Approximate Streaming Regular Pattern Matching

Adrien Mathieu

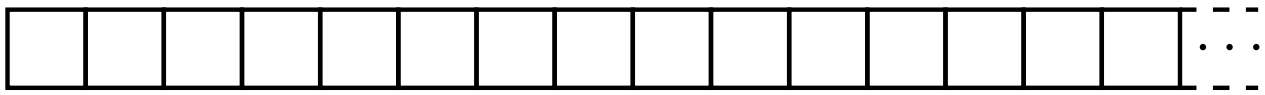
Thursday 29th, February

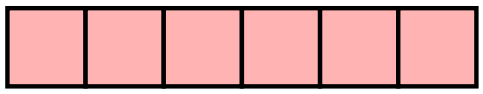
Problem statement

Approximate Streaming Regular Pattern Matching.

Problem statement

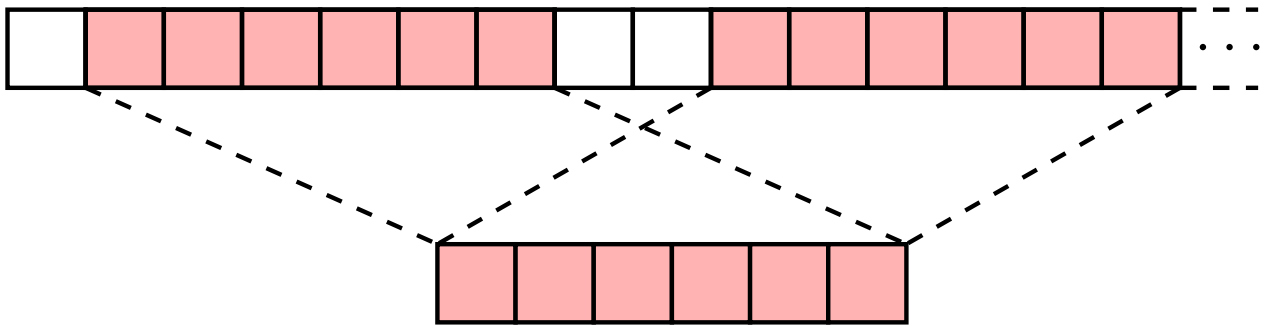
Approximate Streaming Regular Pattern Matching.





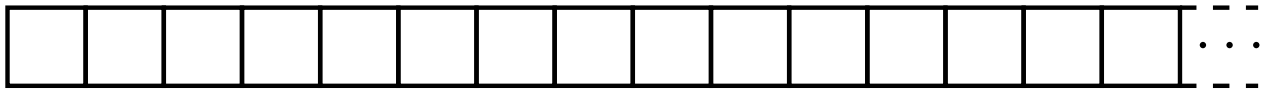
Problem statement

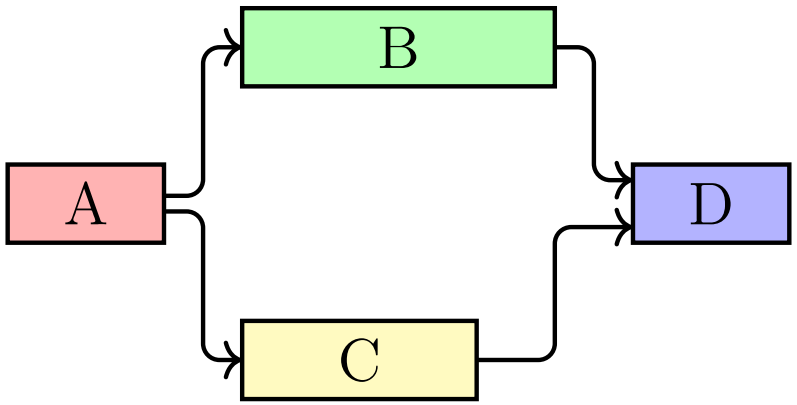
Approximate Streaming Regular Pattern Matching.



Problem statement

Approximate Streaming Regular Pattern Matching.

