7 Randomized divide-and-conquer based on solutions

Divide-and-conquer is a well-known method for developing efficient algorithms. A divide-and-conquer process divides a given instance into smaller instances, solves the smaller instances recursively, and finally “merges” the solutions for the smaller instances into a solution for the original instance. Well-known examples include divide-and-conquer sorting algorithms such as MergeSort and QuickSort.

In this section, we study a new randomized algorithmic technique based on divide-and-conquer that turns out to be very effective for solving parameterized NP-hard problems. Many parameterized NP-hard problems use the size of solutions as their parameter. Therefore, here we are looking for “small” solutions. In order to reduce the recursion depth of the divide-and-conquer process thus speedup the computation, the new method works by recursively dividing the solution, instead of the input instance.

Of course, an immediate question then arises: we are looking for a solution, which certainly implies that the solution is unknown in advance, how do we effectively divide an unknown object?

This is a place where randomization comes to help again. We show that when the size of the solution is small, there is a reasonable probability that the unknown solution is divided in the desired way when the input instance is randomly divided. Therefore, by repeatedly applying random dividing on the instance, we have a good probability to get an instance dividing that has the solution divided nicely.

We again use the max-Path problem as an example to illustrate the technique. At the end, we will mention briefly how the technique is used to solve other NP-hard problems.

7.1 Randomized divide-and-conquer

Recall that the max-Path problem is to construct a k-path of the maximum weight in an undirected and weighted graph. Fix an undirected and weighted graph \( G = (V, E) \). For any subset \( V' \) of vertices in \( G \), denote by \( G[V'] \) the subgraph of \( G \) induced by \( V' \) (i.e., \( G[V'] \) is the graph that has vertex set \( V' \) and contains exactly those edges in \( G \) that have both ends in \( V' \)). The concatenation of two paths \( \rho_1 = \langle v_1, \ldots, v_l \rangle \) and \( \rho_2 = \langle w_1, \ldots, w_h \rangle \) in \( G \), where \([v_l, w_1]\) is an edge in \( G \), is the path \( \langle v_1, \ldots, v_l, w_1, \ldots, w_h \rangle \). We denote by \( \rho_0 \) the special 0-path (i.e., the empty path containing no vertex), and define that the concatenation of \( \rho_0 \) and any path \( \rho \) gives the path \( \rho \). An \( h \)-path \( \rho \) is also called a \((v, h)\)-path if \( v \) is an end vertex of \( \rho \).

We first give a high-level description of the algorithm. Fix a \( k \)-path \( \rho_k \) of the maximum weight in the graph \( G \). Suppose that we randomly partition the vertex set \( V \) of \( G \) into two subsets \( V_L \) and \( V_R \). Note that if the partition \((V_L, V_R)\) has the “first-half” of the path \( \rho_k \) entirely in the induced subgraph \( G[V_L] \) and the “second-half” of the path \( \rho_k \) entirely in the induced subgraph \( G[V_R] \), then we can apply the algorithm recursively to construct the maximum \((k/2)\)-paths in \( G[V_L] \) and in \( G[V_R] \), then concatenate the \((k/2)\)-paths in \( G[V_L] \) and the \((k/2)\)-paths in \( G[V_R] \) to make \( k \)-paths in \( G \). Of course, the success of this process depends on the probability that the random partition \((V_L, V_R)\) of \( V \) divides the path \( \rho_k \) into two “right” halves, which, luckily, turns out to be enough by the probability analysis we will present later. Thus, the general framework of our algorithm looks as follows.

1. \textbf{repeat} sufficiently many times \textbf{do}
   
   randomly partition the vertices of \( G \) into \( V_L \) and \( V_R \);
   
   recursively work on the induced subgraphs \( G[V_L] \) and \( G[V_R] \);
   
   concatenate \((k/2)\)-paths in \( G[V_L] \) and \((k/2)\)-paths in \( G[V_R] \) to make \( k \)-paths in \( G \);

2. return the largest \( k \)-path constructed in step 1.

