5 we put all of the remaining heaps on the line. For convenience we arrange them by their order level, because then it is easier to merge them together with the union operation. In the end we are left with one heap of order 1 and another with the order 3.

For the delete operation, it is similar to this procedure. The only difference is that we first search for the node that we want to delete, then we change that node key to $-\infty$, forcing the node to go up until it reaches the top of the heap, and then we can remove it with the same steps seen in Figure 4 and 5.

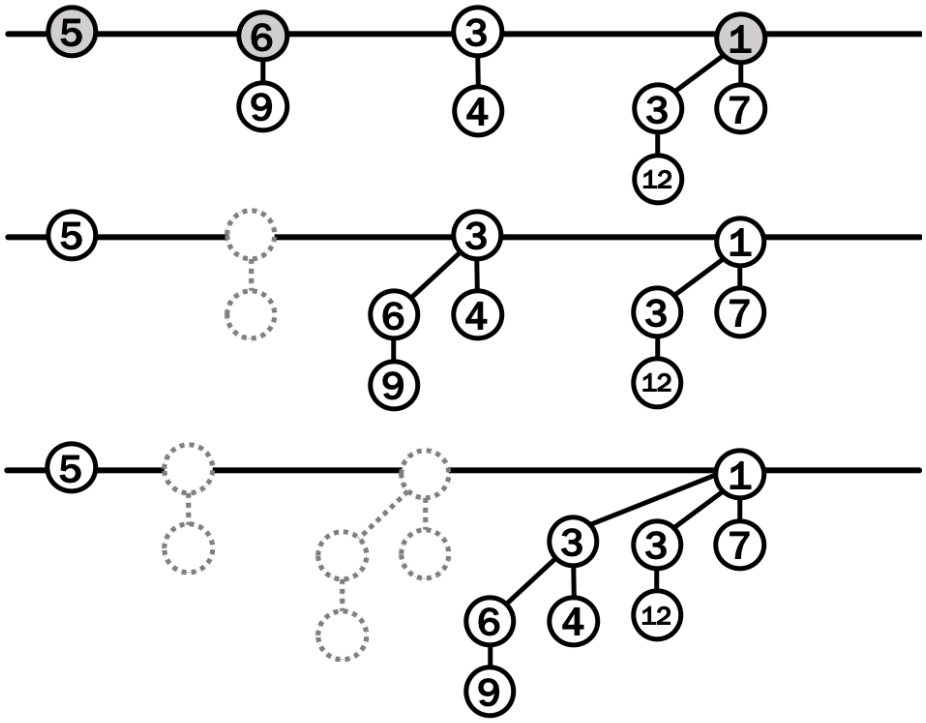


Figure 5: Example of the relocation of the heaps, and the merge of the heaps of the same order.

3 Example

With the same sequence of numbers used in the previous practice, we go inserting one node at the time in the order of *3, 1, 5, 7, 6, 8, 9, 10*. We see the results in Figure 6 and 7.

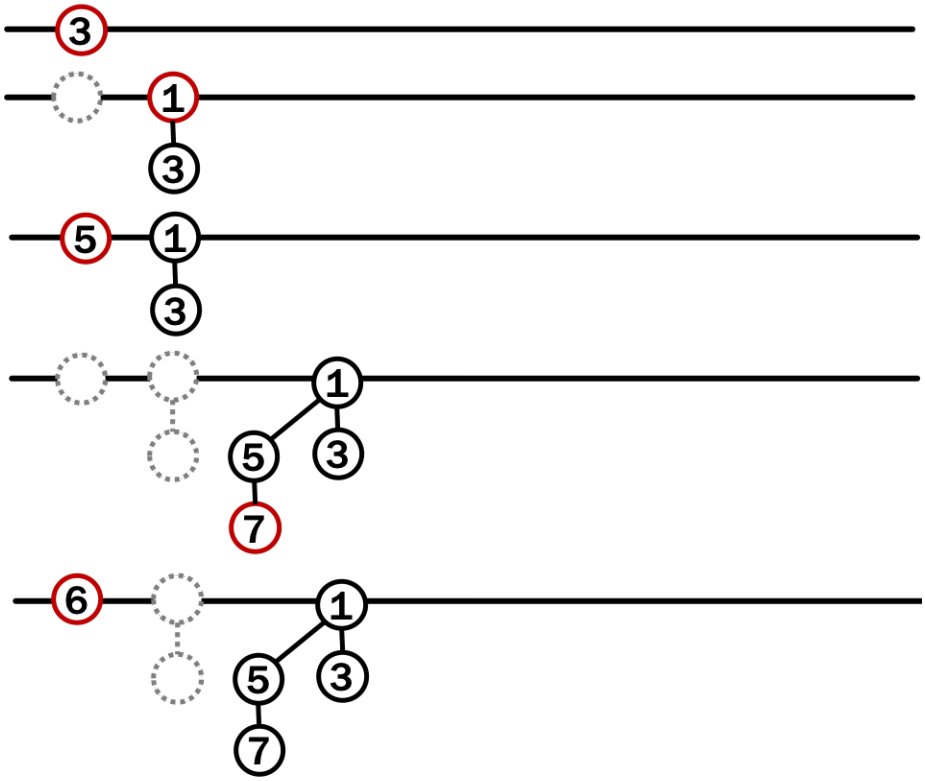


Figure 6: First part of example.

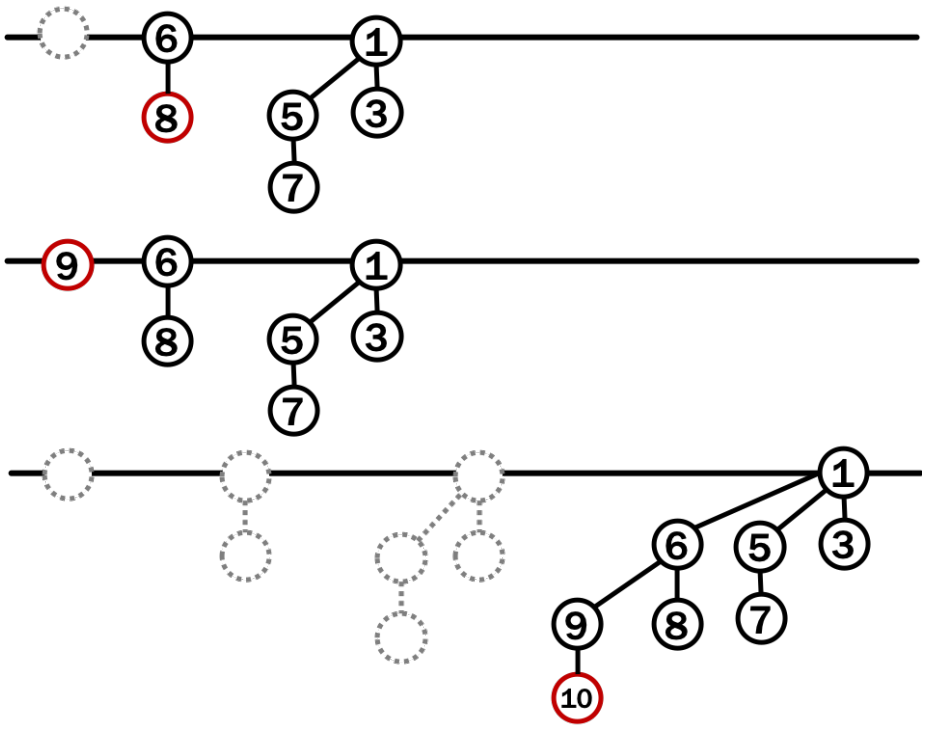


Figure 7: Last part of the example.

Table 1: Performance in running time

Operation	**Binary**	**Binomial**	**Fibonacci**
Find-min	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Delete-min	$\mathcal{O}\log n$	$\mathcal{O}\log n$	$\mathcal{O}\log n$
Insert	$\mathcal{O}\log n$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Dec-key	$\mathcal{O}\log n$	$\mathcal{O}\log n$	$\mathcal{O}(1)$
Merge	$\mathcal{O}\ m\log(n+m)$	$\mathcal{O}\log n$	$\mathcal{O}(1)$

4 Conclusions

In the end, it was a bit difficult for me to understand in which way I could use the binomial heap. The only way I could see it was in a sorting algorithm. In this case, the time it takes to search for the minimum is really fast, because of the way the heaps form and arrange when you put all the elements together and when you take them. I did not understand the fibonacci all that well, but I hope after seeing the explanation of my classmates I can understand better why the fibonacci has a better performance in running time.

References

[1] Binomial heap. https://www.youtube.com/watch?v=7UQd9SYUoNk

[2] Binomial heap https://www.geeksforgeeks.org/binomial-heap-2/

[3] Implement A Binary Heap https://www.youtube.com/watch?v=g9YK6sftDi0

Practice 10

Mayra Cristina Berrones Reyes

April 3, 2020

1 Introduction

For this practice we continue to explore some more data structures related to graphs. The subject is topological ordering or sorting. In this case it is necessary to refresh other concepts first.

1.1 Directed Acyclic Graphs

First of, a graph is a structure of several nodes that are connected by edges. In this case, this edges are depicted by arrows which represents the single directional flow from one node to the other. And acyclic means that there are no loops or cycles in the graph. A directed acyclic graph (DAG) is a type of graph in which, if you start in a certain node and follow the edges that connect this node to another, there is no path in the graph that can get you back to the original node.

Now, in a real world sense, we can see the DAG structure as a representation of a series of activities. The order in which this activities must be performed are depicted by the graph structure, each node representing an activity, some of them connected by lines that represent the flow from one activity to the other. Seeing it this way, there are many real world situation that we can model as a DAG, such as:

- School classes pre requisites
- Event scheduling
- Instructions for assembly.
- Program dependencies

An easy example for this can be seen in Figure 1 where each node is a class, and if, for example we want to take the class of the node 8, we need to take class 1, 2, 4, and 5 before we can take class 8.

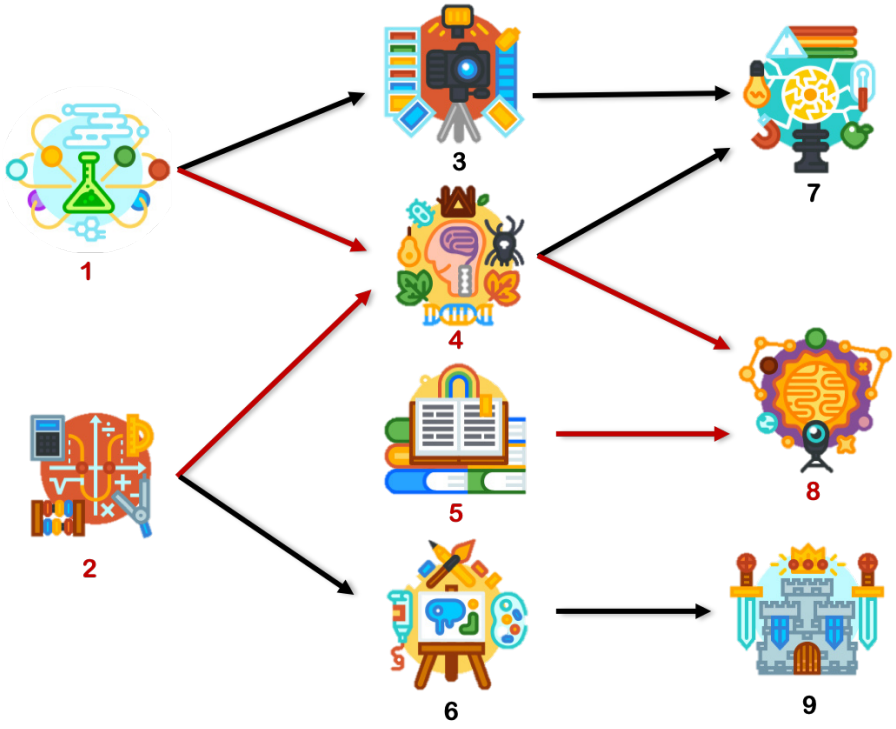


Figure 1: Example of the school class pre requisite.

2 Topological sorting

Topological sorting is an ordering of nodes. To explain it we can look again at Figure 1, and we can look at the nodes as a stack. A stack is a linear data structure that can store items in a last in/ first out (LIFO) or a First in/ last out (FILO) manner. The way a stack works is, when a new element is added, it goes on top of the last element added, and the only way to remove it, is by taking out all of the elements above this item. The insert and delete operations are commonly called push and pop.

There are two conditions in order to find the topological ordering or sorting of a graph:

- The graph must be a directed acyclic graph.
- The initial node in a topological graph should be a node with no incoming edges.

The first condition is explained above. The second condition is because, if the chosen node has incoming edges, then it can not be taken out of the stack. It is important to note that there is not a singular solution for this topological sorting. Now this is important because there are several ways to approach this problem.

