Aalto-yliopisto
Aalto-universitetet
Aalto University

ELEC-E7852
Computational Interaction and Design

Assignment A1a

Combinatorial optimization

Aitor Urruticoechea Puig
aitor.urruticoecheapuig@aalto.fi
Student N°101444219
October 2024

Notice

Acknowledgement

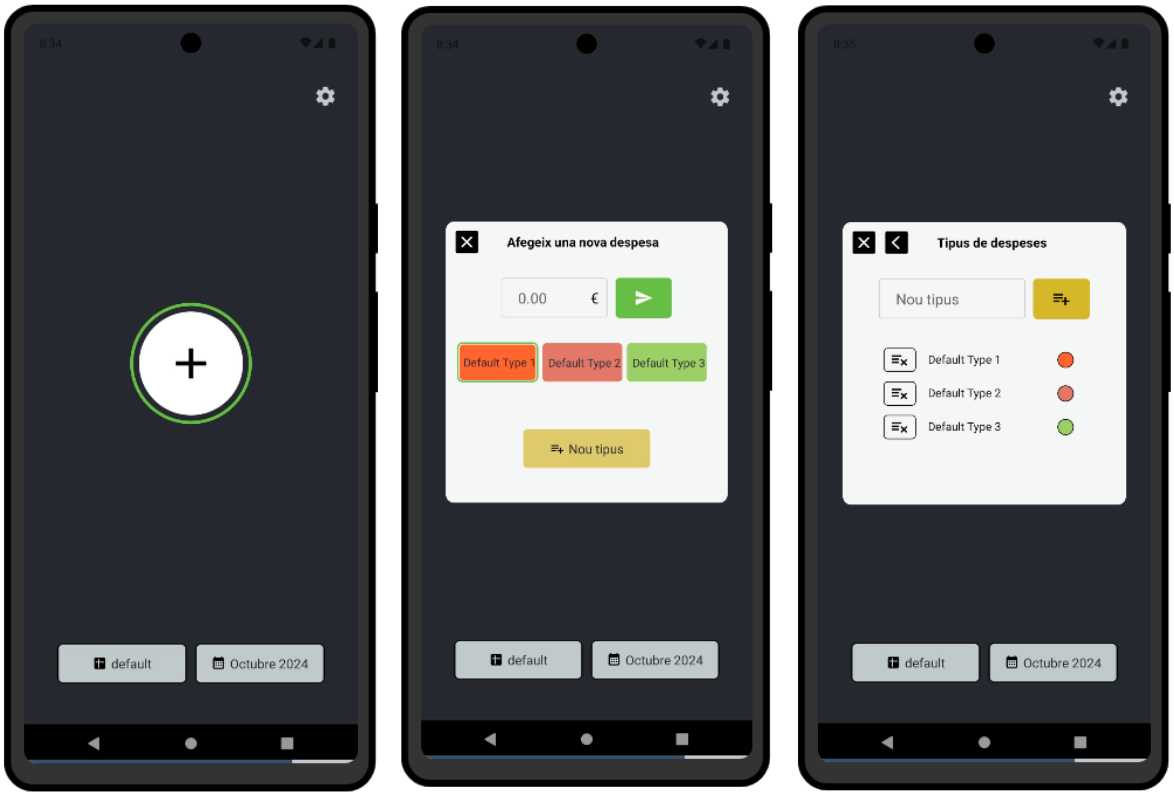
Parts of the code implemented in this work were generated with the assistance of GitHub Copilot. None of its proposals were, however, employed without refinement and necessary tweaks to adapt it to the nuances of the tasks at hand; making it impossible to identify and subsequently mark which lines of code were human or machine-made, for many were the fruit of the combination of both.

Contents

List of Figures

1 Introduction

From "Urruticoechea", the assigned topic is "Assignment of properties to UI elements". As a playground for this assignment, I will be using a side project I have been slowly developing. This is a spending tracking app which should help synchronize all your spendings and update your custom Google Sheets file with ease. Its UI is very much still in development, but can be seen in Figure 1 (unfortunately, the UI is only available in Catalan at the moment). The idea is to simplify the process as much as possible. The user needs only to choose a spending type and add the amount, the system should do the rest (Figure 1b). Spending types can be added, removed, and edited in a separate screen (Figure 1c).



(a) Home screen (b) "Add new spending" window. (c) "Spending types" window.

Figure 1: "DespesApp" spending tracking app overview.

