CMSC 28100-1 / MATH 28100-1
Introduction to Complexity Theory
Fall 2017 – Homework 2
Solution

October 8, 2017

Exercise 1. The *Knapsack Problem* is defined as follows. The input is a set S of $|S| = n$ items, each with a positive integer weight w_i and a non-negative value v_i ($i \in S$), a positive integer knapsack capacity W, and a positive integer target total value V. The output should be "Yes" if there is a subset $S' \subseteq S$ of items such that $\sum_{i\in S'} w_i \leqslant W$ and $\sum_{i\in S'} v_i \geqslant V$ and should be "No" otherwise.

(a) Give an algorithm for the Knapsack Problem. If you give a dynamic programming algorithm, state the precise "English" definitions of your sub-problems, state base cases, and the recurrence. Try to make it as efficient as you can. Prove its correctness and analyze its running time. Does it run in polynomial time?

(b) Define the εW*-Knapsack Problem* for a fixed constant ε as the Knapsack Problem above, except that you have the guarantee that, in the input, we have $\varepsilon W \leqslant w_i$ for every $i \in S$ (i.e., there are no arbitrarily small weights). Can you come up with a polynomial time algorithm for this problem? Or is your algorithm for part (a) already polynomial time for this problem? Why? Analyze. (Note: the expected answer is short and ε is *not* part of the input.)

(c) Define the εV*-Knapsack Problem* for a fixed constant ε as the Knapsack Problem above, except that you have the guarantee that, in the input, we have $\varepsilon V \leqslant v_i$ for every $i \in S$ (i.e., there are no arbitrarily small values). Can you come up with a polynomial time algorithm for this problem? Why? Analyze. (Note: the expected answer is short and ε is *not* part of the input.)

Solution. Before we present the algorithm for item (a), we will prove some properties. First, we assume without loss of generality that $S = \{1, \ldots, n\}$.

First, for $i \in \{1, \ldots, n\}$ and $w \in \{0, \ldots, W\}$, let

$$B(i, w) = \max \left\{ \sum_{j\in J} v_j : J \subseteq \{1, \ldots, i\} \wedge \sum_{j\in J} w_j \leqslant w \right\}.$$

We now prove a small lemma.

Lemma 1.1. *For every $i \in \{1, \dots, n\}$ and $w \in \{0, \dots, W\}$, we have*

$$B(i,w) = \begin{cases} \max\{B(i-1,w), B(i-1,w-w_i)+v_i\}, & \text{if } i \geqslant 2 \text{ and } w \geqslant w_i; \\ B(i-1,w), & \text{if } i \geqslant 2 \text{ and } w < w_i; \\ v_i, & \text{if } i = 1 \text{ and } w \geqslant w_i; \\ 0, & \text{if } i = 1 \text{ and } w < w_i. \end{cases}$$

Proof. The assertions when $i = 1$ are trivial.

Suppose then that $i \geqslant 2$.

Clearly we must have $B(i,w) \geqslant B(i-1,w)$ be the definition of B.

Suppose first that $w < w_i$ and let us prove the other inequality $B(i,w) \leqslant B(i-1,w)$. Let $J \subseteq \{1, \dots, i\}$ be such that $\sum_{j \in J} v_i = B(i,w)$ and $\sum_{j \in J} w_j \leqslant w$. Since $w_i > w$, we know that $i \notin J$, which immediately implies $B(i-1,w) \geqslant \sum_{j \in J} v_j = B(i,w)$ as desired.

Therefore, when $w < w_i$ and $i \geqslant 2$, we have $B(i,w) = B(i-1,w)$.

It remains to prove the case $i \geqslant 2$ and $w \geqslant w_i$. We have already proved that $B(i,w) \geqslant B(i-1,w)$. Let $J \subseteq \{1, \dots, i-1\}$ be such that $\sum_{j \in J} v_i = B(i-1,w-w_i)$ and $\sum_{j \in J} w_j \leqslant w-w_i$. Then we have $\sum_{j \in J \cup \{i\}} w_j \leqslant w$, which implies $B(i,w) \geqslant \sum_{j \in J \cup \{i\}} v_j = B(i-1,w-w_i)+v_i$. Putting this together with the previously proved inequality, we get

$$B(i,w) \geqslant \max\{B(i-1,w), B(i-1,w-w_i)+v_i\}.$$

Let us now prove the other inequality. Let $J \in \{1, \dots, i\}$ be such that $\sum_{j \in J} v_j = B(i,w)$ and $\sum_{j \in J} w_j \leqslant w$.

