The improved static structure continued

Paweł Gawrychowski

2 czerwca 2014

Static predecessor search

Preprocess a sequence of integers $0 \leq a_1 < a_2 < \ldots < a_n < U$ so that later, given any x, we can efficiently determine the largest i such that $a_i \leq x$.

Last week we tried to prove the following.

Beame and Fich 2002

If the allowed space is $\mathcal{O}(n^2)$, then one can achieve $\mathcal{O}(\frac{\log\log U}{\log\log\log U})$ query time.

We showed that their result follows from THE REDUCTION.

Static predecessor search

Preprocess a sequence of integers $0 \leq a_1 < a_2 < \ldots < a_n < U$ so that later, given any x, we can efficiently determine the largest i such that $a_i \leq x$.

Last week we tried to prove the following.

Beame and Fich 2002

If the allowed space is $\mathcal{O}(n^2)$, then one can achieve $\mathcal{O}(\frac{\log\log U}{\log\log\log U})$ query time.

We showed that their result follows from THE REDUCTION.

Main tool

THE REDUCTION

Say that we have an instance of static predecessor search with n keys of length ℓ. Choose integer parameters $q \geq 2$ dividing n and $h \geq 2$ dividing ℓ. We can reduce the original problem into a number of subproblems, each of which is easier in one of two ways:

length-reduced the length of the keys is at most $\frac{\ell}{h}$,

size-reduced the number of keys is reduced to at most $\frac{n}{q}$,

The reduction takes $\mathcal{O}(1)$ time. The space is $\mathcal{O}(q^{3h} + m)$, where $m \in [0, n]$ is some number determined by the reduction. The total size of all size-reduced subproblems is at most $n - m$ and the total size of all length-reduced subproblems is m.

Additionally, we want the reduction to split the original problem into more than one subproblem. This is easy to ensure (for example, by always removing the largest element).

Main tool

THE REDUCTION

Say that we have an instance of static predecessor search with n keys of length ℓ. Choose integer parameters $q \geq 2$ dividing n and $h \geq 2$ dividing ℓ. We can reduce the original problem into a number of subproblems, each of which is easier in one of two ways:

length-reduced the length of the keys is at most $\frac{\ell}{h}$,

size-reduced the number of keys is reduced to at most $\frac{n}{q}$,

The reduction takes $\mathcal{O}(1)$ time. The space is $\mathcal{O}(q^{3h} + m)$, where $m \in [0, n]$ is some number determined by the reduction. The total size of all size-reduced subproblems is at most $n - m$ and the total size of all length-reduced subproblems is m.

Additionally, we want the reduction to split the original problem into more than one subproblem. This is easy to ensure (for example, by always removing the largest element).

Main tool

THE REDUCTION

Say that we have an instance of static predecessor search with n keys of length ℓ. Choose integer parameters $q \geq 2$ dividing n and $h \geq 2$ dividing ℓ. We can reduce the original problem into a number of subproblems, each of which is easier in one of two ways:

length-reduced the length of the keys is at most $\frac{\ell}{h}$,

size-reduced the number of keys is reduced to at most $\frac{n}{q}$,

The reduction takes $\mathcal{O}(1)$ time. The space is $\mathcal{O}(q^{3h} + m)$, where $m \in [0, n]$ is some number determined by the reduction. The total size of all size-reduced subproblems is at most $n - m$ and the total size of all length-reduced subproblems is m.

Additionally, we want the reduction to split the original problem into more than one subproblem. This is easy to ensure (for example, by always removing the largest element).

Main tool

THE REDUCTION

Say that we have an instance of static predecessor search with n keys of length ℓ. Choose integer parameters $q \geq 2$ dividing n and $h \geq 2$ dividing ℓ. We can reduce the original problem into a number of subproblems, each of which is easier in one of two ways:

length-reduced the length of the keys is at most $\frac{\ell}{h}$,

size-reduced the number of keys is reduced to at most $\frac{n}{q}$,

The reduction takes $\mathcal{O}(1)$ time. The space is $\mathcal{O}(q^{3h} + m)$, where $m \in [0, n]$ is some number determined by the reduction. The total size of all size-reduced subproblems is at most $n - m$ and the total size of all length-reduced subproblems is m.