2.1 Kahn's Algorithm

In this way we need use the adjacent list to select the order of the nodes. The steps for this are:

1. Identify the node that has no incoming edges. Select that node to start.
2. Delete the starting node and delete all the outgoing edges from the graph. Place the deleted node in the output list.
3. Repeat step 1 and 2 until the graph is empty.

As we can see in the steps, this algorithm works by starting with the nodes that have no incoming edges. In this case we need to choose the one node that has the same order as the final topological sort, so we need to pick the one with the most outgoing edges. The solution is an ordered list, that it is not necessarily unique.

In Figure 2 we see an example of a graph solved in this way.

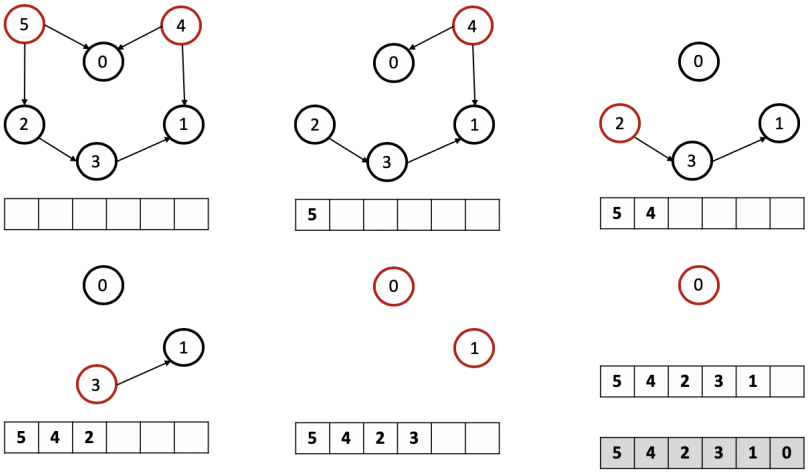


Figure 2: Example using the Kahn's algorithm.

2.2 Depth-first search

An alternative algorithm to solve this problem is based on depth-first search (DFS). In this other way the algorithm loops through each node of the graph in arbitrary order. From this node, it does a DFS and find the last node connected with a path to the starting node, save it on a stack, then delete that node. This step repeats until the graph is empty. In this case, in Figure 3 depicts the way it deletes the nodes. In the end, we print the elements of the stack. Since the only way to pop items from a stack is last ones first, we get the topological ordering. Since each edge and node are visited once, the algorithm runs in linear time.

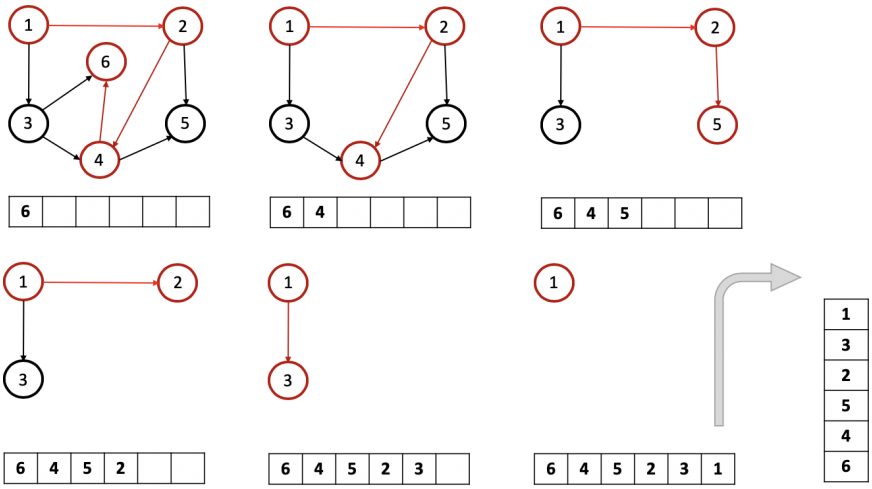


Figure 3: Example using the DFS algorithm

3 Code and terminal

```
from collections import defaultdict
class Graph:
  def __init__(self,vertices):
    self.graph = defaultdict(list) #adjacency List
    self.V = vertices #vertices

  def addEdge(self,u,v):
    self.graph[u].append(v)


  def topologicalSortUtil(self,v,visited,stack):
    visited[v] = True
    for i in self.graph[v]:
      if visited[i] == False:
        self.topologicalSortUtil(i,visited,stack)

    # Push
    stack.insert(0,v)

  def topologicalSort(self):
    visited = [False]*self.V
    stack =[]

    for i in range(self.V):
      if visited[i] == False:
        self.topologicalSortUtil(i,visited,stack)

    print stack

g= Graph(6)
g.addEdge(5, 2);
g.addEdge(5, 0);
g.addEdge(4, 0);
g.addEdge(4, 1);
g.addEdge(2, 3);
g.addEdge(3, 1);

print "Topological Sort"
g.topologicalSort()
```

```
(base) Mayras-MacBook-Pro:desktop mayraberrones$ python topo2.py
Topological Sort
[5, 4, 2, 3, 1, 0]
(base) Mayras-MacBook-Pro:desktop mayraberrones$
```

Figure 4: Result on terminal

4 Conclusions

Since there is not one concrete answer to this algorithm I struggled a little at the beginning because I wanted to do them by hand to understand them. Seeing the

problem as a DFS made it a lot easier to understand. Also the first example with the classes made it more relatable, since it is easier to understand the graph as a series of tasks that you have to complete in order to get to the one that you want.

Since I have worked with the Hamiltonian path in several other practices, the way the algorithm of DFS reminded me a little of finding paths in the graph. After reading some more on this, I found that there is a property on the topological sort that says if all pairs of consecutive nodes in the sorted order are connected by edges, then this form a Hamiltonian path. This property is called uniqueness, and it has been used to test in linear time if a Hamiltonian path exist despite its complexity. This could have been useful when I was studying the subject of reductions.

References

[1] Topological sort algorithms. Graph theory. https://youtu.be/eL-KzMXSXXI

[2] Stack in python https://www.geeksforgeeks.org/stack-in-python/

[3] DAG https://golden.com/wiki/Directed_acyclic_graph-RYDG9

[4] Topological Sorting in Python https://www.codespeedy.com/topological-sorting-in-python/

Practice 11

Mayra Cristina Berrones Reyes

April 10, 2020

1 Introduction

A queue is a data structure similar to a real life scenario of a line in the groceries. The first person in the queue is the first person to pay their groceries. So viewing it as such, a queue follows the First In First Out (FIFO) rule. First item to go in is the first item that comes out too.

2 Priority queues

In this case, a Priority Queue (PQ) is a special type of queue, because the first item to go in may not necessarily be the first to go out. In PQ each item is comes with a certain priority, and it is served according to this priority. In programing this can be shown as an extra key to the item, for example, in the Dijkstra algorithm, the item can be the edge between two nodes, and the extra key the weight of using that edge. However, generally in many examples we find it as the value of the element itself is considered the assigned priority, like we see in any sorting algorithms. In this case the highest or the lowest value can be considered the highest priority element, depending on our needs.

The basic operations of a PQ can be described as:

- **Add**: Adds an item.
- **Remove**: Removes the highest priority item.
- **Peek**: Returns the highest priority item without removing it.
- **Size**: Returns the number of items in the PQ.
- **IsEmpty**: Returns whether the PQ has no items.

A PQ can be implemented using various data structures such as an array, a linked list, a heap data structure or a binary search tree. In this case the functions of add, remove and peek for each structure has different performances as we can see in Table 1.

Table 1: Performance of the different data structures for PQ in running time

	Peek	**Insert**	**Delete**
Array (sorted)	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Array (unsorted)	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Linked list (sorted)	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Linked list (unsorted)	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Binary heap	$\mathcal{O}(1)$	$\mathcal{O}\log n$	$\mathcal{O}\log n$
Binary search tree	$\mathcal{O}(1)$	$\mathcal{O}\log n$	$\mathcal{O}\log n$

The most common data structure used for PQ is binary heap. We are going to show an example of sorting problem in each structure to show why the binary heap is the preferred method.

2.1 Arrays for PQ

Starting with the arrays. This data structure is the least favored one because, unlike all of the other structures, we have to have a fixed number of items in mind to be able to save the memory space. If we are working with an unsorted array, inserting items is very simple, since we can just add them to the end of the queue, so this is the complexity $\mathcal{O}(1)$.

Imagining that we have enough space for out items in the PQ, the next thing we need to do is de-queue our PQ. Here is where the complexity becomes $\mathcal{O}(n)$, because in the worst case scenario, to search the item with the highest priority we must do a linear scan to find it, and if it is at the end of the array, we need to pass all the other elements to get to it.

Same thing happens to remove, in this case, because there can not be an empty space between spaces in arrays, so the worst case scenario in this is if the highest priority item is at the beginning, since we have to shift all elements to cover the vacant space, that is also complexity $\mathcal{O}(n)$.

In the case of the sorted array, the insertion becomes more complex, because we can not just add the element at the end of the array, we must find where to insert the element. This in itself can be done in log time, but then we have to move all the other items to make space for the one we are inserting, which then becomes $\mathcal{O}(n)$.

Peek and remove become constant time $\mathcal{O}(1)$ because if the array is sorted, then the element with the highest priority is at the end. Removing it will not move the other items.

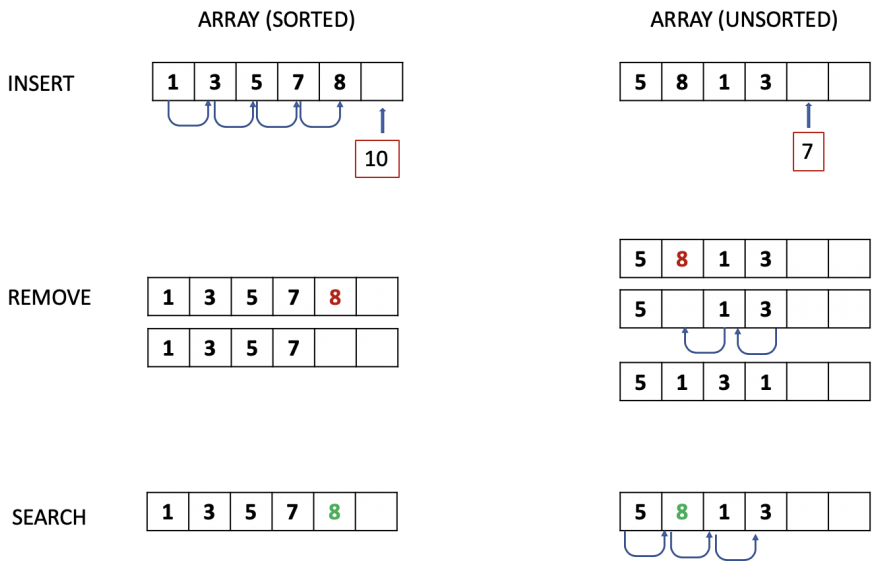


Figure 1: Examples of array features.

2.2 Linked lists for PQ

Like arrays the linked list structure is linear, but unlike arrays, this structure is not stored at a contiguous location in memory. This elements are linked using pointers. They also are dynamic, so they are not constrained by a fixed size.

For the unsorted and sorted linked list the same rules apply as the unsorted and sorted array. The difference is that in the unsorted linked list is in the remove part, where it has the chance to become log because it does not have to rearrange all the other items after removing one.

2.3 Binary search tree for PQ

The structure for a binary search tree (BST) has the following properties:

- The left child of a node contains only the node with lesser key.
- The right child contains the node with a greater key.
- There can not be duplicate nodes.

For this structure the insertion procedure is quite similar to searching. We start from the root node and in a recursive way we go down the tree following the properties of a BST. If the item is already on the tree, then we are done,

since duplicates are not allowed.

For searching it can have a $\mathcal{O}(h)$ complexity because it depends on the size of the tree. This could even turn to a $\mathcal{O}(n)$ complexity, because a BST can degenerate to a linked list in certain cases.

This would not happen if we are working with a self balancing binary search tree such as a AVL tree or a red-black tree. In this case finding the minimum en maximum element will be $\mathcal{O}(1)$. Inserting and removing elements will be $\mathcal{O}\log n$ complexity, because this means that, in order to do this procedure, they do not have to go trough all of the elements in the tree, only a certain part.

2.4 Binary heap for PQ

Among all the other data structures, binary heap structure provides a efficient implementation of priority queues. Heap is a complete binary tree in which the root node is the smaller value of all the trees, in this case the convention is that the smallest element is the one with the highest priority.

It is simple to explain the search or peek procedure, because the highest priority element is always going to be on top. So the complexity is $\mathcal{O}(1)$. To insert a new item to the tree, we put it to the last space (the next leaf from left to right) and then we bubble up the element until it reaches its appropriate place in the tree. Since the tree we are working on is a complete tree, it has minimum height and the worst number of moves has $\mathcal{O}\log n$ complexity.