If $i \in W$, then we have $\sum_{j \in J \setminus \{i\}} w_j \leqslant w-w_i$, which implies $B(i-1,w-w_i) \geqslant \sum_{j \in J \setminus \{i\}} v_j = B(i,w)-v_i$. Hence $B(i,w) \leqslant B(i-1,w-w_i)+v_i$, which implies $B(i,w) \leqslant \max\{B(i-1,w), B(i-1,w-w_i)+v_i\}$.

On the other hand, if $i \notin W$, then $J \subseteq \{1, \dots, i-1\}$, which implies $B(i-1,w) \geqslant \sum_{j \in J} v_j = B(i,w)$, hence $B(i,w) \leqslant \max\{B(i-1,w), B(i-1,w-w_i)+v_i\}$.

Therefore we get

$$B(i,w) = \max\{B(i-1,w), B(i-1,w-w_i)+v_i\},$$

as desired. ◁

Based on Lemma 1.1, since the solution of the problem is simply computing $B(n,W)$ and testing whether it is greater or equal to V. Consider then the following algorithm.

Algorithm 1.1: Knapsack

```
1 for $w \leftarrow 1$ to $w_i$ do $B[1,w] \leftarrow v_1$
2 for $w \leftarrow w_i + 1$ to $W$ do $B[1,w] \leftarrow 0$
3 for $i \leftarrow 2$ to $n$ do
4     for $w \leftarrow 1$ to $w_i$ do $B[i,w] \leftarrow \max\{B[i-1,w], B[i-1,w-w_i]+v_i\}$
5     for $w \leftarrow w_i + 1$ to $W$ do $B[i,w] \leftarrow B[i-1,w]$
6 if $B[n,W] \geqslant V$ then return "Yes"
7 else return "No"
```

Clearly the table $B[i,w]$ in Algorithm 1.1 is filled in non-decreasing order of i. Furthermore, by Lemma 1.1 (and an induction), it follows that $B[i,w]$ is gets the value $B(i,w)$, which implies that Algorithm 1.1 correctly solves the Knapsack Problem.

It is also straightforward to see that the complexity of Algorithm 1.1 is $O(n \cdot W \cdot b_v)$, where b_v is the maximum bitlength among $v_1, \dots, v_n, V$.

Unfortunately, the input size is $O(n \cdot \log W \cdot b_v)$, which implies that Algorithm 1.1 is not polynomial.

Before we solve the other items, consider the following problem.

Let $k \in \mathbb{N}$ be fixed. The *k-Knapsack Problem* is the same as the Knapsack Problem, except that we also require $|S'| \leqslant k$, that is, we want to find a collection of *at most* k items of maximum value, not exceeding the maximum weight (here k is *not* part of the input and is fixed).

Note now that if $|S| = n$, then

$$|\{S' \subseteq S : |S'| \leqslant k\}| = \sum_{i=0}^{k} n^i = O(n^k).$$

This immediately implies that the following algorithm correctly solves the k-Knapsack Problem in time $O(n^{k+1} \cdot b)$, where b is the maximum bitlength of $w_1, \ldots, w_n, v_1, \ldots, v_n, W, V$.

Algorithm 1.2: k-Knapsack

```
1 for $J \subseteq \{1, \ldots, n\}$ with $|J| \leqslant k$ do
2     if $\sum_{j \in J} w_j \leqslant W$ then
3         if $\sum_{j \in J} v_j \geqslant V$ then
4             return "Yes".
5 return "No".
```

We can now solve the other items.

First, note that for item (b), since we are guaranteed that $w_i \geqslant \varepsilon W$, then we know that any $J \subseteq \{1, \ldots, n\}$ with $\sum_{j \in J} w_j \leqslant W$ must have at most $\lfloor 1/\varepsilon \rfloor$ items, which implies that Algorithm 1.2 with $k = \lfloor 1/\varepsilon \rfloor$ correctly solves the εW-Knapsack Problem in time $O(n^{1/\varepsilon} \cdot b)$, which is polynomial since ε is not part of the input.

Note that Algorithm 1.1 of item (a) does not run in polynomial time even with the guarantee that $w_i \geqslant \varepsilon W$.