Additionally, we want the reduction to split the original problem into more than one subproblem. This is easy to ensure (for example, by always removing the largest element).

Main tool

THE REDUCTION

Say that we have an instance of static predecessor search with n keys of length ℓ. Choose integer parameters $q \geq 2$ dividing n and $h \geq 2$ dividing ℓ. We can reduce the original problem into a number of subproblems, each of which is easier in one of two ways:

length-reduced the length of the keys is at most $\frac{\ell}{h}$,

size-reduced the number of keys is reduced to at most $\frac{n}{q}$,

The reduction takes $\mathcal{O}(1)$ time. The space is $\mathcal{O}(q^{3h} + m)$, where $m \in [0, n]$ is some number determined by the reduction. The total size of all size-reduced subproblems is at most $n - m$ and the total size of all length-reduced subproblems is m.

Additionally, we want the reduction to split the original problem into more than one subproblem. This is easy to ensure (for example, by always removing the largest element).

The reduction

The main trick is to use parallel hashing. Say that we want to store a set Z of q keys, each consisting of h characters (not necessarily binary). We denote this by $x = x_1 x_2 \ldots x_h$.

Lemma

Using additional space $\mathcal{O}(q^{3h})$ we can preprocess such a set, so that given a query x, we can find in $\mathcal{O}(1)$ time the longest common prefix (in whole characters) between x and any key in Z.

The idea is simple and neat: use h instances of some perfect hashing in parallel.

The reduction

The main trick is to use parallel hashing. Say that we want to store a set Z of q keys, each consisting of h characters (not necessarily binary). We denote this by $x = x_1 x_2 \ldots x_h$.

Lemma

Using additional space $\mathcal{O}(q^{3h})$ we can preprocess such a set, so that given a query x, we can find in $\mathcal{O}(1)$ time the longest common prefix (in whole characters) between x and any key in Z.

The idea is simple and neat: use h instances of some perfect hashing in parallel.

The reduction

The main trick is to use parallel hashing. Say that we want to store a set Z of q keys, each consisting of h characters (not necessarily binary). We denote this by $x = x_1 x_2 \ldots x_h$.

Lemma

Using additional space $\mathcal{O}(q^{3h})$ we can preprocess such a set, so that given a query x, we can find in $\mathcal{O}(1)$ time the longest common prefix (in whole characters) between x and any key in Z.

The idea is simple and neat: use h instances of some perfect hashing in parallel.

Parallel hashing

We want to find a hash function h, which is perfect on every "column". The function will be applied to all columns at once. We choose H at random from some family of functions $\mathcal{H}$ and require that:

1. H hashes into small $[m]$,
2. for any i and two characters $x \neq y$ which appear in the i-th column, the chance that $H(x) = H(y)$ is less than $\frac{2}{m}$,
3. we can compute $H(x_1), H(x_2), \ldots, H(x_h)$ in $\mathcal{O}(1)$ time, assuming that all $x_1, x_2, \ldots, x_h$ are given in a single machine word.

We choose $m = q^2 h$. Then, if we choose $H \in \mathcal{H}$ at random, the chance that we get a collision in the i-th column is less than $\frac{q^2}{m} = \frac{1}{h}$. Hence the chance that we get collision in **any** column is less than one, so it is possible to choose H so that there is no collision.

We choose a function of the form $H_a(x) = \left\lfloor \frac{a \cdot x}{2^\alpha} \right\rfloor \bmod 2^\beta$.

See the blackboard for an explanation of why it works.

