Paint Shop Problem

by Luís Ramalho

October 4, 2018

Contents

1 Introduction

2 Algorithm

2.1 Options

When I first read the problem a matrix popped up in my mind, and I thought that it was similar to the somewhat famous N-queens problem [2] where one needs to place N non-attacking queens on an $N \times N$ chessboard. The N-queens is an example of a depth-first backtracking algorithm, which helps compute the solution for constraint satisfaction problems such as this one.

> ”A backtracking search for a solution to a CSP can be seen as performing a depth-first traversal of a search tree. The search tree is generated as the search progresses and represents alternative choices that may have to be examined in order to find a solution. [...] A backtracking algorithm visits a node if, at some point in the algorithm's execution, the node is generated. Constraints are used to check whether a node may possibly lead to a solution of the CSP and to prune subtrees containing no solutions. A node in the search tree is a deadend if it does not lead to a solution.” [1]

The options I have looked into when writing the algorithm to solve the issue 1 were basically Wikipedia, which had a nice article about backtracking, and also the pseudocode [5] for a recursive implementation and other implementations of similar problems, such as the N-queens.

2.2 Selection

What led to the current algorithm being selected was that it would take rather a long time to have a brute-force solution for the larger datasets. So, I needed a way to eliminate a possible candidate as soon as I knew that it will not lead to a solution. In the case of the backtracking strategy the actual search tree that is traversed is smaller than all the possible options, because we backtrack as soon as we see that all clients cannot be satisfied. Moreover, it is an algorithm that could be implemented for this particular problem such that we have the possibility of rejecting candidates early on. That fact is very important due to the fact that “the efficiency of the backtracking algorithm depends on reject returning true for candidates that are as close to the root as possible.“ [6]

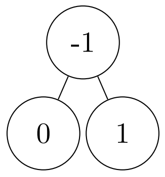
2.3 Pseudocode

The pseudocode for my algorithm is almost identical to the vanilla backtrack one with the exception that I want to return if a solution has been found.

```
function BT(order, candidate)
    if $solution \neq null$ then
        return
    if $reject(order, candidate)$ then
        return
    if $accept(order, candidate)$ then
        output(candidate)
    $solution \leftarrow first(candidate)$
    while $solution \neq null$ do
        bt(order, solution)
        $solution \leftarrow next(solution)$
```

2.4 Partial search tree

The search tree is generated as the search progresses and it is done by creating a simple binary tree at the first -1 it founds, that in turn creates another two possible partial solution candidates that are then run through the clients to check if all are satisfied.



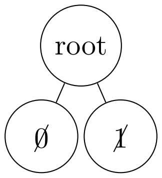
So, let's run through the examples provided in the problem statement.

2.4.1 First example

If we consider the first input example provide in the problem statement:

```
1
2
1 1 0
1 1 1
```

The algorithm tries and generates the following binary tree:



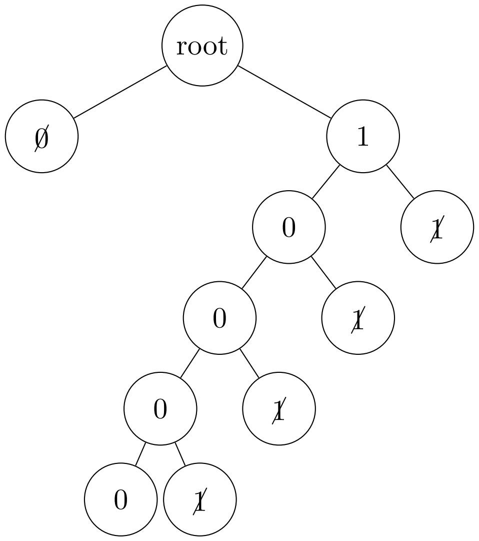
However none of the customers is satisfied with the final solution, i.e. the first customer cannot have the matte as he only wants glossy and the second customer's only preference is matte so glossy will not be accepted. Thus an empty solution is returned which forces the program to output IMPOSSIBLE.

2.4.2 Second example

In the case of the second example provide:

```
5
3
1 1 1
2 1 0 2 0
1 5 0
```

The algorithm should generate the final tree by backtracking as soon as as least one customer is not satisfied with the current solution.