Needless to say, the UI is far from perfect; and many decisions now made without any basis would greatly benefit from combinatorial optimization. Focusing on the assignment of properties to UI elements, one could, for instance, focus on:

- **Button Size Assignment**: Given the set of buttons with different functions and the limited amount of screen space on a mobile interface, determine the optimal size for each button.
- **Icon Assignment**: Given a set of potential icons representing different in-app functions; determine their assignment to each use.
- **Shortcut Assignment**: In the desktop/browser version of the app, it might be helpful to add keyboard shortcut support. Given the set of potential in-app functions, determine the assignment of these shortcuts to each function.

For this assignment, focus will be put on button size, as early testing (even before this assignment) proved that it was a challenge for accessibility and navigation.

2 Assignment Problem Formulation

Thus, from the button size dilemma, an optimization problem can be formulated regarding two key areas: usability, and user learning/effort. In this way, one can word the problem as:

- **Maximize** usability by assigning larger sizes to more frequently used buttons.
- **Minimize** user effort and learning required by having similar buttons have consistent sizes to make the interface easier to navigate.

This initial overview can be further broken down to identify the variables at play:

- There exist a set of n buttons in the app interface.
  - Each button i has a specific function (i.e., "Add Spending", "Cancel", "Set Type 1"...).
  - Redundancy is assumed to be non-existent: no two buttons have the same function.
- There is a maximum available area A.
- Each button i has a priority level p_i, indicating its importance or frequency of use.
- The size of each button s_i is to ultimately represent the area allocated to the button i.
- $m(i, j)$ will be the similarity measure between button i and button j, where higher values indicate higher similarity in their uses.

There is an obvious constraint regarding screen size if one assumes no scrolling features. This can be defined as:

$$\sum_{i=1}^{n} s_i \leq A \tag{1}$$

Since this is a mobile app, buttons need to be clickable by a finger, a minimum size constraint should also be considered:

$$s_i \geq s_{min} \quad \forall i \tag{2}$$

Finally, the maximization problem can be defined as:

$$maximize \sum_{i=1}^{n} p_i \cdot s_i \tag{3}$$

The minimization problem becomes a bit more complex, as the similarity between buttons forces the equation to take into account each pair of buttons. It is here where the similarity score helps in weighting the differences in size, as size differences between buttons with high similarity can be made to matter more than dissimilar buttons. Thus:

$$minimize \sum_{i=1}^{n} \sum_{j=1}^{n} m(i, j) \cdot |s_i - s_j| \tag{4}$$

3 Approach to the solver

The approach to the solver has been iterative and weight-based. Iterative because the objective is to define the optimizer based on a set number of maximum iterations; where the more the algorithm is iterated upon, the better the results are. And weight-based because the critical optimization point for this problem is the goal of having button sizes proportional to their priority values. Since this is the first assignment, the approach has been a "greedy" one: assigned size values will not be further edited in a bid to simplify the optimization process. Further, the solver has focused on the

maximization problem, and for the time being, the minimization one has been set aside. In this sense, the input values will be: `p`, an array of button priority values which will be assumed to be between 0 and 1; A, the maximum available area; s_{min}, the minimum area for each button; and `max_iter`, the maximum allowed iterations that will be allowed for the solver. Note that the number of buttons `n` is necessarily tied to `p`, thus it needs not be a direct input value for the proposed function.

From this beginning, it is important to have early returns for known early edge cases. First, the case where giving the minimum size to each button would already put the total area used above the maximum. Secondly, the case where the given maximum number of iterations is less than 1, where not even an initial size definition for each button would be possible.

```python
def max_button_size(p, A, s_min, max_iter=10):
    n = len(p)

    # Initialization
    sizes = np.ones(n) * s_min
    if np.sum(sizes) >= A:
        raise ValueError("The minimum size of each button is too large, or there are too
        ↪  many buttons.")

    remaining_space = A - np.sum(sizes)

    # Calculate the initial delta increase (if the max iterations are set to less than 1,
    ↪  the initial allocation is returned)
    if max_iter > 0:
        delta_increase = remaining_space / n
    else:
        return sizes
```

In this early snippet, it has already been shown one of the key values that will be used for area allocation: $\delta_{increase}$. The logic behind the proposed optimization calls for each iteration, resulting in an increase in the allocated area for each button, weighted against their priority values. This will be mediated precisely by $\delta_{increase}$.

