JBP031-B-6 Data structures and algorithms 2023
Garbage Collection Group Project

March 1, 2023

Abstract

This project aims to equip students with the essential skills required to solve algorithmic problems. To this end, it covers a range of critical aspects, including algorithm design, data storage strategies, algorithmic correctness verification, and running time analysis. By engaging in this project, you will gain a deep understanding of these fundamental concepts and their importance in developing efficient algorithms.

1 Instructions

The problem description is presented in Section 3. A Python script is provided for completion, where participants must select an appropriate **data structure** that efficiently stores the data and an **algorithm** that solves the problem efficiently. In addition to submitting the code, a report of 2 to 6 pages is required. The report should include an explanation of the chosen algorithm, a thorough analysis of its correctness and running time complexity, and the test results. To facilitate testing, a collection of input cases is supplied on Canvas, alongside the Python script.
You are allowed to collaborate in groups of two members. One member of the team may submit the solution, but both members' names should be included in the report.

2 Grading

- 20 points for explanation of the chosen algorithm
    1. Justify why there is no better/faster algorithm than the one you have selected.
- 20 points for the correctness analysis
- 20 points for the complexity analysis
- 30 points for the test results
- 10 points for the quality of the code

3 Problem Description

The municipal administration of Den Bosch wants to keep its streets clean by placing as many garbage bins as possible on street intersections, particularly in the aftermath of the carnaval festivities. However, this would require the garbage collectors to make more stops on their daily routes, which they are not thrilled about. To reach a compromise, the city and the collectors have agreed that the city will place as many bins as possible, but only one bin will be placed on **adjacent** intersections. The parties are satisfied with this solution, and the city has already ordered the bins. However, the task of placing them in the city has proven to be more challenging than expected. As an expert in Algorithms and Data Structures, the city has enlisted your help to determine if it is feasible to place all the bins in the city.

3.1 Input

The personal of the municipal administration of Den Bosch is expecting to input (via **stdin**) the following data:

- The first line contains integers n, m and k, $(1 \leq n \leq 20, 1 \leq m \leq 100, 1 \leq k \leq 20)$ which denote the number of streets in Den Bosch, number of intersections in Den Bosch and number of bins which are already ordered. It is given that on every intersection a maximum of 4 streets meet.
- Then n lines, each containing two integers a and b, indices of intersections that are connected by a street (and thus adjacent). Intersections are numbered 1 to m. Moreover, a and b are always distinct (so an intersection is never adjacent to itself) and no two streets can connect the same two intersections

3.2 Output

Your output (via `stdout`) should be a string: "possible" if the city can place k bins, "impossible" if it can not.

3.3 Examples

We present two examples that illustrate the input-output description, along with the corresponding graph representation, below.

Sample input 1:

```
9 8 3
1 2
1 4
2 3
2 5
4 5
5 6
5 7
6 8
7 8
```

Sample output 1:

```
possible
```

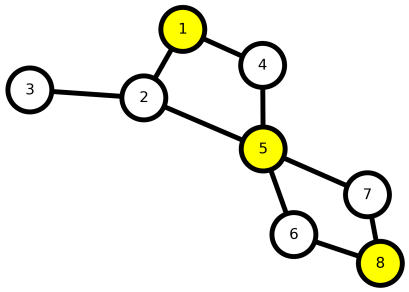

Figure 1: The first line of the input provides information about the size of the road map input, consisting of $n = 9$ streets and $m = 8$ intersections, where the government has requested the placement of $k = 3$ bins. The subsequent lines describe the edges between the vertices (or nodes) in the map. The expected output is "possible," and the corresponding graph representation is shown on the right-hand side, with the streets where the bins will be placed colored in yellow, satisfying the condition of $k = 3$.

Sample input 2:

```
8 7 4
1 2
1 4
2 3
2 5
4 5
5 6
5 7
6 7
```

Sample output 1:

```
impossible
```

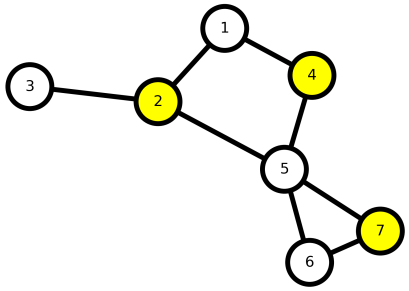

Figure 2: For a request of $n = 8$ streets, $m = 7$ intersections with $k = 4$ bins, the expected output is "impossible,". As shown with the corresponding graph representation on the right-hand side placing more than 3 bins will violate the restriction. Bins are colored in yellow.

