Greedy Modification of Kahn's Algorithm to Solve Circular Dependencies

Renaldy Arief Susanto - 13522022
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
Email (gmail): renaldyariefsusanto@gmail.com

***Abstract*—Circular dependendcy is a phenomenon that can occur in many different contexts. Of course, while it's desirable to prevent them from occuring in the first place, there should also be a technique to resolve them once they have been created. This paper demonstrates the use of a simple technique in which we can resolve (yet not necessarily optimize) all circular dependencies that exist within a system, given that we have defined all of the dependencies.**

Keywords—graphs; cycles; Kahn's algorithm; circular dependencies; greedy;

I. Introduction

The term circular dependency itself is often used in the context of software engineering, as seen in [1] and [2]. Examples include module imports or class dependencies. However, it can perhaps also be used as an umbrella term, which includes all conflicts that can be described as "two or more things that depend on each other to function". For instance, it can happen in scheduling schemas or prerequisites. These are also systems which require careful design, as one may accidentally create a loop of dependencies if too many constraints are defined. Here, I will define a dependency "*A* depends on *B*" as "the object *A* requires that object *B* is defined before it itself can be defined". And thus a circular dependency is defined as a situation where "a group of objects can not be defined as they are dependent of each other."

To prevent them from happening in the first place would be ideal. But what if the problem arises anyway? Perhaps it is better to redesign the system as a whole so as to create a brand new schema which is cleaner. However, if we have a large enough system, this may not be such a trivial task and designing a new algorithm to generate a better dependency chain might take time. It may also be the case that a simple hotfix is required immediately, and we are not required to refactor the system at all.

In any case, this paper demonstrates the use of a greedy technique which attempts to resolve the circular dependency (if present) in a system of dependencies, by selecting and removing as little dependencies as possible. Of course, this may not be desirable, as some dependencies may be crucial. However, the results of this algorithm will allow the user to identify what few dependencies can be modified in order to quickly resolve the issue.

II. Theoretical Basis

A. Using Graphs and Cycles to Model Dependencies

A graph consists of vertices (also often called nodes) and edges. In this paper, the variable V will denote the number of nodes in a graph, and the variable E will denote the number of edges. The vertices may be arbitrarily labeled, for instance using integers. For example, the following graph consists of 5 nodes and 7 edges:

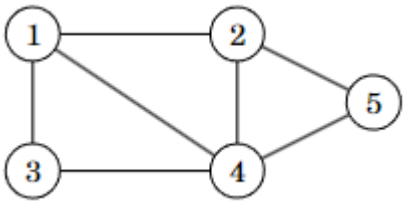

Fig. 1. Example Graph 1 (copied from [3])

A path leads from node a to node b through edges of the graph. The length of a path is the number of edges in it. For example, the above graph contains a path 1 → 3 → 4 → 5 of length 3 from node 1 to node 5:

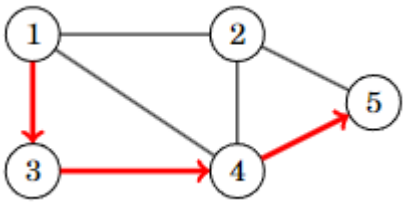

Fig. 2. Example of a path in a graph (copied from [3])

A path is a **cycle** if the first and last node is the same. For example, the above graph contains a cycle 1 →3 →4 →1.

Regarding connectivity, a graph is said to be **connected** if there is a path between any two nodes. For example, the following graph is connected.

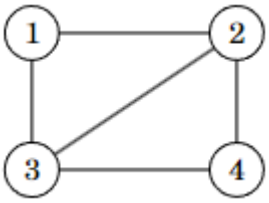

Fig. 3. Example of a connected graph (copied from [3])

The following graph is not connected, because it is not possible to get from node 4 to any other node:

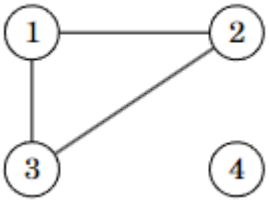

Fig. 4. Example of a disconnected graph (copied from [3])

The connected parts of a graph are called its components. For example, the following graph contains three components: {1, 2, 3}, {4, 5, 6, 7} and {8}.

