

CS2402: Lab assignment #2, some additional help

0.1 How to determine which walls surround a room?

i.e., how to determine which elements of the array current surround an element $M[i][j]$ of the matrix representing the museum?

Light reminder about walls indexing

The walls of the museum are indexed as shown in the text of the lab assignment.

In particular, all vertical walls are indexed from $0 + kn$ to $(n - 1) + kn$, where k indicate which vertical you consider (between the 0^{th} and the $(n - 1)^{\text{th}}$).

Same thing for the horizontal lines, except that before to consider an horizontal line, you ve already considered $n(n - 1)$ walls. So all horizontal wall are therefore indexed from $n(n - 1) + 0 + kn$ to $n(n - 1) + (n - 1) + kn$, where k indicates here which horizontal you consider (between the 0^{th} and the $(n - 1)^{\text{th}}$).

Let's got back to $M[i][j]$

As a consequence of the above reminder, the room referred to by $M[i][j]$ is surrounded by the walls indexed in the array by:

- current [$i + (j - 1) \cdot n$]
- current [$i + j \cdot n$]
- current [$n \cdot (n - 1) + j + (i - 1) \cdot n$]
- current [$n \cdot (n - 1) + j + (i - 1) \cdot n$]

0.2 Algorithm to check for consistency of the current problem against the matrix representing the museum

Let us consider first that the current problem is completely instantiated

i.e., the array representing the problem is made of 0's and 1's only.

Before to go on with the whole algorithm, let us consider one element of the matrix $M[i][j]$. Starting from this element, you want to count how many rooms (given the assignments of walls described by the arrays of 0's and 1's) you can see from room $[i][j]$ and compare it against the v alue it is supposed to be, *i.e.*, $M[i][j]$.

Counting the number of rooms that are visible from room $[i][j]$ consists in going in all 4 directions from room $[i][j]$ and stop when you hit a wall. You stop counting when the

number of rooms that are visible is greater than $M[i][j]$, or when you have checked all rooms aligned in the 4 directions.

Let us review the algorithm for checking if the number $M[i][j]$ of one room $[i][j]$ is consistent with the current assignment of walls `current`. This algorithm is made a method, and called `Checks-element-array`.

```

Method: checks-element-array (int M[i][j], problem current)
returns: boolean value

int room-counter=1; // at least room[i][j] is visible from room[i][j]

boolean wall=false;
// the integers we use to traverse the 4 directions:
int v1 = i - 1, v2 = i, h1 = j - 1, h2 = j

// checks the visible room on top of the current one
while (room-counter ≤ M[i][j]) and (v1 ≥ 1) and (wall=false) {
  if current[n(n - 1) + j + v1.n]=0, then {
    room-counter ++;
    v1 --;
  }
  else // i.e., there is a wall, so no need to check other rooms in this direction
    wall=true;
}
wall=false; // reinitialize wall to false to begin again the while statement

// checks the visible room below the current one
while (room-counter ≤ M[i][j]) and (v2 < n) and (wall=false) {
  if current[n(n - 1) + j + v2.n]=0, then {
    room-counter ++;
    v2 ++;
  }
  else // i.e., there is a wall, so no need to check other rooms in this direction
    wall=true;
}
wall=false; // reinitialize wall to false to begin again the while statement

// checks the visible room to the left of the current one
while (room-counter ≤ M[i][j]) and (h1 ≥ 1) and (wall=false) {
  if current[i + h1.n]=0, then {
    room-counter ++;
    h1 --;
  }
  else // i.e., there is a wall, so no need to check other rooms in this direction
    wall=true;
}
wall=false; // reinitialize wall to false to begin again the while statement

// checks the visible room to the right of the current one
while (room-counter ≤ M[i][j]) and (h2 < n) and (wall=false) {
  if current[i + h2.n]=0, then {
    room-counter ++;
    h2 ++;
  }
  else // i.e., there is a wall, so no need to check other rooms in this direction
    wall=true;
}
if (room-counter= M[i][j]) then return true;
else return false;

```

Then the global algorithm to check a problem (current array) against the matrix representing the museum is as follows:

```
Method: consistent(matrix M, problem current)  
returns: boolean value
```

```
boolean consistent = true;  
int i=0,j=0;  
  
while (i<n) and (consistent) {  
  while (j<n) and (consistent) {  
    if (checks-element-array(M[i][j],current)) then  
      j++;  
    else consistent = false;  
  }  
  i++;  
}  
}  
  
return consistent;
```

Let us consider first that the current problem is completely instantiated

i.e., the array representing the problem is made of 0's and 1's at the beginning, and of 2's at the end.

If you want to adjust this algorithm to the case where not all walls have been decided (*i.e.*, there are still 2's in the array), you have to add a case in each while statement of `checks-element-array`: this case cancels a check for a number of rooms as soon as a 2 is detected, and returns true by default.

Or other option: you can design another algorithm that is going to check only the rooms that are adjacent to your last choice. So that you don't have to retrace the whole matrix just because you've made a decision on wall x_m for instance. I don't provide this algorithm here. This is up to you to end up with such an algorithm.

0.3 General algorithm

The general algorithm is based on DFS, but integrates check-points that verify that the current context (the array of 0's, 1's and 2's) is still consistent with the constraints given by the matrix representing the museum.

Let's have a look at the algorithm. In red are highlighted the parts that are specific to DFS with constraints. The remaining parts (in black) are plain DFS.

```

Method: DFS+constraints(matrix M of size n, problem walls)
returns: problem // where all values of the returned problem are 0's or 1's
                // and are consistent with M

stack S=new stack();
problem current;
int index;

// initialization of the stack, so that it is not empty
S.push(walls); // at this point, walls is an array containing only 2's

while (S is not empty) {
    current=S.top();
    S.pop();

    // determine which wall is the next for decision
    index=find-index(current); // find-index was defined at the extra-class

    // checks whether current is totally instantiated or not
    if (index < current.size()) then {
        // current([index]=0 is array current where current[index] is set to 0
        if (consistent(M,current([index]=0))) then {
            S.push(current([index]=0);

            // current([index]=1 is array current where current[index] is set to 0
            if (consistent(M,current([index]=1))) then {
                S.push(current([index]=1);
            }
        }
        else { // i.e., current is totally instantiated, so we compare it against M
            if (consistent(M,current)) then returns current; }
    }
}

```