The remaining task is to take care of how the \((k/2)\)-paths of maximum weight in the induced subgraphs \( G[V_L] \) and \( G[V_R] \) are recorded and organized, and how these \((k/2)\)-paths are concatenated to
make \( k \)-paths of maximum weight in the original graph \( G \). There is a straightforward implementation for this: for each vertex pair \((v_1, v_2)\) in \( G[V_L] \), record the \((k/2)\)-path of the maximum weight whose two ends are \( v_1 \) and \( v_2 \). Similarly organized the \((k/2)\)-paths in the graph \( G[V_R] \). Now to construct the \( k \)-paths in \( G \), we consider each edge \([v, w]\) in \( G \), where \( v \in V_L \) and \( w \in V_R \), and concatenate the \((k/2)\)-path in \( G[V_L] \) that has \( v \) as an end and the \((k/2)\)-path in \( G[V_R] \) that has \( w \) as an end. With this process we can surely construct the \( k \)-path of the maximum weight between each vertex pair in \( G \). The drawback of this process is its running time and memory space: it takes time \( O(kn^2) \) and space \( O(kn^2) \). In the following, we present an algorithm that uses significantly less time and space.

Let \( P_1 \) be a set of \( l \)-paths in \( G \), and let \( V' \subseteq V \) such that no vertex in \( V' \) is on any path in \( P_1 \). A \((v, h)\)-path is in \( P_1 \odot V' \) if \( v \in V' \) and \( \rho \) is a concatenation of an \( l \)-path in \( P_1 \) and an \((h - l)\)-path in \( G[V'] \). In particular, for \( P_0 = \{ \{0\} \} \), any \((v, 1)\)-path in \( P_0 \odot V' \) consists of the single vertex \( v \) in \( V' \).

On a set \( P_1 \) of \( l \)-paths in \( G \) and \( V' \subseteq V \), where \( P_1 \) contains at most one \((v, l)\)-path for each vertex \( v \), and no vertex in \( V' \) is on any path in \( P_1 \); our algorithm \textsc{FindPaths}(\( P_1, V', h \)) returns a set \( P_{l+h} \) of \((l + h)\)-paths in \( P_1 \odot V' \). In particular, the algorithm \textsc{FindPaths}(\( \{\{0\}\}, V, k \)) returns a set of \( k \)-paths in the graph \( G \). The detailed algorithm is given as follows.

\textbf{Algorithm 13} \textsc{FindPaths}(\( P_1, V', h \))

Input: a set \( P_1 \) of \( l \)-paths; \( V' \subseteq V \) and no vertex in \( V' \) is on a path in \( P_1 \); an integer \( h \geq 1 \); 
Output: a set \( P_{l+h} \) of \((l + h)\)-paths in \( P_1 \odot V' \); 
1. \( P_{l+h} = \emptyset \); 
2. if \( h = 1 \) then 
   2.1. if \( P_1 = \{\{0\}\} \) then \( P_{l+1} \) contains a \((u, 1)\)-path for each vertex \( u \in V' \); return \( P_{l+1} \); 
   2.2. else for each \((w, l)\)-path \( \rho_l \) in \( P_1 \) and each \( u \in V' \), where \([w, u] \) is an edge in \( G \), do 
      2.2.1. concatenate \( \rho_l \) and \( u \) to make a \((u, l + 1)\)-path \( \rho_{l+1} \) in \( P_1 \odot V' \); 
      2.2.2. if \( P_{l+1} \) contains no \((u, l + 1)\)-path then add \( \rho_{l+1} \) to \( P_{l+1} \); 
      2.2.3. else if the \((u, l + 1)\)-path \( \rho'_{l+1} \) in \( P_{l+1} \) has a weight smaller than that of \( \rho_{l+1} \) 
      2.2.4. then replace \( \rho'_{l+1} \) in \( P_{l+1} \) by \( \rho_{l+1} \); 
      2.2.5. return \( P_{l+1} \); 
3. loop \( 3 \cdot 2^h \) times do 
   3.1. randomly partition the vertices in \( V' \) into two parts \( V_L \) and \( V_R \); 
   3.2. \( P_{l[h/2]}^L = \textsc{FindPaths}(P_1, V_L, [h/2]) \); 
   3.3. if \( P_{l[h/2]}^L \neq \emptyset \) then 
      3.3.1. \( P_{l+h}^R = \textsc{FindPaths}(P_{l[h/2]}^L, V_R, [h/2]) \); 
      3.3.2. for each \((u, l + h)\)-path \( \rho_{l+h} \) in \( P_{l+h}^R \) do 
      3.3.3. if \( P_{l+h} \) contains no \((u, l + h)\)-path in \( P_1 \odot V' \) then add \( \rho_{l+h} \) to \( P_{l+h} \); 
      3.3.4. else if the \((u, l + h)\)-path \( \rho'_{l+h} \) in \( P_{l+h} \) has a weight smaller than that of \( \rho_{l+h} \) 
      3.3.5. then replace \( \rho'_{l+h} \) in \( P_{l+h} \) by \( \rho_{l+h} \); 
4. return \( P_{l+h} \).