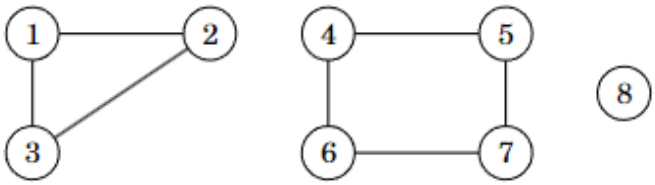

Fig. 5. Example of graph components (copied from [3])

Regarding directions, a graph may have **directed** vertices, and thus considered to be a directed graph. A directed graph is one whose edges may only be traversed in one direction. The graph below is an example. To further clarify, one may traverse from 3 → 1, but not vice versa.

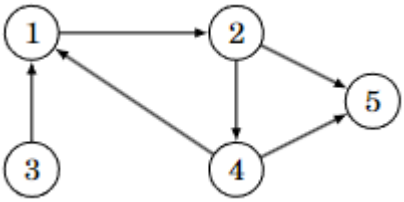

Fig. 6. Example of a directed graph (copied from [3])

We can now model a set of objects and depedencies as a directed graph: the nodes will represent the objects or events, and the vertices shall represent the dependencies. A vertex from node *a* to node *b* means that object *b* depends on object *a*. These are also referred to as **predecessors** and **successors**, respectively (though these terms are more often used for prerequisite based problems, here the meanings are analogous and thus I will be using these terms).

I now bring attention to the importance of **cycles** in this context. If we are given a dependency chain model in the form of a directed graph, a cycle in the graph implies that there exists a **circular dependency**. It is clear that if a cycle exists, we can traverse a particular node in the graph back to itself. This leads to a loop in dependencies and thus needs to be resolved.

Resolving this conflict can perhaps be done in a few ways. One way, as mentioned earlier, is to redefining the objects and dependencies such that a new system is created entirely. Here, however, a much simpler way is to delete dependencies. Again, this may not be ideal given the circumstances, but it's an option nonetheless. Hence, the problem can be reduced to: **What set of edges in the graph can we remove such that we eliminate the cycle but also remove as little as possible?**

B. Kahn's Algorithm and Topological Sorting

Now that we have defined the representation of a directed graph for a set of objects and dependencies, as well as the task of finding the smallest set of edges to remove from the graph, it's time to breakdown the task into smaller subtasks.

First and foremost, we should detect whether or not a cycle actually exists, because if it doesn't, then there is no problem. This is achievable through many ways and is a task that has been covered in detail by hundreds of articles. One such way is by attempting to do a typological sort of the vertices. A typological sort is one where the vertices are validly ordered according to their succession. In this case, a vertex *A* has to appear after *B* if *A* (directly or indirectly) depends on *B*.

More formally, a topologically sorted ordering is an ordering of the nodes of a directed graph such that if there is a path from node *A* to node *B*, then node *A* appears before node *B* in the ordering. For example, for the graph:

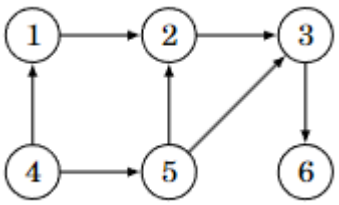

Fig. 7. Example of a directed graph 2 (copied from [3])

one topological sort is [4, 1, 5, 2, 3, 6]:

Fig. 8. Example of a topological sort (copied from [3])

A topological sort will not exist for a graph that contains cycles, as it will have at least two nodes that will simultaneously require to come before and after each other. For example:

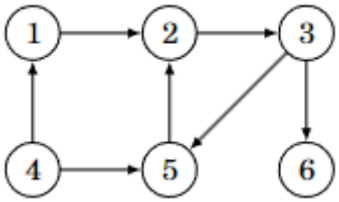

Fig. 9. Example of a cyckic graph (copied from [3])

In this graph, 3 has to appear before 5, yet 5 has to appear before 2 which implies 5 has to appear before 3 as well, which is a contradiction. It is thus concluded that a topological sort is not possible.

Kahn's algorithm is one that can detect a cycle in linear $O(|V| + |E|)$ time. It works by repeatedly finding vertices with no incoming edges (also referred to as **in-degrees** or **predecessors**), removing them from the graph, and updating the incoming edges of the remaining vertices. This process continues until all the vertices have been ordered. It is also important to note that disconnected components are independent of eachother and thus can be ordered arbitrarily (as long as each of them are themselves orederd). The following is a rough step by step of Kahn's algorithm.