To begin the de-queueing we need to remove the element on top. Once this is done, we replace the root node with the last node of the tree (the last leaf from left to right) and then we bubble it down until it reaches its appropiate place in the tree. Since it is similar to the insertion, it also has a complexity of $\mathcal{O}\log n$.

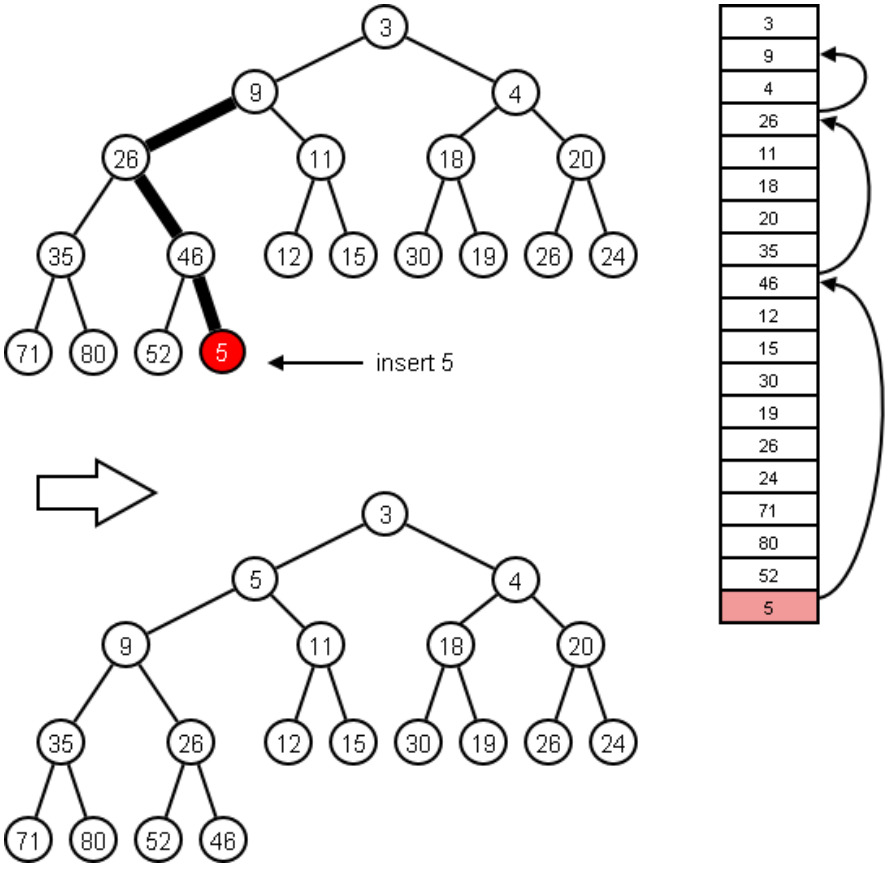


Figure 2: Examples of heap insertion from [1].

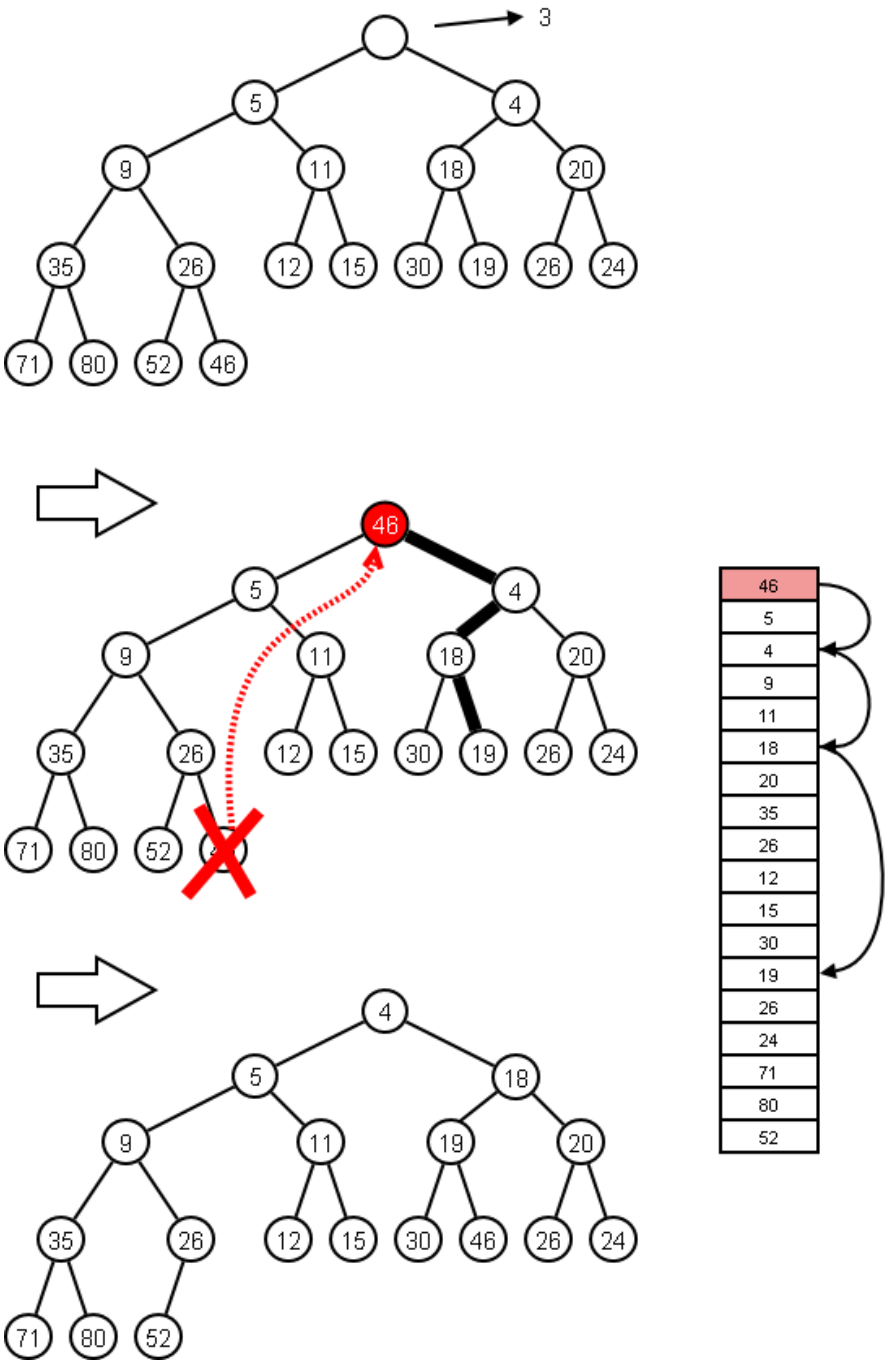


Figure 3: Examples of heap delete from [1]

3 Conclusions

As we stated before, the binary heap structure is the preferred method to solve priority queues. In this case, as in many other practices we have done, it depends on the features of your problem to decide which structure to use.

If we had few elements in our queue, it would be easier to implement a simple array or a linked list. In this case, the array has the advantage to have random access to its elements, and it has more variety of options to use a sorting algorithm, since a linked list can only use a few efficiently. In the case of an unknown set of elements, a linked list would be more appropriated, since has a dynamic size and it is easier to perform the delete and insert tasks.

However, if the number of elements to sort becomes to great, the $\mathcal{O}(n)$ complexity becomes a problem, and the $\mathcal{O} \log n$ complexity of the trees becomes more efficient. In this case I had a hard time deciding which one is best, because from all that I read they seemed to have similar advantages, but when I had the chance to visualize the BST without the self balancing feature, I realized why the heap method is more commonly used. It also seems more cleaner to me in the way that it arranges its nodes.

In Figure 4 I made a BST and a binary heap with the same sequence of numbers (_4, 5, 8, 11, 12,6, 9, 7) and that is what I meant when I said the binary heap looks cleaner than BST without the self balancing.

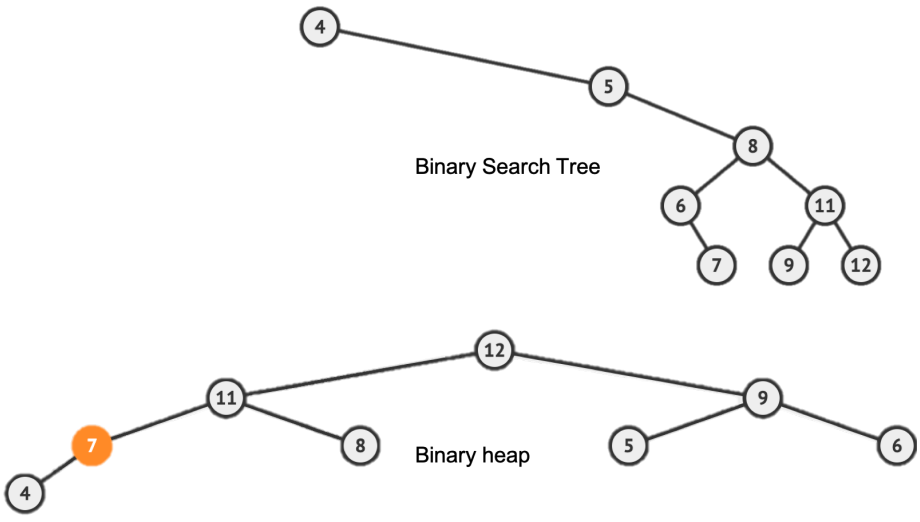


Figure 4: Difference of structure of Binary search tree and binary heap.

References

[1] Priority queues https://cs.lmu.edu/~ray/notes/pqueues/

[2] Data structures - Priority queues https://www.programiz.com/dsa/priority-queue

[3] Linked lists https://www.geeksforgeeks.org/linked-list-set-1-introduction/

[4] Binary Search trees https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/

[5] Binary heaps https://www.geeksforgeeks.org/binary-heap/

[6] Binary search tree visualization https://visualgo.net/en/bst

[7] Binary trees https://www.cs.cmu.edu/~adamchik/15-121/lectures/Trees/trees.html

Practice 12

Mayra Cristina Berrones Reyes

April 17, 2020

1 Introduction

The greatest common divisior (GCD) of two numbers is the largest number that can divide them both without residual numbers. This concept can easily extend to a bigger set of numbers, because the GDC is the largest number dividing each of them.

The GDC is used for a variety of applications in number theory such as modular arithmetic, but it can also be used in simpler applications, such as simplifying fractions. The GDC is traditionally notated as $gdc(a, b)$ [Alexander Katz, 2020]. The problem to solve is, given two non-negative integers a and b, we need to find their GCD, which is the largest number that can divide both a and b. Mathematically we can define it as Equation 1.

$$gcd(a, b) = max\, k. \tag{1}$$

\noindent Where:

$k = 1 \ldots \infty : k|a \wedge k|b$

(In several examples found, they use the "|" symbol as divisibility. Meaning that $k|a$ is "k" divides "a")

In this case, if one of the numbers a or b is zero, while the other is not zero, then the GCD by definition, is the non-zero number. For example $gcd(a, 0) = a$. When both numbers are zero, then the GCD remains undefined, as it can be any arbitrarily large number [Kogler, 2014].

There are several ways to solve the GCD. One of them is by simply listing the factors of each number and determining the largest common one. For example, in Figure 1 we have the number 30 and 24, then we have a list with all of the factors. As we can see, the largest common factor is the number 6, so that is our GCD. This practice however is very inefficient, but for small cases it can be done by hand. There are other methods to solve this, such as the euclidean

algorithm [Alexander Katz, 2020].

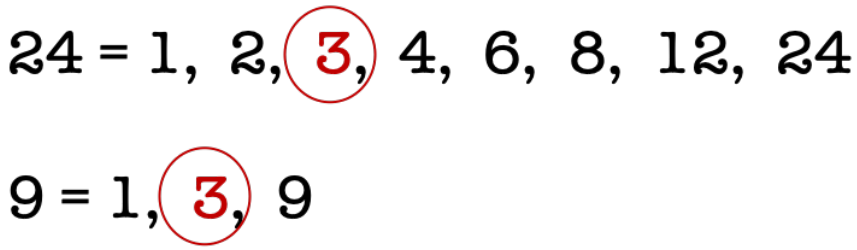


Figure 1: Example of listing the factors of the numbers to find de GCD.

2 Euclidean algorithm

The euclidean algorithm is one of the oldest numerical algorithms that is used commonly to this day. The exact origin of the algorithm is unknown, but it was first described in Euclid's *"Elements"* in 300 B.C. and it solves the problem of computing the GCD of two positive integers.

2.1 GCD by subtraction

The original version of the euclidian algorithm is based on subtraction. We recursively subtract the smaller number from the bigger one [Codility, 2014].

```
def gcd(a, b):
  if a == b:
    return a
  if a > b:
    gcd(a - b, b)
  else:
    gcd(a, b -a)
```

Above we can see the code to solve this problem. In Figure 2 is an example of how this algorithm work, choosing the bigger number, subtracting the smaller one until they both are the same. The worst case complexity for this problem can be linear $\mathcal{O}(n)$, because the value of the integers decreases with every step.