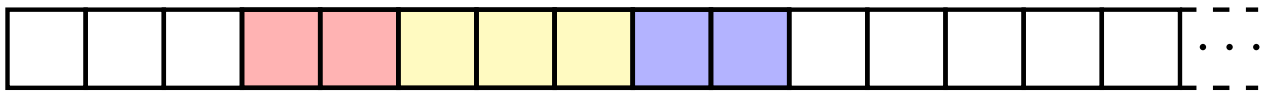


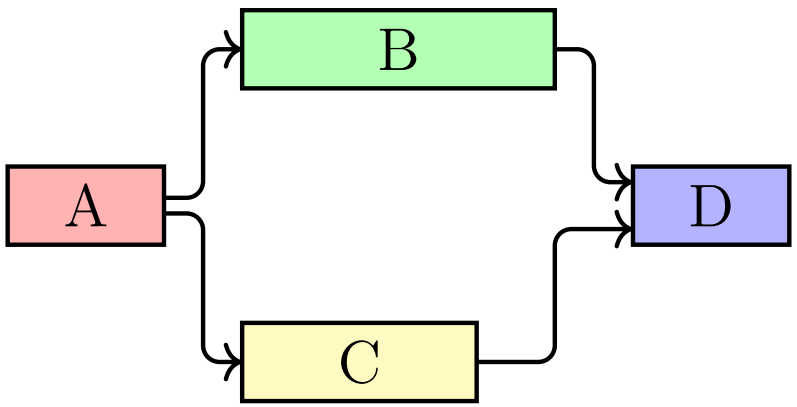


A(B|C)D

Problem statement

Approximate Streaming Regular Pattern Matching.

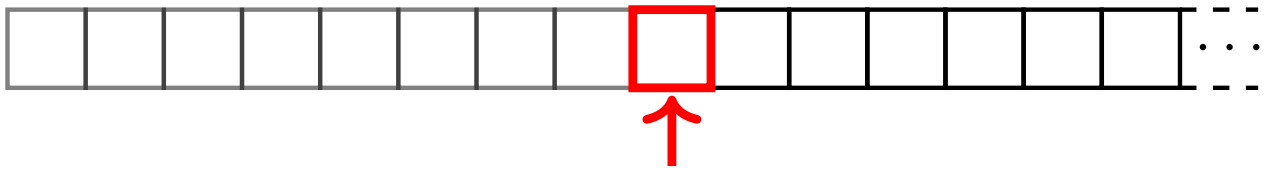




A(B|C)D

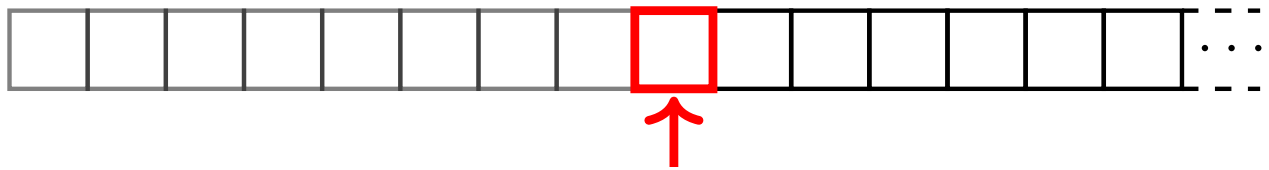
Problem statement

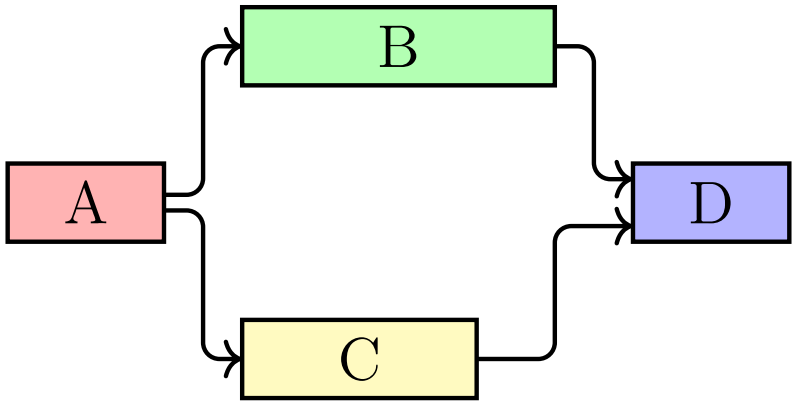
Approximate Streaming Regular Pattern Matching.