1. Add all nodes with in-degree 0 to a queue.
2. If the queue is empty, stop.
3. Otherwise, pop a node from the queue.
4. For each outgoing edge from the removed node, i.e. it's successors, decrement the in-degree of the successor node by 1.
5. If the in-degree of a successor node becomes 0, add it to the queue and remove it from the graph.
6. If the queue is empty and there are still nodes in the graph, the graph contains a cycle and cannot be topologically sorted.
7. Otherwise, repeat from step 2.
8. The nodes in the queue represent the topological ordering of the graph.

C. *Modifying Kahn's Algorithm to Remove Dependencies*

As seen from the previous subsection, Kahn's algorithm is capable of detecting a cycle in the graph (step 6). The algorithm stops because it realizes it isn't possible to achieve it's goal. That being said, because our goal is to remove some set of edges that cause the cycle, in our end result, we will eventually have a combination of dependencies that *do* have a topological sort. Hence, we can simply continue the algorithm by removing some edges until an in-degree of a node becomes zero. We are essentially forcing a topological ordering to be possible.

This poses the important question: **which of the edges/dependencies do we remove such that the algorithm can continue and we remove as few as possible?** We can greedily select the node with the least in-degrees, and remove all of it's dependencies. Intuitively, this is optimal, because for all nodes that have larger in-degrees, we will need to erase more dependencies. However, the formal proof is left as an exercise to the reader.

The new algorithm now obliges us to keep track of all nodes in the queue, regardless of their in-degrees. Since we want to be able to remove the dependencies of the node with the least in-degrees, we will keep the queue sorted in ascending order by each node's in-degree count. As a result, this new algorithm will have a time complexity upperbounded by $O(|V| + |E| \log |V + E|)$ The steps for it is as follows (steps 1-4 exactly the same as Kahn's):

1. Maintain two sets for each node: a successor set and a predecessor set. The in-degree of a node can now be inferred from the size of the predecessor set of this node.
2. Add all nodes to a priority-based queue, with the in-degrees of each node as the priority.
3. If the queue is empty, stop.
4. Otherwise, pop a node from the queue.
5. If the node priority (which is it's predecessor count when it was enqueued) is not the same as it's current predecessor count, repeat from step 3. We will ignore this node as it was modified some time after we last enqueued it.
6. If the predecessor count of this node is not 0, remove all of it's predecessors. This entails removing the edge from the relevant predecessor and successor sets.
7. For each outgoing edge from the removed node, i.e. for all nodes in it's successor set, remove the corresponding edge from both the successor set of this node the and predecessor set of the successor node. Enqueue this node with priority `size(successor set) - 1`.
8. Repeat from step 2.
9. The nodes in the queue represent the topological ordering of the graph.

III. Two Cases

A. *Example One: An Illustration*

Below is a basic example graph to illustrate Kahn's as well as the new algorithm.

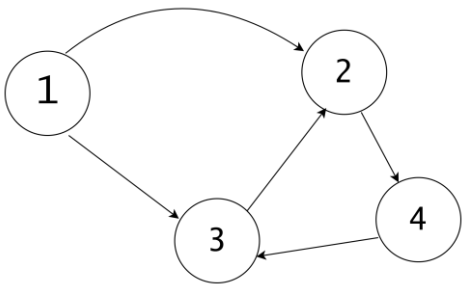

Fig. 10. Example of a graph of dependencies

For Kahn's algorithm, we will first calculate the in-degrees of each node.

Node	1	2	3	4
In-degrees	0	2	2	1

Since 1 is the only node with zero in-degrees, we initialize the queue as {1}. We then pop 1 and decrement the in-degree of each successor node of 1.

Node	1 (popped)	2	3	4
In-degrees	0	1	1	1

No new nodes with 0 in-degrees have been found, and no nodes with 0 in-degrees haven't been popped from the queue. Thus, the algorithm ends and we conclude that a topological sort is not possible.

In the new algorithm, we shall maintain the predecessor and successor sets, as mentioned before.

Node	1	2	3	4
Predecessors	-	1, 3	1, 4	2
Successors	2, 3	4	2	3

Below is a table that illustrates the process of the algorithm (the number in brackets in the priority queue column represents the priority of the queue element).

