**Engineering design method**

**Phase 1**

**Context description:**

A computer distribution company offers its customers the necessary components to build a desktop PC. Each component consists of a name, price, image and component’s type. At the moment the available components are: cpu, motherboard, ram, gpu and storage. For each type of component the company has multiple options that cover a wide range of prices. However, being such a recognized company, stock comes and goes, so the available components can change every time the program is opened; also the large number of components they offer makes it difficult for customers to find the components they need, therefore, the company requires a program with a graphical interface to help customers in their process, this program has to recommend a complete PC to the customer starting from two components chosen or filter the information as the customer selects the parts in case they are not compatible with each other. (The components are reflected as a directed graph where their compatibility is the existence of a relationship, and the price of each part is the weight of the edge, the graph works as a hierarchy, where all the cpus point to all the motherboards, all the motherboards point to the rams, the rams to the gpus and the gpus to the storage).

You have been hired to design a preliminary program that:

* Allows the user to select the components that he wants to buy for his computer and, based on his choices, filters the information in such a way that the user is shown the most compatible parts possible in order to build a computer with highly compatible parts.
* Allows the user to select two reference parts (cpu and storage) and based on these show the complete computer with the lowest possible price.

Note: you must display the list of components chosen and the total price of the build. Since this is a preliminary program, in order to simulate the compatibility and stock of products, you should randomly delete edges each time the program is started, but no component hierarchy should be completely disconnected from another.

**Symptoms and Needs:**

* The customer wants a user-friendly interface.
* The customer wants a program that facilitates the acquisition of a build.
* The customer wants the program to filter components based on whether or not they are compatible.
* The customer wants multiple options to be shown based on two reference parts chosen by the user (cheapest).
* The customer wants the user to be able to build his own build based on the parts that are compatible with each other.

**Problem Identification:**

The customer needs a program that automatically shows builds and recommends components based on the user's choices, and randomly removes a number of relationships so that solutions and builds are not the same every time.

**Requirements specification:**

**Note:** The “requirements analysis” file is located in the “doc” folder.

**Phase 2**

**DFS:**Depth First Search is an algorithm for traversing or searching through graph structures. The algorithm starts at the root node (selecting some arbitrary node as the root node) and explores as far as possible along each branch before backtracking.

So, the basic idea is to start from the root, mark the node, move to the adjacent unmarked nodes and continue this cycle until there are no adjacent unmarked nodes. Then go back and find other roots and go through them. Finally, build a tree structure with the path.

[Depth First Search or DFS for a Graph - GeeksforGeeks](https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/)

**BFS:**Breadth-first search is an algorithm that explores all vertices adjacent to a root vertex and from these continues to explore the other vertices by assigning a predecessor and the minimum distance (in number of vertices) for each one respect to the root. Any node in the graph can be taken as the root and the algorithm will run until it reaches all nodes that have a path from the initial node. Unlike DFS, once the last reachable node is explored, no new root is assigned, so the result will always be a single vertex tree.

[BFS Algorithm - javatpoint](https://www.javatpoint.com/breadth-first-search-algorithm)

**Dijkstra:**This is a “greedy algorithm” that is useful to find the shortest path between two vertices of a graph, without necessarily including all vertices in the graph. It works in directed and undirected graphs and employs an iterative process to find the current shortest path by updating the cumulative distance and using two sets, one with the current shortest path and the other one with the vertices that haven’t been used.

[Dijkstra's Algorithm (programiz.com)](https://www.programiz.com/dsa/dijkstra-algorithm)

**Floyd-Warshall:** This algorithm is in charge of finding all the shortest paths between each pair of vertices in a given weighted and directed graph, the algorithm returns the shortest distances (summed) between each pair of vertices of the graph in a matrix where the minimum distances between nodes will be represented. It does this by changing the weighting estimate between each pair of vertices until it finds an estimate that is considered optimal, for this it uses intermediate vertices.The algorithm only returns the shortest distances between pairs of vertices, but small modifications can be made to return the details of the paths it uses.

[Floyd Warshall Algorithm - geeksforgeeks](https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/)

**Prim:** We start by choosing any edge of minimum weight and selecting it for the tree. We successively add edges to the tree from among those of minimum weight that are incident with a vertex that is already in the tree, and that does not form a cycle with other edges of the tree.

[Computación y Estructuras Discretas I (icesi.edu.co)](https://www.icesi.edu.co/moodle/pluginfile.php/889139/mod_resource/content/12/CyED1_diapositivas30%2822-2%29.pdf)

**Kruskal:** this is a [minimum spanning tree](https://www.programiz.com/dsa/spanning-tree-and-minimum-spanning-tree#minimum-spanning) algorithm that fits into the category of “greedy algorithms”, it takes a graph and finds a set of edges that form a tree that includes every single vertex of the graph and has the minimum weight possible. It works by sorting the edges of the graph from low to high weight, takes out one by one and adds it to a list with the condition that the new edge doesn’t create a cycle, then continues till all vertices are reached.

[Kruskal's Algorithm (programiz.com)](https://www.programiz.com/dsa/kruskal-algorithm)

**Safety measures:**

Measures to ensure the correct functioning of the program are needed in order to minimize the possibility of errors occurring.

