UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Data Structures and Algorithms

Q2 2024/2025

Team Identifier: 43.2

Raül Box Velasco: raul.box@estudiantat.upc.edu

Martina Cusidó Gascón: martina.cusido@estudiantat.upc.edu

Ada Peña Treviño: ada.pena@estudiantat.upc.edu

Aina Serra Ferré: aina.serra.ferre@estudiantat.upc.edu

1. Data Structures and algorithms description

To implement the main data structure (to store the lexicum) and the algorithm, we read and partially followed an abstract by Andrew W. Appel and Guy J. Jacobson:

https://www.cs.cmu.edu/afs/cs/academic/class/15451-s06/www/lectures/scrabble.pdf

1.1. Data Structures

Class DAWG (Directed Acyclic Word Graph):

The main data structure to be implemented was the DAWG. Since a language's lexicon involves a huge amount of data, to work with it we needed a structure that allowed efficient searches while also being as compact as possible. Our implementation of the DAWG fulfills both conditions.

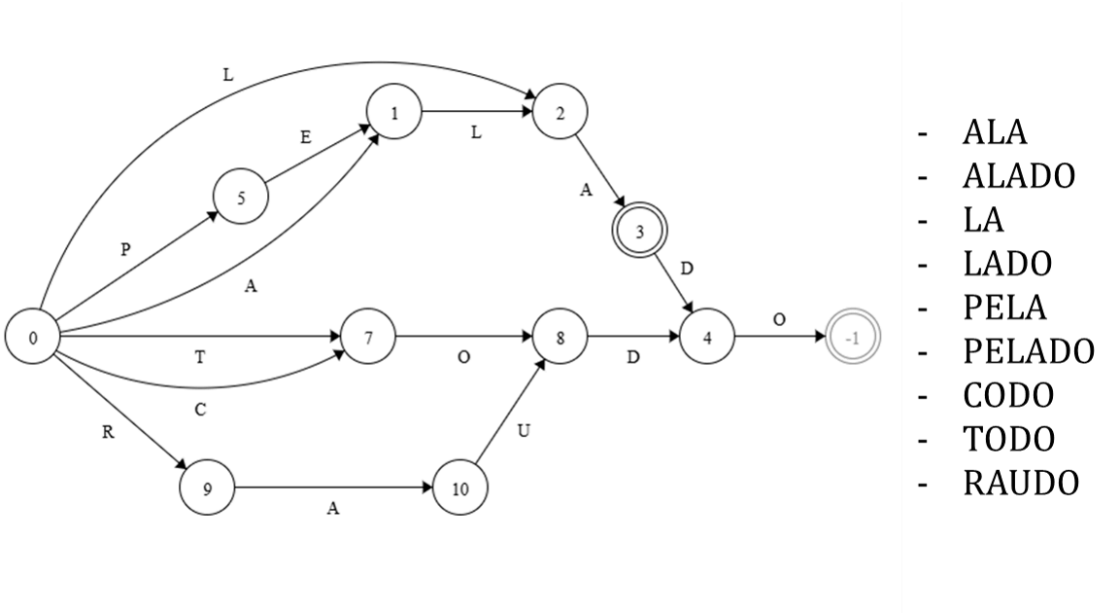

Figura 1: Example of DAWG

For its implementation, we were proposed a one-dimensional structure that would store all the edges contiguously. Each cell of this structure would represent an edge of the graph with the following attributes:

- **Letter:** the letter associated with the transition represented by the edge.
- **Next node:** the identifier of the destination node of the transition, corresponding to the index of the first edge of that node in the vector.
- **Is terminal:** a boolean indicating whether the transition leads to a word accepted in the lexicon.

- **Is last edge:** since all edges are stored contiguously regardless of the node they belong to, a boolean is needed to indicate which edges represent the last one of a node.

We took the proposed idea and reshaped it. Since it was originally designed for the hardware and software resources of the 1980s, it aimed to maximize spatial locality in memory, which is why a one-dimensional structure was suggested. However, since we work with Java—a reference-based, object-oriented language that manages memory in an arbitrary way—it doesn't make much sense to exploit spatial locality, as we constantly face unavoidable memory jumps.

Thus, in order to take advantage of Java's main appeal—class-based programming—and ignoring spatial locality optimization, we chose to create a two-dimensional structure that turns out to be more compact and more intuitive to implement and use.

For the final structure, we decided to create two auxiliary classes: "Edge" and "Node", where the former stores the attributes "letter", "next node", and "is terminal", while the only attribute of Node is a list of Edges. Thanks to the two-dimensional representation of the graph, we can omit the "is last edge" attribute, thereby achieving greater compactness.