Problem statement

Approximate Streaming Regular Pattern Matching.

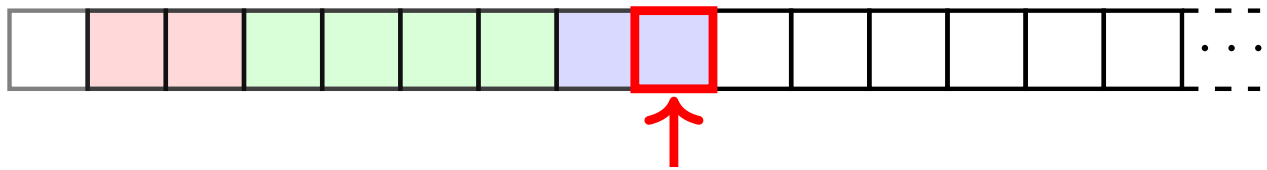


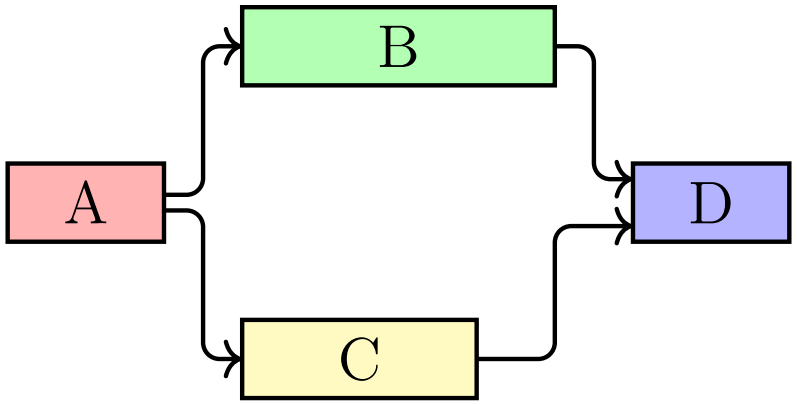


A(B|C)D

Problem statement

Approximate Streaming Regular Pattern Matching.

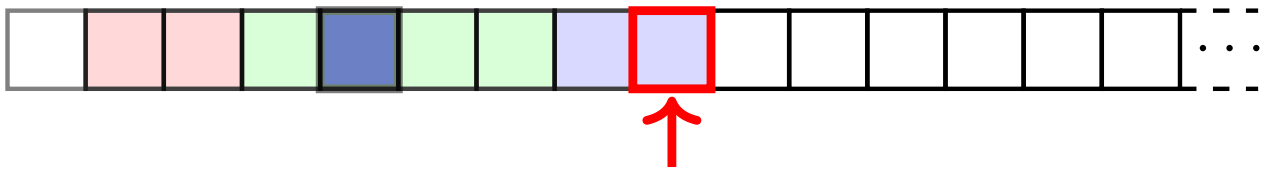


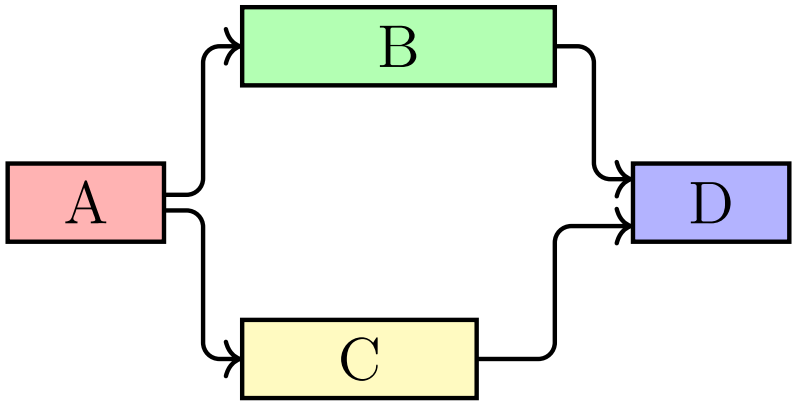


A(B|C)D

Problem statement

Approximate Streaming Regular Pattern Matching.