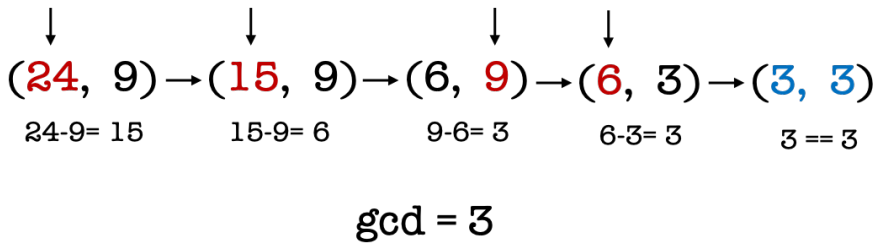


Figure 2: Example of the euclidian algorithm by sustraction.

2.2 GCD by dividing

This is the most common implementation for the euclidean algorithm for GCD. The algorithm for this can be seen in Equation 2.

$$gcd(a,b) = \begin{cases} a & \text{if } b = 0 \\ gcd(b, a \bmod b), & \text{otherwise} \end{cases} \tag{2}$$

This equation is for two given numbers a and b, such that $a \geq b$. The *mod* operation in Equation 2 is the residual of $a \mid b$. In Figure 3 we can see how this equation works, and below the code for it.

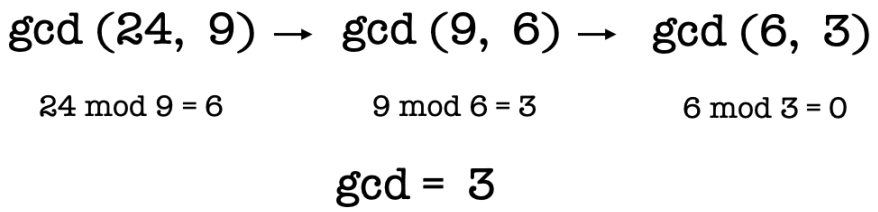


Figure 3: Example of the euclidian algorithm for GCD by dividing.

```
def gcd(a, b):
  if a % b == 0:
    return b
  else:
    return gcd(b, a % b)
```

Now, we can see in each iteration the argument decreases, which means that the algorithm will always terminate, because our elements are non-negative.

For proof of correctness, we already stablished the first part of Equation 2 that if the other number is 0, then the GCD is the non-zero number. So now we need to prove that $gcd(a,b) = gcd(b, a \bmod b)$ for all $a \geq 0$, and $b > 0$.

Let d be the GCD of (a, b). Then by definition $d \mid a$ and $d \mid b$, which means that both a and b are divisible by d.

$$a \, mod \, b = a - b \cdot \left\lfloor \frac{a}{b} \right\rfloor \tag{3}$$

In Equation 3 we represent how the *mod* operation works. From this follows that $d \mid (a \, mod \, b)$ [Trevor, 2015].

How do we know the algorithm works is, if $a = bq + r$ a and b being our integers to find the GCD and q and r some other integers, then we have that $gcd(a, b) = gcd(b, r)$. In Figure 4 we have an example of how this works. First we have the a number represented by 2322. Then the b number which is 654. The q number represents the integer which will multiply the b number, and finally the r represents the residue necessary to get number a. After all recursions we get the GCD as 6.

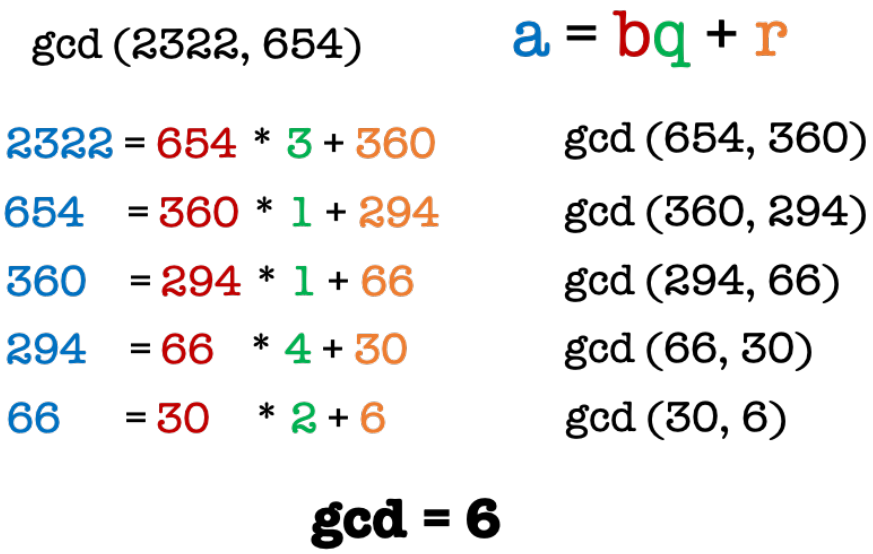


Figure 4: Example of how to get the GCD by dividing.

Now, the only way to prove the algorithm is if $gcd(a, b) = d$ and $gcd(b, r) = d$. We already proved that $d \mid a$ and $d \mid b$, and in Equation 3 we can translate that to $d \mid (a - qb)$. With this, we can use the equation in Figure 4 of $a = bq + r$ we can clear r and we have then $r = a - bq$, which translates as $d \mid r$. So, as shown in Figure 5 we prove both sides and we can conclude that $gcd(a, b) = d$

and $gcd(b, r) = d$.

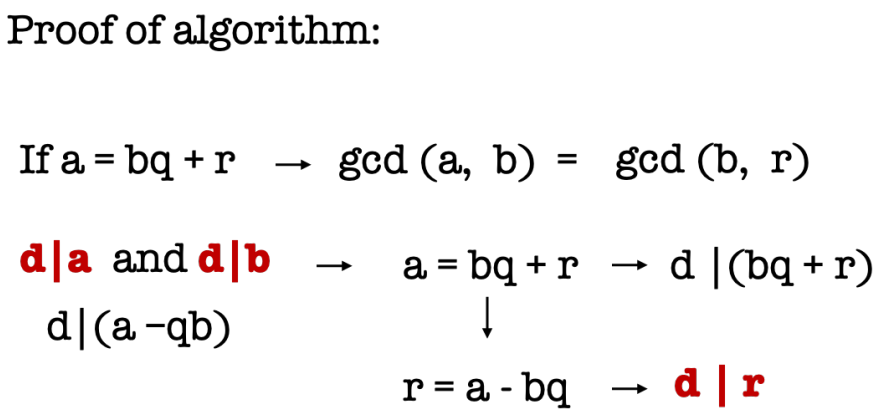


Figure 5: Proof of both sides of the GCD.

For the time complexity, we can estimate the number of recursions this problem may have with the Lame theorem, that establishes a connection between the euclidian algorithm and the Fibonacci sequence. If $a > b \geq 1$ and $b < F_n$ for some n, the euclidean algorithm performs at most $n - 2$ recursive calls. The consecutive Fibonacci numbers are the worst case input for the euclidian algorithm, and given that Fibonacci numbers grow exponentially, we get that this algorithms works in $\mathcal{O}(\log \underline{min}\,(a, b))$.

3 Conclusions

I tried to make this practice more informative for me, than to make it similar to the example on the book, because I struggle a little bit to understand the algebraic format (too many letters and numbers). The part of the Fibonacci sequence to see the time complexity for the algorithm was not completely clear for me, because the Lame theorem is very old, and I got confused with all the different interpretations of it. At the end the only thing clear it was that it was the worst case for the euclidian algorithm because it grows exponentially.

References

[Alexander Katz, 2020] Alexander Katz, Patrick Corn, M. A. (2020). Greatest common divisor. Disponible en: "https://brilliant.org/wiki/greatest-common-divisor/".

[Codility, 2014] Codility (2014). Euclidean algorithm. Disponible en: "https://codility.com/media/train/10-Gcd.pdf".

[Kogler, 2014] Kogler, J. (2014). Euclidean algorithm for computing the greatest common divisor. Disponible en: "https://cp-algorithms.com/algebra/euclid-algorithm.html".

[Trevor, 2015] Trevor (2015). Euclidean algorithm. Disponible en: "https://www.youtube.com/watch?v=cOwyHTiW4KE".

3 pts, quedó corto de la meta

Practice 13

Mayra Cristina Berrones Reyes

April 24, 2020

1 Introduction

If we can take any problem, divide it into smaller sub problems, solve these smaller sub problems and combine their solutions to find the solution for the original problem, then it becomes easier to solve the whole problem. This concept is known as **Divide and conquer**. This concept involves three steps [Gimel'farb, 2016]:

1. **Divide** the problem into multiple small problems.
2. **Conquer** the sub problems by solving them.
3. **Combine** the solutions of the subproblems to find the solution to the actual problem.

In Figure 1 we illustrate a diagram showing these three steps.

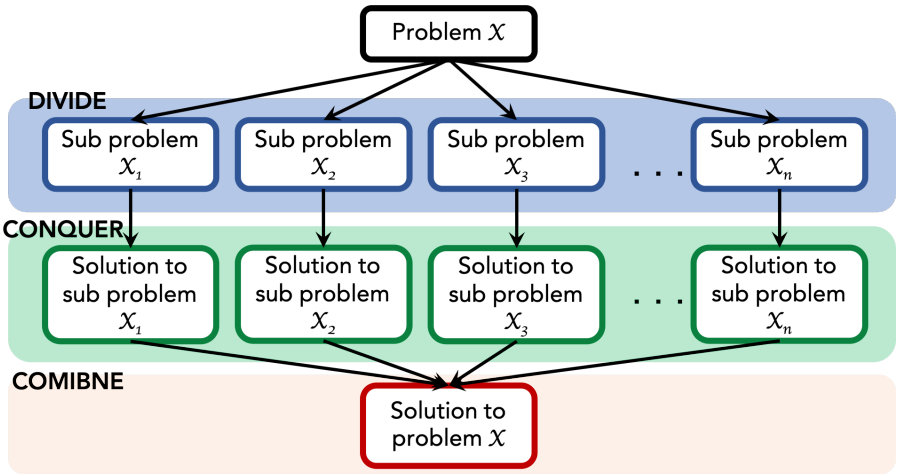


Figure 1: Diagram of divide and conquer steps.

2 Merge sort algorithm

Merge sort uses the divide and conquer rules, breaking a problem into smaller sub problems, the problem in this case being of sorting a given array of numbers. The steps for the merge sort algorithm are as follows:

1. We are going to take a variable x, and store the starting index of our array in it. Then we do the same but for the last index of our array and store it in a variable y.
2. With the formula $(x + y)/2$ we find the middle of the array and store the index as z.
3. Break the array into two sub arrays, from the index of x to z and form the index $z + 1$ to y.
4. We take each of the sub arrays and repeat steps 1 to 3 until the sub arrays are in single elements.
5. We start to merge the sub arrays.

In Figure 2 and 3 we have an example of the steps enumerated above.

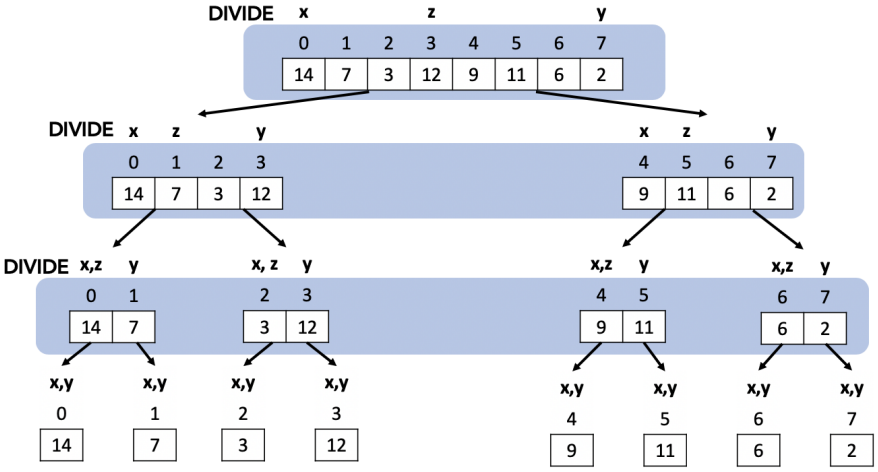


Figure 2: Example of the divide part on the merge sort algorithm.

