Exercise 1

You are given an image as a two-dimensional array of size $m \times n$. Each cell of the array represents a pixel in the image, and contains a number that represents the color of that pixel (for example, using the RGB model).

A segment in the image is a set of pixels that have the same color and are connected: each pixel in the segment can be reached from any other pixel in the segment by a sequence of moves up, down, left, or right. Design an efficient algorithm to find the size of the largest segment in the image.

Solution 1

To solve this problem, we think of the image as a graph. Each pixel is a vertex in the graph. Two pixels have an edge between them if they are adjacent (one is above/below/to the left of/to the right of the other) and have the same color. Notice that two pixels belong to the same segment if and only if they are connected in the graph. Furthermore, the segments correspond to the connected components of the graph, and the largest segment corresponds to the largest connected component of the graph.

The algorithm works as follows. We construct the graph, and initially all vertices are unvisited. Then run BFS or DFS from every vertex that has not been visited, and explore its connected component. We keep track of the size of each connected component, and return the maximum size.

When implementing the algorithm, we do not need to explicitly construct the graph, but instead we can: (i) treat each cell index $(i, j)$ as a vertex, and (ii) to find the list of neighbors of $(i, j)$ in the graph, we add or subtract 1 to $i$ or $j$ and compare the colors of the two neighboring cells. The algorithm can be implemented as follows:

1. For all $i, j$, set visited$[i][j]$ to False.
2. Initialize max_segment_size to 0.
3. For $i = 1, \ldots, m$:
   (a) For $j = 1, \ldots, n$:
      i. If visited$[i][j]$ is True, continue.
      ii. Initialize current_segment_size to 1 and start running BFS from $(i, j)$:
      iii. Initialize a new queue with one element $(i, j)$ and set visited$[i][j]$ to True.
      iv. While the queue is not empty:
         a. Pop the current indices cur_i, cur_j from the queue.
         b. For each pair of indices next_i, next_j that are adjacent to cur_i, cur_j:
            - If visited$[next_i][next_j]$ is False, continue.
            - Otherwise, add $(next_i, next_j)$ to the queue and set visited$[next_i][next_j]$ to True.
            - Increase current_segment_size by 1.
   v. If current_segment_size > max_segment_size, update max_segment_size.
4. Return max_segment_size.
If \(i\), \(j\) is within the boundaries of the array, \(\text{visited}[i][j]\) is False, and the color of cell \((i, j)\) is the same as the color of \((i, j)\):

- Set \(\text{visited}[i][j]\) to True.
- Increase \(\text{current\_segment\_size}\) by 1.
- Add \((i, j)\) to the queue.

v. Set \(\text{max\_segment\_size}\) to the maximum of \(\text{max\_segment\_size}\) and \(\text{current\_segment\_size}\).

4. Return \(\text{max\_segment\_size}\).

The run time of the algorithm is \(O(mn)\). The reasoning is the same as in the analysis of BFS. There are \(mn\) vertices, and each vertex has 4 neighbors. Therefore, each of the \(mn\) vertices is going to be considered a constant number of times (either when exploring one of its neighbors or when in the loop over \((i, j)\) that starts in step 3 of the algorithm). Note that the run time is linear in the number of pixels in the image \((mn)\).

**Exercise 2**

In this problem, we design an algorithm to solve the 2-SAT problem. Suppose we have \(n\) boolean variables

\[
\text{boolean } b_1 \\
\text{boolean } b_2 \\
\ldots \\
\text{boolean } b_n
\]

and \(m\) conditionals

\[
\text{if } a_1 \text{ or } a_2: \quad \text{// conditional 1} \\
\quad \text{// do task 1} \text{ if } a_3 \text{ or } a_4: \quad \text{// conditional 2} \\
\quad \text{// do task 2} \text{ . . .
}
\]

where each conditional is the disjunction of two literals \(a_i\) which are each either a boolean variable \((b_j)\) or some negation of a boolean variable \((\text{not } b_j)\). For example, one such list of conditionals is

\[
\text{if } b_1 \text{ or } \text{not } b_2: \\
\quad \text{// do task 1} \\
\text{if } b_2 \text{ or } b_3: \\
\quad \text{// do task 2} \\
\text{if } \text{not } b_1 \text{ or } \text{not } b_2: \\
\quad \text{// do task 3}
\]

In this problem, we will devise a polynomial time algorithm to assign all \(n\) boolean variables such that every conditional evaluates to true, and every task is executed (if possible). For example, in the above example, the algorithm might output

\[
b_1 = \text{True} \\
b_2 = \text{False} \\
b_3 = \text{True}
\]
Consider a set of $2n$ vertices $V$ where $n$ vertices correspond to the $n$ boolean variables \{b_1, \ldots, b_n\} and the other $n$ correspond to their negations \{\text{not } b_1, \ldots, \text{not } b_n\}. Denote the vertex that corresponds to boolean variable $b_i$ as $v_{b_i}$, and the vertex that corresponds to its negation as $v_{(\text{not } b_i)}$. Construct a directed graph on these vertices such that for every conditional if $c$ or $d$, we add the two directed edges $(v_{\text{not } c}, v_d)$ and $(v_{\text{not } d}, v_c)$ (Note that $c$ is any literal and can be a negation of a variable, and $\text{not } \text{not } c = c$).