A(B|C)D

The Streaming Model

Why the streaming model?

The Streaming Model

Why the streaming model?

The naive algorithm has $\tilde{\mathcal{O}}(1)$ space complexity in read-only model.

The Streaming Model

Why the streaming model?

The naive algorithm has $\tilde{\mathcal{O}}(1)$ space complexity in read-only model.
In the streaming model, it has $\Theta(m)$ space complexity.

The Streaming Model

Why the streaming model?

The naive algorithm has $\tilde{\mathcal{O}}(1)$ space complexity in read-only model.
In the streaming model, it has $\Theta(m)$ space complexity.

In the streaming settings, we aim at a

sublinear space complexity

Without this more restrictive model, it's impossible to tell the difference!

The Streaming Model

Why the streaming model?

The naive algorithm has $\tilde{\mathcal{O}}(1)$ space complexity in read-only model.
In the streaming model, it has $\Theta(m)$ space complexity.

In the streaming settings, we aim at a

sublinear space complexity

Without this more restrictive model, it's impossible to tell the difference!

A sublinear space complexity solution is impossible!

The Streaming Model

Why the streaming model?

The naive algorithm has $\tilde{\mathcal{O}}(1)$ space complexity in read-only model.
In the streaming model, it has $\Theta(m)$ space complexity.

In the streaming settings, we aim at a

sublinear space complexity

Without this more restrictive model, it's impossible to tell the difference!

A sublinear space complexity deterministic solution is impossible!

Where are we going?

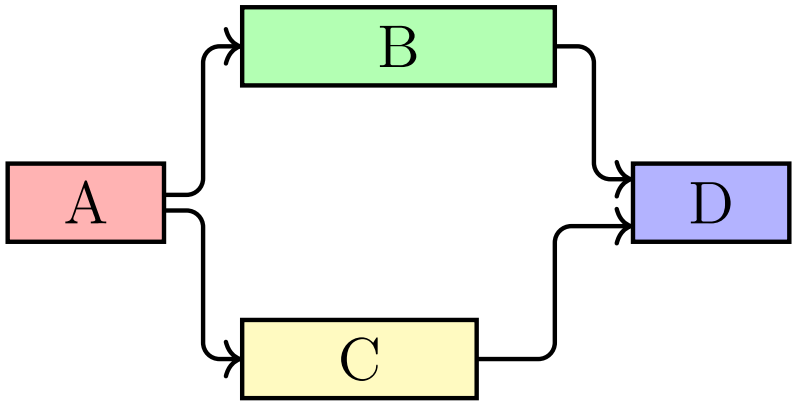
1. Exact Regular Expression Matching
   - Solution Sketch
   - Witnesses
   - Space-Time Tradeoff
   - Backtracking Solution
   - Occurrence Structure
   - Precomputing Backtracking Queries
   - Reducing to a Graph Problem
   - Reducing to a Circuit Problem
2. Generalizing to Approximate Matching
   - General Idea
   - Handling Arithmetic Progressions
   - The Stubborn Edge Case

Solution Sketch

We represent a regular expression as an automaton with ε-transition between atomic strings.

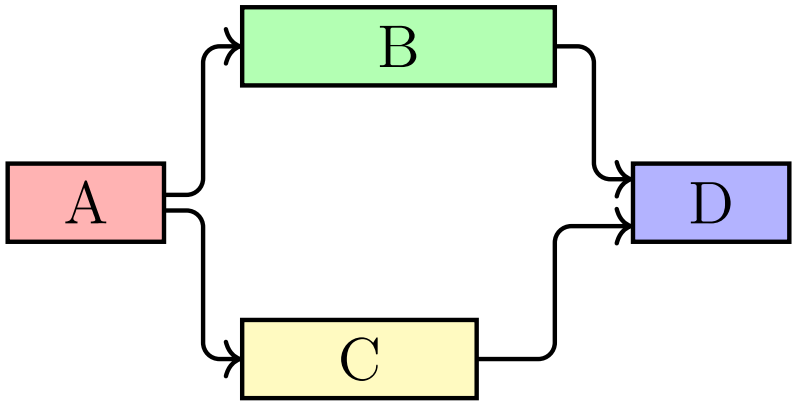
Solution Sketch

We represent a regular expression as an automaton with ε-transition between atomic strings (ex. A(B|C)D).



