2.3 More of the Input vs More of the Output

Sometimes it is not clear at first whether to use "More Of The Input" or "More Of The Output" type of loop invariants. This section gives two similar problems, one of which the first works better and the other in which the second works better.

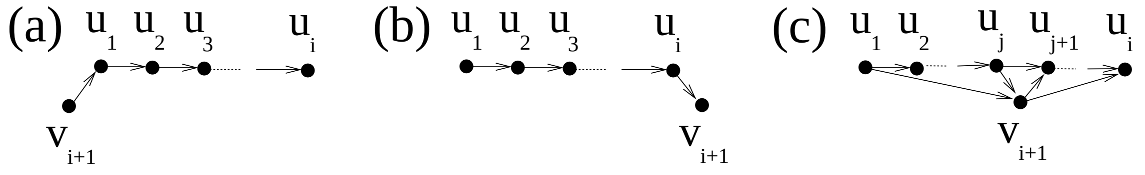
Example 2.3.1 Tournament: A *tournament* is a directed graph (see Section 3.1) formed by taking the complete undirected graph and assigning arbitrary directions on the edges, i.e., a graph $G = (V, E)$ such that for each $u, v \in V$, exactly one of $\langle u, v\rangle$ or $\langle v, u\rangle$ is in E. A *Hamiltonian path* is a path through a graph that can start and finish any where but must visit every node exactly once each. Design an algorithm which finds a Hamiltonian path through it given any tournament. Because this algorithm finds a Hamiltonian path for each tournament, this algorithm, in itself, acts as proof that every tournament has a Hamiltonian path.

More of the Output: It is natural to want to push forward and find the required path through a graph. The measure of progress would be the amount of the path output and the loop invariant would say "I have the first i nodes (edges) in the final path." Maintaining this loop invariant would require extending the path constructed so far by one more node. The problem, however, is that the algorithm might get stuck, when the path constructed so far has no edges leaving the last node to a node that has not yet been visited. This makes the loop invariant as stated false.

Recursive Backtracking: One is then tempted to have the algorithm "backtrack" when it gets stuck trying in a different direction for the path to go. This is a fine algorithm. See recursive backtracking algorithms in Chapter 17. However, unless one is really careful, such algorithms tend to require exponential time.

More of the Input: Instead, try solving this problem using a "more of the input" loop invariant. Assume the nodes are numbered 1 to n in an arbitrary way. The algorithm temporarily pretends that the sub-graph on the first i of the nodes is the entire input instance. The loop invariant is "I currently have a solution for this sub-instance." Such a solution is a hamiltonian path $u_1, \ldots, u_i$ that visits each of the first i nodes exactly once each, which in turn is simply a permutation the first i nodes. Maintaining this loop invariant requires constructing a path for the first $i+1$ nodes. There is no requirement that this new path resembles the previous path. However, for this problem, it can be accomplished by finding a place to insert the $i + 1^{st}$ node within the permutation of the first i nodes. In this way, the algorithm looks a lot like insertion sort.

Case Analysis: When developing an algorithm, a good technique is to see for which input instances the obvious thing works and then try to design another algorithm for the remaining cases.



(a) If $\langle v_{i+1}, u_1\rangle$ is an edge, then the extended path is easily $v_{i+1}, u_1, \ldots, u_i$.

(b) Similarly, if $\langle u_i, v_{i+1}\rangle$ is an edge, then the extended path is easily $u_1, \ldots, u_i, v_{i+1}$.

(c) Otherwise, because the graph is a tournament, both $\langle u_1, v_{i+1}\rangle$ and $\langle v_{i+1}, u_i\rangle$ are edges. Color each node u_j red if $\langle u_j, v_{i+1}\rangle$ is an edge and blue if $\langle v_{i+1}, u_j\rangle$ is. Because u_1 is red and u_i is blue, there must be some place u_j to u_{j+i} in the path where path changes color from red to blue. Because both $\langle u_j, v_{i+1}\rangle$ and $\langle v_{i+1}, u_{j+i}\rangle$ are edges, we can form the extended path $u_1, \ldots, u_j, v_{i+1}, u_{j+i}, \ldots, u_i$.

Example 2.3.2 Euler Tour: An *Euler tour* in an undirected graph is a cycle that passes through each edge exactly once. A graph contains an Eulerian cycle iff it is connected and the degree of each vertex is even. Given such a graph find such a cycle.

More of the Output: We will again start by attempting to solve the problem using the more the output technique, namely, start at any node and build the output path one edge at a time. Not having any real insight into which edge should be taken next, we will choose them in a blind or "greedy" way (see Chapter 16). The loop invariant is that after i steps you have some path through i different edges from some node s to some node v.

Getting Stuck: The next step in designing this algorithm is to determine when, if ever, this simple blind algorithm gets stuck and to either figure out how to avoid this situation or to fix it.