Parallel hashing

We want to find a hash function h, which is perfect on every "column". The function will be applied to all columns at once. We choose H at random from some family of functions $\mathcal{H}$ and require that:

1. H hashes into small $[m]$,
2. for any i and two characters $x \neq y$ which appear in the i-th column, the chance that $H(x) = H(y)$ is less than $\frac{2}{m}$,
3. we can compute $H(x_1), H(x_2), \ldots, H(x_h)$ in $\mathcal{O}(1)$ time, assuming that all $x_1, x_2, \ldots, x_h$ are given in a single machine word.

We choose $m = q^2 h$. Then, if we choose $H \in \mathcal{H}$ at random, the chance that we get a collision in the i-th column is less than $\frac{q^2}{m} = \frac{1}{h}$. Hence the chance that we get collision in **any** column is less than one, so it is possible to choose H so that there is no collision.

We choose a function of the form $H_a(x) = \left\lfloor \frac{a \cdot x}{2^\alpha} \right\rfloor \bmod 2^\beta$.

See the blackboard for an explanation of why it works.

Parallel hashing

We want to find a hash function h, which is perfect on every "column". The function will be applied to all columns at once. We choose H at random from some family of functions $\mathcal{H}$ and require that:

1. H hashes into small $[m]$,
2. for any i and two characters $x \neq y$ which appear in the i-th column, the chance that $H(x) = H(y)$ is less than $\frac{2}{m}$,
3. we can compute $H(x_1), H(x_2), \ldots, H(x_h)$ in $\mathcal{O}(1)$ time, assuming that all $x_1, x_2, \ldots, x_h$ are given in a single machine word.

We choose $m = q^2 h$. Then, if we choose $H \in \mathcal{H}$ at random, the chance that we get a collision in the i-th column is less than $\frac{q^2}{m} = \frac{1}{h}$. Hence the chance that we get collision in **any** column is less than one, so it is possible to choose H so that there is no collision.

We choose a function of the form $H_a(x) = \left\lfloor \frac{a \cdot x}{2^\alpha} \right\rfloor \bmod 2^\beta$.

See the blackboard for an explanation of why it works.

Parallel hashing

We want to find a hash function h, which is perfect on every "column". The function will be applied to all columns at once. We choose H at random from some family of functions $\mathcal{H}$ and require that:

1. H hashes into small $[m]$,
2. for any i and two characters $x \neq y$ which appear in the i-th column, the chance that $H(x) = H(y)$ is less than $\frac{2}{m}$,
3. we can compute $H(x_1), H(x_2), \ldots, H(x_h)$ in $\mathcal{O}(1)$ time, assuming that all $x_1, x_2, \ldots, x_h$ are given in a single machine word.

We choose $m = q^2 h$. Then, if we choose $H \in \mathcal{H}$ at random, the chance that we get a collision in the i-th column is less than $\frac{q^2}{m} = \frac{1}{h}$. Hence the chance that we get collision in **any** column is less than one, so it is possible to choose H so that there is no collision.

We choose a function of the form $H_a(x) = \left\lfloor \frac{a \cdot x}{2^\alpha} \right\rfloor \bmod 2^\beta$.

See the blackboard for an explanation of why it works.

Parallel hashing

We want to find a hash function h, which is perfect on every "column". The function will be applied to all columns at once. We choose H at random from some family of functions $\mathcal{H}$ and require that:

1. H hashes into small $[m]$,
2. for any i and two characters $x \neq y$ which appear in the i-th column, the chance that $H(x) = H(y)$ is less than $\frac{2}{m}$,
3. we can compute $H(x_1), H(x_2), \ldots, H(x_h)$ in $\mathcal{O}(1)$ time, assuming that all $x_1, x_2, \ldots, x_h$ are given in a single machine word.

We choose $m = q^2 h$. Then, if we choose $H \in \mathcal{H}$ at random, the chance that we get a collision in the i-th column is less than $\frac{q^2}{m} = \frac{1}{h}$. Hence the chance that we get collision in **any** column is less than one, so it is possible to choose H so that there is no collision.

We choose a function of the form $H_a(x) = \left\lfloor \frac{a \cdot x}{2^\alpha} \right\rfloor \bmod 2^\beta$.