For item (c), since we are guaranteed that $v_i \geqslant \varepsilon V$, then we know that if there exists $J \subseteq \{1, \ldots, n\}$ with $\sum_{j \in J} w_j \leqslant W$ and $\sum_{j \in J} v_j \geqslant V$, then any subset $J' \subseteq J$ with $|J'| \geqslant 1/\varepsilon$ must also satisfy $\sum_{j \in J'} w_j \leqslant W$ and $\sum_{j \in J'} v_j \geqslant V$. This implies that Algorithm 1.2 with $k = \lfloor 1/\varepsilon \rfloor$ correctly solves the εV-Knapsack Problem in time $O(n^{1/\varepsilon} \cdot b)$, which is polynomial. $\triangleleft$

Exercise 2. We are given an array $A[1..n]$ of n rational numbers with $n \geqslant 3$ and the special property that $A[1] \geqslant A[2]$ and $A[n-1] \leqslant A[n]$. We say that an element $A[x]$ is a *local minimum of* A if $A[x-1] \geqslant A[x]$ and $A[x] \leqslant A[x+1]$. First, note that, given the condition $A[1] \geqslant A[2]$ and $A[n-1] \leqslant A[n]$, the array A must have at least one local minimum, and it is trivial to find it in $O(b \cdot n)$ time, where b is the largest bitlength of an element of A. Give an algorithm to find and return a local minimum of A in $O(b \cdot (\log n)^2)$ time. Analyze the running time of your algorithm. Prove the correctness of your algorithm.

Solution. Consider the following algorithm.

Algorithm 2.1: Local minimum

```
1 if $n = 3$ then return 1.
2 if $n = 4$ then
3     if $A[2] \leqslant A[3]$ then return 2.
4     return 3.
5 $i \leftarrow \lceil n/2 \rceil$.
6 if $A[i-1] < A[i]$ then return Localminimum($A[1..i]$).
7 if $A[i] > A[i+1]$ then return $i - 1 +$ Localminimum($A[i..n]$).
8 return $i$.
```

First note that if $n \geqslant 5$, then we must have $3 \leqslant \lceil n/2 \rceil \leqslant n-2$, which implies that the recursive calls effectively reduce the size of the array and yield an array of size at least 3. Furthermore, the conditions for the recursive calls ensure that the array A' passed recursively satisfies $A'[1] \geqslant A'[2]$ and $A'[n'-1] \leqslant A'[n']$ (where n' is the size of A').

It is easy to see that Algorithm 2.1 correctly solves the problem if $n = 3$ or $n = 4$.

On the other hand, if $n \geqslant 5$ and we assume by induction that Algorithm 2.1 correctly solves the problem for arrays of size strictly less than n, then we have two cases.

In the first case, the point $i = \lceil n/2 \rceil$ is a local minimum and clearly the algorithm correctly finds it (the last two "ifs" test false).

In the second case, the point $i = \lceil n/2 \rceil$ is not a local minimum, so we must either have $A[i-1] < A[i]$ or $A[i] > A[i+1]$. The first implies that A has a local minimum in $2, \ldots, i-1$, which, by inductive hypothesis, is correctly found by the recursive call. The latter implies that A has a local minimum in $i+1, \ldots, n$, which by inductive hypothesis (and noting that we correctly adjust the index returned), is correctly found by the recursive call.

Therefore, Algorithm 2.1 correctly solves the problem.

Let us now analyze the time complexity of the algorithm. Let $T(n, b)$ be the time complexity of Algorithm 2.1 when A has n entries, each with bitlength at most b. By inspecting the algorithm (and since $n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$), it is clear that we have

$$T(n,b) \leqslant \max\left\{T\left(\left\lceil \frac{n}{2} \right\rceil, b\right), T\left(\left\lfloor \frac{n}{2} \right\rfloor + 1, b\right)\right\} + O(b \log n) \tag{1}$$

whenever $n \geqslant 5$.

(We will solve this recursion completely formally. Usually one handwaves some of what follows.)

Consider the function $U(n, b)$ defined inductively by letting

$$U(3,b) = U(4,b) = \max\{T(3,b), T(4,b)\}$$

and

$$U(n,b) = \max\left\{U\left(\left\lceil \frac{n}{2} \right\rceil, b\right), U\left(\left\lfloor \frac{n}{2} \right\rfloor + 1, b\right)\right\} + C \cdot b \log n,$$

for $n \geqslant 5$, where $C > 0$ is a constant such that the function g that is $O(b \log n)$ in (1) satisfies $g(n) \leqslant Cb \log_2(n-1)$ for every $n \geqslant 3$ (this is possible since $O(b \log n)$ is the same as $O(b \log_2(n-1))$).