Iteration 0	Popped Node	Priority Queue
	-	1 (0), 3 (1), 4 (1), 2 (2)

Node	1	2	3	4
Predecessors	-	1, 3	1, 4	2
Successors	2, 3	4	2	3

Explanation: this is the intialization step.

Iteration 1	Popped Node	Priority Queue
	1	2 (1) 3 (1), 3(1), 4 (1), 2 (2)

Node	1	2	3	4
Predecessors	-	3	4	2
Successors	-	4	2	3

Explanation: 1 is popped.

1. First check that it's priority is the same as it's current predecessor count (this is true).
2. Next, since the predecessor count is 0, we can skip step 6. Now, 1's successors are removed and enqueued with the new predecessor count as the priority.

Iteration 2	Popped Node	Priority Queue
	2	4 (0), 3(1), 3(1), 4 (1), 2 (2)

Node	1	2	3	4
Predecessors	-	-	4	-
Successors	-	-	-	3

Explanation: 2 is popped.

1. First check that it's priority is the same as it's current predecessor count (this is true).
2. Next, since the predecessor count is not 0, we first need to remove all of it's dependencies first. Remove 3 from 2's predecessors and remove 2 from 3's successors.
3. Finally, we enqueue 2's successors. The node 4 is enqueued with priority 0.

Iteration 3	Popped Node	Priority Queue
	4	3 (0), 3(1), 3(1), 4 (1), 2 (2)

Node	1	2	3	4
Predecessors	-	-	-	-
Successors	-	-	-	-

Explanation: 4 is popped.

1. First check that it's priority is the same as it's current predecessor count (this is true).
2. Next, since the predecessor count is not 0, we first need to remove all of it's dependencies first. Remove 4 from 3's predecessors and remove 3 from 4's successors.
3. Finally, we enqueue 4's successors. The node 3 is enqueued with priority 0.

Iteration 4	Popped Node	Priority Queue
	3	3(1), 3(1), 4 (1), 2 (2)

Node	1	2	3	4
Predecessors	-	-	-	-
Successors	-	-	-	-

Explanation: 3 is popped.

1. First check that it's priority is the same as it's current predecessor count (this is true).
2. Next, this node no longer has any successors or predecessors. Thus, we don't need to do anything.

Iteration 5-8	Popped Node	Priority Queue
	-	-

Node	1	2	3	4
Predecessors	-	-	-	-
Successors	-	-	-	-

Explanation:

1. Before iteration 5, all nodes in the queue have invalid priorities. This is because at this point, all the predecessors are 0.
2. Thus, for each iteration 5 until 8, all we're doing is just popping the queue and ignoring the popped node.
3. We reach an empty queue, and thus have completed the algorithm

The result of the algorithm is as follows:

1. We have removed the $3 \rightarrow 2$ edge in iteration 2. This means we have erased the dependency "2 depends on 3".
2. After removing that dependency (and none others), we were able to topologically sort the graphs vertices.
3. This implies that removing the $3 \rightarrow 2$ edge is enough to resolve the circular dependency in this graph.
4. Additionally, we have obtained a topological ordering of the vertices, which is the order of the popped nodes: $1 \rightarrow 2 \rightarrow 4 \rightarrow 3$.

B. Example Two: A Case Study

We now move on to a second, more elaborate example. Say we have a software project in which the classes are interdependent. An example of this is a university management system comprising several classes: Professor, Course, Student, Department, Schedule, Room, Faculty, and University. Each class depends on others to model the complex interactions within the university. It is crucial to ensure that the class schema avoids circular dependencies which could pose an issue during compilation. Below is a table of example dependencies this system could have.

Class	Depends On
Professor	Department, Faculty
Course	Department
Student	Department
Faculty	-
Schedule	Course, Room
Room	-
Department	Proffesor, Faculty
University	Proffesor, Course, Student, Department, Schedule, Room, Faculty

The result of applying the algorithm to this set of dependencies is as follows:

1. The removed dependencies are "Course depends on Department" and "Department depends on Professor"
2. By removing these two dependencies (and none others), we have obtained the system with resolved circular dependencies by removing as few dependencies as possible.
3. Additionally, we have obtained a topological ordering of the classes: Faculty → Room → Course → Schedule → Department → Proffesor → Student → University