2.1 Complexity analysis of Merge sort

The merge sort algorithm is very fast, and it is also a stable sort, which means that the equal elements in the array are ordered in the same order in the sorted list. The time complexity of the merge sort in all the 3 possible cases (Best,

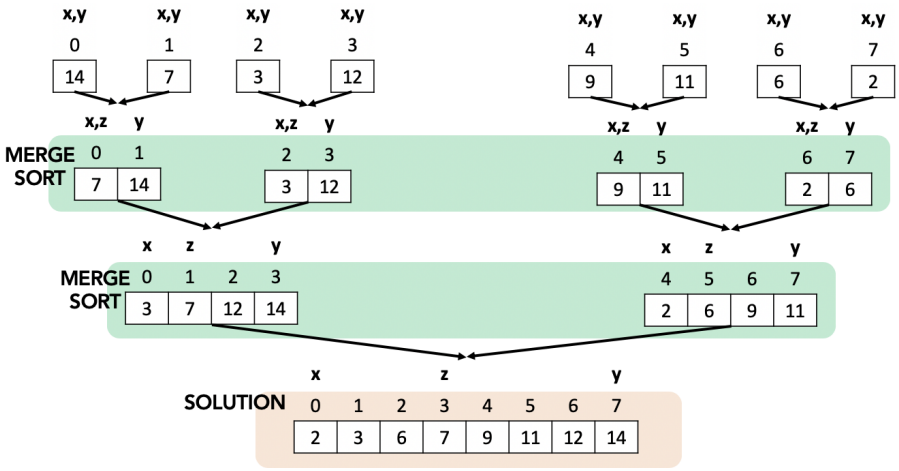


Figure 3: Example of the merge and sort part of the merge sort algorithm.

worst and average cases) is always $\mathcal{O}(n * \log\ n)$. This is because merge sort always divides the array in two halves and takes linear time to merge two halves. It also needs the same amount of additional space as the unsorted array, so this is not recomended for searches in large unsorted arrays. This is however, the best technique for sorting linked lists [Ahlawat, 2020].

Now we explain why the complexity of this algorithm is $\mathcal{O}(n * \log\ n)$ [G, 2016].

$$T(n) = 2T\left(\frac{n}{2}\right) + cn \tag{1}$$

where:

- $T(n) =$ represents the runtime of the algorithm.
- n is the number of elements in the array.
- $2T\left(\frac{n}{2}\right)$ represents solving the sub problems through recursion.
- cn represents that merging them together will take constant time c of n.

In this case, if the value of n in Equation 1 is 1, then $T(n) = c$ which means that the runtime of the algorithm is constant time. With that we can put it as Equation 2.

$$\begin{cases} T(1) & = c, \\ T(n) & = 2T\left(\frac{n}{2}\right) + cn \end{cases} \tag{2}$$

Now, with the iterative application, we can get Equation 3

$$\begin{aligned}
T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\
T\left(\frac{n}{2}\right) &= 2T\left(\frac{n}{4}\right) + \frac{cn}{2} \\
T(n) &= 2\left(2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn \\
T(n) &= 4T\left(\frac{n}{4}\right) + cn + cn \\
T(n) &= 4T\left(\frac{n}{4}\right) + 2cn \\
T\left(\frac{n}{4}\right) &= 2T\left(\frac{n}{8}\right) + \frac{cn}{4} \\
T(n) &= 4\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + 2cn \\
T(n) &= 8T\left(\frac{n}{8}\right) + 2cn + cn \\
T(n) &= 8T\left(\frac{n}{8}\right) + 3cn
\end{aligned} \tag{3}$$

Equation 3 represents the progression of the iterative application, and in the end, we begin to see a pattern, form which we can form Equation 4

$$T(n) = 2^k * T\left(\frac{n}{2^k}\right) + k * cn \tag{4}$$

From Equation 4 we need to get to the $\mathcal{O}(n * \log n)$ time complexity, and for this, we need to establish a few things. So we have that

$$n = 2^k$$

so we can say that

$$\frac{n}{2^k} = \frac{2^k}{2^k} = 1$$

so finally we have

$$T\left(\frac{n}{2^k}\right) = T(1).$$

Now to get the log we have the formula

$$\log_{base} result = exponent \qquad \text{\textbf{\textbackslash text\{result\}}}$$

taking into consideration the $n = 2^k$ we then have

$$\log_2 n = k$$

Now we take Equation 4 and replace the elements

$$\begin{aligned} T(n) &= 2^k * T\left(\frac{n}{2^k}\right) + k * cn \\ T(n) &= 2^{\log_2 n} * T(1) + \log n * cn \\ T(n) &= 2^k * c + \log n * cn \\ T(n) &= n * c + \log n * cn \\ T(n) &= n + n(\log n) \end{aligned} \tag{5}$$

In Equation 5 we can see the few considerations above to follow the changes on the equation. In the end, we need to compare and weigh which side of that sum has a greater impact on the equation. We can conclude that $n(\log n)$ is a slightly bigger number than n so, then we have the complexity as $\mathcal{O}(n * \log n)$

3 Conclusions

In this case, I finally understood every equation, because I took it step by step, and whenever I was dubious about parts of the equation, I took the example from Figure 2 and 3 to see if everything matched. An example of this is when I tried to understand the changes made in Equation 5 when it incorporated the log function, I did not understand why the $n = 2^k$, but following the example, if I substitute n for the 8 elements in the example, and find the value of k with the $\log_2 n$ I got

$n = 8$
$\log_2 n = \log_2 8 = 3$

so $k = 3$ and finally, with $n = 2^k$ is the same as $8 = 2^3 = 8$. And then all the rest makes sense.

References

[Ahlawat, 2020] Ahlawat, A. (2020). Merge sort algorithm. Avialable: "https://www.studytonight.com/data-structures/merge-sort".

[G, 2016] G, C. (2016). Course of computer science algorithms. Avialable: "https://www.youtube.com/channel/UCPjbFOR74Ns0jTorAd3Fgdg".

[Gimel'farb, 2016] Gimel'farb, G. (2016). Algorithm mergesort complexity. Avialable: "https://www.cs.auckland.ac.nz/compsci220s1c/lectures/2016S1C/CS220-Lecture09.pdf".

Practice 14: Amortized analysis of Red – Black trees

Mayra Cristina Berrones Reyes

May 1, 2020

1 Introduction

Amortized analysis, as its name portraits, is a worst case analysis of a sequence of operations. This is used for algorithms where we have a majority of fast operations, commonly known as cheap operations, but every now and then we find a very slow operation, or expensive operation. Amortized analysis allows us to have an average worst case time that is lower than the worst case time of a particular expensive operation. There are three techniques used for this analysis [CIS, 2011]:

1. **Aggregate method** where we analyse the total running time for a sequence of operations.
2. **Accounting method**, also known as the bankers method, is where we put an extra charge on cheap operations and use it to pay for the expensive operations later on.
3. **Potential method**, also known as the physicists method, we have to derive a potential function the extra amount of work we can do in every step. The potential either increases or decreases with each operation, but it can never go below zero.

2 Red – black tree

We previously worked with red – black trees. They are a self balancing binary search tree, where we need to follow some rules. It is important that we remember them, because they determine the time complexity of some of the operations.

1. Every node has a color, either red or black.
2. The root of the tree must always be black.

3. There can not be two adjacent red nodes, which means that a red node can not have a red parent or red child.

4. Every path from the root node to any of the null decedents (leafs) has to have the same number of black nodes.

Now, the red – black tree has three main operations. Insertion, deletion and search. For the insertion and deletion operation, we need to make some changes to the tree, so it keeps complying with the features we mentioned before, so we have other operations inside the insertion and deletion, which are rotate left, rotate right, an change the color of the node, and we use them depending on where the node we inserted or deleted is.

3 Amortized analysis of red – black tree

For this amortized analysis we are using the **potential method**. So, firstly we suppose that we can define the potential function Φ on states of a data structure with the following properties Rhodes, 2020:

- $\Phi(h_0) = 0$, where h_0 is the initial state of the data structure.
- $\Phi(h_m) \leq 0$, for all states h_m of the data structure during the computation.

The way this analysis works is similar to the concept of the bankers method in the way that keeps track of the balance we have available to pay for more expensive operations, but in this case we take into consideration the whole history of the computation of the state we are in.

Now we can define the amortized time of an operation as Equation 1 where T_i is the actual cost of the operation and $\Phi(h_i)$ and $\Phi(h_i - 1)$ are the states of the data after and before the operation respectively. Thus the amortized time is the actual time plus the change in potential. Ideally, the change in potential should be positive for low-cost operations and negative for high-cost operations. Demaine, 2016

$$A_i = T_i - \Phi(h_i) - \Phi(h_i) \quad (1)$$

Now consider a sequence of n operations taking actual times $c_0, c_1, c_2, ..., c_n$?1 and producing data structures $h_1, h_2, ..., h_n$ starting from h_0. The total amortized time is the sum of the individual amortized times:

Table 1: Credit assignment for the tree

Features	**Credits assigned**
A black node with 1 red child	0
A black node with 2 black children	1
A black node with 2 red children	2

Table 2: Rules for assigning credit in insertion operation

Features	**Credits assigned**
Attach a node to a black node and terminate insertion	-1
Update colors and move up	-1
Perform single/double rotation and terminate	+2

$$
\begin{aligned}
&(c_0 + \Phi(h_1) - \Phi(h_0)) + (c_1 + \Phi(h_2) - \Phi(h_1)) + ... + (c_n - 1 + \Phi(h_n) - \Phi(h_n - 1)), \\
&= c_0 + c_1 + ...c_n - 1 + \Phi(h_n) - \Phi(h_0) \\
&= c_0 + c_1 + ... + c_n - 1 + \Phi(h_n).
\end{aligned} \tag{2}
$$

Therefore the amortized time for a sequence of operations overestimates of the actual time by $\Phi(h_n)$, which by assumption is always positive. Thus the total amortized time is always an upper bound on the actual time. Now we can take Equation 2 and clean it to look like Equation 3 [Ashish Bahendwar et al., 2018].

$$
\sum_{i-1}^{m} T_i = \sum_{i-1}^{m} (A_i - \Phi(h_n) - \Phi(h_n - 1)) = \Phi(h_0) - \Phi(h_m) + \sum_{i-1}^{m} A_i. \tag{3}
$$

Now that we explained our equations, we need to establish the rules and the credits that we are going to assign.

4 Example

We have a red – black tree in Figure 1 and then we use the insert function to insert the number 25. In Figure 1 we see the before and after of the tree, and in Table 5 we see how we can calculate de potential with the equations seen previously and following the rules on Table 1, 2, and 3. T_i takes the value of 4, because in Table 4 it says that this value is going to be as many nodes were

Table 3: Rules for assigning credit in deletion operation

Features	**Credits assigned**
Delete a black node and move up	-1
Update colors and move up	-2
Update colors and terminate deletion	-1
Perform single rotation and terminate deletion	-1 / +2
Perform double rotation and terminate deletion	+2

Table 4: Assumed value of T_i

Features	**Credits assigned**
Left rotate	1
Right rotate	1
Recolor node	Number of nodes recolored

Table 5: Potential before and after de insertion operation of the node 25

Nodes	8	5	17	15	18	25	Potential
$\Phi(h_0)$	1	1	2	-	-	-	4
$\Phi(h_1)$	0	1	-	1	0	-	2

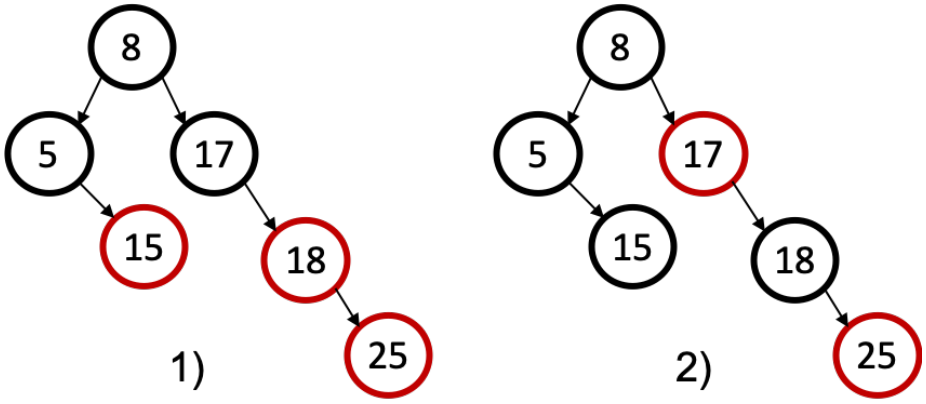


Figure 1: Example of an insertion in a red – black tree.

Table 6: Potential before and after the deletion of node 40

Nodes	8	5	17	15	25	18	40	Potential
$\Phi(h_0)$	0	1	-	1	2	-	-	4
$\Phi(h_1)$	0	1	-	1	0	1	-	3

recolored, and in the example we have 4 nodes that changed their color.

$$\begin{aligned} A_i &= T_i - \Phi(h_i) - \Phi(h_i) \\ &= 4 - 2 - 4 = -2 \end{aligned} \tag{4}$$

So now we have

$$\begin{aligned} &= \Phi(h_0) - \Phi(h_m) + \sum_{i-1}^{m} A_i \\ &= 4 - 2 + (-2) = 0 \end{aligned} \tag{5}$$

Now for the deletion function we have the potential from before and after in Table 5 and Figure 2 we can see how the tree accommodates to loose the node 40.