* **Exceptions:**

Exception handling in java is an effective way to handle runtime errors so that the application does not crash its execution. When an exception occurs an object is generated containing a name, error description and system status, both default java exceptions and user designed exceptions( to handle specific cases) can be "trapped" to allow the normal flow of the system.

Source:<https://www.geeksforgeeks.org/exceptions-in-java/#:~:text=Exception%20is%20an%20unwanted%20or,method%2C%20it%20creates%20an%20object>.

* **Tests:**

For testing there is JUnit, a Java framework that allows you to generate test scenarios linked to the classes/objects you need to check and use them to perform unit tests with the help of “assertions” that allow you to make assumptions that will be checked when the test is executed.

Source:<https://www.vogella.com/tutorials/JUnit/article.html>

* **generics:**

The use of generics allows "types" to be parameters when defining classes, interfaces and methods. It is used to improve software stability because of the checking of "types" at compile time, thus reducing run-time errors that can be problematic.

Source:<https://docs.oracle.com/javase/tutorial/java/generics/why.html>

**Phase 3**

For this point we used brainstorming and the construction of scenarios, with the first technique, and with help of the previous research and preliminary ideas, multiple implementation options were obtained for the program; and in the second technique a few hypothetical scenarios that could happen in the program were built as well as the expected response for each case, thus collecting ideas that had been overlooked and that serve to solve these possible scenarios.

**Proposed solutions:**

We decided to divide the solutions into different modules that collect the main functionalities of the program in order to obtain a combination of solutions (one per module) that together provide an answer to the complete problem.

1. **Auto Build:**
   1. Use dijkstra's algorithm
   2. Use floyd warshall's algorithm
   3. Use prim’s algorithm
   4. Use kruskal’s algorithm
2. **Build with Help (Filter not compatible):**
   1. Use BFS algorithm.
   2. Use a DFS algorithm.
3. **UI:**
   1. Use an UI created on console
   2. Use javaFX to create an UI so it’s more user friendly and understandable
   3. Use java.Swing to create an UI so it’s more user friendly and understandable

**Phase 4**

**Discarded solutions:**

1. Auto Build:

1.3 and 1.4 both algorithms need to be used in a non-directed graph to work correctly, so it might happen that they return more than one type of component, for example two cpus, since we want to avoid that they were discarded.

1. Build with Help (Filter not compatible):

2.2 DFS travels among all nodes, even if they are disconnected from each other so we should avoid that.

1. UI:

3.1 It might seem too aggressive from an user’s perspective, specially since we would need to find a way to represent our graph in a way that’s understandable for the user.

**Possible solutions:**

1. Auto Build

1.1. Dijkstra's Algorithm:

* Returns the value of the shortest path between to components, and we can set the predecessors of each node so we can reconstruct the path

1.2. Floyd Warshall’s algorithm

* + Returns a matrix with all the shortest path from a node i, to a node j, and we can set the predecessors of each node so we can reconstruct the path

1. Build with Help:

2.1 BFS :

* BFS only goes through the vertex that are connected, so that facilitates our work to find the components compatible with the user’s choice, since we can return an array with the nodes painted in black we will have an array with all the compatible components from a particular piece.

1. UI:

3.2 JavaFX ui

* As programmers we will have an ui to work with to create or own ui, making it more friendly and easy for us.

3.3 Java.Swing

* Contains a set of GUI bases were the layout is based on JPanel

**Phase 5**

**Evaluation Criteria:**

To find the best solution for each module, it is necessary to have an evaluation system that assigns a value to each option according to the fulfillment of measurable criteria. To achieve this, we established three criteria were each one has three possible values (1,2,3) to evaluate its compliance:

1. that is efficient.
2. That it is not difficult to implement.
3. That is maintainable.

**Evaluación de propuestas por módulo:**

1. **Auto Build:**

|  | **Criteria 1** | **Criteria 2** | **Criteria 3** | **Total** |
| --- | --- | --- | --- | --- |
| 1.1 Dijkstra’s | **3** | **3** | **3** | **9** |
| 1.2 Floyd Warshall’s | **3** | **3** | **3** | **9** |

1. **Build with Help:**

|  | **Criteria 1** | **Criteria 2** | **Criteria 3** | **Total** |
| --- | --- | --- | --- | --- |
| 2.1. BFS | **3** | **3** | **3** | **9** |

1. **UI:**

|  | **Criteria 1** | **Criteria 2** | **Criteria 3** | **Total** |
| --- | --- | --- | --- | --- |
| 3.2 JavaFX | **2** | **3** | **3** | **8** |
| 3.3 Java.Swing | **3** | **1** | **3** | **7** |

**Finally we decided to go with:**

* Dijkstra's Algorithm since we only want the shortest path between two components, not between every component, and we can make our Dijkstra’s return the predecessors list to show the build to the user via UI.
* BFS, it was the only remaining. Only goes through the vertex that are connected, so that facilitates our work to find the components compatible with the user’s choice, since we can return an array with the nodes painted in black we will have an array with all the compatible components from a particular piece.
* JavaFX since we have more experience with it, and has an UI that allows us to drag and drop components and set methods way easier.