We claim that $T(n,b) \leqslant U(n,b)$ for all $n \geqslant 3$ and all $b \geqslant 1$. This is clearly true if $n \leqslant 4$.

Suppose then that $n \geqslant 5$ and that $T(m,b) \leqslant U(m,b)$ for every $m < n$ by induction. Then since $3 \leqslant \lceil n/2 \rceil \leqslant \lfloor n/2 \rfloor + 1 \leqslant n-1$, equation 1 implies

$$T(n,b) \leqslant \max\left\{U\left(\left\lceil \frac{n}{2} \right\rceil, b\right), U\left(\left\lfloor \frac{n}{2} \right\rfloor + 1, b\right)\right\} + C \cdot b \log_2(n-1) = U(n,b),$$

as desired.

Therefore $T(n,b) \leqslant U(n,b)$ for all $n \geqslant 3$ and all $b \geqslant 1$. In particular, we have $T = O(U)$.

We now claim that U is non-decreasing in the first coordinate. We clearly have $U(3,b) \leqslant U(4,b)$. Suppose then that $n \geqslant 5$ and that $U(m-1,b) \leqslant U(m,b)$ for every $m < n$ by induction. Then since $3 \leqslant \lceil n/2 \rceil \leqslant \lfloor n/2 \rfloor + 1 \leqslant n-1$, we have

$$\begin{aligned} U(n,b) &= \max\left\{U\left(\left\lceil \frac{n}{2} \right\rceil, b\right), U\left(\left\lfloor \frac{n}{2} \right\rfloor + 1, b\right)\right\} + C \cdot b \log_2(n-1) \\ &\geqslant \max\left\{U\left(\left\lceil \frac{n-1}{2} \right\rceil, b\right), U\left(\left\lfloor \frac{n-1}{2} \right\rfloor + 1, b\right)\right\} + C \cdot b \log_2(n-2) \\ &= U(n-1,b), \end{aligned}$$

since $\log_2$ is increasing.

Therefore $U(n-1,b) \leqslant U(n,b)$ for every $n \geqslant 4$ and every $b \geqslant 1$.

Let us now prove by induction in $k \geqslant 1$ that $U(2^k+1,b) \leqslant U(3,b) + C \cdot b \cdot k^2$.

For $k = 1$, this is trivial. So suppose $k \geqslant 2$ and that the result holds for $k-1$. Then we have

$$\begin{aligned} U(2^k+1,b) &= \max\{U(2^{k-1}+1,b), U(2^{k-1}+1,b)\} + C \cdot b \cdot \log_2(2^k) \\ &\leqslant U(3,b) + C \cdot b \cdot (k-1)^2 + C \cdot b \cdot k \\ &\leqslant C \cdot b \cdot k^2, \end{aligned}$$

as desired.

We now claim that $U(n,b) = O(b \cdot (\log n)^2)$.

First note that for $n = 2^k+1$ for some $k \geqslant 1$, we have already proved $U(n,b) \leqslant C \cdot b \cdot (\log n)^2$.

On the other hand, if $n \geqslant 3$ is not of the form 2^k+1 for some $k \geqslant 1$, then there at least exists $k \geqslant 1$ such that $n \leqslant 2^k + 1 \leqslant 2n$. Since U is non-decreasing in the first coordinate, we get

$$U(n,b) \leqslant U(2^k+1,b) \leqslant C \cdot b \cdot k^2 \leqslant C \cdot b \cdot (\log(2n-1))^2.$$

Therefore $U(n,b) = O(b \cdot (\log n)^2)$, which implies $T(n,b) = O(b \cdot (\log n)^2)$. ◁

Exercise 3. A *cycle* in an undirected simple graph $G = (V,E)$ is a sequence of vertices

$$(v_0, v_1, \ldots, v_k)$$

with $v_0 = v_k$, $k \geqslant 3$ and $\{v_i, v_{i+1}\} \in E(G)$, $v_i \neq v_{i+1}$ and $v_i \in V(G)$ for every $i \in \{0,1,\ldots,k-1\}$.

In this exercise, we consider that access to a graph G with vertex set $V(G) = \{1,\ldots,n\}$ is given by the following subroutines.