We first study the success probability of the algorithm \textsc{FindPaths}. We have the following theorem (where \( e = 2.718 \cdots \) is the base of the natural logarithm):

\textbf{Theorem 7.1} Let \( P_1 \) be a set of \( l \)-paths and let \( V' \) be a vertex subset that contains no vertex in any path in \( P_1 \). Let \( v \) be any vertex in \( V' \) with \((v, l + h)\)-paths existing in \( P_1 \odot V' \). Then with a probability larger than \( 1 - 1/e \), the algorithm \textsc{FindPaths}(\( P_1, V', h \)) returns a set \( P_{l+h} \) that contains a \((v, l + h)\)-path in \( P_1 \odot V' \) whose weight is the maximum over all \((v, l + h)\)-paths in \( P_1 \odot V' \).

\textbf{Proof.} First note that by steps 2.4–2.6 and steps 3.6–3.8, if the set \( P_{l+h} \) contains a \((v, l + h)\)-path \( \rho \), then \( \rho \) must be a valid \((v, l + h)\)-path in \( P_1 \odot V' \). Therefore, if there is no \((v, l + h)\)-path in \( P_1 \odot V' \), then the set \( P_{l+h} \) returned by the algorithm cannot contain a \((v, l + h)\)-path.

In the following discussion, by a “maximum-weighted \((v, l + h)\)-path in \( P_1 \odot V' \), we really mean a \((v, l + h)\)-path in \( P_1 \odot V' \) whose weight is the maximum over all \((v, l + h)\)-paths in \( P_1 \odot V' \). In particular, this does not imply that the path has the maximum weight over all \((l + h)\)-paths in the graph \( G \).
By the assumptions of the theorem, there are \((v,l+h)\)-paths in \(P_l \odot V'\). Thus, let 

\[ \rho_{l+h} = \langle u_1, \ldots, u_l, w_1, \ldots, w_h \rangle \]

be a maximum-weighted \((v, l+h)\)-path in \(P_l \odot V'\), where \(w_h = v, \langle u_1, \ldots, u_l \rangle \) is an \(l\)-path in \(P_l\), and \(w_1, \ldots, w_h\) are vertices in \(V'\). We prove the theorem by induction on \(h\).

Consider the case \(h = 1\). If \(P_l = \{p_0\}\) (in this case \(l = 0\)), then the set \(P_{l+1}\) returned by step 2.1 contains the (unique) \((v, 1)\)-path in \(P_l \odot V'\), which is obviously a maximum weighted \((v, 1)\)-path in \(P_l \odot V'\). If \(l > 0\), then when the \((u_1, l)\)-path \(\langle u_1, \ldots, u_l \rangle\) in \(P_l\) and the vertex \(w_h = w_1 = v\) are examined in step 2.2, the path \(\rho_{l+1}\) is constructed in step 2.3, so steps 2.4–2.6 ensure that a maximum-weighted \((v, l+1)\)-path in \(P_l \odot V'\) is included in the set \(P_{l+1}\). Therefore, for the case of \(h = 1\), with a probability 1, the set \(P_{l+h}\) returned by the algorithm contains a maximum-weighted \((v, l+h)\)-path in \(P_l \odot V'\). This proves the theorem for the case \(h = 1\).