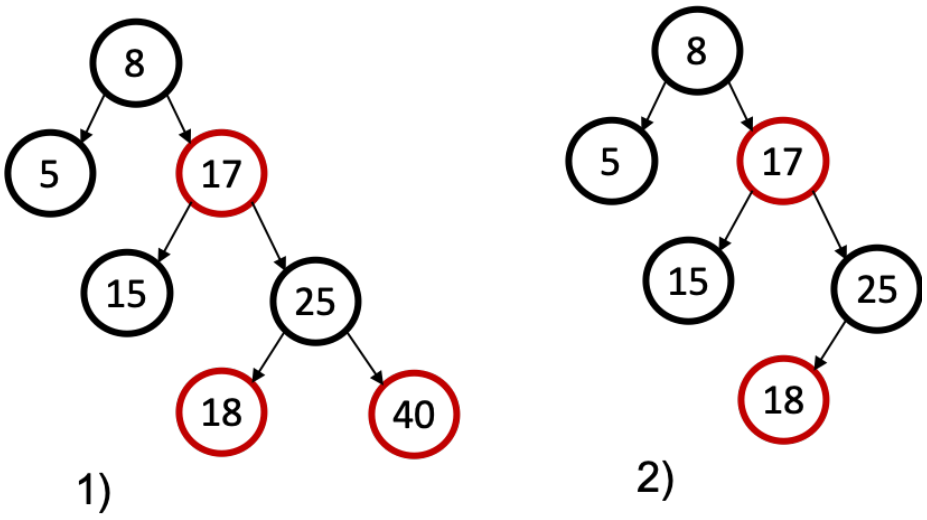


Figure 2: Example of a deletion in a red – black tree.

5 Conclusions

Again with this practice, it wasn't until I made an example by hand that I fully understood the problem of amortization. In this case, I had to re read the red – black trees, and I found that the time complexity for this algorithm is:

- Insert – $\mathcal{O}(\log\ n)$
- Delete – $\mathcal{O}(\log\ n)$
- Search – $\mathcal{O}(\log\ n)$
- Rotate right – $\mathcal{O}(1)$
- Rotate left – $\mathcal{O}(1)$

And as I understood, the purpose of the amortized cost is to be sort of like a upper bound, to have a more accurate prediction of the performance the algorithm may have. In this case, and after reading about the Accounting method, it sounded like I have to give a bigger cost to the "cheaper" operations, so that I have like saved credit for the more expensive ones. In my case, rotate left and right are the cheapest ones, and they also happen quite often, because they are used in the insertion and delete operations.

According to Figure 3 the next best notation is $\mathcal{O}(\log\ n)$.

At this point I don't know if I am understanding it correctly, but my amortized costs are:

- Insert – $\mathcal{O}(\log\ n)$
- Delete – $\mathcal{O}(\log\ n)$
- Search – $\mathcal{O}(\log\ n)$
- Rotate right – $\mathcal{O}(\log\ n)$
- Rotate left – $\mathcal{O}(\log\ n)$

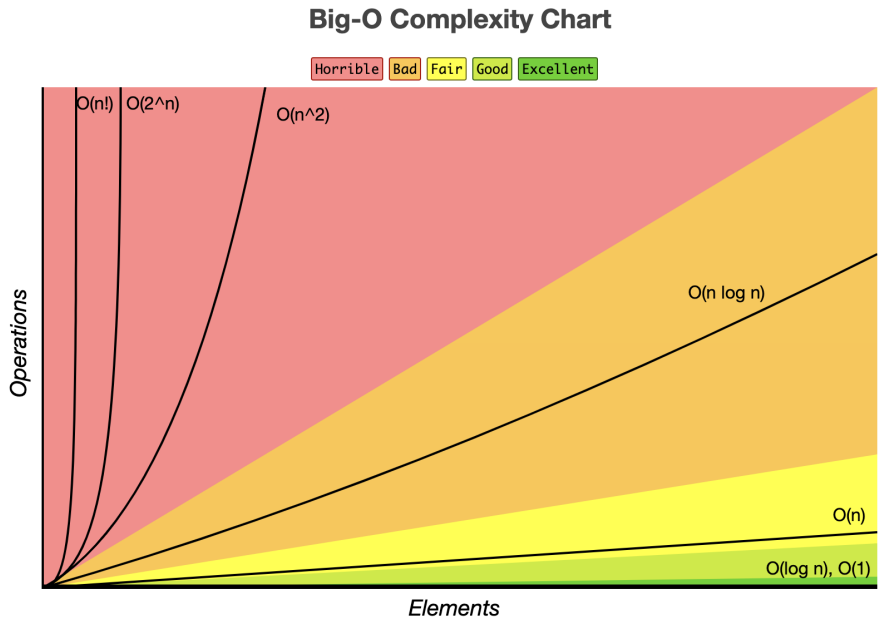


Figure 3: Big-O Complexity Chart from https://www.bigocheatsheet.com/.

References

[Ashish Bahendwar et al., 2018] Ashish Bahendwar, I., Bhardwaj, R., and Mundada, S. (2018). Amortized complexity analysis for red-black trees and splay trees. *Isha Ashish Bahendwar, RuchitPurshottam Bhardwaj, Prof. SG Mundada (2018) Amortized Complexity Analysis for Red-Black Trees and Splay Trees IJIRCST*, 6.

[CIS, 2011] CIS, C. U. (2011). Lecture 20: Amortized analysis. Available: "http://www.cs.cornell.edu/courses/cs3110/2011sp/Lectures/lec20-amortized/amortized.htm".

[Demaine, 2016] Demaine, E. (2016). Amortization: Amortized analysis. Available: "https://www.youtube.com/watch?v=3MpzavN3Mco&t=237s".

[Rhodes, 2020] Rhodes, N. (2020). Amortized analysis: Bankers method. Available "https://www.coursera.org/lecture/data-structures/amortized-analysis-bankers-method-X6a5I".

Practice 15. Branch and bound for the Travel Salesman Problem

Mayra Cristina Berrones Reyes

May 7, 2020

1 Introduction

The traveling salesman problem (TSP) consists of a set of nodes that represent "cities", and and joining them there are edges with a cost or "distance" that we need to pay to visit all the cities. In this case, we are trying to find the shortest or cheapest way of "visiting all the cities". The ordering is called tour or circuit.

This type of problem is similar to the Hamiltonian cycle. Thanks to a previous practice [1] we can say that the TSP problem is then $\mathcal{NP}-$complete.

This problem is very old, and can be known by different names. There are also several ways to solve this problem, but in this practice we are going to be solving it by the branch and bound method. This consists in a exhaustive search for the best solution in a set. Each branching step reduces the search space, in order to make it easier than the original [Applegate et al., 2006].

2 Solving TSP with branch and bound

In this section we are getting an example with four nodes. We can see the distance matrix in Equation 1. With a code in Python [2] we generate a random matrix, and then we solve it step by step using the branch and bound method, in specific, the best bound first search strategy which rules to the subset with the smaller lower bound [Smith, 1979].

[1] https://github.com/mayraberrones94/Analisis_Algoritmos/blob/master/Practica_5/Practica5.pdf

[2] https://github.com/mayraberrones94/Analisis_Algoritmos/blob/master/Practica_15/tsp.py

$$\begin{bmatrix} 0 & 31 & 44 & 12 \\ 6 & 0 & 46 & 44 \\ 26 & 4 & 0 & 38 \\ 24 & 16 & 10 & 0 \end{bmatrix} \tag{1}$$

$$\begin{bmatrix} \infty & 31 & 44 & 12 \\ 6 & \infty & 46 & 44 \\ 26 & 4 & \infty & 38 \\ 24 & 16 & 10 & \infty \end{bmatrix} \begin{matrix} 12 \\ 6 \\ 4 \\ 10 \end{matrix} = 32$$

In the above matrix we can see that we changed the 0 to ∞, because in this case, the 0 mean something different. The first step is to see which number is the smallest per row. In this matrix they are colored red. The sum of all this numbers is the value of our first reduced matrix. Next we subtract the smaller number to all the elements of the row, and we end up with a matrix like the one seen below, and we will call it our reduced matrix.

$$\begin{bmatrix} \infty & 19 & 32 & 0 \\ 0 & \infty & 40 & 38 \\ 22 & 0 & \infty & 34 \\ 14 & 6 & 0 & \infty \end{bmatrix}$$

To begin our search, first we name node 1 as our root. The reduced cost (r) of this node is going to be the sum we made in the reduced matrix, in this case is 32. Now we take the reduced matrix and begin our search with all the nodes that are adjacent to the node 1. The rules to this are:

1. We turn all the row in our reduced matrix of our parent node to ∞.
2. Then we also turn all our column of the child node we are going to explore to ∞
3. If the node we are exploring is not the last node, we also turn to ∞ the intersection number of the explored node and the root node.
4. We search first by row, then by column for the smaller number. If its zero the row/column stays the same. If not, we subtract it form the row/column and we make a sum of all of this elements.
5. The sum of all this smaller numbers in rows and columns will take the name of r_i.

The formula we are going to be using to get the cost of each node is in Equation 2:

$$c(nodeP, nodeC) + r + r_i = \text{Cost of the node} \tag{2}$$

So now that we know the rules, we begin to explore the nodes that are adjacent to the node parent. We begin with $c(1,2)$:

$$\begin{bmatrix} \infty & 19 & 32 & 0 \\ 0 & \infty & 40 & 38 \\ 22 & 0 & \infty & 34 \\ 14 & 6 & 0 & \infty \end{bmatrix} \rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 40 & 38 \\ 22 & \infty & \infty & 34 \\ 14 & \infty & 0 & \infty \end{bmatrix} \begin{matrix} 0 \\ 38 \\ 22 \\ 0 \end{matrix} = 60 \rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 2 & 0 \\ 0 & \infty & \infty & 12 \\ 14 & \infty & 0 & \infty \end{bmatrix}$$

$$c(1,2) + r + r_i = 19 + 32 + 60 = 111 \tag{3}$$

The result for the first exploration is a cost of 111. So now we move to the right to the next node $c(1,3)$

$$\begin{bmatrix} \infty & 19 & 32 & 0 \\ 0 & \infty & 40 & 38 \\ 22 & 0 & \infty & 34 \\ 14 & 6 & 0 & \infty \end{bmatrix} \rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 38 \\ \infty & 0 & \infty & 34 \\ 14 & 6 & \infty & \infty \end{bmatrix} \begin{matrix} 0 \\ 0 \\ 0 \\ 6 \end{matrix} \rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 38 \\ \infty & 0 & \infty & 34 \\ 8 & 0 & \infty & \infty \end{bmatrix} \rightarrow$$

$$\begin{bmatrix} 0 & 0 & 0 & 34 \end{bmatrix} \rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 2 \\ \infty & 0 & \infty & 0 \\ 8 & 0 & \infty & \infty \end{bmatrix} \rightarrow 6 + 34 = 40$$

$$c(1,3) + r + r_i = 32 + 32 + 40 = 104 \tag{4}$$

For the next exploration we have a cost of 104, so lastly we move to the right again to the last node adjacent to 1, $c(1,4)$.

$$\begin{bmatrix} \infty & 19 & 32 & 0 \\ 0 & \infty & 40 & 38 \\ 22 & 0 & \infty & 34 \\ 14 & 6 & 0 & \infty \end{bmatrix} \rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & 40 & \infty \\ 22 & 0 & \infty & \infty \\ \infty & 6 & 0 & \infty \end{bmatrix} \begin{matrix} 0 \\ 0 \\ 0 \\ 0 \end{matrix} \rightarrow = 0$$

$$c(1,4) + r + r_i = 0 + 32 + 0 = 32 \tag{5}$$

Finally, in this last exploration we can see the best cost to the adjacent nodes of one, which is 32 and is the relation of $c(1,4)$. The strategy of best bound first search is depicted in Figure 1, where we deprecate the other nodes adjacent to node 1, and we continue to explore the node with the lower bound of cost.

Now that we have to explore the adjacent nodes of the node 4, our reduced matrix changes to the one we made for the exploration of $c(1,4)$. The remaining nodes we have to explore are node 2 and 3. We begin with $c(4,2)$.

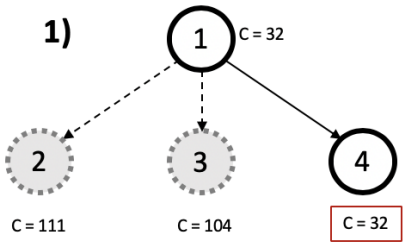


Figure 1: Exploration of nodes adjacent to the root node.

