A fast exact sequential algorithm for the partial digest problem

Mostafa M. Abbas¹ and Hazem M. Bahig²*

From 15th International Conference On Bioinformatics (INCOB 2016)
Queenstown, Singapore. 21-23 September 2016

Abstract

Background: Restriction site analysis involves determining the locations of restriction sites after the process of digestion by reconstructing their positions based on the lengths of the cut DNA. Using different reaction times with a single enzyme to cut DNA is a technique known as a partial digestion. Determining the exact locations of restriction sites following a partial digestion is challenging due to the computational time required even with the best known practical algorithm.

Results: In this paper, we introduce an efficient algorithm to find the exact solution for the partial digest problem. The algorithm is able to find all possible solutions for the input and works by traversing the solution tree with a breadth-first search in two stages and deleting all repeated subproblems. Two types of simulated data, random and Zhang, are used to measure the efficiency of the algorithm. We also apply the algorithm to real data for the Luciferase gene and the E. coli K12 genome.

Conclusion: Our algorithm is a fast tool to find the exact solution for the partial digest problem. The percentage of improvement is more than 75% over the best known practical algorithm for the worst case. For large numbers of inputs, our algorithm is able to solve the problem in a suitable time, while the best known practical algorithm is unable.

Keywords: Restriction site analysis, Digestion process, Partial digest problem, DNA, Bioinformatics algorithm, Breadth first search

Background

In 1970, Hamilton Smith discovered that long DNA molecules could be digested into a set of restriction fragments by the restriction enzyme HindIII based on the occurrence of restriction sites with the sequences GTGCAC or GTTAAAC [1]. Since that time, restriction enzymes have played a crucial role in many biological experiments, including genome editing [2], gene cloning [3], protein expression [4, 5] and genome mapping [6–11]. Restriction enzymes are commonly used to physically map genomes. In physical mapping, the restriction enzymes are used to cut a DNA molecule at restriction sites with the goal of identifying the locations of the restriction sites after digestion. Their positions in the genome are determined by analyzing the lengths of the digested DNA. Based on the experimental assumptions of digestion, there are two main types of digestions, a partial digest [9] and a double digest [10]. Constructing an accurate physical map following a partial digestion is a fundamental problem in genome analysis. In this work, we consider the partial digestion.

In a partial digestion experiment, one restriction enzyme is used to cut one or more target DNA molecules at several specific restriction sites. The digestion results in a collection of short DNA fragments, and the lengths of these fragments are recorded in multiset \( A \). Attempting to reconstruct the locations of the restriction sites in the target DNA molecules using multiset \( A \) is known as the Partial Digest Problem, PDP. Some modifications have been introduced into the partial digestion process to produce simplified variants of PDP. These variants include: the simplified partial digest problem, SPDP [12], the labeled partial digest problem, LPDP [13] and the probed partial digest problem, PPDP [14]. In this work, we consider the PDP.

Several algorithms [15–20] have been developed to solve the PDP. Some of these algorithms have short running times, but the solutions are not exact. These algorithms
are based on heuristic and approximation strategies. Other algorithms require a long running time in the worst case, but the solution is exact for any instance. These algorithms [18–20] are based on brute force or branch and bound strategies.

In this research paper, we describe an algorithm with a suitable run time that generates an exact solution for the PDP. The previous algorithms that yield an exact PDP solution can be divided into impractical and practical types. The impractical solutions are based on the brute force strategy and polynomial factorization [19]. The best known practical algorithm for PDP is the algorithm proposed by Skiena et al. [20], which is based on the branch and bound strategy. The algorithm is practical because the running time of the algorithm is $O\left(n^2 \log n\right)$ for an average case, while exponential amounts of time were required for the worst case.

### Problem formulation and related definitions

Before we give the formal definition of the PDP, we need the following related definitions.

**Definition 1** [21]: The difference of two multisets $D$ and $L$ denoted by $D \setminus L$ such that $D \setminus L = \{x : x \in D \text{ and } C(D, x) = C(D, x) - C(L, x) > 0\}$, where $C(D, x)$ denotes the number of occurrences of element $x \in D$ in $D$.

**Definition 2** [21]: The sum or (disjoint union) of two multisets $D$ and $L$ denoted by $D \cup L$ such that $D \cup L = \{x : x \in D \text{ or } x \in L\}$ and $C(D \cup L, x) = C(D, x) + C(L, x)$.

**Definition 3** [22]: The differences of element $y$ and a set $X$ denoted by $\Delta(y, X)$ such that $\Delta(y, X) = \{y - x, j : x \in X\}$.

**Definition 4** [22]: The differences for a set $X$, denoted by $\Delta X$, is a multiset such that: $\Delta X = \{x_i - x_j, 0 \leq i < j \leq n - 1\}$.

**Remark 1**: We can write $\Delta X$ in another form $\Delta X = \bigcup_{i=0}^{n-1} \Delta_i(x_i, (X \setminus x_i))$, where $X = \{x_0, x_1, \ldots, x_{n-1}\}$.

