Practical Assignment 1

Cas Haaijman, s4372662, Tijmen van der Kemp, s4446887

November 2019

1 Explanation of the algorithm

1.1 Parsing the problem into a data structure

We're building a graph from the problem, where streets are edges and crossroads are vertices. That way, we see that this problem is equivalent to the Independent Set Problem, which is famously a problem with an exponential running time. Converting the bins-crossroads problem into the graph is something that's done in polynomial time, so we're not going to give it much attention here.

1.2 Entities

We used the following entities (trivial stuff like getters, setters, equals, hashcode, have been omitted for the sake of brevity, unless they're interesting):

```
class Vertex {
    int id;
}

class Edge {
    Vertex v1;
    Vertex v2;

    // Since the edges aren't directional, Edge(2, 1) and Edge(1, 2) are the same.
    boolean equals(Edge other) {
        return (v1.equals(other.v1) && v2.equals(other.v2))
                || (v1.equals(other.v2) && v2.equals(other.v1));
    }
}

class Graph {
    Set<Vertex> vertices;
    Set<Edge> edges;
    Map<Vertex, Set<Vertex>> adjacencyLists;
}
```

```
class Problem {
    Graph graph;
    int numberOfBins;
}

// This basically says Algorithm { Problem problem, List<Problem> impossibleProblems }
// but splitting Problem into the graph and the number of bins was more convenient.
class Algorithm {
    int numberOfBins;
    Graph graph;
    // This is a collection of problems that have proven to be unsolvable, we use this for
    // quick lookup to disqualify new problems
    HashMap<Graph, Integer> impossibleProblems;
}
```

1.3 Functions

We use the following functions:

```
doCheapChecks();
fillAllGradeZeroVertices();
fillAsManyAsPossibleGradeOneVertices();
tryToPlaceABinOnGradeTwoOrMoreVertex();
```

The `doCheapChecks()` function is technically not a function, but can instead be directly replaced by the following code:

```
doCheapChecks =
    if (numberOfBins <= 0 || testVertexLowerBound() || TuranTest()) {
        return true;
    }
    if (graph.isEmpty() || hasBeenCheckedBefore()) {
        return false;
    }
```

In other words, the entire algorithm terminates with either true or false if one of these conditions has been met. These conditions are explained further in the correctness analysis subsection 2.1.

The `fillAllGradeZero()` function searches for all vertices with grade zero and then removes them and all edges from and to that vertex from the graph. It 'fills' all of these vertices with bins, so the number of bins is reduced by the amount of vertices removed.

The `fillAsManyAsPossibleGradeOneVertices()` function removes one-by-one all vertices of grade one and its neighbour and their edges from the graph. Each time it does this, the number of bins is reduced by one.

The `tryToPlaceABinOnGradeTwoOrMoreVertex()` is where the recursive part of the function comes in. It tries to remove the lowest grade vertex or its neighbours in the graph. It does this by removing one of these points and this points neighbours and then calling the `solve` function of a new algorithm entity to see if this resulting graph is solvable with one less bin to place. If it is, the function returns true and if it isn't, the function saves the resulting graph and the number of bins minus one in the list of impossible problems.

1.4 The Algorithm

Our algorithm is this recursive recipe:

```
boolean solve() {
    doCheapChecks();
    fillAllGradeZeroVertices();
    doCheapChecks();
    fillAsManyAsPossibleGradeOneVertices();
    doCheapChecks();
    fillAllGradeZeroVertices();
    doCheapChecks();
    return tryToPlaceABinOnGradeTwoOrMoreVertex();
}
```

2 Correctness analysis

2.1 `doCheapChecks()`

Since we don't need to provide an independent set, we can determine with some low running time checks whether it's always theoretically possible or impossible to find an independent set.

1. `numberofbins <= 0`: If we have no more bins left, it is always possible to place all bins.
2. `testVertexLowerBound()`: If the number of bins is less than a fifth of the number of vertices, we can place all the bins. This is because we can put every vertex in one of five groups that its neighbours are not in (if you have neighbours in groups 1, 2, 4, 5, you get put in group 3), and by the pigeon hole principle, there's always a group with $\lceil\frac{|V|}{5}\rceil$ vertices in them, on all of which you can place bins.
3. `TuranTest()`: According to a corollary to Turán's theorem, there's always an independent set of size $\frac{v}{\frac{2e}{v}+1}$, so if the number of bins is less than or equal to that, the problem is always possible. More on this in the subsection 2.1.1.