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & 40 & \infty \\ 22 & 0 & \infty & \infty \\ \infty & 6 & 0 & \infty \end{bmatrix} \rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 40 & \infty \\ 22 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix} \begin{matrix} 0 \\ 40 \\ 20 \\ 0 \end{matrix} \rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty \\ 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix} = 60$$

$$c(4,2) + r + r_i = 6 + 32 + 60 = 98 \tag{6}$$

The cost for this exploration is 98. So now we move to the next node, $c(4,3)$.

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & 40 & \infty \\ 22 & 0 & \infty & \infty \\ \infty & 6 & 0 & \infty \end{bmatrix} \rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix} \begin{matrix} 0 \\ 0 \\ 0 \\ 0 \end{matrix} = 0$$

$$c(4,3) + r + r_i = 0 + 32 + 0 = 32 \tag{7}$$

Now we find that the cost of the exploration of $c(4,3)$ is the best lower bound, so we take this node, and repeat the same steps we did when we changed parent node. We take the reduced matrix of node 3, and explore the last node we have available, $c(3,2)$.

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix} \rightarrow \begin{matrix} 0 \\ 0 \\ 0 \\ 0 \end{matrix} = 0$$

$$c(3,2) + r + r_i = 0 + 32 + 0 = 32 \tag{8}$$

We easily find the lower bound here, because most of our matrix is already filled with ∞. In Figure 2 we see the rest of the exploration, where we deprecate the node 2 when node 4 was the parent, and the final path taken with this

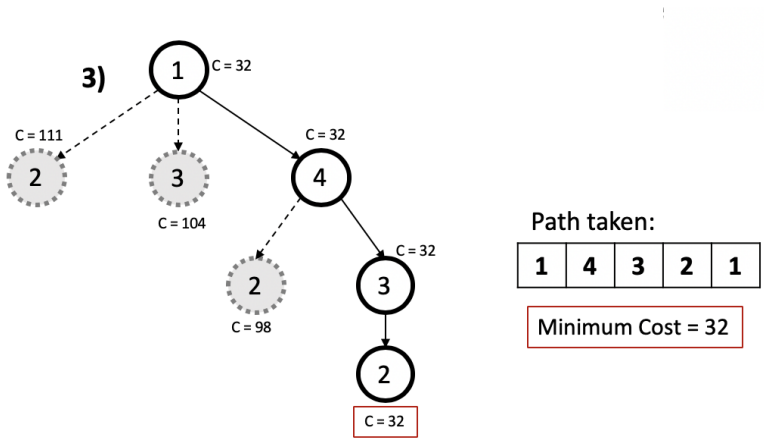


Figure 2: Diagram of divide and conquer steps.

algorithm. At the end we find that the lowest bound was 32.

Finally in Figure 3 we have the output of the terminal, and we can see that the results match.

```
[Path Taken :  1 3 2 7 5 4 6 1 (base) Mayras-MacBook-Pro:Elisa mayraberrones$ pyt
hon3 tsp.py
 [[27 31 44 12]
  [ 6  6 46 44]
  [26  4 17 38]
  [24 16 10 42]]

[[ 0 31 44 12]
  [ 6  0 46 44]
  [26  4  0 38]
  [24 16 10  0]]

Minimum cost : 32
Path Taken :  1 4 3 2 1 (base) Mayras-MacBook-Pro:Elisa mayraberrones$
```

Figure 3: Terminal with the results of the branch and bound of the TSP.

3 Conclusions

For the time complexity I could not find much. We know that TSP is $\mathcal{NP}-$ complete, but one of the books [Smith, 1979] mentions that the complexity depends mostly on the size of the problem so I made a little experiment with diferent size of matrix, starting with the experiment shown in this practice of four nodes, up to 25 nodes. We can see in Figure 4 that the time grows exponentially around the 17 and 20 node, so we can confirm that this type of problem is feasible to

solve this way on small instances.

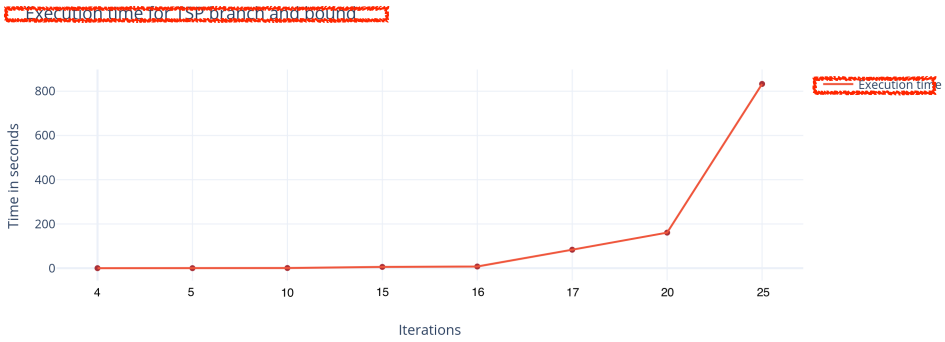


Figure 4: Execution time for branch and bound for TSP.

References

[Applegate et al., 2006] Applegate, D. L., Bixby, R. E., Chvatal, V., and Cook, W. J. (2006). *The traveling salesman problem: a computational study*. Princeton university press.

[Smith, 1979] Smith, D. R. (1979). *On the computational complexity of branch and bound search strategies*. Naval post graduate school, Monterrey California.

6 pts; retroalimentación por videoconferencia

Practice 16. Approximation algorithms

Mayra Cristina Berrones Reyes

May 14, 2020

1 Introduction

In the subject of optimization, most problems presented are $\mathcal{NP}$-hard, thus have no polynomial algorithms to find their solutions that we know of. Approximation algorithms are an efficient way to find approximate solutions to this type of problems.

An α-approximations a polynomial time algorithm that for all instances of the problem it produces a solution, whose value is within the factor of α and the value of the optimal solution. The factor α is called the approximation ratio Williamson and Shmoys, 2011.

2 A 2-Approximation Algorithm for TSP

In other practices we establish that it is $\mathcal{NP}$-complete to decide if a given graph $G = (V, E)$ has a Hamiltonian cycle. An approximation algorithm for TSP can be used to solve a Hamiltonian cycle, since both problems are very similar Schaeffer, 2020.

The input for this algorithm is a distance matrix of $n \times n$. The numbers inside the matrix represent d where $d_{i,j}$ denotes the cost of travelling from city i to city j. This matrix is defined by the following features Biswas, 2015:

1. $d_{i,j} \geq 0$ for all $i, j \in \{1, \ldots, n\}$
2. $d_{i,j} = d_{j,i}$ for all $i, j \in \{1, \ldots, n\}$
3. $d_{i,k} \leq d_{i,j} + d_{j,k}$ for all $i, j, k \in \{1, \ldots, n\}$
4. $d_{i,i} = 0$ for all $i \in \{1, \ldots, n\}$

The number one feature is the non negative rule of the TSP. The second rule aplies to the symmetric form of the matrix, which means it is the same distance from node i to j as is from j to i. Then we have the triangle inequality feature.

This basically states that the distance between node i and k is less or equal to the distance between node i and j, plus the distance between j and k. This can be seen in Figure 1

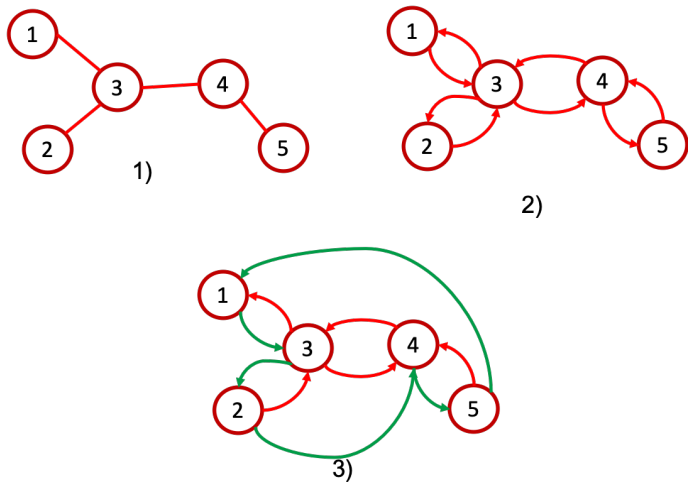


Figure 1: Example of the 2-approximation algorithm. In step one we see our MST. In step two we transverse our tree and form sequence S_1. Finally step three we have our shortcuts made and form sequence S_2.

The Triangle-Inequality holds in many practical situations. When the cost function satisfies the triangle inequality, we can design an approximate algorithm for TSP that returns a tour whose cost is never more than twice the cost of an optimal tour [Cormen et al., 2009].

Now, knowing this features, we can begin to compute our path. We can think of d as a complete weighted graph G whose vertex set is $V = \{1, \ldots, n\}$ and for which the weight of the edge ij is $d_{i,j}$. So now we first need to find the minimum spanning tree (MST) of this graph. The MST can be found in $\mathcal{O}(n^2)$, being n the number of vertex. There are several algorithms to find this MST, for example, the Prim's algorithm.

The steps to build a MST are as follows:

1. Choose a starting node or root node for the starting and ending point of the salesman.
2. Construct a MST from that root node.
3. List the vertices visited in preorder walk of the constructed MST and add the root node at the end.

In the example on Figure 1 we can see the diagram of a given graph, and then we show the MST constructed taking the node 1 as root. The pre order transversal of this example is

$$S_1 = 1, 3, 2, 3, 4, 5, 4, 3, 1 .$$

For transversing the tree, in this example, we obtain the sequence

$$S_1 = 1, 3, 2, 3, 4, 5, 4, 3, 1 .$$

Now, remove all but the first occurrence of each vertex in his sequence (except 1):

$$S_2 = 1, 3, 2, 4, 5, 1 .$$

Note that S_2 is a permutation of $\{1, \ldots, n\}$ since S_1 contains every element of $\{1, \ldots, n\}$ at least once and therefore S_2 contains every element exactly once. Therefore, S_2 is a valid solution for the TSP. So we have T that will represent the MST which is the connected subgraph of G of minimum weight.

Each edge of T is used exactly twice S_1, so

$$w(S_1) = 2w(T) .$$

By the triangle inequality

$$w(S_2) \leq w(S_1) .$$

Remember that T is a minimum spanning tree; it's a minimum weight connected subgraph of G. But a travelling salesman tour is also a connected subgraph of G, so

$$w(T) \leq w(C^\star) .$$

where $C^\star$ is any optimal travelling salesman tour. Putting all these together, we have

$$w(S_2) \leq w(S_1) \leq 2w(T) \leq 2w(C^\star) .$$

We have just found a solution, S_2, to the TSP whose weight is at most twice of the optimal solution.

3 Christofides algorithm for TSP

The strategy for this algorithm is to construct an Eulerian tour whose cost is at most α, the short cut it to get an α-approximation solution. A connected graph is Eulerian if and only if every vertex has even degree. The steps to carry out this algorithm are as follows:

1. Find minimum spanning tree (T).
2. Let O be a set of nodes with odd degree in T. Find a minimum cost perfect matching on O.

3. Add the set of edges of the perfect matching to T and find an Eulerian tour.

4. Shortcut the Eulerian tour.

This means that we have found a solution to the traveling salesman problem whose cost is

$$w(T) + w(M)$$

We already know that $w(T) \leq w(C^\star)$. For M, we rely on an algorithm for the minimum weight perfect matching problem, which runs in $O(n^3)$ time. Still, we need to relate $w(M)$ to $w(C^\star)$.

Finally, by triangle inequality, shortcutting previously visited vertices does not increase the cost $w(C) \leq w(C^\star)$. What's more, C can be partitioned into two perfect matching by alternately coloring its edges red and blue. Since the total cost of the two matchings is at most $w(C)$, the cheaper one has cost at most $1/2$ of $w(C)$. In other words, one of these two matchings has weight at most $w(C)/2$

$$w(M) \leq w(C)/2 \leq w(C^\star)/2.$$

So in the end we found a solution that has a weight of

$$w(T) + w(M) \leq w(C^\star) + w(C^\star)/2 = \frac{3}{2} w(C^\star)$$

In Figure 2 we see an example of the Christofides algorithm.

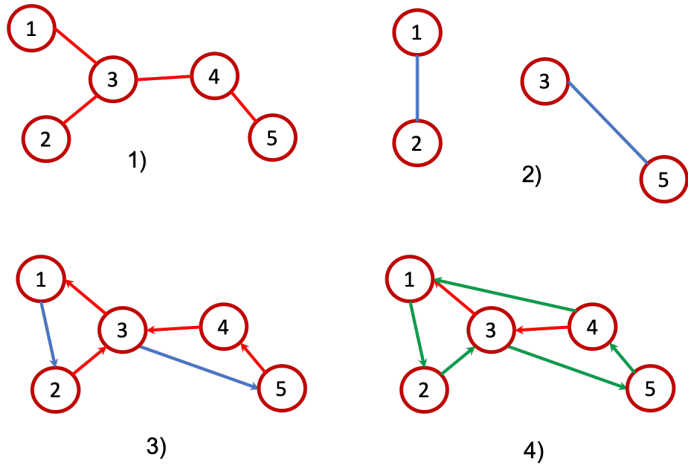


Figure 2: Example of the Christofides algorithm. In step one we have our MST. In step 2, we do or two matching in color blue. In step three we add the other vertices, and finally in step four we see our tour.