The tree is generated in the following steps:

1. It generates a placeholder list the size of the paint colours with all nodes set to -1.

   ```
   [-1, -1, -1, -1, -1]
   ```

2. In the first step the algorithm creates a subtree, containing a 0 and a 1, for the first element with a -1. Due to the preference of the first client, the solution leading with a 0 is rejected and the other one is accepted. Thus, the new solution candidate is:

   ```
   [1, -1, -1, -1, -1]
   ```

3. The algorithm continues to create binary subtrees for each subsequent -1 and asking if all clients are happy with the solution candidate. In the case of the second example provided, all clients are satisfied with the solution being glossy for every batch.

   ```
   [1, -1, -1, -1, -1]
   [1, 0, -1, -1, -1]
   [1, 0, 0, -1, -1]
   [1, 0, 0, 0, -1]
   [1, 0, 0, 0, 0]
   ```

4. When the algorithm reaches the number of paint colours it outputs the solution.

   ```
   1 0 0 0 0
   ```

2.5 Time complexity

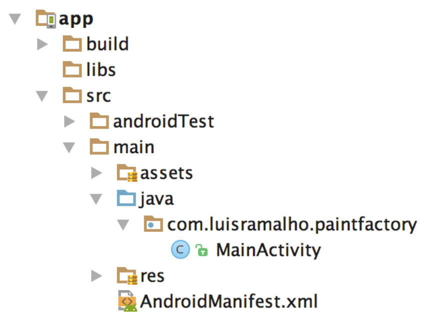
The time complexity for the algorithm is $O(n)$.

3 Android App

The android application was built such that an URL that points to a raw file can be typed or pasted and the application will try and compute the batches of paint that satisfy all customers.

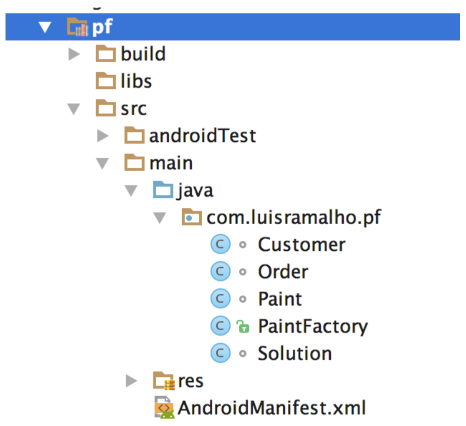
3.1 Library Implementation

The technologies I have evaluated when in implementing the library were not that many since I reckon I could build it in vanilla Java. Below you can see that in the library there is the logic of the algorithm and the app has a single activity.



```
▼ app
  ▶ build
    libs
  ▼ src
    ▶ androidTest
    ▼ main
      ▶ assets
      ▼ java
        ▼ com.luisramalho.paintfactory
            MainActivity
      ▶ res
        AndroidManifest.xml
```

(a) Contents of the app



```
▼ pf
  ▶ build
    libs
  ▼ src
    ▶ androidTest
    ▼ main
      ▼ java
        ▼ com.luisramalho.pf
            Customer
            Order
            Paint
            PaintFactory
            Solution
      ▶ res
        AndroidManifest.xml
```

(b) Contents of the library

Figure 1: Folder structure for the app and library (pf)

I have used was AsyncTask which allows the application to perform background operations and publish the results on the UI thread, this was necessary when I decided to fetch the file from an URL. Moreover, I could not resist adding ButterKnife [4].

3.2 Source Code

The source code is included in the folder PaintFactory in the deliverable zip file.

3.2.1 Instructions

I have used Android Studio to implement the application, so you should be able to open the source files attached and be able to run it in either an emulator or device. The folder to be opened is named PaintFactory, and the app folder has the MainActivity, the library is named pf.

3.3 Installer packages

The .apk file is in the deliverables zip and it is named app-release.apk, in addition I also provide a public URL for your convenience:

`https://dl.dropboxusercontent.com/u/88167266/app-release.apk`

3.3.1 Instructions for device or emulator

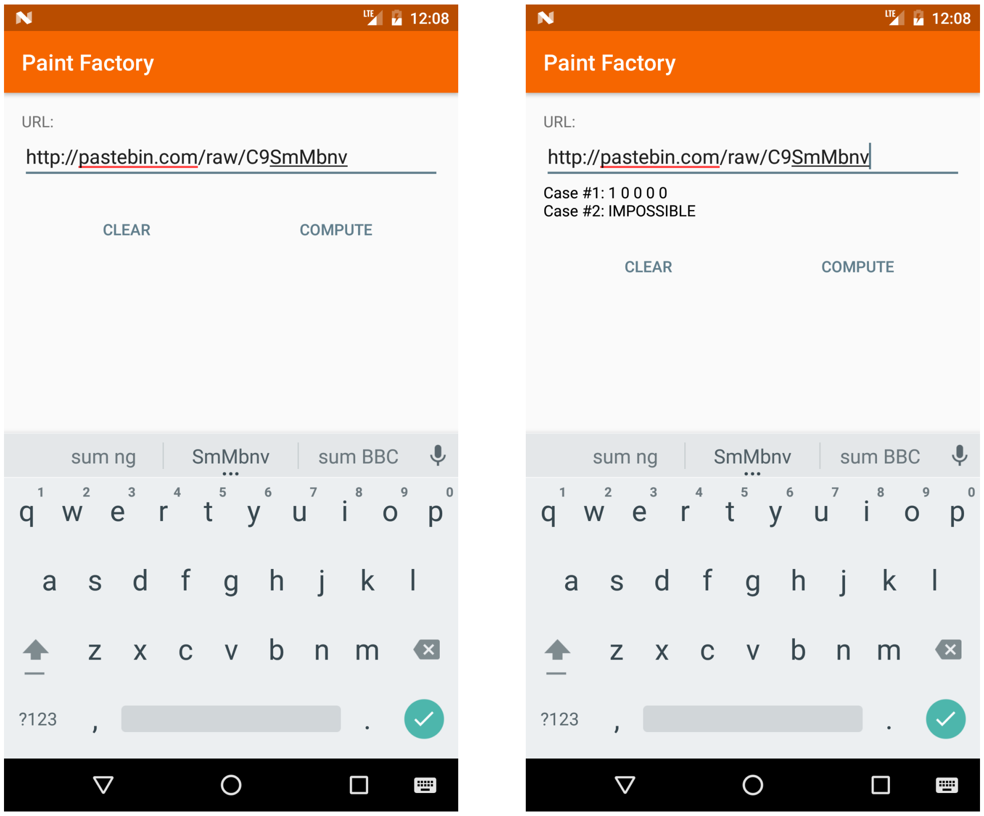
1. Allow installation of apps from sources other than the Play Story by going to:

   Settings $\rightarrow$ Security $\rightarrow$ Unknown sources

2. Download or upload the app to the device;
3. Click on the .apk file to install.

3.3.2 App running on the emulator

The screenshots below show the application running on the emulator, it basically consists of an input view and two buttons. The clear button clears the input field and the result.



http://pastebin.com/raw/C9SmMbnv

```
Case #1: 1 0 0 0 0
Case #2: IMPOSSIBLE
```

(a) Input the URL of the file (b) After pressing compute

Figure 2: Example of computing the sample input from a pastebin raw file [3].

An easy way to paste an URL to the input field is using the command below when the emulator or device is running.

```
adb shell input text 'http://pastebin.com/raw/C9SmMbnv'
```

I have added a few simple exception handling, but the application expects the file to be formatted properly and following the input guidelines stated in the problem statement.

4 Conclusion

In conclusion it was a really fun challenge to solve. I have tested it for a really large dataset T > 4000000 and it took ≈ 170 seconds on an old Nexus 5. At first I had a simple multi-line text field but it would not work for such large datasets so I just removed it and made the URL the only option for data input.

References

[1] Susan L Gerhart and Lawrence Yelowitz. Control structure abstractions of the backtracking programming technique. *IEEE Transactions on Software Engineering*, pages 285–292, 1976.

[2] Google. The n-queens problem. `https://developers.google.com/optimization/puzzles/queens`, Oct 2016.

[3] Pastebin. Pastebin sample problem. `http://pastebin.com/raw/C9SmMbnv`, Nov 2016.

[4] Jake Wharton. Butter knife. `http://jakewharton.github.io/butterknife/`, Nov 2016.

[5] Wikipedia. Backtracking pseudocode. `https://en.wikipedia.org/wiki/Backtracking#Pseudocode`, Nov 2016.

[6] Wikipedia. Backtracking usage considerations. `https://en.wikipedia.org/wiki/Backtracking#Usage_considerations`, Nov 2016.