Solution Sketch

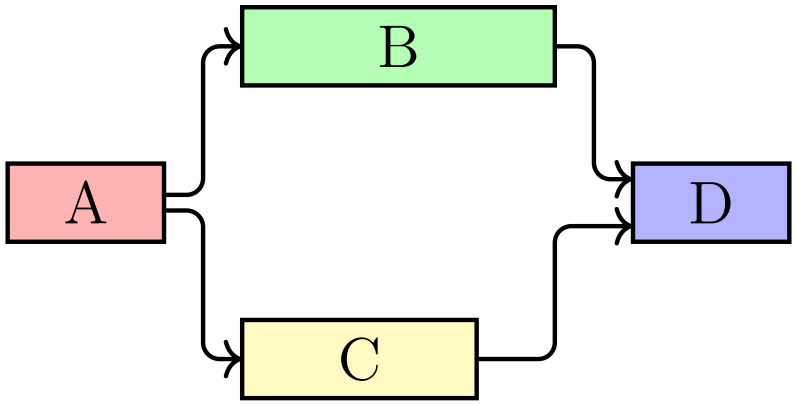
We represent a regular expression as an automaton with ε-transition between atomic strings (ex. A(B|C)D).



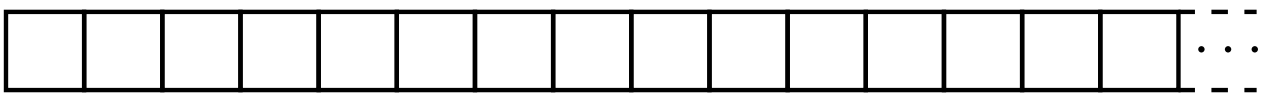
We search for occurrences of each of these atomic strings independently.

Solution Sketch

We represent a regular expression as an automaton with ε-transition between atomic strings (ex. A(B|C)D).

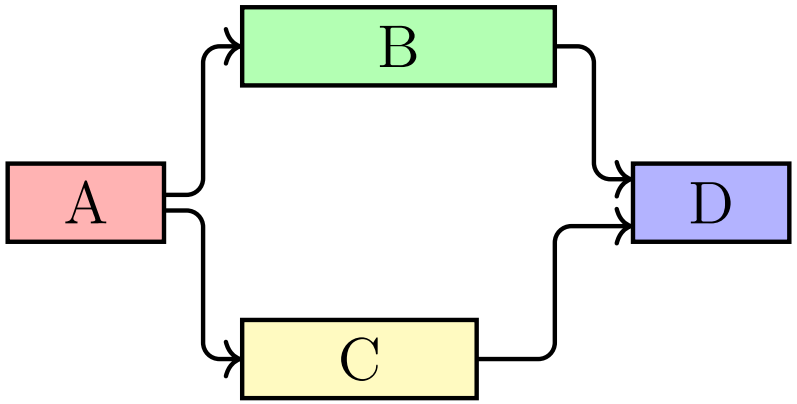


We search for occurrences of each of these atomic strings independently.

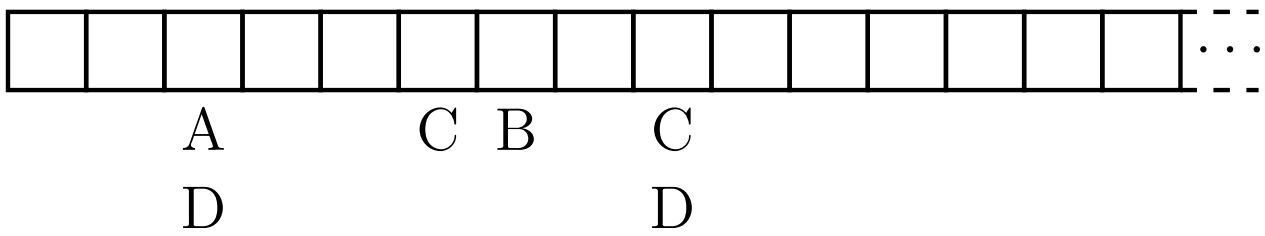


Solution Sketch

We represent a regular expression as an automaton with ε-transition between atomic strings (ex. A(B|C)D).