After initializing $\delta_{increase}$ by dividing the remaining space (after giving every button the minimum area s_{min}) by the number of buttons `n`; one can enter the optimization loop itself. During each iteration, the buttons will be randomly sorted and visited. In each visit to each button, the algorithm will attempt to increase its allocated space by $\delta_{increase} \cdot$ p[i]. If that is allowed under the maximum area constraint, this will be done and the remaining area will be updated accordingly. Alternatively, if this would violate that constraint, the iteration is finished, and $\delta_{increase}$ is scaled down for the next iteration. This scaling down can be done in multiple ways. For this implementation, an exponential decay approach has been chosen. Similar to radioactive decay, $\delta_{increase}$ will be halved each time it results in a forbidden area increase.

The random approach to ordering the button visits deserves some attention. Due to the way these visits are interrupted each time $\delta_{increase}$, visit order benefits from this randomness. Should one try to order these visits by, for instance, by always visiting first the higher-priority buttons, one could end up visiting disproportionately more those buttons than the others, as eventually $\delta_{increase}$ would be too big and require updating before reaching the buttons at the very button of the priority list. By changing the visit order each iteration, on the flip side, this preference for some buttons over the others gets minimized as the iteration number increases.

```
    while remaining_space > 0 and iteration < max_iter:
        random_indices = np.random.permutation(n)
        # Assign new space to the buttons, according to their priority values
        for i in random_indices:
            # The size delta increase of each button is proportional to its priority value
            i_increase = delta_increase * p[i]
            if i_increase <= remaining_space:
                sizes[i] += i_increase
                remaining_space -= i_increase

            # Unless the button increase would exceed the available space
            else:
                break

        # Delta increase decay
        delta_increase *= 0.5 # Decay
        iteration += 1
```

Finally, one can return the output values, as well as the used size for the aggregate of all buttons. This latter one will help when evaluating this algorithm quantitatively.

```
    return sizes, sum(sizes)
```

4 Results

Qualitatively, it is of interest to evaluate this function by comparing how well the allocated spaces for each button corresponds to each button priority indicator. As a toy example, let us consider an available area $A = 1000$; a minimum button size of $s_{min} = 20$; and a set of four buttons with priorities $[0.2, 0.3, 0.5, 1]$ (let us assume the buttons for "Settings", "Cancel", "Add Spending Type", and "Add Spending"). With 10 iterations of the proposed algorithm, the found optimal sizer are, respectively, $[111.9, 157.9, 249.8, 479.6]$; using 999.2 area units.

This is considerably good, as one can easily see how, roughly, the button with a priority of 0.5 has an assigned area roughly half of the button with a priority of 1, and also approximately the equivalent area to the area sum of the buttons with priorities 0.2 and 0.3. This seems to work well with other toy examples. Keeping A, s_{min}, and the number of iterations unchanged; but assigning weights randomly to groups of 5 buttons; the algorithm is able to keep up in the outputting of correspondently scaled buttons with the minimal fluctuations expected from the random component (see Table 1).

		Buttons					Area Used
Test 2	Priorities	0.4	0.7	0.8	1	0.5	1000
	Areas	128	165.7	236	290	180.3	
Test 3	Priorities	0.9	0.2	0	0.7	0.4	891.2
	Areas	343.7	91.9	20	271.8	163.9	
Test 4	Priorities	0.7	0.4	0.9	0.5	1	999.9
	Areas	161.8	129.1	263	155.7	290.4	
Test 5	Priorities	0.2	0.5	0.9	0.7	0.3	999.9
	Areas	90	195.1	333.9	256.2	124.6	

Table 1: Toy example results with button priorities and assigned area sizes.

Quantitatively, it is of interest how the used area improves with an increase in the number of iterations. Since the algorithm assigns incrementally smaller deltas of area to the buttons, for smaller iteration numbers it might be hard to fully use the whole available area A. However, as the number of iterations increases, the algorithm is able to better distribute the available space and make a better use of the available space (where better use is understood as making use of a bigger portion of this available space). In order to visualize this improvement over iteration amount, let us keep the same value for s_{min} and keep the first two toy examples. A will now be defined as 10^6 area units; which in turn allows for an increase in the number of buttons n to have different toy examples for more extreme cases. Namely, obtained by assigning random values to sets of 10, 10^3, and 10^4 buttons respectively.

