Cycle Detection: Brent's Algorithm

Nitin Verma
mathsanew.com

August 27, 2021

In an earlier article titled *Cycle Detection: Floyd's Algorithm* [1], the generic cycle detection problem was introduced, and it was solved via Floyd's algorithm. Now we will discuss the *Brent's Cycle Detection Algorithm* for this problem, named after R. P. Brent [2]. Refer to page 1 of article [1] for the problem definition and related notations.

The Algorithm

First we observe that, there must exist integer r such that e_r belongs to the cycle and e_r equals at least one element among its next r elements $(e_{r+1}, e_{r+2}, \ldots, e_{r+r})$. Specifically, any integer r satisfies this condition if and only if:

(a) $r \geq l$ (e_r belongs to the cycle) and,

(b) $r \geq n$ (e_r equals any of its next r elements).

Note that, ($r \geq l$ and $r \geq n$) can also be written as $r \geq \max(l, n)$.

So, for any given l and n, all integers $\max(l, n)$ onwards satisfy the criteria for r. The Brent's algorithm attempts to find r which is the minimum power of 2, 2^p (p is a non-negative integer), such that $2^p \geq \max(l, n)$. Now onwards, we will use r to denote this specific integer 2^p.

By finding r and locating e_r, the algorithm will have located some element in the cycle (since e_r belongs to the cycle), and will then proceed to find l and n.

The algorithm works as follows. It sequentially checks if the candidates $r' = 2^0, 2^1, 2^2, \ldots$, satisfy the criteria for the desired r. That is, whether $e_{r'}$ equals any of its next r' elements or not. To do that, it maintains two references M (mark) and R. For each r', M is kept at $e_{r'}$ while R steps through the next r' elements comparing them with M ($e_{r'}$). Note that this stepping of R will finally bring it at $e_{r'+r'} = e_{2r'}$. So, for checking the next candidate r', which is $2r'$, this reference R at $e_{2r'}$ can be used to set the mark M for that next candidate.

Note that a candidate r' does not satisfy the criteria if, either (a) $e_{r'}$ is outside the cycle ($r' < l$), or (b) $e_{r'}$ is inside the cycle but is distinct from its next r' elements ($r' < n$).

Following is an implementation of this algorithm in C. The above described part of this method will be referred as "Cycle-Searching" (the other part "Find l" will be discussed below). Whenever a reference, say R, is updated to $f(R)$, we will call it one "step" taken by R.

Note that when the desired r is reached, the value of n can be found by counting the steps R has taken after e_r to reach the nearest element equaling e_r.

```
/* Parameter f is a function pointer.
   Output values of n and l are returned via pointers pn and pl.

   The elements e{i} are referred using type "void *". So,
   function f has input and output type as "void *". */

void brent(void *e0, void* f(void*), int *pn, int *pl)
{
  void *R, *M, *R0;
  int i, r, cycle_found, count, j, n, l;

  /***** Cycle-Searching *****/

  /* initializations for the loop below */
  M = f(e0);
  r = 1;
  R = f(M);
  i = 2;
  cycle_found = (R == M);

  /* outer-loop invariants:

       1. M = e{r}

       2. R = e{i}

       3. (cycle_found AND (i = r + n) AND (R = M)) OR
          (!cycle_found AND (i = 2r)) */
```

```
while(!cycle_found)
{
  r = 2*r;

  M = R;

  /* Now, M = R = e{r}. R will take (upto) r steps till e{2r} */

  /* inner-loop invariant: R = e{i} */

  while(i < 2*r && !cycle_found)
  {
    R = f(R);
    i = i + 1;
    cycle_found = (R == M);
  }
}

n = i - r;

/***** Find l *****/

/* (M = e{r}) holds */

if(r%n != 0)
{
  j = r + n - r%n;
  i = r;

  while(i < j)
  {
    M = f(M);
    i = i + 1;
  }

  /* (M = e{j}) holds */
}

/* (M = e{multiple-of-n}) holds */
R0 = e0;
count = 0;

while(R0 != M)
{
  R0 = f(R0);
  M = f(M);
  count++;
}
```

```
  /* (R0 = M = e{l}) holds */

  l = count;

  *pn = n;
  *pl = l;
}
```

This algorithm finds minimum power of 2, $r = 2^p$, such that $2^p \geq \max(l, n)$. Since 2^p is the minimum possible, we must have $2^{p-1} < \max(l, n)$, which can also be written as $2^p < 2 \cdot \max(l, n)$. So,

$$r < 2 \cdot \max(l, n)$$

Also, this algorithm (the cycle-searching part) steps R total $r + n$ times. So, the number of calls to f performed by it is upper-bounded by:

$$2 \cdot \max(l, n) + n$$

Finding l

We initialize two references R_0 and R_n at e_0 and e_n respectively. R_n can be placed at e_n by stepping n times from e_0. Now, after l steps, R_0 will be at index l and R_n will be at index (due to equation (1) in [1]):

$$l + (n + l - l) \bmod n = l$$

So, to find l, we can iteratively step R_0 and R_n till they meet, while counting the steps. This step count will be l.

Alternatively, we can use another more efficient approach. When the cycle-searching loop terminates, M is at e_r. We will first place M at e_j, where j is a multiple of n. If $n \mid r$, we are done with $j = r$. Otherwise, we use the fact that $(r - r \bmod n)$, and hence $(r + n - r \bmod n)$, is always a multiple of n. So, we step M $(n - r \bmod n)$ times to bring it at e_j with $j = (r + n - r \bmod n)$.

Now, initialize a reference R_0 at e_0. After l steps, it will be at index l. Also, M, which is at e_j, after l steps will be at index (due to equation (1) in [1]):

$$l + (j + l - l) \bmod n = l + j \bmod n = l \quad \{\text{since } n \mid j\}$$

So, to find l, we can iteratively step R_0 and M till they meet, while counting the steps. This step count will be l.

The "Find l" part in method *brent*() implements this approach.

The earlier approach steps R_0 and R_n l and $n + l$ times respectively. This approach steps R_0 l times, but steps M l or $(n - r \bmod n) + l$ times (based on $n \mid r$ or not). Note that, $(n - r \bmod n) < n$ when $n \nmid r$.

$\blacksquare$

References

[1] Nitin Verma. *Cycle Detection: Floyd's Algorithm.* `https://mathsanew.com/articles/cycle_detection.pdf` (2021).

[2] R. P. Brent. *An Improved Monte Carlo Factorization Algorithm.* BIT Numer. Math., Vol 20 (1980), 176–184. `https://maths-people.anu.edu.au/~brent/pd/rpb051i.pdf`.