Multi-Collision Search using Cycle Detection Algorithms

Design Lab Project
Autumn Semester, 2022-23
by Rahul Aditya (18CS30032)

Under the supervision of
Prof. Somindu Chaya Ramanna

Abstract

This paper presents an algorithm for collision search that could very well be extended to solve multi-collision search problem. The algorithm uses a stack-based cycle-detection algorithm as a precursor. The stack based cycle detection algorithm has traditionally been known to provide time-memory trade-off. Performance reaches close to the theoretically best performance when multiple stacks are used in the precursor algorithm at the cost of increased memory usage. Our algorithm for collision search problem can also be parallelized in the final step. Average case time complexity has been the sole parameter of analysis in this paper, which is another reason why the stack-based algorithm has been used for cycle detection.

1 Introduction

Several cycle detection algorithms have been proposed and each of them had its objectives. Formally, the **cycle detection problem** is defined as the following:

Given a finite set D, an initial point $x_0 \in D$ and a function $f : D \to D$ defining the sequence $\{x_i\}$ as $x_i = f(x_{i-1})$ for $i \geq 1$, the cycle detection algorithm requires to find two integers $i \neq j$ such that $x_i = x_j$. The algorithm must also output the cycle length.

It is to be noted that for a finite set D, the sequence $\{x_i\}$ induces a prefix followed by a cycle. The prefix consists of distinct integers $x_0, x_1, ..., x_\mu$, where μ is the length of the prefix. Similarly, the cycle consists of distinct integers $x_\mu, x_{\mu+1}, ..., x_{\mu+\lambda-1}$. For any $i \geq \mu$, $x_i = x_{i+\lambda}$.

Several algorithms have been proposed to solve the cycle detection problem. In this paper, Nivasch's stack algorithm has been used for cycle detection [1]. This is because it is one of the most efficient algorithms for cycle detection in terms of average time complexity. The memory complexity of Nivasch's stack algorithm is also logarithmic. Sedgewick's algorithm aims to optimize worst-case time complexity [2]. The distinguished point algorithm [3], another cycle detection algorithm, requires less memory but may fail to halt. A uniform partitioning technique applied to Nivasch's stack algorithm offers a time-memory tradeoff. This is why Nivasch's stack algorithm is our choice for the cycle detection algorithm.

In addition to cycle detection, extra effort has to be made for collision search. This paper presents one such approach that is efficient in terms of average case time complexity. Formally, the **collision search** problem is defined as the following:

Given a finite set D, an initial point $x_0 \in D$ and a function $f : D \to D$ defining the sequence $\{x_i\}$ as $x_i = f(x_{i-1})$ for $i \geq 1$, the collision search problem requires to find two points $x \neq y$ such that $f(x) = f(y)$. In relation to the sequence $\{x_i\}$ induced by the function f described above, the collision search algorithm should output point $x_{\mu-1}$ and $x_{\mu+\lambda-1}$.

2 Nivasch's Stack Algorithm

A stack of pairs (x_i, i) is maintained such that all x_i's and i's on the stack form an increasing sequence. The stack is initially empty. At each step j, we look at the topmost element, (x_i, i), of the stack. If $x_i = x_j$, then the cycle length is $j - i$. If $x_i < x_j$, then (x_j, j) is pushed on top of the stack. If $x_i > x_j$, then (x_i, i) is popped off the stack and the condition is checked again with the then topmost element of the stack until either one of the two conditions is satisfied or the stack becomes empty.

Theorem 1. The algorithm halts on the smallest value on top of the stack, at some time in $[\mu + \lambda, \mu + 2\lambda)$.

Proof. Let the smallest element in the induced cycle be x_{min}. A cycle cannot be detected until all the elements are traversed once. Once, x_{min} is pushed on the stack, it cannot be popped as no element to be pushed on the stack is less than x_{min} because only the cycle elements would henceforth be pushed. Hence, the cycle is detected in the second iteration of the cycle when x_{min} is detected again. If x_{min} lies at the start of the cycle as x_μ then the algorithm halts in $\mu + \lambda$ steps. In the otherwise extreme case, if x_{min} lies at the end of the cycle as $x_{\mu+\lambda-1}$ then the algorithm halts in $\mu + 2\lambda - 1$ steps.

Theorem 2. The average case time complexity of the algorithm is $\mu + \frac{3}{2}\lambda$ if the function f is a deterministic function chosen at random from among all possible functions having the same domain and codomain.

Proof. If the function f is random, then x_{min} is equally likely to occupy any of the λ positions in the cycle. Hence, the average case time complexity of the algorithm is the average of time steps $[\mu + \lambda, \mu + 2\lambda)$ derived in **Theorem 1**. Hence, the average case time complexity is $\mu + \frac{3}{2}\lambda$.

Theorem 3. For some time step $n < \mu + \lambda$, the expected stack size $E[S_n] = O(log(n))$.

Proof. Let X_0, X_1, X_n be indicators variables such that $\forall i = 1, 2, 3, ..., n, X_i = 1$ if (x_i, i) is present on the stack otherwise $X_i = 0$. Consider the stack at time $t = n$.