The final structure consists of a list where each cell represents a node, and each node is a list of edges.

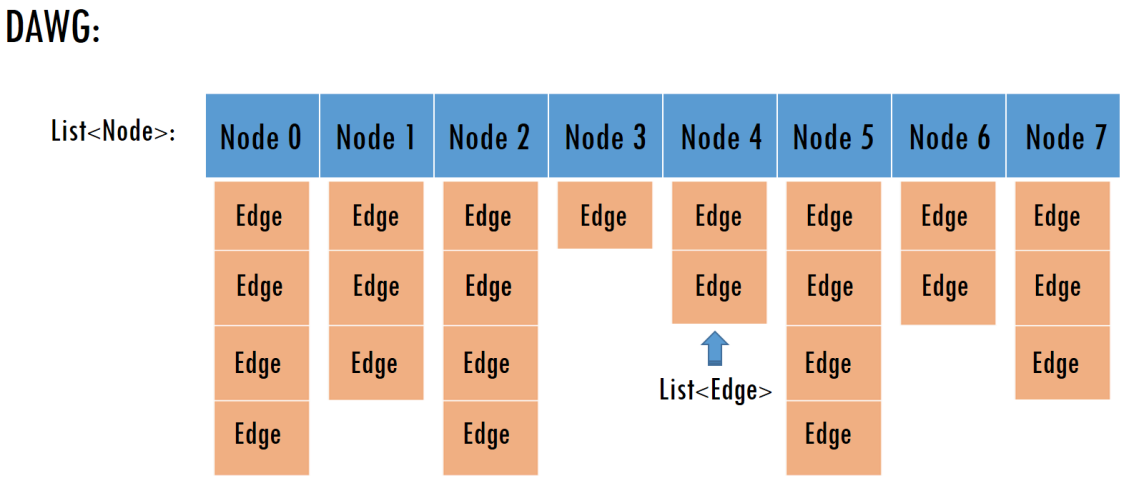

Figura 2: Visual representationde of our DAWG structure

To represent all this information, we thought the best choice was to use the ArrayList class from the "java.util" library, since the asymptotic cost of its operations is as follows:

- **Lookup** [get(i)] : O(1)
- **Insertion** [add(e)] :

- O(1) ➜ In case the structure does not need to be resized.
- O(n) ➜ In case the available space needs to be expanded.

- **Deletion** [remove(i)] : O(n)

Since the main interest is fast search operations and we don't need to perform any deletions, we opted for this class despite the near-linear cost of its creation, as it will only be executed at the start of the game. In the end, the DAWG is structured:

DAWG ≡ List<Node> ≡ List<List<Edge>> Edge ≡ {letter: String;
nextNode: int;
isTerminal: boolean;}

The asymptotic cost of its creation is calculated as follows, where 'n' is the number of words in the lexicon:

$$(n - \log(n)) * O(1) + \log(n) * O(n) \equiv O(n) + O(n * \log(n)) \equiv \mathbf{O(n * log(n))}$$

The first half of the initial expression corresponds to the ≈n insertions where the list does not need to be resized, while the second half accounts for the log(n) insertions that will require resizing. We know that the number of insertions that trigger a resize is log(n) because the ArrayList class doubles its size each time it expands.

On the other hand, the asymptotic cost of performing a lookup is calculated, in the worst case, as follows:

$$w * (O(1) + O(e)) \equiv O(w) + O(w * e) \equiv \mathbf{O(w*e)}$$

where 'w' corresponds to the length of the word being searched, and 'e' corresponds to the number of Edges of the largest Node visited during the search. Thus, we end up with a search that is linear with respect to the length of the word and the possible suffixes for each prefix subword.

- **Class Bossa (Bag):**

The class "Bossa" (Bag) must store all instances of "Fitxa" (Tile) that are not currently in play. Throughout the game, it will undergo insertions and deletions, with deletions being much more frequent. Since no predefined Java class offers deletions in constant time, we decided to implement a new class called "FastDeleteList" where retrieval, insertion, and deletion operations all run in constant time.

To achieve such efficient operations, we sacrifice the List's ability to maintain insertion order but retain the ability to have duplicate elements.

This class is implemented using a combination of a Map<T, Set<Integer>> and a List<T>. The list stores the elements to be held in an ArrayList, and the map associates each value T in the list with the set of indices where it appears.

When we perform an insertion in this structure, two internal insertions must be made: one into the map's set and one into the list. Insertions into a HashSet are O(1), while insertions into an ArrayList, as we mentioned earlier in the DAWG explanation, are constant unless the space needs to be expanded. Therefore, we pay a linear cost when creating the bag, but insertions during the game will be constant time since the list will never need to resize.

