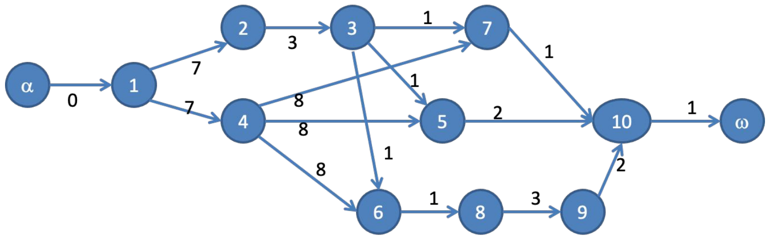
12/04/2022

Graph Theory Project : Scheduling Graphs



Tutor : [redacted]

L3 INT2 Group 1

[redacted]

Pablo SANCHEZ

Introduction
What is scheduling

Scheduling graphs are a subpart of graphs that represents a list of tasks to be made, and the order in which some have to be executed.

It can have several applications in the real world.

We have coded a program that can, from a simple constraint table as an input, compute the corresponding scheduling table.

Here is what we did :

SUMMARY

1. Project Description
2. Program construction
3. Difficulties encountered and what we did well
4. Conclusion

Project description

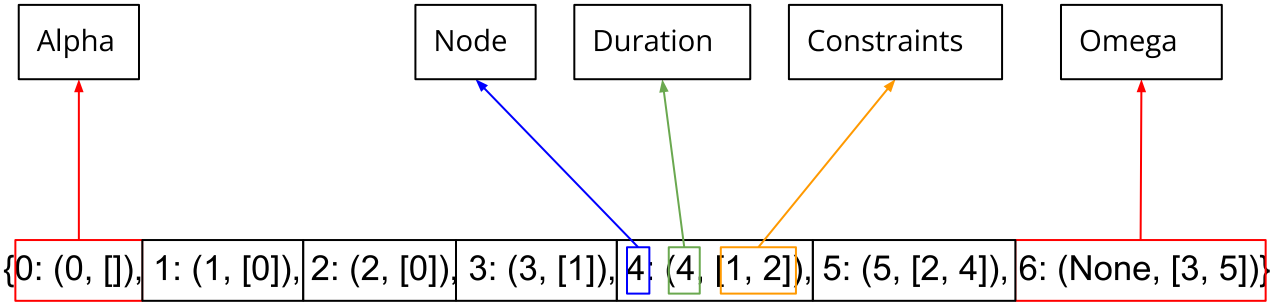
The aim of the project was to create a program that would, from tables given as inputs in txt files and that would represent scheduling graphs, compute their earliest and latest dates calendar as well as the floats.

We divided this project into 3 parts:

- Reading the table and saving it into memory, while adding alpha and omega.
- Checking if the graph represented by the constraint table is a scheduling graph.
- Computing the earliest and latest dates calendar and the floats.

Project construction

This is how the data are stored in the memory:




```
1 1
2 2
3 3 1
4 4 1 2
5 5 2 4
```

Program construction

Reading from file, adding alpha and omega - 1

```python
# This function gets the full constraint table.
def get_constraint_table(file):

    # First we load the table from the memory ...
    constraint_table = read_from_file(file)

    # ... then we compute alpha and omega and we return the table.
    add_alpha(constraint_table)
    add_omega(constraint_table)
    return constraint_table
```

```python
# This function reads the file and saves it in the memory.
def read_from_file(file):
    constraint_table = {}
    for line in file:

        # For each line, we need to: remove the space, remove the \n,
        # turn the strings into ints, and put the whole thing in an array.
        # The following line does it all:
        values = list(map(lambda x: int(x), line.rstrip().split(' ')))

        # Then, we save the values in our dictionnary following this pattern:
        # {Key: (value, [constraints])}
        constraint_table[values[0]] = (values[1], values[2:])
    return constraint_table
```

For each line, we separate each character and put them in a list.

We add the first value as the key, the 2nd value as the duration, and all the other ones as constraints.

Program construction

Reading from file, adding alpha and omega - 2

```python
# This function adds an alpha to the table read from memory.
def add_alpha(constraint_table):

    # The duration of alpha is 0.
    constraint_table[0] = (0, [])
    for i in constraint_table:

        # If the constraint array of a key is empty
        # (and if the key is not 0 => Not alpha): we add 0 in the table
        if len(constraint_table[i][1]) == 0 and i != 0:
            constraint_table[i][1].append(0)
```

We initialize alpha: duration of 0 and no constraints.