X_n is always equal to 1 because the last element x_n must be present on the stack. Hence, $P[X_n = 1] = 1$. For any x_i to be present on the stack, x_i must be less than each of $x_{i+1}, x_{i+2}, ..., x_n$. Since the function is random, we can reduce this problem to the following:

Given a permutation of length $n - i + 1$ where each element $x = 1, 2, 3, ..., n - i + 1$ in the permutation corresponds to $i, i+1, ..., n$, find the probability that $x = 1$ occurs at the initial location when the permutation is shuffled totally at random.

The total number of ways of permuting a list of $n - i + 1$ elements is $(n - i + 1)!$. The favorable cases correspond to fixing 1 at the initial position and permuting the rest of the $n - i$ elements. This can be done in $(n - i)!$ ways. Hence, $P[X_i = 1] = \frac{(n-i)!}{(n-i+1)!} = \frac{1}{n-i+1}$. $E[X_i]$ can be derived as follows:

$E[X_i] = P[X_i = 1] * 1 + P[X_i = 0] * 0$
$\Rightarrow E[X_i] = P[X_i = 1]$
$\Rightarrow E[X_i] = \frac{1}{n-i+1}$

The stack size S_n can be written as $S_n = X_0 + X_1 + X_2 + ... + X_n$
$\Rightarrow E[S_n] = E[X_0 + X_1 + X_2 + ... + X_n]$
$\Rightarrow E[S_n] = E[X_0] + E[X_1] + E[X_2] + ... + E[X_n]$
$\Rightarrow E[S_n] = \frac{1}{n+1} + \frac{1}{n} + ... + \frac{1}{1}$
$\Rightarrow E[S_n] = H(n + 1) = O(log(n))$

Theorem 4. The stack size S_n is almost surely less than $rlog(n)$ for any constant $r > e$.

Proof. Let the expectation of stack size S_n be represented by E. By Chernoff's bound, $P[S_n >= rE] \leq (\frac{e^{r-1}}{r^r})^E \; \forall r \geq 1$. The RHS could be solved in the following way:

$(\frac{e^{r-1}}{r^r})^E = (\frac{e}{r})^E * e^{-E}$. For $r > e$, $\frac{e}{r} < 1$, and hence $(\frac{e}{r})^E * e^{-E} < e^{-E}$. Further, according to **Theorem 3**, $E = O(log(n))$. This implies that $(\frac{e}{r})^E * e^{-E} < O(\frac{1}{n})$. Hence, $P[S_n >= rE] < O(\frac{1}{n})$.

For $n \to \infty$, the RHS being bounded by $O(\frac{1}{n}) \to 0$. In other words, the probability that the stack size is greater than or equals rE for all $r > e$ is nearly 0. Hence, the stack size, S_n is almost surely less than $rlog(n)$ for all $r > e$.

2.1 Performance under a random function

It has been proved before that under a random function f operating on a finite set D whose size is n such that n is large and the initial value x_0 is chosen at random from the set D, the joint probability distribution function of μ and λ is approximately given by [4]:

$$w(\mu, \lambda) = \frac{1}{n} e^{-\frac{(\mu+\lambda)^2}{2n}}$$

Each of μ and λ has expectation $\sqrt{\frac{\pi n}{8}}$ [4].

2.2 Multi-stack algorithm

Extending the idea of maintaining just one stack for finding x_{min} in the stack algorithm, we can divide the set D into k disjoint classes and maintain a stack for each of the classes, preferably making sure that each of them has the same size [1]. For example, k could be a power of 2, and each x_i is assigned to one of the classes $0, 1, 2, ..., (k-1)$ according to the value x_i mod k. This algorithm works because each x_i, when encountered again, is assigned the same class as before and hence the cycle length that would be given as output would still be the same. In addition to this, the run time per step practically remains unchanged as every time step, the operations would be done on a single stack.

Each class $j, 0 \leq j < k$, has it's own minimum $x_{min,j}$ instead of just one x_{min} as in the conventional stack algorithm. This gives the algorithm to halt early as it stops as soon as $x_{min,j}$ is encountered the second time for any of the k classes. Assuming that each $x_{min,j}$ occurs uniformly at random in the cycle, the average running time complexity reduces to $\mu+\lambda(1+\frac{1}{k+1})$. However, the memory complexity increases from $O(log(\mu+\lambda))$ to $O(klog(\mu+\lambda))$.

Hence, the Multi-stack algorithm offers a time-memory tradeoff. If we use $O(klog(\mu+\lambda))$ memory, the algorithm halts at time $\frac{\lambda}{k}$ above the absolute minimum $\mu + \lambda$.

3 Collision Search

This section describes the algorithm for finding a collision pair by making use of Nivasch's Stack algorithms. Specifically, concerning the sequence $\{x_i\}$ induced by the hash function f, the described algorithm aims to find $x_{\mu-1}$ and $x_{\mu+\lambda-1}$.