4 Conclusions

In several other practices where we realize the difficulty or the computational time it takes to solve some algorithms, it is really interesting to see the ways that exist to approach the $\mathcal{NP}$-hard problems. It seems relatively more easy to understand this subject after practices with similar subjects such as reductions and other complexity problems we made along some other practices.

References

[Biswas, 2015] Biswas, A. S. (2015). R9.approximation algorithms: Traveling salesman problem.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.

[Schaeffer, 2020] Schaeffer, E. (2020). Complejidad computacional de problemas y el analisis y diseño de algoritmos. Available "https://elisa.dyndns-web.com/teaching/aa/pdf/aa.pdf".

[Williamson and Shmoys, 2011] Williamson, D. P. and Shmoys, D. B. (2011). *The design of approximation algorithms*. Cambridge university press.

5 PTS; retroalimentación por videoconferencia

Practice 17. Randomized algorithms

Mayra Cristina Berrones Reyes

May 22, 2020

1 Introduction

A randomized algorithm uses random numbers to decide what to do next. Some of these algorithms have a deterministic time complexity, and can be classified in two categories.

1.1 Las Vegas algorithm

Las Vegas algorithm is a randomized algorithm that always gives the correct result but gambles with resources. However, the runtime of a Las Vegas algorithm differs depending on the input. These features makes these algorithms more suitable for cases where the number of possible solutions is limited. For example, Randomized QuickSort always sorts an input array and expected worst case time complexity of QuickSort is $\mathcal{O}(n \log n)$ Sharma, 2015.

1.2 Monte Carlo algorithm

Monte Carlo algorithms use repeated random sampling to obtain numerical results.They are typically used to simulate the behaviour of other systems.The output may be incorrect with a certain, typically small, probability. It a big amount of sampling methods that are so complex they are usually performed with the aid of a computer.

2 Approximate value of π using Monte Carlo

In this case we are using the Monte Carlo algorithm to make an estimate of the π number. As we know π is a name given to the ratio of the circumference of a circle to the diameter. To be able to estimate π using the Monte Carlo algorithm, we start by generating a large number of random points and see how many fall in the circle enclosed by the unit square.

We know that the area of the square is 1 unit square, and the circle is

$$\pi * \frac{1}{2}^2 = \frac{\pi}{4}.$$

The way it works then, is if points (x, y), with $-1 < x < 1$ and $-1 < y < 1$, are placed randomly, the ratio of points that fall within the unit circle will be close to $\pi/4$. A Monte Carlo simulation would randomly place points in the square and use the percentage of points inside the circle to estimate the value of π. In randomized and simulation algorithms like Monte Carlo, the more the number of iterations, the more accurate the result is [to Go, 2020].

In Figure 1 we can see the different results in approximation to the π number with 100, 1,000, 10,000 and 100,000 samples.

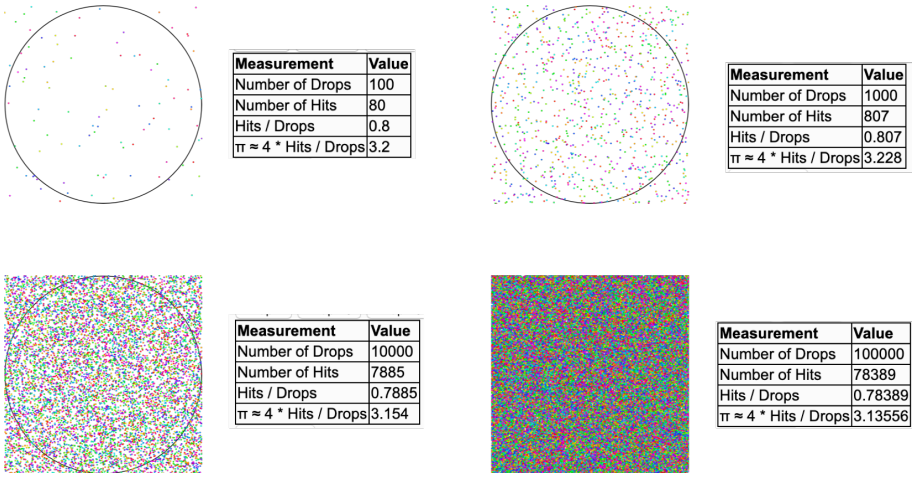


Figure 1: Example of π approximations with different number of sampling. This simulator can be found in [Nick Exner, 2014] **falta punto**

Now, to be able to calculate the error (ε) of each experimentation we can evaluate the integral

$$\int f dV$$

where f is a general function and the domain of integration is of arbitrary dimension. We then take the approximation integral for Monte Carlo to be

$$\int f dV = \frac{1}{N} \sum_{i=1,N} f(x_i) + O\left(\frac{1}{\sqrt{N}}\right)$$

where N represents the randomly scattered point through the integration domain, calcultating f at each point. Let x_i denote the ith point. The integration error is associated with the Monte-Carlo method as a function of the number of points, N. It can be seen that there is very little change in the rate at which the error falls off with increasing N as the dimensionality of the integral varies[1].

So giving an error to the sampling of 100 it is roughly a $\varepsilon = 0.01$ Comparing that to the error of the 100,000 sampling we have $\varepsilon = .000000031$ which proves this assumption correct.

3 Conclusions

This experiment is commonly used to prove the uses of the Monte Carlo algorithm. But the way it is constructed and thanks to the visualization found on this investigation got me curious about how we can implement this problem to the set covering problem, using the same idea of randomly placing points inside a plane, and seeing how much space I can cover like that.

References

[Fitzpatrick, 2006] Fitzpatrick, R. (2006). Monte-carlo integration. Available "http://farside.ph.utexas.edu/teaching/329/lectures/node109.html".

[Nick Exner, 2014] Nick Exner, E. R. (2014). Estimation of pi - mste. Available "https://mste.illinois.edu/activity/estpi/".

[Sharma, 2015] Sharma, A. (2015). Randomized algorithms. set 2 (classification and applications). Available "https://www.geeksforgeeks.org/randomized-algorithms-set-2-classification-and-applications/".

[to Go, 2020] to Go, A. (2020). Las vegas vs. monte carlo algorithms. Available "https://yourbasic.org/algorithms/las-vegas/".

[1] For more information on how the integral of Monte Carlo got there, see Fitzpatrick, 2006

Retroalimentación por videoconferencia;
6 pts por redacción y discusión

Extra: Discussion of phase transitions.

Mayra Cristina Berrones Reyes

May 29, 2020

1 Discussion

For this discussion, we read some publications and articles that explained the phase transition problem. The most basic form that we understood, was the notion that a phase transition is the one that happens once a certain feature of the problem changes and it reaches what we know as a critical point. A widely used example is the one of the water temperature. In this case, the temperature is going to change little by little, and the critical point in this experiment will happen when the water boils and turns into gas, or solidifies because is too cold.

Since most of the examples and common phase transition examples we found were about physics, magnetic examples and such, we had a hard time placing optimization problems in this type of solutions. In a conference transcript of a phase transitions workshop [Zhang, 2002] we found a graphic that helped us understand a little bit better this concept. First they start with a complexity analysis on different types of problems, such as a tree search. In Figure 1 we see the easy-hard transition in optimal tree search.

In this figure we have the order parameter as:

- b is the mean branching factor.
- p is the probability that an edge has a cost 0.
- bp is the expected number of children having the same cost as the parent.

So in this case the complexity is determined by bp.

In this same conference transcript we found another example of a TSP problem, similar to the one described in the documentation for this problem [Schaeffer, 2020]. In this case the control parameter changes. In Figure 2 we can see the comparison they made. They used as the changing parameter the number of the distance, and the different iterations are from different size of cities they have to travel to.

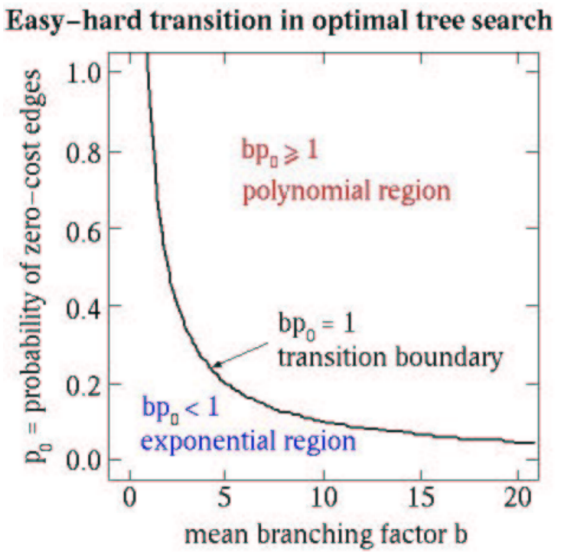


Figure 1: Easy-hard transition in optimal tree search.

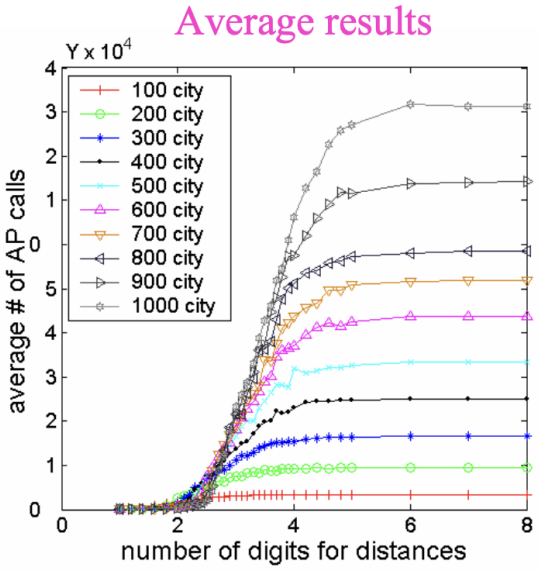


Figure 2: Average results for the TSP with the changing parameter of distance between cities.

After this images, the practice 15 came to mind, because we found a similar problem than this. In that practice we made some experimentation with the number of cities we had to visit using a method of branch and bound to find the solution. In Figure 3 we can see how after 10 cities the complexity starts to rise and by the 25 iteration the complexity becomes exponential.

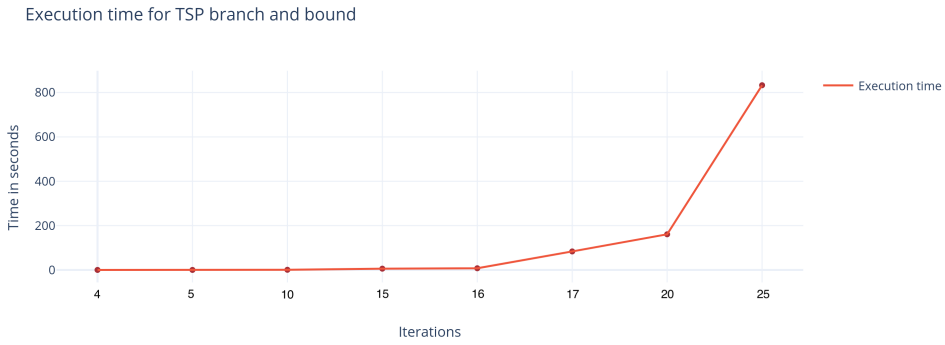


Figure 3: Execution time for branch and bound for TSP.

To solve this kind of problems we found various explanations with the Ising Model. The simplified way they explained is that, we suppose we have what they called spins. These spins have a direction of up and down. Now what if one of the spins gets flipped at a random position because of a change in the changing parameter, which can cause the neighboring spins to flip in the same direction. And this process will go on and on, and completely ordered state will not remain stable [Singh, 2020].

This was easy to understand with magnetic forces or the temperature example, but we failed to connect it to the problems we first described as to how or when this type of critical points are going to be reached, other than trial and error, as we tried to do in Figure 3.

References

[Schaeffer, 2020] Schaeffer, E. (2020). Complejidad computacional de problemas y el analisis y diseño de algoritmos. Available "https://elisa.dyndns-web.com/teaching/aa/pdf/aa.pdf".

[Singh, 2020] Singh, S. P. (2020). The ising model: Brief introduction and its application. DOI: 10.5772/intechopen.90875.

[Zhang, 2002] Zhang, W. (2002). Phase transitions, backbones, measurement precision, and phase-inspired approximation. Available "http://www.ipam.ucla.edu/abstract/?tid=1654&pcode=PTAC2002".