See the blackboard for an explanation of why it works.

Parallel hashing

Now we can compute $H(x) = H(x_1), H(x_2), \ldots, H(x_h)$ in $\mathcal{O}(1)$ time. For the time being assume that $h \leq q$, then because $m = q^2 h$, considering all possible values of x, there are just q^{3h} different values of $H(x)$!

For every such different possible $H(x)$, we store the answer, i.e., z which maximizes the length of the longest common prefix between $H(x)$ and $H(z)$, where $z \in Z$.

Because H is injective on every column, z which maximizes the length of the longest common prefix between $H(x)$ and $H(z)$ also maximizes the longest common prefix between x and z.

Parallel hashing

Now we can compute $H(x) = H(x_1), H(x_2), \ldots, H(x_h)$ in $\mathcal{O}(1)$ time. For the time being assume that $h \leq q$, then because $m = q^2 h$, considering all possible values of x, there are just q^{3h} different values of $H(x)$! For every such different possible $H(x)$, we store the answer, i.e., z which maximizes the length of the longest common prefix between $H(x)$ and $H(z)$, where $z \in Z$.

Because H is injective on every column, z which maximizes the length of the longest common prefix between $H(x)$ and $H(z)$ also maximizes the longest common prefix between x and z.

Parallel hashing

Now we can compute $H(x) = H(x_1), H(x_2), \ldots, H(x_h)$ in $\mathcal{O}(1)$ time. For the time being assume that $h \leq q$, then because $m = q^2 h$, considering all possible values of x, there are just q^{3h} different values of $H(x)$! For every such different possible $H(x)$, we store the answer, i.e., z which maximizes the length of the longest common prefix between $H(x)$ and $H(z)$, where $z \in Z$.

Because H is injective on every column, z which maximizes the length of the longest common prefix between $H(x)$ and $H(z)$ also maximizes the longest common prefix between x and z.

Why $h \leq q$?

We claim that if $h > q$, then some of the columns don't really matter. Imagine the trie of depth h containing all the keys. Then there are at most q depths such that there is a branching (i.e., with outdegree > 1) node at such depth. We call these depths important.

We consider a set of "truncated" keys. From every key we keep only characters corresponding to edges at important depths. We apply the previous method on the resulting sufficiently short keys.

Now we claim that the predecessor x in the original set can be found in $\mathcal{O}(1)$ once we know the predecessor of x in the set of shorter keys, if we allow additional space $\mathcal{O}(q \cdot h)$ to store some data for every node. This is because we want to navigate the trie following the characters of x. If we ignore the edges at all non-important depths, we might follow the unique edge outgoing from some node, even though the next character of x is actually different. But the found key still maximizes the length of the longest common prefix, see the blackboard.

Remember this trick.

Why $h \leq q$?

We claim that if $h > q$, then some of the columns don't really matter. Imagine the trie of depth h containing all the keys. Then there are at most q depths such that there is a branching (i.e., with outdegree > 1) node at such depth. We call these depths important.

We consider a set of "truncated" keys. From every key we keep only characters corresponding to edges at important depths. We apply the previous method on the resulting sufficiently short keys.

Now we claim that the predecessor x in the original set can be found in $\mathcal{O}(1)$ once we know the predecessor of x in the set of shorter keys, if we allow additional space $\mathcal{O}(q \cdot h)$ to store some data for every node. This is because we want to navigate the trie following the characters of x. If we ignore the edges at all non-important depths, we might follow the unique edge outgoing from some node, even though the next character of x is actually different. But the found key still maximizes the length of the longest common prefix, see the blackboard.

Remember this trick.

Why $h \leq q$?

We claim that if $h > q$, then some of the columns don't really matter. Imagine the trie of depth h containing all the keys. Then there are at most q depths such that there is a branching (i.e., with outdegree > 1) node at such depth. We call these depths important.

We consider a set of "truncated" keys. From every key we keep only characters corresponding to edges at important depths. We apply the previous method on the resulting sufficiently short keys.