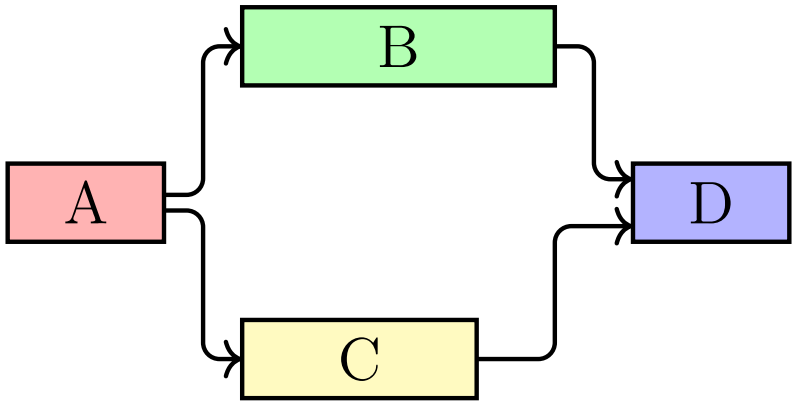


We search for occurrences of each of these atomic strings independently.

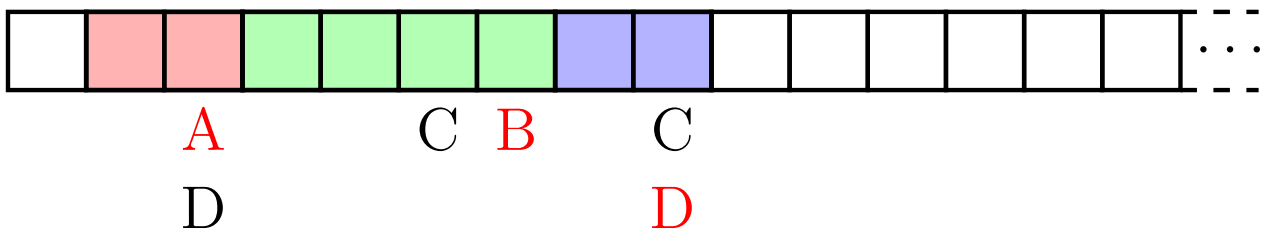


Solution Sketch

We represent a regular expression as an automaton with ε-transition between atomic strings (ex. A(B|C)D).

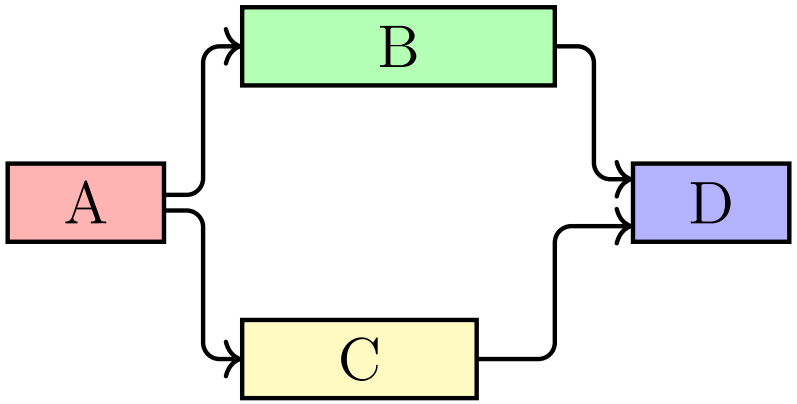


We search for occurrences of each of these atomic strings independently.



Solution Sketch

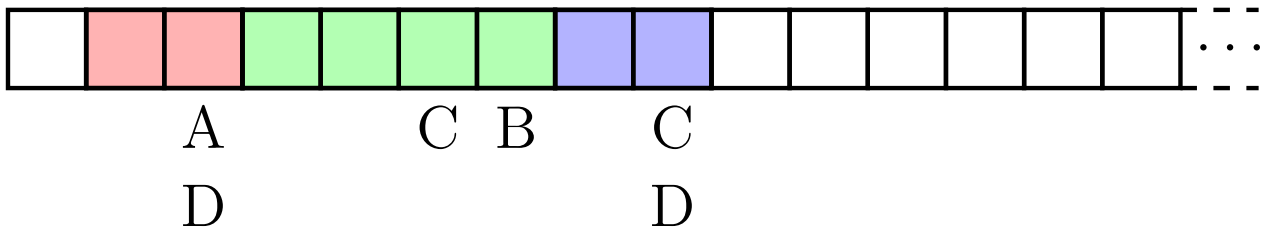
We represent a regular expression as an automaton with ε-transition between atomic strings (ex. A(B|C)D).