- The subroutine vertices() returns the number of vertices n in time $O(\log n)$ (which is the bitlength of n).
- The subroutine nextneighbor(i,j) returns the smallest $k \geqslant j$ such that $\{i,k\} \in E(G)$ and returns "NONE" if no such k exists. This subroutine also takes time $O(\log n)$.

Design an algorithm that, using the subroutines above, runs in time $O(n \log n)$ and returns a cycle of the graph G if one exists and returns "ACYCLIC" if G does not contain any cycle. Prove the correctness of your algorithm.

Observation: the graph may have much more than $O(n)$ edges (it can have up to $\Omega(n^2)$ edges), so your algorithm cannot inspect all edges of the graph. Note also that your algorithm can only make at most $O(n)$ calls to the subroutines above.

Solution. Let us suppose that $V(G) = \{1, \ldots, n\}$ and consider the following algorithm.

Algorithm 3.1: Detect cycle

```
1  $n \leftarrow$ vertices().
2  for $i \leftarrow 1$ to $n$ do $V[i] \leftarrow 0$
3  Let $S$ be an empty stack.
4  for $s \leftarrow 1$ to $n$ do
5      if $V[s] = 0$ then
6          Push $(s, 1)$ to $S$.
7          $V[s] \leftarrow 1$.
8          while $S$ is not empty do
9              Pop the top element $(v, i)$ from $S$.
10             $i \leftarrow$ nextneighbor$(v, i)$.
11             if $S$ is non-empty and $i = z$ for the top element $(z, k)$ of $S$ then
12                 $i \leftarrow$ nextneighbor$(v, i)$.
13             if $i \neq$ "NONE" then
14                 if $V[i] = 1$ then
15                     $w_1 \leftarrow v$
16                     $k \leftarrow 2$
17                     while The top element of $S$ is not $(i, j)$ for some $j$ do
18                         Pop the top element $(w, j)$ of $S$.
19                         $w_k \leftarrow w$
20                         $k \leftarrow k + 1$
21                     $w_k \leftarrow i$
22                     return $(w_1, w_2, \ldots, w_k, v)$
23                 else
24                     Push $(v, i + 1)$ to $S$.
25                     Push $(i, 1)$ to $S$.
26                     $V[i] \leftarrow 1$
27 return "ACYCLIC".
```

Let us prove the correctness of the algorithm above.

First we claim thatif the elements of the stack S are $((v_1, i_1), \ldots, (v_k, i_k))$ at some point, then $\{v_j, v_{j+1}\} \in E(G)$ for every $j \in \{1, \ldots, k-1\}$.

This is clearly true if S is empty or has at most one element, so this holds when the algorithm enters the loop of line 8 for the first time. Note that the only other place in which elements are pushed on S is in the else line 23, in this case, the algorithm popped (v, i) in line 9 and ensured through lines 10, 11 and 13 that i is changed to a value such that $\{v, i\} \in E(G)$. In the else of line 23, the algorithm then pushes (v, i) then $(i, 1)$. If (w, j) is the element before (v, i) in S, we have $\{v, w\} \in E(G)$ by induction (before v was popped from S) and we proved that $\{v, i\} \in E(G)$.

Therefore the claim holds.

Let us call a pair (w, j) a representative of v if $w = v$.

We claim that for every $v \in V(G)$, at all points the stack S can have at most one representative of v.

First note that for the push operation of line 24 to happen, the pop operation of line 9 must have happened. This means that line 24 can only replace representatives of v, but not add them. Furthermore, it always replaces them with a representative (v, j) with $j \geqslant 2$. Hence, the only form in which a representative of v enters S without replacing one is if it is of the form $(v, 1)$. But for $(v, 1)$ to be pushed on S, we must have had $V[v] = 0$ and it is immediately set to 1 afterward, so it can be pushed on S at most once.

Therefore at all points the stack S can have at most one representative of v.

Note also that since when we replace a representative (v, j) of v by another (v, r) on line 24, we always have $r > j$, so it follows that each pair $(v, j) \in V(G) \times V(G)$ can enter S at most once.

Note furthermore that the only pairs that can be pushed on S are of the form $(v, i+1)$ with either $\{v, i\} \in E(G)$ or $i = 0$.

Let us say that v enters S when some representative of v enters S. Let us say that v leaves S when line 13 tests false in the same iteration when line 9 popped a representative of v (this way we do not count when representatives of v are replaced).

We claim that for every $v \in V(G)$, if v leaves S, then each neighbor of v must have entered S in some iteration before t. Consider all the representatives of v that enter S in order $(v, i_0), (v, i_1), \dots, (v, i_\ell)$.