**The Partial Digest Problem, PDP** [11]: Given a multiset of $N = \binom{n}{2}$ positive integers $D = \{d_0, d_1, d_2, \ldots, d_{n-1}\}$. Is there a set of $n$ integers $X = \{x_0, x_1, x_2, \ldots, x_{n-1}\}$ such that $\Delta X = D$?

We also need two propositions. The first proposition is used to give another formula for the difference between three multiple sets. The formula will be used to prove the correctness of our proposed method. The second proposition is used to illustrate how to construct an example for the worst case, which leads to an exponential time for Skiena’s algorithm [23].

**Proposition 1**: Let $D, L$ and $Z$ be three multisets, then $(D \setminus L) \cup Z = D \setminus (L \cup Z)$.

**Proposition 2** [23]: Let $0 < e < \frac{1}{12}n$, $A_1 = \{1 - ne, \ldots, 1 - 2e, 1 - e\}$, $A_2 = \{e, 2e, \ldots, ne\}$, $A_3 = \{(n + 1)e, (n + 2)e, \ldots, 2ne\}$, $A_4 = \{(n + 1)e, (2n + 2)e, \ldots, 3ne\}$, $A_5 = \{1 - 3ne, \ldots, 1 - (2n + 1)e, 1 - (2n + 1)e\}$ and $D = F \cup G$ where $F$ and $G$ are disjoint sets satisfying $F \cup G^* = A_3$ and $G^* = \{1 - g : g \in G\}$. Let $A = A_1 \cup A_2 \cup A_4 \cup A_5 \cup D \cup \{0, 1\}$, we can choose $D$ such that giving $\Delta A$ to Skiena’s algorithm will take it at least $\Omega(2^{n-1})$ time to find $A$.

### Methods

In this section, we present three algorithms. The first one is the best previous practical algorithm, while the other two algorithms are the proposed algorithms.

#### Best previous practical algorithm

The main goal of PDP is to reconstruct the elements, $x_0$, from a multiset, $D$, of $N = n(n - 1)/2$ integers by finding a set $X$ such that $\Delta X = D$. The best known algorithm for solving the PDP is based on the branch and bound strategy. The main idea of this algorithm is to construct the set $X$ incrementally. The algorithm is based on the depth-first search algorithm with two bounding conditions. We refer to this method as Algorithm BBd (branch and bound based on depth).

**Algorithm BBd**

**Input**: A multiset of integers, $D$.

**Output**: The solution set $S$.

**Begin**

1. Initialize the set $X$ with empty.
2. Delete the maximum element from the set $D$ and assign it to the element width.
3. Assign the set $\{0, \text{width}\}$ to the set $X$.
4. $\text{Place}(D, X)$

**End**

**Procedure** Place($D, X$)

**Begin**

1. if the set $D$ is empty then
2. if the set $X$ does not belong to $S$, then
3. Add the set $X$ to the set $S$.
4. return
5. end if
6. end if
7. Find the maximum element from set $D$ and assign it to the element $y$.
8. if the set $\Delta(y, X)$ is a subset of the set $D$ then
9. Add the element $y$ to the set $X$.
10. Remove the elements of the set $\Delta(y, X)$ from $D$.
11. Place($D, X$)
12. Remove the element $y$ from $X$.
13. Add the elements of $\Delta(y, X)$ to $D$.
14. end if
15. end if
16. if $\text{width}$ is a subset of the set $D$ then
17. Add width to $X$ and remove the elements of $\Delta(\text{width}, X)$ from $D$.
18. Place($D, X$)
19. Remove width from $X$ and add the elements of $\Delta(\text{width}, X)$ to $D$.
20. end if
21. end if

**End**

**Observation**

Figure 1 represents the execution of the algorithm BBd on $D = \{1, 2, 3, 5, 6, 7, 8, 9, 10\}$. In the figure, we use the following notations.
i. Each node in the solution tree for the PDP represents a pair of sets \( (D_k, X_k) \) where the index \( k \) represents the level number, and the index \( i \) represents the node number in the level \( k \).

ii. The number \( t \) inside the circle at the top right of each node represents the \( t \)-th calling for the procedure \( \text{Place} \).

iii. The symbol “\( \sqsubset \)” is used when the current node does not generate any new elements.