Class	Revised Dependencies
Professor	Department, Faculty
Course	-
Student	Department
Faculty	-
Schedule	Course, Room
Room	-
Department	Faculty
University	Proffesor, Course, Student, Department, Schedule, Room, Faculty

IV. IMPLEMENTATION

I have implemented the algorithm in Python. However, first the algorithm should read input from a file `input.txt` which has the following format:

1. The first line contains the names of all objects seperated by a space
2. Then for each every lines after that: the first line is empty, the second line is an object *A*, and the third line is a list of all objects that depend on *A*
3. Below is an example of the file that describes the second example in section III.

```
Proffesor Course Student Department Schedule Room
Faculty University

Proffesor
Department Faculty

Student
Department

Department
Proffesor Faculty

Schedule
Course Room

Course
Department

University
Proffesor Course Student Department Schedule Room
Faculty
```

Below is the algorithm to process the input file:

```
succs_list = dict()
preds_list = dict()


""" Input """
with open("input.txt", "r") as f :

    # Helper function
    def get() :
        return list(f.readline().strip().split())
```

```
    # Initialize the successor list and predecessor
list for each node
    nodes = get()
    for node in nodes :
        succs_list[node] = set()
        preds_list[node] = set()

    # Read each line describing the predecessor of
the node
    while True:

        if not f.readline() : break

        successor = get()[0]
        predecessors = get()

        for pred in predecessors :
            succs_list[pred].add(successor)
            preds_list[successor].add(pred)
```

And below is the code for the implementation of the algorithm:

```
""" Process the graph """

from heapq import heapify, heappush, heappop

seq = []
erased = []

pq = [(len(preds_list[node]), node)
      for node in preds_list]
heapify(pq)

while pq:

    pred_count, node = heappop(pq)

    if pred_count != len(preds_list[node]) :
        continue

    if len(preds_list[node]) != 0 :
        for pred in preds_list[node] :
            erased.append((pred, node))
            succs_list[pred].remove(node)
        preds_list[node] = set()

    for suc in succs_list[node] :
        preds_list[suc].remove(node)
        heappush(pq, (len(preds_list[suc]), suc))

    seq.append(node)

print(" → ".join(seq))
print(erased)
```

V. Conclusion

In this paper, I have demonstrated a simple technique to resolve circular dependencies by. First, the objects and dependencies are modelled as a directed graph. Then, Kahn's algorithm is modified to select and remove the edges of the graph that are causing a cycle, thus removing the circular dependency. The output of this algorithm is then the dependencies it has removed as well as the topological order of the objects after the removal.

The algorithm is optimal in the sense that the amount of dependencies that are removed is as few as possible. However, in a large system of interconnected dependencies, it may not be desirable to remove certain dependencies. Which is why, the algorithm should only be considered as a temporary solution, and the dependencies should eventually be properly resolved by design.

VI. Acknowledgments

I would like to express my gratitude to the lecturers of ITB Algorithm Strategies IF2211, Mrs. Ulfa Nur Maulidevi, Mrs. Harlili, Mrs. Fariska Zakhralativa, and Mr. Rinaldi Munir for sharing their knowledges and guiding the students throughout the learning process in the class. And of course, I would also like to thank my friends and family who have provided me their support and accompanied me day to day throughout the entire semester. They have made my days a lot more meaningful.

VII. References

[1] Verjans, M. "What is a circular dependency and how can I solve it?" Stack Overflow, 2016 https://stackoverflow.com/questions/38042130/

[2] Oyetoyan, Tosin Daniel & Falleri, Jean-Rémy & Dietrich, Jens & Jezek, Kamil. (2015). Circular Dependencies and Change-Proneness: An Empirical Study. 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings. 10.1109/SANER.2015.7081834.

[3] Laaksonen, Antti. Competitive Programming Handbook 1st ed., Springer, July 2018.

[4] A. B. Kahn. 1962. Topological sorting of large networks. Commun. ACM 5, 11 (Nov. 1962), p 558–562.

[5] Alexander A, Penerapan Topological Sorting pada Penjadwalan Proses, May 2017. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2016-2017/Makalah2017/Makalah-IF2211-2017-055.pdf

VIII. Statement

I hereby declare that the contents of the paper I have written is my own writing, not an adaptation or translation of another author's paper, and not plagiarised.

Bandung, 12 June 2024

Renaldy Arief Susanto - 13522022