Lookups also have constant cost, as they rely on Java's get(i) operation, which, as previously mentioned, runs in O(1) time.

Deletions are the functionality that most exploits the combined use of the HashMap and the ArrayList. We divide deletions into two cases, depending on the function used: removeAt(i) and remove(T).

When performing a removeAt(i), the element at position i in the ArrayList is swapped with the last element, followed by removing the last position—an operation with constant cost. Finally, we remove the index from the corresponding HashSet in the HashMap, and if the set becomes empty, we also remove the entry from the map. Both removals are also O(1) operations.

If we want to perform a remove(T), we query the HashMap to find the indices where the value T is located; HashMap lookups are O(1). Once we have the indices, for each position to be removed from the ArrayList, we delegate the removal to removeAt(), whose constant cost we already justified.

Thus, we end up with a class that, although it duplicates memory space compared to a conventional list, allows deletions in constant time.

- **Class Taulell (Board):**

When implementing the board, a clearly two-dimensional structure, we initially considered using the same approach as with the DAWG and representing it as a List<List<Casella>>. However, since the board has a fixed size and doesn't need to be resized at any point (unlike the DAWG, which is constantly resized during creation), we ultimately chose to use a primitive array "Casella[ ][ ]", which is faster and uses less memory.

- **Class Ranking:**

We implemented the Ranking class using the most intuitive and straightforward approach possible: Java's PriorityQueue. We considered two types of rankings: rankingGlobal and rankingLocal — the first one for all players in the application and the second for each individual game. Each ranking is represented by a PriorityQueue<JugadorPuntuacio>, where JugadorPuntuacio is an auxiliary class (since Java doesn't support structs) that contains the player's ID and their current score.

1.2. Algorithm

The implemented algorithm is a backtracking algorithm with multiple highly effective pruning techniques that take into account the words existing in the dictionary, the tiles available to the player on their rack, and the tiles already placed on the board. Below is a brief explanation of the general functioning of the algorithm and the pruning methods, followed by a justification of why this algorithm is a good choice compared to other alternatives.

General Functioning:

Broadly speaking, what the algorithm does (without considering the pruning methods, which are explained later) is identifying the points on the board where it is possible to insert words. These points are called *anchors* and are all the adjacent positions to an already occupied cell (if there are no tiles on the board yet, the only anchor is considered to be the center of the board).

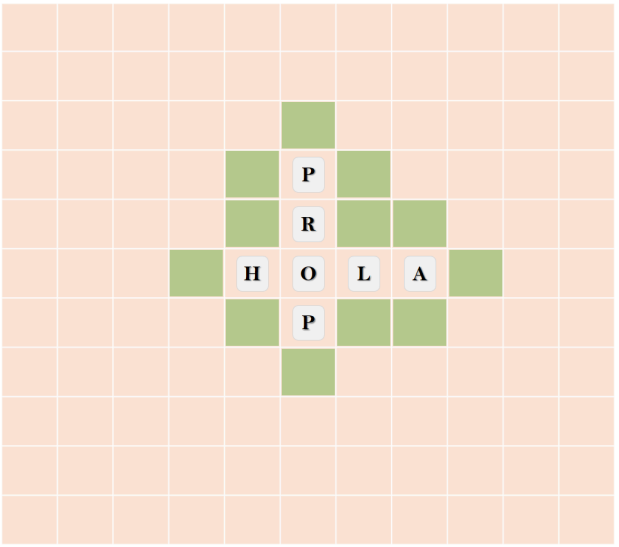

Figura 3: Board with the anchor cells highlighted.

Once the anchors are identified, the algorithm proceeds to generate words for each anchor. To simplify the program and avoid code duplication, a single algorithm is created to search only for horizontal words. It is executed twice: once on the original board and once on the transposed board—the second pass produces the vertical words.

In each of these passes, a brute-force algorithm is executed, divided into two parts that we've implemented as two separate functions: generateLeft(...) and extendRight(...).

The first function takes the anchor and generates all possible combinations (considering some pruning techniques explained later) using the letters from the rack up to a maximum length. This length is determined by calculating the distance from the anchor to the nearest cell to the left that is either an anchor or already occupied.

In the following visual example, the green cells are anchors, the red one is the anchor currently being processed by the algorithm, and the red tiles represent the combinations generated by the algorithm if the rack contained the following tiles: {A, B, C}.