It is clear that at the level 0, the root contains the multiset \( D_0 = \{1, 2, 3, 5, 6, 7, 8, 10\} \) and the set \( X_0^0 = \{0, 10\} \). In general, \( X_0^0 = \{0, \text{width}\} \) and \( D_0^0 = D \setminus \{\text{width}\} \), where \( D \) is the input of the PDP. Additionally, each node \( (D_k, X_k) \) at the level \( k, k > 0 \), of the solution tree generates at most two nodes at the level \( k + 1 \) as follows:

1. Add an element \( y \) to \( X_k \) to generate \( X_{k+1} = X_k \cup \{y\} \).
2. Remove the elements of \( \Delta(y, X_k) \) from \( D_k \) to generate \( D_{k+1} = D_k \setminus \Delta(y, X_k) \).

where \( y = \text{Max}(D_k) \) or \( y = \text{width} \cdot \text{Max}(D_k) \). We also observe from Fig. 1 the following:

1. There are two identical subproblems \( (D_k^2, X_k^2) \) and \( (D_k^3, X_k^3) \), such that \( D_k^2 = D_k^3 = \{1, 3, 5, 7\} \) and \( X_k^2 = X_k^3 = \{0, 2, 3, 8, 10\} \).
2. The two identical subproblems lead to two identical solutions, \( \{0, 2, 7, 8, 10\} \) and \( \{0, 2, 3, 8, 10\} \).

### First proposed method

In this subsection, we propose an efficient method that reduces the running time of the PDP. The method is based on traversing the solution tree for the PDP using the breadth-first strategy instead of the depth-first strategy. We also consider the two bounding conditions that are used in algorithm BBd. Moreover, we remove all identical subproblems at the same level. The main steps of our method are as follows:

1. Build the solution tree for the PDP using the breadth-first strategy, level by level.
2. Before creating the nodes of the new level, we remove all repeated nodes existing in the current level such that any node appears only once in the current level.

Algorithm BBb (branch and bound based on breadth) shows the steps of our proposed method to solve the PDP. The input of the algorithm is the multiset \( D \) that consists of \( n(n-1)/2 \) elements. Initially the algorithm starts with two sets and two lists. The two sets are \( X_0^0 = \{0, \text{width}\} \) and \( D_0^0 = D \setminus \{\text{width}\} \). The two lists are \( L_X = \{X_0^0\} \) and \( L_D = \{D_0^0\} \). In general, \( L_D \) and \( L_X \) represent the lists of sets, \( D_k \) and \( X_k \), respectively, at the current level, \( k \), of the solution tree. The main step of the proposed algorithm is a while loop that represents the number of levels in the solution tree for the PDP. In each iteration \( k \) of the while loop, we will generate the elements of the next level, \( k + 1 \), by calling the procedure \( \text{GenerateNextLevel}(L_D, L_X, S, k) \). The initial value of the procedure is the three lists of sets, \( L_D, L_X \) and \( S \), for the current level \( k \). The outputs of the procedure are the three lists of sets, \( L_D, L_X \) and \( S \), for the level \( k + 1 \).

The body of the procedure \( \text{GenerateNextLevel} \) consists of an initialization and a loop. In the initialization, we will use two auxiliary lists, \( ALD \) and \( ALX \), which contain the sets \( D_{k+1} \) and \( X_{k+1} \), respectively, for the next level in the solution tree. The initial value of the two lists is empty. The main loop in the \( \text{GenerateNextLevel} \) procedure represents the process of generating the elements of the next level and storing it in the two auxiliary lists \( ALD \) and \( ALX \). Each pair of sets, \( D_k \in L_D \) and \( X_k \in L_X \), will generate at most two pairs of sets, as follows:

(i) \( D_{k+1}^1 = D_k^1 \setminus \Delta(y, X_k^1) \) and \( X_{k+1}^1 = X_k^1 \cup \{y\} \) if the condition \( \Delta(y, X_k^1) \subseteq D_k^1 \) is true.

(ii) \( D_{k+1}^2 = D_k^2 \setminus \{\text{width} - y, X_k^2\} \) and \( X_{k+1}^2 = X_k^2 \cup \{\text{width} - y\} \) if the condition \( \Delta(\text{width} - y, X_k^2) \subseteq D_k^2 \) is true.

The two sets, \( X_{k+1}^1 \) and \( D_{k+1}^1 \) and in a similar way, \( X_{k+1}^2 \) and \( D_{k+1}^2 \), will be added to the auxiliary lists \( ALD \) and \( ALX \), respectively if set \( X_{k+1}^1 \) does not exist in list \( ALD \).

The main loop of the \( \text{GenerateNextLevel} \) procedure will terminate when list \( L_D \) is empty. This means that all of the elements of the current level \( k \) are replaced by new elements in the next level \( k + 1 \). In this case, we assign the lists \( ALD \) and \( ALX \) to \( L_D \) and \( L_X \), respectively. Figure 2 illustrates how this idea works.
Algorithm BBb
Input: A multiset of integers, \( D \).
Output: The solution set \( S \).
Begin
1. Initialize the set \( S \) with empty.
2. Delete the maximum element from the set \( D \) and assign it to the element \( width \).
3. Assign the set \( \{0, \ width \} \) to the set \( X \).
4. Add the set \( D \) to the list \( L_0 \).
5. Add the set \( X \) to the list \( L_0 \).
6. while the list \( L_0 \) is not empty do
7. \( \text{GenerateNextLevel}(L_D, L_X, S) \)
8. end while
End