Now suppose that \(h > 1\). Let \(h_1 = \lfloor h/2 \rfloor\). We rewrite the path \(\rho_{l+h}\) as 

\[ \rho_{l+h} = \langle u_1, \ldots, u_l, w_1, \ldots, w_h \rangle, \]

Let \(E_D\) be the event that step 3.1 includes the vertices \(w_1, \ldots, w_{h_1}\) in \(V_L\) and includes the vertices \(w_{h_{1}+1}, \ldots, w_h\) in \(V_R\). Let \(E_L\) be the event that the recursive call in step 3.2 returns a set \(P^L_{l+h_1}\) that contains a maximum-weighted \((w_{h_1}, l+h_1)\)-path in \(P_l \odot V_L\), and let \(E_R\) be the event that the recursive call in step 3.4 returns a set \(P^R_{l+h}\) that contains a maximum-weighted \((w_h, l+h)\)-path in \(P_{l+h} \odot V_R\). Note that the events \(E_D, E_L, E_R\) are not independent. For example, if the vertex \(w_{h_1}\) is not in the set \(V_L\) (thus, the event \(E_D\) fails), then \(Pr[E_L] = 0\).

We consider what happens under the event \(E_D \cap E_L \cap E_R\). Under the event \(E_D\), the vertices \(w_1, \ldots, w_{h_1}\) are in \(V_L\) and the vertices \(w_{h_{1}+1}, \ldots, w_h\) are in \(V_R\). Thus, the path \(\rho_{l+h_1} = \langle u_1, \ldots, u_l, w_1, \ldots, w_h \rangle\) is a \((w_{h_1}+1, l+h_1)\)-path in \(P_l \odot V_L\). The path \(\rho_{l+h}\) must be a maximum-weighted \((w_{h_1}, l+h_1)\)-path in \(P_l \odot V_L\) otherwise, a maximum-weighted \((w_{h_1}, l+h_1)\)-path \(\rho_{l+h_1}'\) in \(P_l \odot V_L\) concatenated with the path \((w_{h_{1}+1}, \ldots, w_h)\) would give a \((w_h, l+h)\)-path in \(P_l \odot V'\) whose weight is larger than the maximum-weighted \((w_h, l+h)\)-path \(\rho_{l+h}\) in \(P_l \odot V'\) (note that under the event \(E_D\), none of the vertices \(w_{h_{1}+1}, \ldots, w_h\) can be on \(\rho_{l+h_1}'\)). Now since the event \(E_L\) assumes that the set \(P^L_{l+h_1}\) returned in step 3.2 contains a maximum-weighted \((w_{h_1}, l+h_1)\)-path in \(P_l \odot V_L\), the event \(E_D \cap E_L \cap E_R\), the set \(P^L_{l+h_1}\), returned in step 3.2 contains a maximum-weighted \((w_{h_1}, l+h_1)\)-path \(\tilde{\rho}_{l+h_1}\) in \(P_l \odot V_L\) whose weight is equal to that of \(\rho_{l+h_1}\) (note that \(\tilde{\rho}_{l+h_1}\) and \(\rho_{l+h_1}\) are not necessarily the same). Now the path \(\tilde{\rho}_{l+h_1}\) concatenated with the path \(\langle w_{h_{1}+1}, \ldots, w_h \rangle\) is a \((w_{h_1}, l+h_1) + \lfloor h/2 \rfloor\)-path in \(P^L_{l+h_1} \odot V_R\). Since all paths in the set \(P^L_{l+h_1}\) are \((l+h_1)\)-paths in \(P_l \odot V_L\), every \((w_{h_1}, l+h_1) + \lfloor h/2 \rfloor\)-path in \(P^L_{l+h_1} \odot V_R\) is a \((w_h, l+h)\)-path in \(P_l \odot V'\). In particular, the \((w_{h_1}, l+h_1) + \lfloor h/2 \rfloor\)-path \(\bar{\rho}_{l+h_1}\) in \(P^L_{l+h_1}\) and the path \(\langle w_{h_1}, \ldots, w_h \rangle\) is a \((w_h, l+h)\)-path in \(P_l \odot V'\) whose weight is equal to that of \(\rho_{l+h}\). This implies that a maximum-weighted \((w_{h_1}, l+h_1) + \lfloor h/2 \rfloor\)-path in \(P^L_{l+h_1} \odot V_R\) is also a maximum-weighted \((w_{h_1}, l+h)\)-path in \(P_l \odot V'\). Now since the event \(E_R\) assumes that step 3.4 returns a set \(P^R_{l+h}\) that contains a maximum-weighted \((w_{h_1}, l+h_1) + \lfloor h/2 \rfloor\)-path in \(P^L_{l+h_1} \odot V_R\), under the event \(E_D \cap E_L \cap E_R\), the set \(P^R_{l+h}\) returned in step 3.4 contains a maximum-weighted \((w_{h_1}, l+h)\)-path in \(P_l \odot V'\), which is added to the set \(P_{l+h}\) in steps 3.5-3.8. In conclusion, for the given vertex \(w_h = v\), under the event \(E_D \cap E_L \cap E_R\), the set \(P_{l+h}\) contains a maximum-weighted \((v, l+h)\)-path in \(P_l \odot V'\).