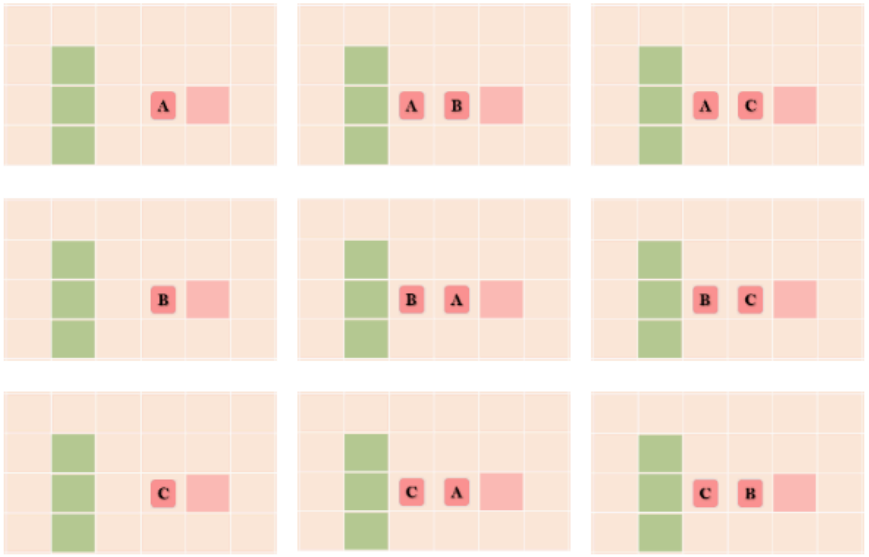

Figura 4: Primera fase de la generació de paraules del l'algorisme

For each subword generated by the generateLeft(...) function, a call is made to extendRight(...), which, taking into account the pruning techniques explained in the next section, extends the subwords using the remaining letters from the rack. If any of these words appear in the dictionary, form a valid play, and yield more points than the best word found so far, it will be saved as the best move found.

Backtracking pruning:

An algorithm like this with no pruning could take hours or even days to generate a single move. That's why several pruning techniques are used when searching for the best move:

- **Anchors**: Instead of considering every cell on the board, we only consider those that are adjacent to an already placed word.
- **Rack**: Rather than trying every letter in the dictionary at each new position, we only consider the tiles currently available on the player's rack.
- **Crosschecks**: If in addition to the main word we also generate perpendicular (cross) words, cross-checks help limit generation to letters that result in valid words in both directions.
- **Dictionary**: Instead of blindly considering all available rack letters at each step, we take into account the current subword being built and only consider letters that can validly follow it. This pruning relies on the DAWG implementation and is crucial to achieving an efficient algorithm.

Thanks to these pruning techniques, the algorithm is able to generate the best possible move quickly and efficiently.

Why it is a good choice:

The main strength of the algorithm lies in its pruning strategies. To begin our reasoning, we start from the following expression for the algorithm's asymptotic cost:

$$O(A * B * C) \equiv O(A * \Sigma^k * \Sigma^l) \equiv O(A * \Sigma^{k+l}) \equiv O(A * \Sigma^R)$$

Where A corresponds to the number of anchors on the board, B corresponds to the cost of generateLeft(...), C to the cost of extendRight(), R to the number of tiles on the rack, and Σ corresponds to the number of symbols in the alphabet. In the worst case, B is calculated Σ^k, where k is the maximum length of the generated prefixes, while C, on the other hand, is calculated as Σ^l, where l is the length of the extension.

Since the case of extending the word with a letter already present on the board has negligible cost (in fact, it further prunes the values to explore in upcoming evaluations), we can ignore it. We then observe that the maximum k+l is equivalent to the number of tiles in the rack, which are the cases that truly affect the cost, thus yielding a cost of Σ^R for word generation per anchor.

However, since thanks to the pruning we do not evaluate all the letters of the alphabet at each call, but only those available on the rack at that moment, we can express this Σ^R as R^R, or more precisely: $f(R) = \sum_{i=1}^{R} \binom{R}{i} * i! \simeq R!$ Therefore, we arrive at the expression:

$$O(A * f(R)) \equiv O(A * (R!))$$

One might think that when working with a backtracking algorithm, which is expected to have exponential cost, obtaining a factorial cost (asymptotically larger) is a bad result. However, if we think about it, since the values we work with are always bounded by a maximum value, the factorial result actually ends up being smaller than the exponential. The first expression obtained expressed the cost of the algorithm as Σ^R, where the value of Σ would be 26 and in the worst case R would be 7. Thus:

$$26^7 \geq 7! \Rightarrow \Sigma^R \geq R!$$

The use of the DAWG limits the branches considered only to valid suffixes, and the cross-checks discard letters that cannot form valid perpendicular words, which reduces the actual cost $f(R) \ll R!$ in most cases.

We therefore conclude that this algorithm, accompanied by the pruning strategies, is a very good choice for our objectives, as it considerably reduces computation time.