Procedure GenerateNextLevel(L_D, L_X, S)
Begin
1. Initialize the auxiliary lists \( AL_D \) and \( AL_X \) with empty.
2. while the list \( L_D \) is not empty do
3. Remove the first element from \( L_D \) and store it to \( D \).
4. Remove the first element from \( L_X \) and store it to \( X \).
5. if the set \( D \) is empty then
6. if the set \( X \) does not exist in the set \( S \) then
7. Add the set \( X \) to the set \( S \).
8. continue // take a new iteration
9. end if
10. end if
11. Find the maximum element from the set \( D \) and assign it to the element \( y \).
12. if the set \( \Delta(y, X) \) is a subset of set \( D \) then
13. Add the element \( y \) to the set \( X \).
14. Remove the elements of set \( \Delta(y, X) \) from \( D \).
15. if set \( X \) does not exist in the list \( AL_X \) then
16. Add the set \( X \) to the list \( AL_X \).
17. Add the set \( D \) to the list \( AL_D \).
18. end if
19. Remove the element \( y \) from the set \( X \).
20. Add the elements of the set \( \Delta(y, X) \) to \( D \).
21. end if
22. if the set \( \Delta(width - y, X) \) is a subset of set \( D \) then
23. Add the element \( width - y \) to the set \( X \).
24. Remove the elements of the set \( \Delta(width - y, X) \) from \( D \).
25. if set \( X \) does not exist in the list \( AL_X \) then
26. Add the set \( X \) to the list \( AL_X \).
27. Add the set \( D \) to the list \( AL_D \).
28. end if
29. end if
30. end while
31. Assign the list \( AL_X \) to \( L_X \) and \( AL_D \) to \( L_D \).
End

Now, we investigate how to test the equality between two nodes \((D_k, X_k^i)\) and \((D_k, X_k^j)\) in the solution tree for the PDP. The test can be performed by comparing the equality between \(D_k\) and \(D_k\) and the equality between \(X_k^i\) and \(X_k^j\). The following theorems and corollary prove that the two nodes \((D_k, X_k^i)\) and \((D_k, X_k^j)\) are equal if the set \(X_k^i\) is equal to the set \(X_k^j\).

**Theorem 1:** Given a subproblem \((D_k, X_k^i)\) in the solution tree for the PDP, the relation \(D_k = D \setminus \Delta X_k^i\) is valid, where \(k \geq 0\), \(D\) is the input of the PDP, \(D_k\) is a modified version of \(D\) produced by removing some of its elements and \(X_k^i\) contains \(k + 2\) elements of the candidate solution.

**Proof:** We will prove the theorem using mathematical induction.

First, we prove that the relation is true at \(k = 0\).

The algorithm starts by finding the maximum elements of the set \(D, width\), and updates the value of the sets \(D\) and \(X\). So, \(X_0^0 = \{0, width\}\) and \(D_0^0 = D \setminus \{\text{width}\}\). From the definition of \(\Delta\), we have, \(\Delta X_0^0 = \{\text{width}\}\). Therefore, \(D_0^0 = D \setminus \Delta X_0^0\).

Second, we assume that the relation is true at \(k\) (i.e., \(D_k = D \setminus \Delta X_k^i\)).

Third, we prove that the relation is true at \(k + 1\).

From the algorithm, the set \(D_{k+1}\) can be constructed from a node at level \(k\), say \((D_k, X_k)\). Therefore, \(D_{k+1} = D_k \setminus \Delta(y, X_k)\) and \(X_{k+1} = X_k \cup \{y\}\). This implies that \(D_{k+1} = D \setminus \Delta X_k^i\) and \(X_{k+1} = X_k \cup \{y\}\). Therefore, \(D_{k+1} = D \setminus \Delta X_k^i\).

**Theorem 2:** If there are two subproblems \((D_k, X_k^i)\) and \((D_k, X_k^j)\) such that \(X_k^i = X_k^j\), then \(D_k = D_k\).

**Proof:** From Theorem 1, the following equations are valid \(D_k = D \setminus \Delta X_k^i\) and \(D_k = D \setminus \Delta X_k^j\) for the subproblems \((D_k, X_k^i)\) and \((D_k, X_k^j)\), respectively. Because \(X_k^i = X_k^j\), then \(D_k^i = D_k^j\).

**Corollary 1:** If there are two subproblems \((D_k, X_k^i)\) and \((D_k, X_k^j)\) such that \(X_k^i = X_k^j\), then \((D_k^i, X_k^i) = (D_k^j, X_k^j)\).

**Final proposed algorithm**

We proposed an enhanced version of algorithm BBb. The proposed algorithm improves the running time and storage of the BBb algorithm especially for the worst case. In the proposed algorithm, we try to reduce the memory consumption of the BBb algorithm without increasing its running time. The improved algorithm depends on the following two steps:

1. Building the solution tree for the PDP until a specific level \(\alpha\) is reached by using the BBb algorithm.
2. For each node in the level \(\alpha\), building the remainder subtrees individually with the BBb algorithm.