3.1 Algorithm for Collision Search

The stack algorithms, whether it uses single-stack or multi-stack, output two integers i, j such that $i \neq j$ and $x_i = x_j$. This gives the cycle length λ directly as $\lambda = j - i$. For collision pair search, maintain two values $a = x_0$, and $b = x_i$. The algorithm can be divided into the following two parts:

Sequential part. If $i > \lambda$, apply the hash function f on $a = x_0,\ (i - \lambda)$ times. Otherwise, apply the hash function f on $b = x_i,\ (\lambda - i)$ times.

Parallel part. Keep applying the hash function f on a and b simultaneously until $f(a) = f(b)$. When the algorithm halts, output the collision pair, a and b.

It is to be noted that the **Sequential part** of the algorithm cannot be parallelized but the **Parallel part** of the algorithm, if properly synchronized, can be done on two processors in parallel.

3.2 Correctness Proof

The **Sequential part** of the algorithm involves moving one of a or b until $a = x_0$ is as equidistant from x_μ as that of $b = x_i$ from $x_{\mu+\lambda}$. The number of times hash function f must be applied to a such that a moves from x_0 to x_μ are $\mu + 1$ (Let us name it A). On the other hand, the number of times hash function f must be applied to b such that b moves from x_i to $x_{\mu+\lambda}$ is $(\mu + \lambda - i + 1)$ (Let us name it B). To be able to make sure that both hit the same point in **Parallel part** on applying the hash function f the same number of times, we will have to look at the difference of A and B. The difference $B - A = \lambda - i$.

If $i > \lambda$ then $A > B$. In other words, value $a = x_0$ is farther from the common hash value than the value $b = x_i$. Hence, function f applies on a, $(i - \lambda)$ times. Otherwise, $b = x_i$ is farther from the common hash value than the value $a = x_0$. Hence, in this case, function f applies on b, $(\lambda - i)$ times.

The correctness part is ensured as the two values, a and b, are now equidistant from the common hash value, x_μ. Both the pointers are hence moved simultaneously until $f(a) = f(b)$, at which time $a = x_{\mu-1}$ and $b = x_{\mu+\lambda-1}$.

3.3 Average Case Time Complexity

Under the assumption that the hash function $f : D \to D$ is chosen at random among all possible hash functions, the average case time complexity for the single-stack cycle detection algorithm is $\mu + \frac{3}{2}\lambda$.

For calculating the average case time complexity for searching the collision pair, note that under the same assumption, x_{min} occurs uniformly at random in the cycle. This makes the average number of times the function f be applied to value b to be $\frac{\lambda}{2}$. μ is the exact number of times that the hash function f be applied to $a = x_0$ to reach x_μ. Hence, the average case time complexity for finding the collision search is $\mu + \frac{\lambda}{2}$.

The total average-case time complexity is thus the sum of average-case time complexities for cycle detection and collision search which is $(\mu + \frac{3}{2}\lambda) + (\mu + \frac{\lambda}{2}) = 2(\lambda + \mu)$. Under the assumptions stated in section **2.1**, expected values for λ and μ are both $\sqrt{\frac{\pi n}{8}}$. Hence, the average-case time complexity is $\sqrt{2\pi n}$. This time complexity can be further reduced considerably if we make use of the multi-stack cycle detection algorithm.

3.4 Extension to $k = 3$ collision search problem

In essence, we can think of the described algorithm that gives a collision pair (x, y) when given an initial value to start with. The use of the described algorithm reduces the $k = 3$ collision search problem to the typical **Birthday Paradox** problem.

For each iteration i, choose an initial value x_i^{init} at random and use the described algorithm to get a collision pair (x_i, y_i) such that $x_i \neq y_i$ and $f(x_i) = f(y_i)$. This process is repeated for k iterations such that for some $j < k$, one or many of the following conditions are satisfied:

$$x_j = x_k \ or \ y_j = x_k \ or \ x_j = y_k \ or \ y_j = y_k$$

Since the **Birthday Paradox** problem is expected to get a hit by sampling $O(\sqrt{n})$ values, our algorithm can solve the $k = 3$ collision search problem by generating $O(\sqrt{n})$ collision pairs.

Alternatively, we can make use of generic algorithms for 3-collisions by making use of algorithms that generalize multi-collision problems by using parallel processors for speedup. [5]

References

(1) Nivasch, G. Cycle detection using a stack. *Information Processing Letters* **2004**, *90*, 135–140.

(2) Sedgewick, R.; Szymanski, T. G.; Yao, A. C. The complexity of finding cycles in periodic functions. *SIAM Journal on Computing* **1982**, *11*, 376–390.

(3) Van Oorschot, P. C.; Wiener, M. J. Parallel collision search with cryptanalytic applications. *Journal of cryptology* **1999**, *12*, 1–28.

(4) Brent, R. P. An improved Monte Carlo factorization algorithm. *BIT Numerical Mathematics* **1980**, *20*, 176–184.

(5) Joux, A.; Lucks, S. In *International Conference on the Theory and Application of Cryptology and Information Security*, 2009, pp 347–363.