a. Consider an assignment to the boolean variables. Show that all tasks are executed if and only if for every edge $(a, b)$ in the graph, if $a = \text{True}$, then $b = \text{True}$. Hence, we can consider edges of the graphs as “implications”: $a$ implies $b$.

b. Show that if any boolean variable $b_i$ lies in the same strongly connected component as its negation $(\text{not } b_i)$ in this graph, then there is no assignment of the $n$ boolean variables that satisfies all $m$ clauses.

c. Show that if there is a path in the graph $(a \rightarrow b)$ then there is a path $(\text{not } b \rightarrow \text{not } a)$.

d. In the following parts, we assume that no literal is in the same strongly connected component as its negation. Consider one strongly connected component $S \subset V$ of this graph. Show that the negations of every literal $a \in S$ are contained in a single other strongly connected component $\overline{S} \subset V$.

e. If a particular strongly connected component $S$ is a sink node of the SCC meta-graph (i.e., there are no edges from any vertex in $S$ to any vertex outside $S$), what do we know about $\overline{S}$?

f. Design an algorithm that finds an assignment that leads to an execution of all tasks (assuming no literal is in the same strongly connected component as its negation). Prove that your algorithm gives a valid truth assignment to each literal that satisfies every conditional, and show that it has running time polynomial in $n$ and $m$.

**Solution 2**

a. To prove the first direction, assume that we are given an assignment such that all the tasks are executed. Consider an edge $(v_{\text{not } a}, v_b)$ in the graph. This edge was added to the graph due to a conditional of the form

$$\text{if } a \text{ or } b:$$

In order for the conditional to evaluate to True, if $a = \text{False}$ (or equivalently, $\text{not } a$ is True), then we must have $b = \text{True}$. Therefore, if the assignment satisfies all the conditionals, for every edge $(v_{\text{not } a}, v_b)$, if $\text{not } a$ is True, $b$ must be True.

To show the other direction, suppose that edges represent implications (for an edge $(a,b)$, if $a$ is True in the given assignment, $b$ is True), and there is a conditional that evaluates to false. Without loss of generality let it be the above conditional. Then $\text{not } a$ is True and $b$ is False, but this cannot be, since $\text{not } a$ implies $b$. So we have a contradiction, and we have shown the “only if” direction.

b. By the definition of a strongly connected component, if a literal $a$ and its complement $\text{not } a$ are in the same strongly connected component, then there is a path in the graph from $(a \rightarrow \text{not } a)$ and a path back from $(\text{not } a \rightarrow a)$. Given an assignment where $a$ is True, consider the path from $a$ to $\text{not } a$. There must be an edge along the path that points from a literal that is True to a literal that is False. From part a, we know that the assignment does not satisfy all the conditionals. The same argument works for assignment where $a$ is False, but instead of the path from $a$ to $\text{not } a$, we use the path from $\text{not } a$ to $a$. 


c. For every edge in the graph \((a, b)\), there is also an edge \((\text{not } b, \text{not } a)\). Consider a path \((a_1, a_2, \ldots, a_n)\). Then all the edges \((\text{not } a_i, \text{not } a_{i-1})\) are also in the graph, and the path \((\text{not } a_n, \ldots, \text{not } a_1)\) also exists.

d. Let \(S\) be a SCC. By the definition of a SCC, for every pair of nodes \(a \in S\) and \(b \in S\), there is a path from \(a\) to \(b\). So there is also a path from \(\text{not } b\) to \(\text{not } a\). So then there is a path between the negations of every two nodes in \(S\). So the negations of all nodes in \(S\) form a SCC.

e. If \(S\) is a sink, then \(\bar{S}\) must be a source (that is, there are no edges that go from a node outside \(\bar{S}\) to \(\bar{S}\)). Assume toward contradiction that there is an edge \((v, v')\) such that \(v' \in \bar{S}\) and \(v \notin \bar{S}\). Then there must be an edge \((\text{not } v', \text{not } v)\). We know from part d that \(v' \in S\) and that \(v \notin S\). Since \(S\) is a sink, the existence of the edge \((\text{not } v', \text{not } v)\) leads to a contradiction. Therefore, \(\bar{S}\) must be a source.

f. Algorithm:

(a) Compute the implication graph and its SCCs. If any SCC contains both a variable and its negation, return unsatisfiable.

(b) Run topological sort on the SCC meta-graph.

(c) Consider the SCCs in reverse topological order. If the literals in the current SCC have not been assigned values, set them all to True. (Their negations will be set to False.)

All the steps in the algorithm can be done in polynomial time. To prove the correctness of the algorithm, from part a, we need to show that the returned assignment does not result in any edges that point from a vertex assigned True to a vertex assigned False. The intuition is that we apply part e iteratively. Since we consider the SCCs in reverse topological order, the first SCC we pick is a sink. Consider the set of variables used by the literals in that SCC. We assign True to the literals in the sink SCC and False to their negations in the source SCC. Therefore, any edge that involves any of the variables in the source/sink SCC cannot point from a vertex assigned True to a vertex assigned False. Intuitively, we can now remove these source and sink SCCs from the graph, and apply the same logic to the remaining SCCs iteratively, until all variables are assigned. (The correctness of the algorithm can be proved formally by induction on the number of SCCs in the graph.)