Now we claim that the predecessor x in the original set can be found in $\mathcal{O}(1)$ once we know the predecessor of x in the set of shorter keys, if we allow additional space $\mathcal{O}(q \cdot h)$ to store some data for every node.

This is because we want to navigate the trie following the characters of x. If we ignore the edges at all non-important depths, we might follow the unique edge outgoing from some node, even though the next character of x is actually different. But the found key still maximizes the length of the longest common prefix, see the blackboard.

Remember this trick.

Why $h \leq q$?

We claim that if $h > q$, then some of the columns don't really matter. Imagine the trie of depth h containing all the keys. Then there are at most q depths such that there is a branching (i.e., with outdegree > 1) node at such depth. We call these depths important.

We consider a set of "truncated" keys. From every key we keep only characters corresponding to edges at important depths. We apply the previous method on the resulting sufficiently short keys.

Now we claim that the predecessor x in the original set can be found in $\mathcal{O}(1)$ once we know the predecessor of x in the set of shorter keys, if we allow additional space $\mathcal{O}(q \cdot h)$ to store some data for every node. This is because we want to navigate the trie following the characters of x. If we ignore the edges at all non-important depths, we might follow the unique edge outgoing from some node, even though the next character of x is actually different. But the found key still maximizes the length of the longest common prefix, see the blackboard.

Remember this trick.

The reduction

Let Y be the original set of keys. Again, we treat every key x as a sequence of h characters $x_1, x_2, \ldots, x_h$.

Indirection, first try

We choose q evenly spaced keys, call them Z. We apply parallel hashing lemma to Z, this takes $\mathcal{O}(q^{3h})$ space.

The keys between two consecutive keys from Z form a size-reduced subproblem. Edges outgoing from the same node form a length-reduced subproblem. Well, more or less...

The reduction

Let Y be the original set of keys. Again, we treat every key x as a sequence of h characters $x_1, x_2, \ldots, x_h$.

Indirection, first try

We choose q evenly spaced keys, call them Z. We apply parallel hashing lemma to Z, this takes $\mathcal{O}(q^{3h})$ space.

The keys between two consecutive keys from Z form a size-reduced subproblem. Edges outgoing from the same node form a length-reduced subproblem. Well, more or less...

The reduction

Let Y be the original set of keys. Again, we treat every key x as a sequence of h characters $x_1, x_2, \ldots, x_h$.

Indirection, first try

We choose q evenly spaced keys, call them Z. We apply parallel hashing lemma to Z, this takes $\mathcal{O}(q^{3h})$ space.

The keys between two consecutive keys from Z form a size-reduced subproblem. Edges outgoing from the same node form a length-reduced subproblem. Well, more or less...

The reduction

We imagine the whole situation as follows: we have a $2^{\ell/h}$-ary trie containing all the keys. Z corresponds to some subtrie of that trie. The depth of the trie is h. Consider an arbitrary query x.

We consider the next character of x after its longest common prefix with any $z \in Z$. This next characters goes outside of the subtrie spanned by Z. If there is at least one keys in $Y \setminus Z$ with such next character, we form a size-reduced subproblem containing the keys in $Y \setminus Z$ with such prefix.

More precisely, all but one: we store the maximum separately. As in the van Emde Boas tree, and for similar reasons.

The reduction

We imagine the whole situation as follows: we have a $2^{\ell/h}$-ary trie containing all the keys. Z corresponds to some subtrie of that trie. The depth of the trie is h. Consider an arbitrary query x.
We consider the next character of x after its longest common prefix with any $z \in Z$. This next characters goes outside of the subtrie spanned by Z. If there is at least one keys in $Y \setminus Z$ with such next character, we form a size-reduced subproblem containing the keys in $Y \setminus Z$ with such prefix.

More precisely, all but one: we store the maximum separately. As in the van Emde Boas tree, and for similar reasons.