4. `graph.isEmpty()`: If the graph is empty (and there are still bins left), the problem is impossible.

5. `hasBeenCheckedBefore()`: If the graph is one we've already determined to be impossible for a certain number of bins k, we check if the current number of bins is $\geq k$; if so, this problem is also impossible.

2.1.1 Turán's Theorem

Assuming Turán's Theorem, we can show that in a graph G, with v vertices and e edges, there's an independent set of size $\geq \frac{v}{\frac{2e}{v}+1}$.
Let G' be the complement graph for G, and r' be the maximal size of a *clique* in G'. If a set of vertices is a clique in G', it's an independent set in G, because where first every vertex in the set is connected, in the complement graph none are.
Turán's theorem says that in a graph G' with v vertices, such that G' is $K_{r'+1}$-free, then the number of edges in that graph is at most $\frac{r'-1}{r'} \cdot \frac{v^2}{2}$.
Since we picked r' to be the maximal set of a clique in G', G' is automatically $K_{r'+1}$-free.
So now we have a relationship between the number of edges (e') in G', and the max independent set size r' in G. But there's also a relationship between e' and e: $e' = \frac{v(v-1)}{2} - e$.
This means:

$$\frac{v(v-1)}{2} - e \leq \frac{r'-1}{r'} \cdot \frac{v^2}{2}$$

$$\frac{v^2}{2} - \frac{v}{2} - e \leq \frac{r'-1}{r'} \cdot \frac{v^2}{2}$$

$$-\frac{v}{2} - e \leq (\frac{r'-1}{r'} - 1) \cdot \frac{v^2}{2}$$

$$-\frac{v}{2} - e \leq -\frac{1}{r'} \cdot \frac{v^2}{2}$$

$$v + 2e \geq \frac{v^2}{r'}$$

$$r' \geq \frac{v^2}{2e+v}$$

$$r' \geq \frac{v}{\frac{2e}{v}+1}$$

2.2 `fillAllGradeZeroVertices()` and `fillAsManyAsPossibleGradeOneVertices()`

If a graph contains vertices that have less then 2 edges connected to them, then there is no disadvantage to filling these points. We can look at the graph as a list of possible candidates for being filled. Whenever we try to fill one of the points, its neighbours are no longer candidates so they and the filled points get removed from the list. The only way filling a point can be detrimental to the maximum filled list is if filling the point results in more than one points to be removed from the list.

For points with grade zero, this is obvious. For points with grade one, if it's not immediately clear why it's always a good thing to fill them, look at it this way: Say there was a (possibly maximum) independent set of size r. And say there's the following subsituation: we have a vertex with grade one (g_1) that's connected to another vertex, possibly with a higher grade (g_2). Assume that g_2 has a bin, and therefore g_1 doesn't. Observe that all other neighbours of g_2 also can't have bins. We can now place the bin on g_1 instead of g_2, which results in the same number of bins, but the restrictions on g_2's neighbours are now lifted, so we could possibly place more bins.

2.3 `tryToPlaceABinOnGradeTwoOrMoreVertex()`

Whenever we find a graph that has no vertices of grade one or less, we use this function. This function picks a random vertex with the lowest grade v, fills v and then creates an induced subgraph with all vertices except v and its neighbours. If this subgraph can support an independent set of size k, then the original graph can support an independent set of size $k + 1$ by filling v. More generally, if any induced subgraph created by removing a vertex and its neighbours can support an independent set in the same way. Therefore, if this function returns true, we know that the problem is possible.

If the function returns false however, we only know that placing a bin on v does not result in a possible configuration of bins. To see that the entire graph has no possible configuration, we also have to check the neighbours of v. This is because all maximum independent sets (MISs) have a bin on either v or one of its neighbours. To see why this is the case, here's a simple proof from contradiction: assume that there is an MIS with a bin on none of those vertices. Now we can put a bin on v for a larger MIS, so the original MIS couldn't exist.

So to summarise, to find out whether our graph can support k bins, we only have to check a maximum of 5 induced subgraphs: the one where we place a bin on a random vertex v and those where we place a bin on either one of its (max 4) neighbours.

3 Complexity analysis

All non-recursive functions of our algorithm have polynomial time-complexity, while the recursive function `tryToPlaceABinOnGradeTwoOrMoreVertex()` has exponential time-complexity. Therefore this is the only function that matters for the limiting behaviour of our algorithm.

For figuring out the time-complexity, we want to look at the worst possible case the algorithm can operate on. We will start with a pessimistic simplification and try to reduce the complexity from there.