Figure 3 represents the idea described above for the proposed algorithm. We called the algorithm BBb2, branch and bound based on breadth two times.
To determine the best value of $\alpha$, we need to compute the memory complexity of the proposed algorithm for the worst case. Each node in level $k$ will be replaced by at most two nodes in the level $k + 1$, $0 \leq k < \alpha$. Therefore, the total number of nodes at level $\alpha$ is $2^\alpha$. In each node, we store two sets, $D_\alpha$ and $X_\alpha$, of total size $O(n^2)$. Hence, the total amount of storage necessary to reach the level $\alpha$ is $O(n^2 2^\alpha)$ memory for BBb2. In the second step of the BBb2 algorithm, we apply BBb on each node individually. The maximum number of remaining levels is $n - \alpha$ and the total amount of memory required for the second step for the worst case is $O(n^2 2^{n-\alpha})$. Hence, the memory complexity of the BBb2 algorithm for the worst case is given by:

$$M(\alpha) = O(n^2 2^\alpha) + O(n^2 2^{n-\alpha})$$

Let $\alpha_M$ be the number of levels that lead to the minimum memory required by BBb2. The value of $\alpha_M$ can be computed by determining the number of levels, $\alpha$, that minimizes the memory consumption $M(\alpha)$, for $0 \leq \alpha \leq n - 1$. The $\text{Find}_{\alpha_M}$ procedure computes the value of $\alpha_M$ for the instance $n$. The pseudocode of the $\text{Find}_{\alpha_M}$ procedure is as follows:

Procedure $\text{Find}_{\alpha_M}$ ($n$, $\alpha_M$)

\[\begin{align*}
1. \quad n &= 1 + \sqrt{1 + 8N} \\
2. \quad \alpha &= \alpha_M = 1 \\
3. \quad M_{\text{min}} &= n^2 (2^\alpha + 2^{n-\alpha}) \\
4. \quad \text{for } \alpha = 2 \text{ to } n \text{ do} \\
5. \quad \quad M &= n^2 (2^\alpha + 2^{n-\alpha}) \\
6. \quad \quad \text{if } M < M_{\text{min}} \text{ then} \\
7. \quad \quad \quad M_{\text{min}} &= M \\
8. \quad \quad \quad \alpha_M &= \alpha \\
9. \quad \quad \text{end if} \\
10. \quad \text{end for}
\end{align*}\]
In Algorithm BBb2, we start by finding the value of $\alpha M$ and then we build the solution tree until the level $\alpha M$ is reached by using the breadth-first strategy and deleting all similar elements. At the level $\alpha M$, we have at most $2\alpha M$ elements. After obtaining these elements, we consider each node as a root and then expand this node using the BBb algorithm. For the worst case, the number of handled elements at levels $\alpha M + 1, \alpha M + 2, ..., n$ are $2, 2^2, ..., 2^{n-\alpha M}$ respectively, while the number of handled elements during execution of the BBb algorithm at levels $\alpha M + 1, \alpha M + 2, ..., n$ are $2^{\alpha M+1}, 2^{\alpha M+2}, ..., 2^n$ respectively. Therefore, BBb2 reduces the memory required by the BBb algorithm.

Test methodology
In this section, we present the methodology that is used to evaluate the performance of the algorithms, BBd, BBb and BBb2, according to their running times and memory consumptions.

Platform Specification
We implemented the algorithms on a Dual Octa-core processor machine (Intel Xeon E5-2690) with 128 GB RAM. Each processor has a 2.9 GHz speed with 20 MB of cache. The algorithms were implemented in C++ programs. The programs were compiled using g++ with the -O3 flag under 64-bit Red Hat Enterprise Linux 4.4. In the experiments, we used a single core only.

Simulated data
We studied the performance of the three algorithms, BBd, BBb and BBb2, on different types of data and different sized data sets. We used two types of data described in previous studies. The first data set was a random data set (RD) [16], while the second data set is the Zhang data (ZD) [16, 17, 23]. The RD was used to measure the performance of the algorithm for an average case; while the ZD was used to measure the performance of the algorithm for the worst case. In terms of data set size, there are two factors that affect the performance of the algorithms for each data set. The first factor is the number of elements in the set $X$, $n$, while the second factor is the values of the $n$ elements, $M$ (maximum elements of $X$).

In the case of the RD, we assumed that there were $n$ restriction sites in the DNA segments distributed randomly. Each input instance of the simulated data is a multiset $D = \Delta X$ such that the set $X$ contains $n$ positive numbers randomly selected and each number is less than or equal to $M$. In the ZD, the locations of the restriction sites were selected randomly according to Proposition 2 [23].