Let \(E\) be the event that in an execution of steps 3.1–3.4, the set \(P^R_{l+h}\) constructed in step 3.4 contains a maximum-weighted \((v, l+h)\)-path in \(P_l \odot V'\), then by the above discussion, \(E_D \cup E_L \cup E_R \subseteq E\). So,

\[ Pr[E] \geq Pr[E_D \cup E_L \cup E_R] = Pr[E_D] \cdot Pr[E_L | E_D] \cdot Pr[E_R | E_D \cap E_L]. \quad (14) \]

Because we randomly partition the vertex set \(V'\), each vertex on the path \(\rho_{l+h}\) has an equal probability (\(= 1/2\)) to be placed either in \(V_L\) or in \(V_R\). Thus, \(Pr[E_D] = 1/2^h\). Under the event \(E_D\), the \((w_{h_1}, l+h_1)\)-path \(\rho_{l+h_1}\) is a \((w_{h_1}, l+h_1)\)-path in \(P_l \odot V_L\). Thus, the condition in the theorem (i.e., there are \((w_{h_1}, l+h_1)\)-paths in \(P_l \odot V_L\)) is satisfied for the instance \((P_l, V_L, h_1)\) so we can apply the inductive hypothesis on \(h_1 < h\), which gives \(Pr[E_L | E_D] > 1 - 1/e\). Finally, under the event \(E_D \cap E_L\), the path \(\rho_{l+h}\) is a \((w_{h_1}, l+h_1) + \lfloor h/2 \rfloor\)-path in \(P^L_{l+h} \odot V_R\) (and every maximum-weighted \((w_{h_1}, l+h_1) + \lfloor h/2 \rfloor\)-path in
\(P_{l+h_1}^L \odot V_R\) is a maximum-weighted \((w_h, l + h)\)-path in \(P_1 \odot V'\), thus the instance \((P_{l+h_1}^L, V_R, |h/2|)\) satisfies the condition of the theorem (i.e., there are \((w_h, (l + h_1) + |h/2|)\)-paths in \(P_{l+h_1}^L \odot V_R\)), so we can again apply the inductive hypothesis on \(|h/2| < h\), which gives \(\Pr[E_R | E_D \cap E_L] > 1 - 1/e\). Bringing all these into (14), we get

\[
\Pr[E] \geq \Pr[E_D] \cdot \Pr[E_L | E_D] \cdot \Pr[E_R | E_D \cap E_L] > \frac{1}{2^h} \cdot \left(1 - \frac{1}{e}\right)^2 > \frac{1}{3 \cdot 2^h}.
\]

By steps 3.5-3.8 of the algorithm, the set \(P_{l+h}\) is updated based on \(P_{l+h}^R\). Thus, the above analysis shows that with a probability larger than \(1/(3 \cdot 2^h)\), each execution of the steps 3.1-3.8 includes a maximum-weighted \((v, l + h)\)-path in \(P_1 \odot V'\) in the set \(P_{l+h}\). Since step 3 of the algorithm loops \(3 \cdot 2^h\) times, the overall probability that the algorithm returns a set \(P_{l+h}\) that contains a maximum-weighted \((v, l + h)\)-path in \(P_1 \odot V'\) is larger than

\[
1 - \left(1 - \frac{1}{3 \cdot 2^h}\right)^{3 \cdot 2^h} > 1 - \frac{1}{e}.
\]

Thus, the inductive proof goes through, and the theorem gets proved. \(\square\)

We study the time and space complexity of the algorithm \textsc{FindPaths} in the next subsection.