We initialize alpha: duration of 0 and no constraints.

```python
# This function adds an omega to the table read from memory.
def add_omega(constraint_table):
    omega = len(constraint_table)

    # The duration of omega is null.
    constraint_table[omega] = (None, [])

    # We go through the constraint table ...
    for i in constraint_table:

        # ... and we add in the constraints of omega the elements that do not exist as constraints for other nodes.
        if i != omega and not verify_existence_dico(i, constraint_table):
            constraint_table[omega][1].append(i)
```

We set Omega to have no duration.

We simply add to the constraints of omega any node that isn't pointing to any other node.

Program construction

Checking if scheduling graph - 1

A graph is a scheduling graph if the following conditions are respected. In bold, the conditions that do NOT depend on the initial table but only on our code. Those were hard coded and thus do not need to be checked, as it would be a waste of time and resources.

- **A single entry point**
- **A single exit point**
- No cycle
- **Same weights for all outgoing edges of a vertex**
- **Outgoing edges of the entry vertex have zero**
- No negative edges

Program construction

Checking if scheduling graph - 2

- Single entry point and all edges going from alpha to 0

```
# This function adds an alpha to the table read from mem
def add_alpha(constraint_table):

    # The duration of alpha is 0.
    constraint_table[0] = (0, [])
    for i in constraint_table:

        # If the constraint array of a key is empty (and
        if len(constraint_table[i][1]) == 0 and i != 0:
            constraint_table[i][1].append(0)
```

- Single output point Omega

```
# This function adds an omega to the table read from memory.
def add_omega(constraint_table):
    omega = len(constraint_table)

    # The duration of omega is null.
    constraint_table[omega] = (None, [])

    # We go through the constraint table ...
    for i in constraint_table:

        # ... and we add in the constraints of omega the elements that do
        if i != omega and not verify_existence_dico(i, constraint_table):
            constraint_table[omega][1].append(i)
```

Program construction

Checking if scheduling graph - 5

No cycle: looping until the graph is totally deleted, and looking for new entries.

```
def get_entries(dico, deleted):
    entries = []
    for i in dico:
        # if len(set(dico([i][1])).symmetric_difference(set(deleted)) == []):
        if i not in deleted and (list_differences(dico[i][1], deleted)) == []:
            entries.append(i)
    return entries
```

Returns all the elements that are in given list but not in the second one

```
#to check if our table represents a scheduling graph : return 1 if its the case, 0 otherwise
def check_scheduling_graph(dico):
```

```
if len(entries) == 0:
    print("There are no more entry points while there are still vertices.\n-> There is a cycle")
    return 0
```

Program construction

Checking if scheduling graph - 5

No negative values: looping in the table in search of a negative value.

We return 0, the scheduling algorithms won't happen if the condition is not respected.

```
### 2nd CHECK : check there are no negative-weight edges
# We want the number of negative weight to be 0. Otherwise, we return 0
if negative_weight_cpt(dico)==0:
    print("\nThere are no negative-weight edges")
else :
    print("\nSome edges have negative-weight")
    return 0
```

Program construction

Computing the schedule table - 1 - Ranks

```python
# This functions gets the ranks of the graph. They are stored in a double list.
def get_ranks():
    ranks = []
    count = 0
    i = 0

    # We continue as long as the count (of the elements added in the ranks list) is smaller than the number of nodes.
    while len(constraint_table) > count:
        ranks.append([])

        # We need to create a deep copy of the rank list so that an element that is added to a rank
        # does not influe on the other elements that need to be in the same rank.
        current_ranks = copy.deepcopy(ranks)

        # We loop through the constraint table ...
        for j in constraint_table:

            # ... And we verify each time if the element verifies 2 conditions:
            # 1) It is not already in the rank list (with verify_existence() )
            # 2) All its predecessors are already in the rank list (with to_add() )
            if ((not verify_existence(j, ranks)) and (to_add(j, current_ranks))):
                ranks[i].append(j)

                # After appending the element that verified the 2 conditions to the list, we update the counter.
                count += 1
        i += 1
    return ranks
```