3.4 Algorithm example

Algorithm 1 Find Non-Adjacent Vertices

Require: A graph $G = (V, E)$
Ensure: A list of non-adjacent vertex pairs

```
nonAdjacentVertices ← ∅
for each vertex v ∈ V do
    for each vertex w ∈ V do
        if v ≠ w and v is not adjacent to w then
            nonAdjacentVertices ← nonAdjacentVertices ∪ {(v, w)}
        end if
    end for
end for
return nonAdjacentVertices
```

Algorithm 1 takes a graph as input and returns a list of pairs of non-adjacent vertices in the graph. It works by iterating through every pair of vertices in the graph and checking if they are non-adjacent. If a pair of vertices is non-adjacent, it is added to the list of nonAdjacentVertices.
Note that this algorithm has a time complexity of $O(n^2)$, where n is the number of vertices in the graph. This is because it iterates through every pair of vertices in the graph. If the graph is very large, this algorithm may be slow, and more efficient algorithms may be needed.
Your task is to create a function bins_validation

```python
def validate_number_of_bins():
    """
    Provide an algorithm that solves the given task
    Provide a valid input-output description:
        what your algorithm is expecting to take as input and return as output
    """
    pass
```

4 Skeleton code

4.1 Code description

The code we provide builds

```
class Vertex(object):
    """ Vertex describes information of a single vertex in a Graph. """

    # used for faster access
    __slots__ = ["vid", "neighbours"]

    def __init__(self, vid: int, *args) -> None:
        """
        Construct a Vertex instance
        :param vid: Integer identifying this Vertex: Vertex ID
        :param args: Set of vertices adjacent to this Vertex
        """
        super().__init__(*args)
        self.vid = vid
        # TODO: Initialize a variable called neighbours with your chosen data structure
        # example: self.neighbours =

    def add_edge(self, u: "Vertex") -> None:
        """
        Creates an edge between this Vertex and Vertex u
        if vertex is the same then raise exception
        :param u: An adjacent Vertex that creates and edge with this Vertex
        """
        if self.vid == u.vid:
            raise Exception("Edges pointing to the same vertex are not allowed!")

        # TODO: Add Vertex "u" to this vertex by inserting this object to your
        #  self.neighbours data structure

    def __hash__(self) -> int:
        """
        Returns Vertex' ID
        identification allows this class (instance) to be hashed more efficiently
        :returns: This Vertex's identification: self.id
        :rtype: int
        """
        return self.vid


class Graph(object):
    """ Graph is composed of a set of Vertices, Edges and the number of bins.
    Two bins should not be placed adjacent to each other."""

    __slots__ = ["vertices", "n", "m", "k"]

    def __init__(self, *args, **kwargs) -> None:
        """
        Construct a Graph instance
```

```
        vertices: Set of Vertex instances in the graph
        n : Integer, number of nodes in the graph.
        m : Integer, number of edges in the graph.
        b : Integer, number of bins to be placed in the graph.
        """

        super().__init__(*args, **kwargs)
        self.vertices = {}
        self.n = None
        self.m = None
        self.k = None

    def parse_input(self) -> None:
        """
        Parse stdin input
        """
        # TODO: add condition to check number valid input characters

        # Get number of streets, intersections and bins (respectively)
        n, m, k = input().split()

        self.n, self.m, self.k = int(n), int(m), int(k)

        # Creates list of object type Vertex (Intersections) identified by an id
        vertices = [Vertex(i) for i in range(self.m)]

        # iterate over number of edges
        for _ in range(self.n):
            i, j = input().split()
            i = int(i) - 1
            j = int(j) - 1
            # add edges as undirected graph
            vertices[i].add_edge(vertices[j])
            vertices[j].add_edge(vertices[i])

        # Convert to set and store as class attributes
        self.vertices = set(vertices)

    def add_edges_from_list(self, elist: "list") -> None:
        """
        For testing when running on IDE
        :param elist:
        :return:
        """

        # get nodes from list of tuples
        nodes = list(set([item for t in elist for item in t]))
        self.n, self.m, self.k = int(len(elist)), int(len(nodes)), int(3)

        vertices = [Vertex(i) for i in range(self.m)]

        # iterate over list
        for edge in elist:
            i, j = edge[0], edge[1]
```

```
        i = int(i) - 1
        j = int(j) - 1
        # add edges as undirected graph
        vertices[i].add_edge(vertices[j])
        vertices[j].add_edge(vertices[i])

    # Convert to set and store as class attributes
    self.vertices = set(vertices)
```