### 7.2 The complexity of the algorithm \textsc{FindPaths}

Fix an input graph \(G\). Suppose that \(G\) has \(n\) vertices and \(q\) edges. Let \(m\) be the size of the input \((P_1, V', h)\) to the algorithm \textsc{FindPaths}, which is of the order \(O(l \cdot n + q)\).

We first study the space complexity of the algorithm. A recursive call to \textsc{FindPaths}(\(P_1, V', h\)) uses \(O(n(l + h))\) extra space for storing the sets \(P_{l+h_1}^L, P_{l+h}^R,\) and \(P_{l+h}\) (note that for each vertex \(v\) in the graph \(G\), each of these sets contains at most one \((v, *)\)-path). Since the recursive depth of the algorithm \textsc{FindPaths}(\(P_1, V', h\)) is \(O(\log h)\), we conclude that the space complexity of the algorithm \textsc{FindPaths}(\(P_1, V', h\)) is \(O(n(l + h) \log h + q)\). In particular, if we apply the algorithm on an instance \((G, k)\) of the \textsc{Max-Path} problem, then the space complexity of the algorithm is \(O(nk \log k + q)\). Recall that the straightforward implementation we suggested for the problem uses \(O(kn^2)\) space.

We now consider the time complexity of the algorithm \textsc{FindPaths}. Let \(T(h, m)\) be the running time of the algorithm \textsc{FindPaths}(\(P_1, V', h\)), where \(m\) is the size of the instance \((P_1, V', h)\). Clearly we have \(T(1, m) = O(m)\). From the algorithm, we have the following recurrence relation when \(h > 1:\)

\[
T(h, m) \leq 3 \cdot 2^h [cm + T(\lfloor h/2 \rfloor, m) + T(\lceil h/2 \rceil, m)],
\]

where \(c > 0\) is a constant. We show how to solve this recurrence equation.

**Lemma 7.2** The function \(T(h, m)\) in (15) satisfies \(T(h, m) \leq c_04^h h^3 m\), where \(c_0\) is a constant.

**Proof.** To simplify our descriptions, we assume that \(h\) is a power of 2. Thus, the recurrence relation (15) can be written as

\[
T(h, m) \leq 3 \cdot 2^h [cm + 2T(h/2, m)] = 3c \cdot 2^h m + 6 \cdot 2^h T(h/2, m).
\]

Replacing with \(T(h/2, m) \leq 3c \cdot 2^{h/2} m + 6 \cdot 2^{h/2} T(h/2^2, m)\), we get

\[
T(h, m) \leq 3c \cdot 2^h m + 6c \cdot 2^{h/2} m + 6^2 \cdot 2^{h/2} T(h/2^2, m).
\]

For a general integer \(p, 0 < p \leq \log h\), we can derive:

\[
T(h, m) \leq 3cm \cdot 2^h \sum_{i=0}^{p-1} 6^i 2^{h-i} + 6^p 2^{2h-h/2^p-1} T(h/2^p, m).
\]
Now let \( p = \log h \) in (17), and recall that \( T(1,m) \leq c'm \) for a constant \( c' \), we get
\[
T(h,m) \leq 4^h 6^{\log h} m(c + c') = 4^h 6^{\log h} m(c + c') \leq c_0 h^3 m,
\]
where \( c_0 = c + c' \) is a constant, and note that \( \log 6 < 3 \). This completes the proof of the lemma.

We can conclude with

**Theorem 7.3** The algorithm \( \text{FindPaths}(P_t, V', h) \) runs in \( O(4^h h^3 m) \) time and \( O(n(l + h) \log h + q) \) space on a graph of \( n \) vertices and \( q \) edges, where \( m = O(1 \cdot n + q) \) is the size of the instance \((P_t, V', h)\).