- Ranks correspond to the distance to alpha in terms of nodes

- Similar to the method we used for the cycle verification → we verify that all the predecessors are already in ranks

Program construction

Computing the schedule table - 2 - Order

```python
# This function returns a list of the elements in the order of which they need to be computed by the early dates and late dates algorithms.
def get_order(ranks, index):
    nodes = []

    # We loop through the ranks nested list and add the elements in a new 1 dimensional list.
    for i in ranks:
        for j in i:

            # The index will be either 0 (we add the values at the beginning, creating the list in the reverse order),
            # or the max size of the list (we add the values at the end, creating the list in the right order).
            nodes.insert(index, j)
    return nodes
```

- Storing in one-dimensional array to get our tasks.
- Using insert, either our index = 0 and we will therefore be creating the list in the inverse order, or our index = max size of the graph and we will be creating the list in the right order.

Program construction

Computing the schedule table - 3 - Earliest date

```
# This function returns a dico with the earliest dates for each key.
def get_earliest_dates(nodes):
    earliest_dates = {}

    # We loop through the node_list (that was computed using the get_order function).
    for i in nodes:
        # For every value, we compute its earliest dates using get_largest().
        earliest_dates[i] = get_largest(constraint_table[i][1], earliest_dates)
    return earliest_dates
```

The earliest date takes the largest value found.

Program construction

Computing the schedule table - 4 - Earliest date

We sum the earliest date of the predecessor * (constraint) with its duration.

```python
# This function returns the earliest date for a given node x, represented by its list of predecessors.
# It is used by get_earliest_dates().
def get_largest(x, earliest_dates):

    # We set up our tuple. It represents: (duration, source). We begin with an empty one: (duration: -1, source: None).
    largest = (-1, None)

    # If we have alpha (no predecessors), we return (0, 0) that represents (duration: 0, source: alpha).
    if len(x) == 0:
        return (0, 0)

    # We loop through the predecessors.
    for i in x:

        # Each time, we verify if the computed element (earliest_dates[i][0] + constraint_table[i][0]) is larger than the previously computed ones.
        # If it is the case, it means that this newly computed element is the early date that we need to use.
        # It is going to be returned once we are sure that no others predecessors produces a bigger early date.
        if largest[0] < 0 or earliest_dates[i][0] + constraint_table[i][0] > largest[0]:

            # earliest_dates[i][0] + constraint_table[i][0] corresponds to: value of the predecessor in the early date table + its duration.
            largest = (earliest_dates[i][0] + constraint_table[i][0], i)
    return largest
```

If a new element is larger than the current one, it becomes the new largest one.
We return the last one found.

* We know that the predecessor is already on the table, because we compute the dates in the order given by the get_order algorithm.

Program construction

Computing the schedule table - 5 - Latest date

```
# This function returns a dico with the latest dates for each key.
# It is similar to get_earliest_dates()
def get_latest_dates(nodes):
    latest_dates = {}
    for i in nodes:
        latest_dates[i] = get_smallest(i, latest_dates)
    return latest_dates
```

The latest date takes the smallest value found.

Program construction

Computing the schedule table - 6 - Latest date

We subtract the own duration of the node to its successor's* latest date.

```python
# This function returns the latest date for a given node i.
# It is used by get_latest_dates().
def get_smallest(i, latest_dates):

    # x represents the list of successors of i.
    x = successor_table[i][1]
    smallest = (-1, None)

    # If we have omega (no successors), we return (duration: early date of omega, source: omega).
    if len(x) == 0:
        return (earliest_dates[i][0], i)

    # Same principle as for the early dates.
    for j in x:
        if smallest[0] < 0 or latest_dates[j][0] - successor_table[i][0] < smallest[0]:
            smallest = (latest_dates[j][0] - successor_table[i][0], j)
    return smallest
```

If a new element is smaller than the current one, it becomes the new smallest one.
We return the last found.

* We know that the successor is already on the table, because we compute the dates in the order given by the get_order algorithm.