The reduction

We imagine the whole situation as follows: we have a $2^{\ell/h}$-ary trie containing all the keys. Z corresponds to some subtrie of that trie. The depth of the trie is h. Consider an arbitrary query x.
We consider the next character of x after its longest common prefix with any $z \in Z$. This next characters goes outside of the subtrie spanned by Z. If there is at least one keys in $Y \setminus Z$ with such next character, we form a size-reduced subproblem containing the keys in $Y \setminus Z$ with such prefix.

More precisely, all but one: we store the maximum separately. As in the van Emde Boas tree, and for similar reasons.

The reduction

The other case is that the next character is not in the whole trie at all. Then the predecessor of x can be found by locating the predecessor of the next character among the children of the node corresponding to the longest common prefix.

This forms a length-reduced subproblem, because we want to find the predecessor of the next character in a set of characters.

Again, we store the maximum separately.

The space for this step is $\mathcal{O}(1)$ per a node of the subtrie, but this is OK.

The total size of all size-reduced subproblems is $n - m$, and the total size of all length-reduced subproblems is m, where m is $|Z|$ plus the number of edges with starting in the subtrie and going outside.

With a slightly more accurate accounting, the additional space can be made $\mathcal{O}(m)$ plus the huge table for the parallel hashing.

The reduction

The other case is that the next character is not in the whole trie at all. Then the predecessor of x can be found by locating the predecessor of the next character among the children of the node corresponding to the longest common prefix.
This forms a length-reduced subproblem, because we want to find the predecessor of the next character in a set of characters.

Again, we store the maximum separately.

The space for this step is $\mathcal{O}(1)$ per a node of the subtrie, but this is OK.

The total size of all size-reduced subproblems is $n - m$, and the total size of all length-reduced subproblems is m, where m is $|Z|$ plus the number of edges with starting in the subtrie and going outside.

With a slightly more accurate accounting, the additional space can be made $\mathcal{O}(m)$ plus the huge table for the parallel hashing.

The reduction

The other case is that the next character is not in the whole trie at all. Then the predecessor of x can be found by locating the predecessor of the next character among the children of the node corresponding to the longest common prefix.
This forms a length-reduced subproblem, because we want to find the predecessor of the next character in a set of characters.

Again, we store the maximum separately.

The space for this step is $\mathcal{O}(1)$ per a node of the subtrie, but this is OK.

The total size of all size-reduced subproblems is $n - m$, and the total size of all length-reduced subproblems is m, where m is $|Z|$ plus the number of edges with starting in the subtrie and going outside.

With a slightly more accurate accounting, the additional space can be made $\mathcal{O}(m)$ plus the huge table for the parallel hashing.

The reduction

The other case is that the next character is not in the whole trie at all. Then the predecessor of x can be found by locating the predecessor of the next character among the children of the node corresponding to the longest common prefix.
This forms a length-reduced subproblem, because we want to find the predecessor of the next character in a set of characters.

Again, we store the maximum separately.

The space for this step is $\mathcal{O}(1)$ per a node of the subtrie, but this is OK.

The total size of all size-reduced subproblems is $n - m$, and the total size of all length-reduced subproblems is m, where m is $|Z|$ plus the number of edges with starting in the subtrie and going outside.

With a slightly more accurate accounting, the additional space can be made $\mathcal{O}(m)$ plus the huge table for the parallel hashing.

The reduction

The other case is that the next character is not in the whole trie at all. Then the predecessor of x can be found by locating the predecessor of the next character among the children of the node corresponding to the longest common prefix.
This forms a length-reduced subproblem, because we want to find the predecessor of the next character in a set of characters.

Again, we store the maximum separately.

The space for this step is $\mathcal{O}(1)$ per a node of the subtrie, but this is OK.

The total size of all size-reduced subproblems is $n - m$, and the total size of all length-reduced subproblems is m, where m is $|Z|$ plus the number of edges with starting in the subtrie and going outside.

With a slightly more accurate accounting, the additional space can be made $\mathcal{O}(m)$ plus the huge table for the parallel hashing.

Questions?