Recall that our algorithm has the following structure:

```
boolean solve() {
    doCheapChecks();
    fillAllGradeZeroVertices();
    doCheapChecks();
    fillAsManyAsPossibleGradeOneVertices();
    doCheapChecks();
    fillAllGradeZeroVertices();
    doCheapChecks();
    return tryToPlaceABinOnGradeTwoOrMoreVertex();
}
```

3.1 Everything except the recursive part

In this section, we will provide the time complexity upper bound of every function that is not part of the recursive function.

3.1.1 Cheap checks

1. `numberofbins <= 0`: This is $O(1)$.
2. `testVertexLowerBound(): return numberOfBins <= Math.ceil(graph.numberOfVertices() / 5.0)`: as long as division and Math.ceil are $O(1)$ with regard to `numberOfVertices`, this is $O(1)$.
3. `TuranTest(): int E = graph.numberOfEdges(); int V = graph.numberOfVertices(); return numberOfBins <= V / (E * 2.0 / V + 1);` This is also $O(1)$
4. `graph.isEmpty()`: $O(1)$.
5. `hasBeenCheckedBefore()`:

   ```
   if (impossibleProblems.containsKey(graph)) {
       return numberOfBins >= impossibleProblems.get(graph);
   }
   return false;
   ```

   Since `containsKey` is just a `get` that throws away the value, and the javadoc of HashMap says at https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html that "This implementation provides constant-time performance for the basic operations (get and put)", this is dependent on the complexity of our hashcode function. The hash of our graph is comprised of the hashes of all our vertices and edges, and those hashes are in turn calculated in $O(1)$. So each operation on the hashmap is $O(|V| + |E|)$ Since the maximum grade of each vertex is 4, we see that $|E|$ is at most $2 \cdot |V|$, therefore $O(|V|^2)$.

3.1.2 fillAllGradeZeroVertices

```
int fillAllGradeZeroVertices() {
    Set<Vertex> gradeZeroVertices = graph.getVerticesWithGrade(0);
    int numberRemoved = gradeZeroVertices.size();
    graph = GraphHandler.removeVerticesFromGraph(graph, gradeZeroVertices);
    numberOfBins -= numberRemoved;
    return numberRemoved;
}

Set<Vertex> getVerticesWithGrade(int grade) {
    return adjacencyLists.entrySet().stream()
            .filter(entry -> entry.getValue().size() == grade)
            .map(Map.Entry::getKey).collect(Collectors.toSet());
}
```

`getVerticesWithGrade` loops over all adjacency lists and picks out the ones that have a size equal to the grade that we're looking for. `size` is $O(1)$ because we chose HashSet as our implementation of an adjacencylist. So `getVerticesWithGrade` is $O(|V|)$.

```
1  Graph removeVerticesFromGraph(Graph graph, Set<Vertex> vertices) {
2      // subtract the set of edges to be removed
3      Set<Vertex> verticesLeftOver = subtractSets(graph.getVertices(), vertices);
4      // each edge that contains one of the removed vertices is removed as well
5      Set<Edge> edgesLeftOver = graph.getEdges();
6      for(Vertex vertex : vertices) {
7          Set<Edge> edgesFromOrToVertex = edgesFromOrToVertex(graph, vertex);
8          edgesLeftOver = subtractSets(edgesLeftOver, edgesFromOrToVertex);
9      }
10     return new Graph(verticesLeftOver, edgesLeftOver);
11 }
12
13 <T> Set<T> subtractSets(Set<T> set1, Set<T> set2) {
14     return set1.stream()
15     .filter(element -> !set2.contains(element))
16     .collect(Collectors.toSet());
17 }
```

The subtracting of sets is $O(|\texttt{set1}|) \cdot O(\texttt{set2.contains()})$

We define $|V|$ to be the number of vertices in the graph, r the number of vertices to be removed, which is at most $|V|$. The removing of vertices from graphs is the following complexity:

- line 3 has complexity $O(|V|^2)$
- line 5 has complexity $O(1)$

- the loop from line 6 to 9 has complexity $O(r)$
- line 7 has complexity $O(4)$: there are at most 4 neighbours, and we get them from the adjacencylist.
- line 8 has complexity $O(|E| \cdot 4) = O(|V|)$
- line 10 has complexity $O(|V| + |E|) = O(|V|)$

The total complexity of `removeVerticesFromGraph` is $O(|V|^2) + O(|V|^2) + O(|V|) = O(|V|^2)$.