These five button sets will then be run through the algorithm by incremental numbers of iterations, from 2 to 15, which will allow for the plotting of an iterations versus used area graph (Figure 2). In it, there is a clear improvement of results, specially for increased iteration numbers under 9. Above that approximate threshold, it is unclear if more iterations are worth it, as the improvement yielded by each extra iteration diminishes quite quickly.

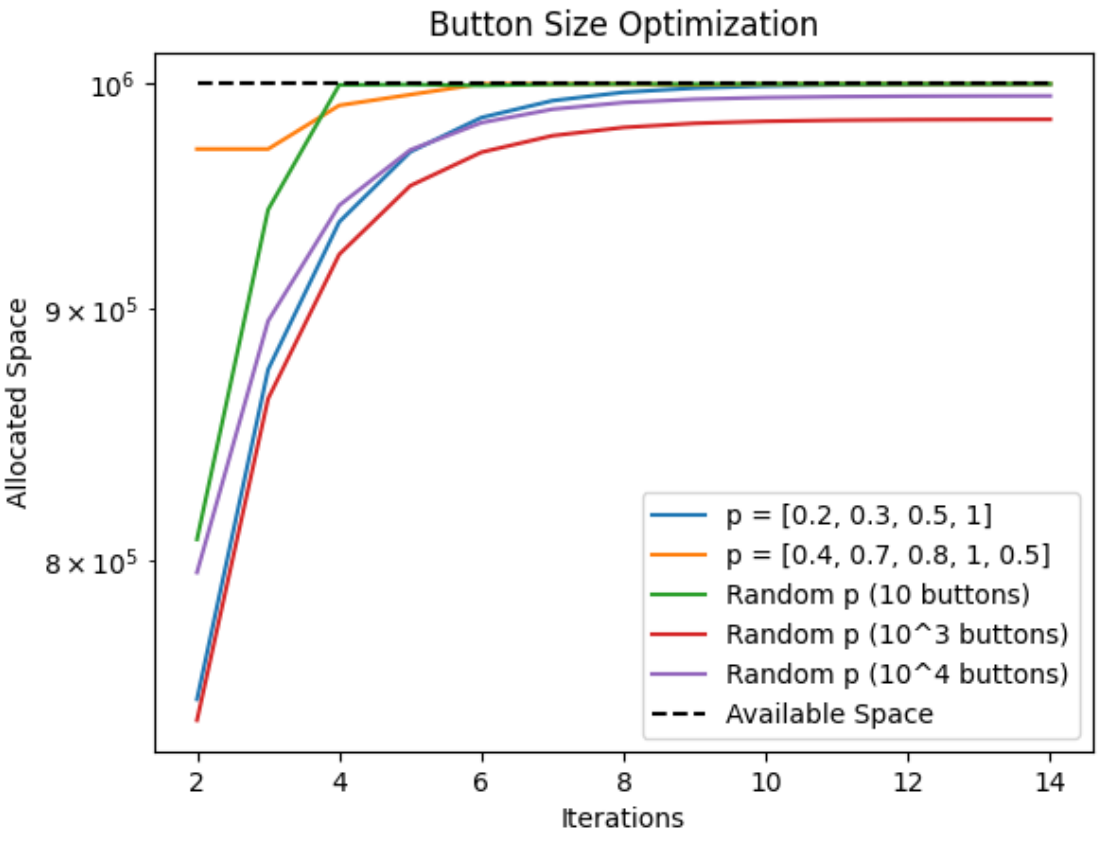


Figure 2: For different toy examples, total used area by buttons as a function of number of iterations.

5 Conclusion

From the developed optimization algorithm, it is clear that this simple approach yields great results. Button areas are correctly scaled to their priority values, and a more complete use of the area is computed as iterations increase. Nonetheless, it is important to recognize that many simplifications have been done in achieving these results. The minimization problem, regarding button similarity, has been completely ignored and will have to be re-examined (potentially in assignment A1b). Similarly, it is unclear if the formulation of the problem correctly reflects the need of the working case. For instance, even if the exit button is the one most pressed, does this mean that should proportionately occupy, let us say, half of the space allocated for buttons?

Even further at the root of the problem, one has to wonder: to what extent this system of button size allocation further reinforces behaviours, rather than conduct them? Are the most clicked buttons the ones that need more space assigned to them, or are they the most clicked buttons precisely because they have had much more space allocated to them? The author looks forward to future assignments, where this kind of design questions can be further explored with the nuance they deserve.