Making Progress: If $s \neq v$, then the end node v must be adjacent to an odd number of edges that are in the path. See Figure **??**.a. This is because there is the last edge in the path and then for every edge in the path coming into the node there is one leaving. Hence, because v has even degree it follows that v is adjacent to at least one edge that is not in the path. Follow this edge extending the path by one edge. This maintains the loop invariant while making progress. This process can only get stuck when the path happens to cycle around back to the starting node giving $s = v$. In such a case, join the path here to form a cycle.

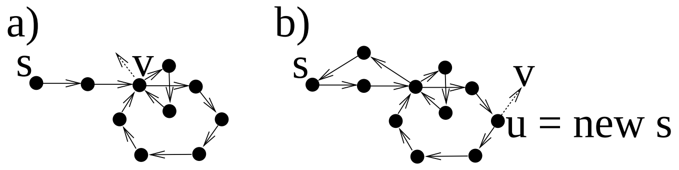


Figure 2.2: Euler Algorithm

Ending: If the cycle created covers all of the edges, then we are done.

Getting Unstuck: If the cycle we have created from our chosen node s back to s does not cover all the edges, then we look for a node u with in this cycle that is adjacent to an edge not in the cycle. See Figure **??**.b. Change s to be this new node u. We break the cycle at u giving us a path from u back to u. The difference with this path is that we can extend it past u along this unvisited edge. Again the loop invariant has been maintained while making progress.

***u* Exists:** The only thing remaining to prove is that when v comes around to meet s again and we are not done, then there is in fact a node u in the path that is adjacent to an edge not in the path. Because we are not done then there is an edge e in the graph that is not in our path. Because the graph is connected, there must be a path in the graph from e to our constructed path. The node u at which this connecting path meets our constructed path must be as required because the last edge $\{u, w\}$ in the connecting path is not in our constructed path.

Extended Loop Invariant: To avoid having to find such a node u when it is needed, we extend the loop invariant to state that in addition to the path, the algorithm remembers some node u other than s and v that is in the path and is adjacent to an edge not in the path.

Exercise 2.3.1 *(See solution in Section V) Iterative Cake Cutting: The famous algorithm for fairly cutting a cake in two is for one person to cut the cake in the place that he believes is half and for the other person to choose which "half" he likes. One player may value the icing more while the other the cake more, but it does not matter. The second player is guaranteed to get a piece that he considers to be worth at least a half because he choose between two pieces whose sum worth for him is at least a one. Because the first person cut it in half according to his own criteria, he is happy which ever piece is left for him. Our goal is write an iterative algorithm which solves this same problem for n players.*

To make our life easier, we view a cake not as three dimensional thing, but as the line from zero to one. Different players value different subintervals of the cake differently. To express this, he assigns some numeric value to each subinterval. For example, if player p_i's name is written on the subinterval $[\frac{i-1}{2n}, \frac{i}{2n}]$ of cake then he might allocate a higher numeric value to it, say $\frac{1}{2}$. The only requirement is that the sum total value of the cake is one.

Your algorithm is only allowed the following two operations. In an evaluation query, $v = Eval(p, [a, b])$, the algorithm asks a player p how much v he values a particular subinterval $[a, b]$ of the whole cake $[0, 1]$. In a cut query, $b = Cut(p, a, v)$, the protocol asks the player p to identify the shortest subinterval $[a, b]$ starting at a given left endpoint a, with a given value v. In the above example, $Eval(p_i, [\frac{i-1}{2n}, \frac{i}{2n}])$ returns $\frac{1}{2}$ and $Cut(p_i, \frac{i-1}{2n}, \frac{1}{2})$ returns $\frac{i}{2n}$. Using these the two player algorithm is as follows.

algorithm $Partition2(\{p_1, p_2\}, [a, b])$

$\langle$***pre−cond***$\rangle$: *p_1 and p_2 are players.*
$[a, b] \subseteq [0, 1]$ is a subinterval of the whole cake.

$\langle$***post−cond***$\rangle$: *Returns a partitioning of $[a, b]$ into two disjoint pieces $[a_1, b_1]$ and $[a_2, b_2]$ so that player p_i values $[a_i, b_i]$ at least half as much as he values $[a, b]$.*

```
begin
    $v_1 = Eval(p_1, [a, b])$
    $c = Cut(p_1, a, \frac{v_1}{2})$
    if( $Eval(p_2, [a, c]) \le Eval(p_2, [c, b])$ ) then
        $[a_1, b_1] = [a, c]$ and $[a_2, b_2] = [c, b]$
    else
        $[a_1, b_1] = [c, b]$ and $[a_2, b_2] = [a, c]$
    end if
    return($[a_1, b_1]$ and $[a_2, b_2]$)
end algorithm
```