### 7.3 Final remarks

To amplify the success probability of the algorithm \( \text{FindPaths} \) for constructing a maximum-weighted \((v, k)\)-path for a vertex \( v \) in a graph \( G = (V, E) \), we simply run the algorithm \( \text{FindPaths}(\{\rho\}, V, k) \) \( t \) times for a sufficiently large constant \( t \), and pick the largest \((v, k)\)-path constructed in this process. By Theorem 7.1, with a probability less than \( 1/e \), an execution of the algorithm \( \text{FindPaths}(\{\rho\}, V, k) \) fails in finding a maximum-weighted \((v, k)\)-path in \( G \). Therefore, the probability that all \( t \) executions of the algorithm fail in finding a maximum-weighted \((v, k)\)-path in \( G \) is less than \( 1/e^t \). In conclusion, the probability that this process finds a maximum-weighted \((v, k)\)-path in \( G \) is larger than \( 1 - 1/e^t \). For instance, if we let \( t = 10 \), then the probability that this process finds a maximum-weighted \((v, k)\)-path in \( G \) is larger than 0.9999. Note that this does not change the asymptotic order of the time and space complexity of the algorithm.

To solve the \textsc{max-path} problem, for an instance \((G, k)\) of \textsc{max-path}, where \( G = (V, E) \), we again run the algorithm \( \text{FindPaths}(\{\rho\}, V, k) \) \( t \) times. However, now for each execution we pick from its output \( P_{0:k} \) the \((v, k)\)-path for some vertex \( v \) whose weight is the largest among all paths in \( P_{0:k} \). The output of this process is the \( k \)-path with the largest weight among all those we picked for the \( t \) executions of the algorithm. Note that this process may pick \( k \)-paths with no common end in two executions of the algorithm. Therefore, we need to justify the validity of this process since we do not know which vertex is an end of a maximum-weighted \( k \)-path in the graph \( G \). Let \( w_0 \) be an end of a maximum-weighted \( k \)-path in \( G \). Note that Theorem 7.1 holds true for any fixed vertex \( v \). In particular, with a probability larger than \( 1 - 1/e \), the set \( P_{0:k} \) returned by an execution of the algorithm contains a maximum-weighted \((w_0, k)\)-path \( \rho \) (in \( \{\rho\} \cap V \)), so the probability that all the \( t \) executions of the algorithm fail in finding a maximum-weighted \((w_0, k)\)-path is less than \( 1/e^t \), and this process returns a maximum-weighted \((w_0, k)\)-path \( \rho \) with a probability larger than \( 1 - 1/e^t \) (even we do not know what \( w_0 \) is). But since there is a \((w_0, k)\)-path that is a maximum-weighted \( k \)-path in \( G \), the path \( \rho \) must also be a maximum-weighted \( k \)-path in \( G \). Thus, with such a probability, the path whose weight is the largest among all executions of the algorithm is a maximum-weighted \( k \)-path in \( G \).

**Theorem 7.4** There is a randomized algorithm of time \( O(4^k k^3 m) \) and space \( O(nk \log k + m) \) that solves the \textsc{max-path} problem with an arbitrarily small error bound.

Theorem 7.4 gives an algorithm for the \textsc{max-path} problem that improves the algorithm \textsc{PathPerm} for the problem, which is based on random permutation and has time complexity \( O(k(n + m)k!) \), as well as the algorithm \textsc{ColorPath} for the problem, which is based on color-coding and has time complexity \( O(5.44^k (n + m)) \) (see previous section). The algorithm \textsc{FindPaths} can also be used directly to solve the \textsc{max-path} problem on directed graphs, as long as we interpret the edge \([w, u]\) in step 2.2 of the algorithm as a directed edge from \( w \) to \( u \). The proof of Theorem 7.1 can be applied to directed graphs with no change. Based on the randomized divide-and-conquer technique presented in this section, we can develop faster parameterized algorithms for other NP-hard problems, such as the \textsc{3D-Matching} problem and the \textsc{3-Set Packing} problem. See reference [3]. Currently, the algorithm in Theorem 7.4 is still the fastest algorithm for solving the \textsc{max-path} problem on weighted graphs. For unweighted graphs, Williams [19] has developed an \( O(2^k n^{O(1)}) \) randomized algorithm, which is based on algebraic techniques.