We know that $i_0 = 1$ and $\{v, i_r - 1\} \in E(G)$ for all $r \in \{1, \dots, \ell\}$. But also note that in the iteration when (v, i_r) is pushed on S we also push $(i_r, 1)$ on S, the pair (v, i_{r-1}) must have been popped, and for every $i_{r-1} \leqslant k < i_r$, we either have $\{v, k\} \notin E(G)$ or k is was in the top of S after the pop of line 9.

Therefore all neighbors of v must enter S before v leaves S.

Let us now prove that if line 14 tests true (i.e., we have $V[i] = 1$), then there must exist a representative of i in S.

First note that line 14 can test true only at most once since it has a return statement in its block. Since $V[i] = 1$, we know that at some point i must have entered S. But in the iteration when line 14 tests true, line 13 must have also tested true, which means that v has not yet left S and since i is a neighbor of v, it follows that i also has not yet left S. Therefore there must exist a representative of i in S.

Note that this implies that the loop of line 17 actually ends. This along with the fact that each pair (v, i) can enter S at most once implies that the algorithm always halts.

Note also that if S before this loop was $(v_1, i_1), \dots, (v_r, i_r)$ with $i = i_\ell$ and the top being (v_1, i_1), then by the end of the loop we have $k = \ell - 1$ and $v_s = w_{s+1}$ for all $s \in \{1, \dots, k\}$, which implies (by our first claim) that $(w_1, w_2, \dots, w_k, v)$ is a cycle of G since $w_1 = v$, $w_k = i$ and $\{v, i\} \in E(G)$.

Therefore, if the algorithm returns a cycle, then it is indeed a cycle in G.

Define now the graphs G_t inductively as follows. We first let G_0 be the graph without any vertices and we follow the execution the execution of Algorithm 3.1 and each time some vertex i enters S, we define G_{t+1} from G_t by adding i to $V(G_t)$ and if v entered S in line 25, we also add the edge $\{v, i\}$ to G_{t-1}.

It is easy to see that at any point the vertices of G_t are precisely the $v \in V(G)$ such that $V[v] = 1$. Furthermore, since each edge added is an edge of G, the graph G_t is always a subgraph of G. Also, each edge added is incident to a new vertex, so G_t does not have any cycles.

But since the maximum number of edges an acyclic graph on n vertices can have is $n - 1$, it follows that we only ever define G_t for $t \leqslant n - 1$. On the other hand, on each iteration of the while of line 8, either a vertex leaves S or a vertex enters S, so it must run a total of at most $2n - 2$ times, each of which has time complexity $O(\log n)$, except if the loop of line 17 runs. But the loop of line 17 runs at most once in total, and it has complexity $O(n \log n)$ (as the stack never has more than n elements). The loop of line 4 clearly runs n times.

Therefore, we get that the algorithm has time complexity of $O(n \log n)$.

It remains to prove that if the algorithm returns "ACYCLIC", then G does not have any cycle.

Suppose that the algorithm returns "ACYCLIC", then the test of line 14 is always false.

From the first lines of the loop of line 4, it follows that every vertex must eventually enter S, hence must eventually enter some G_t. On the other hand, since the loop of line 8 ends without line 14 testing true, it follows that every vertex must eventually leave S.

We claim that every edge of G is in some G_t.

Fix some edge $\{u, w\} \in E(G)$ and suppose u enters S before w. Consider all the representatives of w that enter S in order $(w, i_0), (w, i_1), \dots, (w, i_\ell)$ and let $r \geqslant 0$ be the maximum such that $i_r \leqslant u$.

Consider the iteration when (w, i_r) is popped from S. At this point u is in S since it entered before w and can only leave after w leaves (since the representatives of w keep getting replaced and the representative of u is further down on the stack). Since the test of line 14 is false and $\mathrm{nextneighbor}(w, i_r) = u$, it follows that the test of line 11 must have been true, which implies that in the beginning of the iteration the two top elements of S where (w, i_r) and a representative of u. This implies that $(w, i_0) = (w, 1)$ must have been pushed on line 25 in an iteration when $v = u$ and $i = w$, which implies that we added the edge $\{u, w\}$ to some G_t.

Let T be the maximum such that we defined G_T. Since all edges of G are eventually added to some G_t, it follows that $G_T = G$, but G_T does not have any cycle, so G does not have any cycle as desired. $\triangleleft$