The values of $n$ in the case of the RD are 100, 200, 300,... and 1000, while the values of $n$ in the case of ZD are 15, 20, 25, 30,... and 90. The range of $n$ for the ZD is small because the running time for the algorithm was greater than 1 day when $n > 90$. We also used different values of $M$ as follows: $M = n^q q$, where $q = 10, 100, 1000$ and 10,000.

Running time and memory
For each value of $n$, we ran each algorithm 50 times with different inputs for the RD. For the ZD, we reduced this number to 20 due to the increased running time of the algorithms, especially BBd. The running time for each algorithm for a fixed $n$ represents the average time for all instances studied. If the running time of an algorithm was greater than 24 h for an input instance then the algorithm was terminated. Therefore, the value of the algorithm for this input instance was omitted from the figures and the results.

We also measured the standard error of the mean (SEM) and coefficient of variation (CV) for the three algorithms for RD and ZD. Finally, we used a non-parametric statistical test which is Wilcoxon-signed-rank test [24] to determine if there are a significant differences between the three algorithms in running time or memory consumption on RD and ZD. We applied the Wilcoxon-signed-rank test, at a significance level of 0.05, to the following pairs: BBd & BBb, BBd & BBb2 and BBb & BBb2 algorithms with respect to running time and memory consumption for RD and ZD.

We measured the running time in seconds using a C++ function. We also measured the memory consumed by the algorithm in megabytes using the Linux command top.

Results and discussion
The results from measuring the running times of the three algorithms on the simulated data are shown in
Fig. 4 and (Additional file 1: Figure F1) for the RD and Fig. 5 for the ZD.

In the case of the RD, the results showed that the running times of the proposed algorithms, Bb and Bb2, were less than the running time of Bbd for all values of \( n \) and \( M \) as shown in Fig. 4. For large values of \( n \) and small values of \( q \), the Bbd algorithm required a large amount of time to find a solution, while the proposed algorithms, Bb and Bbb2, found the solution in a suitable time. For example, in the case of \( n \geq 300 \) and \( M = n \times 10 \), the running time for the Bbd algorithm was greater than 24 h, while both proposed algorithms found the solution in time less than 13 min, as shown in Fig. 4a and b. However, the difference in running times between the Bbd algorithm and the proposed algorithms, Bb and Bbb2, decreased with increasing values of \( M \). This behavior is attributed to the probability of repeated subproblems decreasing with increasing values of \( M \). Therefore, both algorithms, Bb and Bbb2, spend a large amount of time checking for the repetition of elements with a low probability of repetitions. Additionally, for small \( M \) values the probability of repeated subproblems is high, thereby increasing the running time of the Bbd algorithm. If we fixed the value of \( n \), the running time for all algorithms decreased with increasing \( M \) values as shown in (Additional file 1: Figures F1 a-c). Both proposed algorithms behaved similarly with increasing \( M \) values. We also observed that the difference in the running times for two successive values of \( M \) is relatively small for both proposed algorithms (especially when \( M \) is large) as shown in (Additional file 1: Figures F1 b and c). On the other hand, the difference in the running times for the Bbd algorithm for two successive values of \( M \) is large when \( M = n \times 100 \) and \( M = n \times 1000 \) as shown in (Additional file 1: Figure F1a). Finally, we found that there was little difference, increasing or decreasing, between the running times of Bb and Bbb2 for all values of \( n \) and \( M \). In general, with large values of \( n \) and \( M \), the Bbb2 algorithm was faster than the Bb algorithm. For example, in the case of \( n \geq 700 \) and \( M = n \times q \) and \( q = 1000 \) and 10,000, Bbb2 was slightly better than Bb. For the ZD, there was no change in the running time when we used different values of \( M \) for the three algorithms. Therefore, we used the ZD with the \( M \) value fixed at \( M = n \times 100 \). The consistent performance of the algorithms using the ZD with different values of \( M \) was due to the systematic selection of the elements of \( A \) according to Proposition 2.

The performance of the three algorithms for the Zhang data set instances is given in Fig. 5. We observed that the running time of Bbb2 was less than Bb and Bb was less than Bbd for all instances. The percentage of running time improvement for Bb and Bbb2 with respect to Bbd increased with increasing \( n \). In our studied cases, the running time was improved by at least 75%. Moreover, the Bbd and Bb algorithms cannot solve any instances
with \( n \geq 60 \) and \( n \geq 80 \) in time less than 24 h respectively. In the other side, the BBb2 algorithm solves any instance with \( n \leq 90 \) in time less than 24 h.

The improved running time of BBb algorithm can be attributed to its reduction of the number of subproblems handled at the levels \( \alpha_M + 1, \alpha_M + 2, \ldots, n \) in the solution tree for the PDP. Therefore, the time spent checking subproblem repetition is reduced.