Add $O(|V|)$ to that, and we get the complexity of `fillAllGradeZeroVertices`.

3.1.3 fillAsManyAsPossibleGradeOneVertices

```
1  void fillAsManyAsPossibleGradeOneVertices() {
2      // each time update which vertex to remove
3      Optional<Vertex> gradeOneVertex = graph.findVertexWithGrade(1);
4      while (gradeOneVertex.isPresent() && numberOfBins > 0) {
5          removeGradeOneVertexAndItsNeighbour(gradeOneVertex.get());
6          gradeOneVertex = graph.findVertexWithGrade(1);
7      }
8  }
9
10 void removeGradeOneVertexAndItsNeighbour(Vertex gradeOne) {
11     numberOfBins--;
12     Set<Vertex> verticesToRemove = graph.getVertexAndNeighbours(gradeOne);
13     graph = GraphHandler.removeVerticesFromGraph(graph, verticesToRemove);
14 }
```

Looking at `removeGradeOneVertexAndItsNeighbour`:

Since we used a hashmap as our implementation for the adjacencyLists, getting a vertex and its neighbours is an $O(5)$ task. We earlier calculated `removeVerticesFromGraph` as being a $O(|V|^2)$ task.

Now looking at `fillAsManyAsPossibleGradeOneVertices`:

`findVertexWithGrade` has complexity $O(|V|)$ as we saw in the previous subsection. The loop at line 4 loops a maximum of $O(|V|)$ times. So in total, the complexity of this method is $O(|V|^3)$.

3.2 The recursive part

```
1  boolean tryToPlaceABinOnGradeTwoOrMoreVertex() {
2      return getVertexWithLowestGradeAndItsNeighbours().stream()
3          .sorted(Comparator.comparing(graph::getGrade))
4          .map(graph::getVertexAndNeighbours)
5          .map(vertexAndNeighbours -> GraphHandler.removeVerticesFromGraph(
6              graph, vertexAndNeighbours))
7          .anyMatch(this::tryToSolveSubproblem);
}

Set<Vertex> getVertexWithLowestGradeAndItsNeighbours() {
    Vertex vertex = graph.getVertexWithMinimalGrade();
    return graph.getVertexAndNeighbours(vertex);
}

Vertex getVertexWithMinimalGrade() {
    return adjacencyLists.entrySet().stream()
            .min(Comparator.comparing(entry -> entry.getValue().size()))
            .map(Map.Entry::getKey)
            .orElseThrow(NullPointerException::new);
}
```

`getVertexWithMinimalGrade`: there are exactly $|V|$ adjacencylists. Comparing them to get a vertex with minimal grade is, in theory, an $O(|V|)$ task, but we could probably optimise this because we know that the size of an adjacencylist is 2, 3, or 4.

`getVertexWithLowestGradeAndItsNeighbours` is therefore an $O(|V|+5) = O(|V|)$ task. It will return at most 5 vertices.

`tryToPlaceABinOnGradeTwoOrMoreVertex` so far is $O(|V|)$. Sorting that list is a linear task since it's at most 5 vertices.

We then map each element to that element and their neighbours. ($O(5)$)

Then, we do removeVerticesFromGraph for each of those vertices. ($O(|V|^2)$)

Then, we try to solve those subproblems.

`tryToSolveSubproblem`: We see that of the rest of the entire algorithm, our complexity so far is $O(|V|^3)$. Following down this line, the worst case will look like this:

$$|V|^3 + k \cdot ((|V| - k)^3 + l \cdot ((|V| - l)^3 + m \cdot (...$$

where $k, l, m...$ are the the chosen vertex and its neighbours (so between 3 and 5, since there are no grade 1 or lower points anymore).

Read this as: "solve the problem with $|V|$ vertices the easy way in $|V|^3$, then solve k subproblems, each of those will take you $(|V| - k)^3$ time to solve the easy way, and then it subdivides into l problems, which..."

The next step in the worst case will have a problem with 3 fewer vertices, as that will result in the highest number of steps.

We make three of these new subproblems, so we get the following sum: $\sum_{x=0}^{|V|/3} 3^x(|V| - 3x)^3$

According to WolframAlpha this sum results in

$$\sum_{x=0}^{v/3} 3^x(v - 3x)^3 = 1/128(-32v^3 - 360v^2 - 1620v + 3105(3^{v/3} - 1))$$

which results in complexity $O(3^{|V|/3})$