Program construction

Computing the schedule table - 7 - Floats

```
# This function returns a dico with the flaots for each key.
def get_floats(nodes, latest_dates, earliest_dates):
    floats = {}

    # We loop through the nodes and, for each value, substract the earliest dates to the latest dates.
    for i in nodes:
        floats[i] = latest_dates[i][0] - earliest_dates[i][0]
    return floats
```

- Computing for each value the corresponding float by subtracting the earliest dates to the latest dates.
  Float = Latest dates - Earliest dates

Program construction

Computing the schedule table - 8 - Critical path

```
# This function gets the critical path.
def get_critical_path():

    # We start by omega (last node)
    Tmp = len(earliest_dates) - 1
    critical_path = [latest_dates[Tmp][1]]

    # While we didn't reach alpha...
    while Tmp != 0:

        # ... we add the longest date constraint (or the source of its earliest date) of the last added node to the critical path...
        critical_path.insert(0, earliest_dates[Tmp][1])

        # ... then this node (the source of the earliest date of the last added node) becomes the new current node (for the next node to add).
        Tmp = earliest_dates[Tmp][1]
    return critical_path
```

We start by the last task (omega) we go back to alpha through the latest dates. This will only select the tasks with a float of 0.

Difficulties Encountered

Technical difficulties

Some parts of the program took us some time to figure out how to solve them:

- Getting used to python again,
- Deciding in which form to store the graph in the memory,
- Debugging,
- Optimizing the code, for faster computation and easier reading.

Difficulties Encountered

Human difficulties

Not only was the coding sometimes tricky, we also had to deal with some human difficulties:

- Managing our schedule,
- Group team,
- Deep understanding of the lesson and its algorithms was necessary.

Conclusion

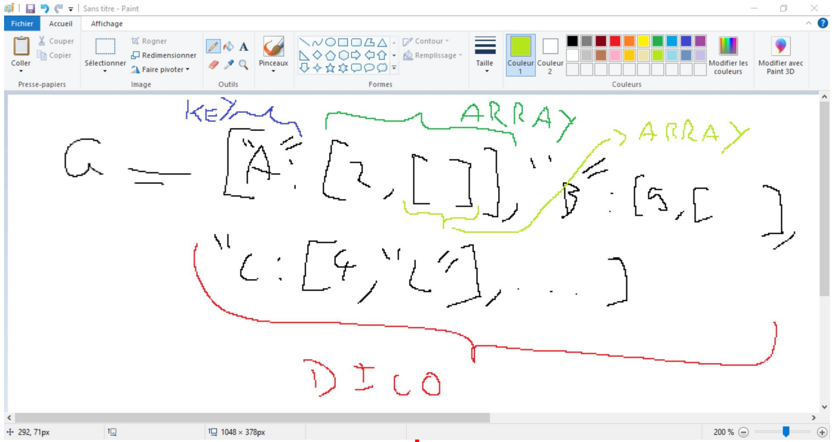
Despite the difficulties that we encountered, we managed to get through the project and overall, we are pretty proud of what we achieved.

Here are few points that we are proud of :

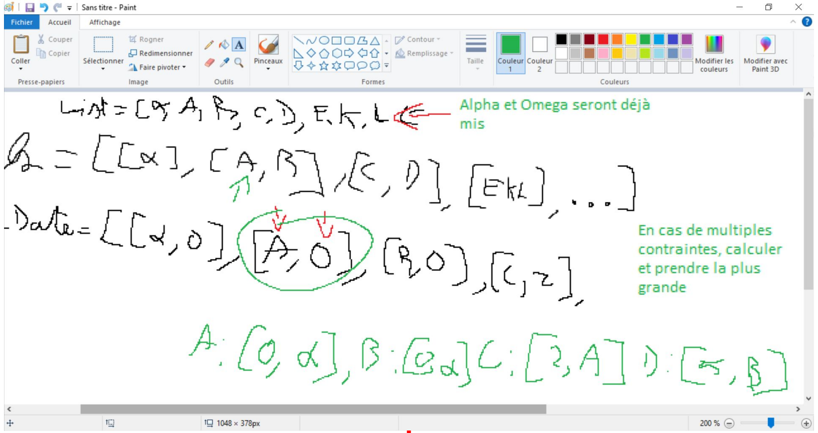
- We planned everything in our code before writing it, thanks to useful tools such as paints,
- Good communication overall thanks to Discord and Teams.
- Good scheduling, we didn’t wait the last minute to deliver the work we needed to do this project.

Conclusion

Some examples for the previsualisation of the project with paints



How we would save information from the table into memory.



how to code the ranks algorithm.