The results of measuring SEM and CV for the running time of the three algorithms are shown in (Additional file 2: Tables S1–S5) for RD and ZD. The results show that the values of SEM and CV for BBd algorithm were very large compared to the values of SEM and CV for BBb and BBb2 algorithms in case of RD. For the ZD, the values of SEM and CV of BBb and BBb2 algorithms were less than the values of SEM and CV of BBd algorithm in the most instances. Moreover, the application of Wilcoxon-signed-rank test shows that there was a significant difference between the following pairs of the algorithms as shown in (Additional file 2: Tables S6–S14):

i. BBd and BBb in case of RD and ZD.
ii. BBd and BBb2 in case of RD and ZD.
iii. BBb and BBb2 in case of ZD.

We also evaluated the algorithms in terms of their memory consumptions. The results of measuring the memory consumed by the three algorithms on the simulated data are shown in (Additional file 3: Figures F2 a–d) for the RD and Fig. 6 for the ZD. For the RD, the results show that for small values of \( M \) such as \( M = n * 10 \) and \( n * 100 \), the BBb and BBb2 algorithms consumed less memory than the BBd algorithm. In the case of large \( M \) values, the BBd algorithm consumed less memory than the BBb and BBb2 algorithms, but small \( M \) values increased the number of repeated subproblems. Thus, the number of elements in each level is large for the BBd algorithm. Therefore, the BBd algorithm consumed more memory than the two proposed algorithms. In general, the memory consumption of the BBb algorithm is a little better than that of the BBb2 algorithm.

For the ZD, less memory was consumed by BBb2 than BBb for all instances. So, the BBb2 algorithm significantly reduced the memory required by the BBb algorithm. The percentage of improvement in memory consumption of BBb2 compared to BBb increases as \( n \) increases. In general, the memory required by the BBd algorithm is less than the BBb and BBb2 algorithms.

The results of measuring SEM and CV for the memory of the three algorithms are shown in (Additional file 4: Tables S15–S19) for the RD and the ZD. The results show that there were differences, decreasing and increasing,
in the values of SEM and CV of the three algorithms for the RD. For the ZD, the values of SEM and CV of BBd algorithm were less than the proposed algorithms in the most instances. Moreover, the application of Wilcoxon-signed-rank test, see (Additional file 4: Tables S20-S28), shows that there was a significant difference between the following pairs of the algorithms in the following cases:

i. BBd and BBb on ZD.
ii. BBd and BBb2 on ZD.
iii. BBb and BBb2 on ZD.

Real data
We tested the final proposed BBb2 algorithm on real digestion experiments and simulated digestion experiments.

Luciferase gene
We extracted the data from a partial digestion of the luciferase gene [25] of length 2009. The partial digestion was performed with the enzyme TaqI, which cuts the gene at the tcga sequence motif. The output of the partial digestion process is the multiset \( D \) consisting of the distances between the tcga locations on the luciferase gene, which is \( D = \{9, 30, 100, 170, 293, 302, 393, 402, 462, 562, 632, 732, 855, 864, 945, 954, 975, 984, 1025, 1034, 1247, 1277, 1347, 1377, 1809, 1839, 1893, 1979, 2009\}.

Our proposed algorithms take the multiset \( D \) as input and output two solutions in the set \( S = \{0, 170, 632, 732, 1025, 1034, 1277, 1377, 1839, 1893, 1979, 2009\} \). The solution \( X = \{0, 30, 975, 984, 1277, 1377, 1839, 1893, 2009\} \), represents the solution for the real data. This means that the map of tcga on luciferase gene at the locations 30, 975, 984, 1277, 1377, and 1839.

E. Coli K12 genome digestion simulation
We also tested our proposed algorithms with a simulated partial digestion experiment using the E. coli K12 genome version 3 (NC_000913.3, downloaded 11 March 2016) and a set of restriction enzymes (https://www.en.wikipedia.org/wiki/Restriction_enzyme). The size of the genome is 4,641,652 bp. For each restriction enzyme, we simulated the partial digestion experiment by cutting the E. coli K12 genome at all occurrences of the corresponding restriction site. Then, we recorded the lengths of \( N \) cut fragments in \( D \). The restriction enzymes, restriction sites, the number of cut fragments (\( N \)), \( n \), times and memory consumptions are shown in Table 1. Finally, we applied the final proposed algorithm to the extracted sets and in all cases the outputs contained the correct set of restriction site positions. It is clear that from Table 1, the running time and memory consumed of the algorithm increases with increase the value of \( n \) and \( N \).

Conclusions
In this paper, we addressed the PDP and proposed two algorithms, BBb and BBb2. In the BBb algorithm, we built the solution tree for the PDP in the breadth-first manner instead of the depth-first manner taking into consideration two conditions of pruning and deleting all repeated subproblems in the same level. The BBb solves many instances which are not solved by BBd in time less than 24 h. The main disadvantage of BBb is that the memory consumed grows exponentially. In BBb2, we reduced the memory required by BBb by solving the problem using two stages, with each stage working in the breadth-first manner. We also determined the number of the levels, which leads to reduced memory consumption without increasing the running time.

We measured the efficiency of the proposed algorithm compared to the best known practical algorithm on the basis of time and memory consumption. In the evaluation, we considered the following parameters: (1) types of data, RD and ZD; (2) value of \( n \); and (3) value of \( M \). In the case of running time, the BBb2 algorithm is faster than other algorithms. The efficiency increased when the ZD was used. In the case of memory, the BBd algorithm consumed less memory than other algorithms, but the running time was very slow especially for the ZD. Finally, we applied the BBb2 algorithm on Luciferase gene and the E. coli K12 genome.

| Restriction enzyme | Restriction site | N  | n  | Time (Seconds) | Memory (MB) |
|--------------------|-----------------|----|----|----------------|-------------|
| NotI              | GCGGCCGC        | 780| 40 | 0.007          | 0.2         |
| XbaI              | TCTAGA          | 1891| 62 | 0.017          | 3.6         |
| Smal              | CCCGGG          | 103,285| 455 | 56.2           | 8.9         |
| BamHI             | GATATCC         | 135,460| 521 | 72             | 11.7        |
| HindIII           | AAGCTT          | 170,820| 585 | 138.6          | 13.6        |
| EcoRI             | GAATTC          | 228,826| 677 | 360            | 15.5        |
| PvuII             | CAGCTG          | 1,599,366| 1789 | 24,012         | 98.8        |
| EcoRV             | GATATC          | 1,999,000| 2000 | 42,840         | 979         |

**Abbreviations:** \( N \) is the number of cut fragments, \( n \) is the number of restriction sites.
Acknowledgements

The experimental study of this work was done where the first author was at KINDI Center for Computing Research, College of Engineering, Qatar University, Doha, Qatar.

Declarations

This article has been published as part of BMC Bioinformatics Volume 17 Supplement 19, 2016. 15th International Conference On Bioinformatics (INCOB 2016): bioinformatics. The full contents of the supplement are available online at https://www.biomedcentral.com/articles/supplements/volume-17-supplement-19.

Funding

Publication charges for this article have been funded by Qatar Computing Research Institute (QCR), Hamad Bin Khalifa University, Qatar Foundation.

Availability of data and materials

Data used in the article are publicly available. Availability of software exists at: https://www.github.com/mostafa-abbas/PDP.

Authors’ contributions

Both authors contributed to theoretical and practical study equally. Both authors wrote and approved the manuscript.

Competing interests

The authors declare that they have no competing interests.

Consent for publication

Not applicable.

Ethics approval and consent to participate

All the datasets used in this study were obtained from open-access databases and previously published by other authors. No human or animal data were collected. Therefore, no informed consent forms were needed from this study.

Additional files

Additional file 1: Supplementary figure. Figure F1 a–c represents the behavior of the running time for BBd, BBb, and BBb2 algorithms with different values of M and fixed value of n. The values on the y-axis are in log-scale. In Figure a, the x-axis does not include the value of \( M = n \times 10 \), because the running time is greater than 24 h. (PDF 8 kb)

Additional file 2: Supplementary data. Calculation of the Standard Error of Mean (SEM), Coefficient of Variation (CV), and Wilcoxon Signed-Rank test for the running time. Tables S1–S5. in Sheet 1 and 2, represent the calculation of SEM and CV for the running time of BBd, BBb, and BBb2 algorithms in case of random data and Zhang data respectively. Tables S6–S8. In Sheet 3, represent the Wilcoxon Signed-Rank test between BBd and BBb algorithms for the running time of Random and Zhang data. Tables S9–S11. In Sheet 4, represent the Wilcoxon Signed-Rank test between BBd and BBb2 algorithms for the running time of Random and Zhang data. Tables S12–S14. In Sheet 5, represent the Wilcoxon Signed-Rank test between BBd and BBb2 algorithms for the running time of Random and Zhang data. (XLS 168 kb)

Additional file 3: Supplementary figure. Figure F2 a–d represents the memory consumed for BBd, BBb, and BBb2 algorithms on random data. (PDF 88 kb)

Additional file 4: Supplementary data. Calculation of the Standard Error of Mean (SEM), Coefficient of Variation (CV), and Wilcoxon Signed-Rank test for the memory. Tables S15–S19. in Sheet 1 and 2, represent the calculation of SEM and CV for the memory consumption of BBd, BBb, and BBb2 algorithms in case of random data and Zhang data respectively. Tables S20–S22. In Sheet 3, represent the Wilcoxon Signed-Rank test between BBd and BBb algorithms for the memory consumption of Random and Zhang data. Tables S23–S25. In Sheet 4, represent the Wilcoxon Signed-Rank test between BBd and BBb2 algorithms for the memory consumption of Random and Zhang data. Tables S26–S28. In Sheet 5, represent the Wilcoxon Signed-Rank test between BBd and BBb2 algorithms for the memory consumption of Random and Zhang data. (XLS 167 kb)