We search for occurrences of each of these atomic strings independently.

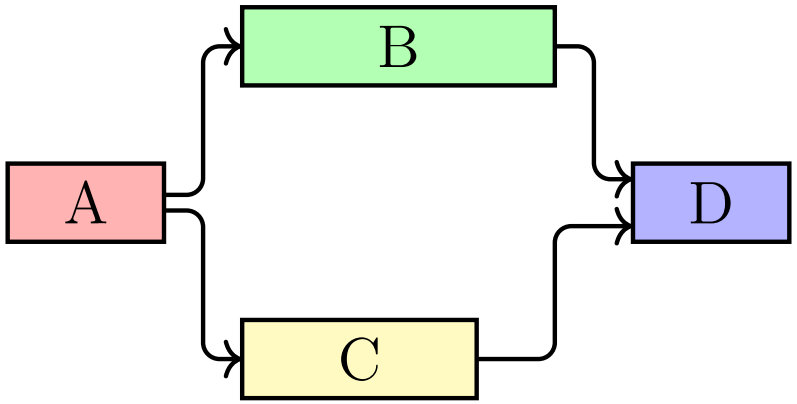
Definition

Occurrences that end partial matches are called witnesses.



Solution Sketch

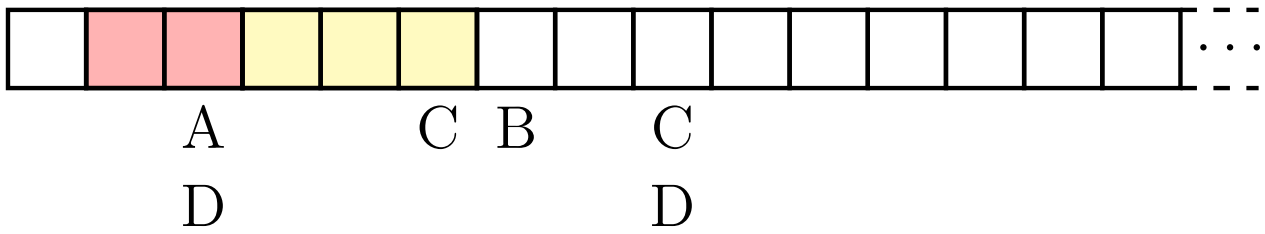
We represent a regular expression as an automaton with ε-transition between atomic strings (ex. A(B|C)D).



We search for occurrences of each of these atomic strings independently.

Definition

Occurrences that end partial matches are called witnesses.

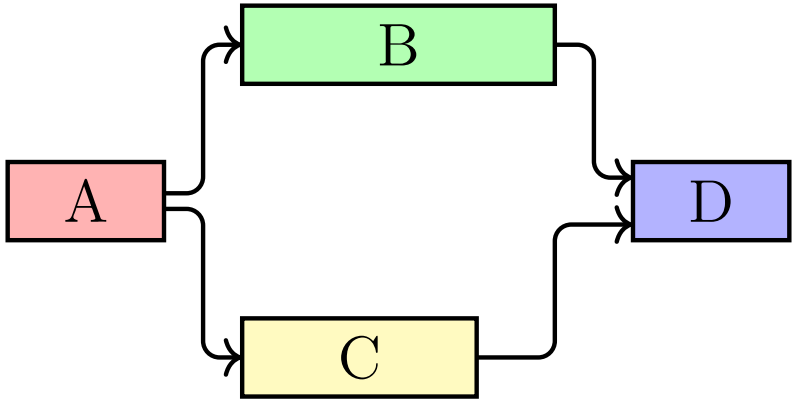


Witnesses

Remembering the witnesses is enough to find partial matches.

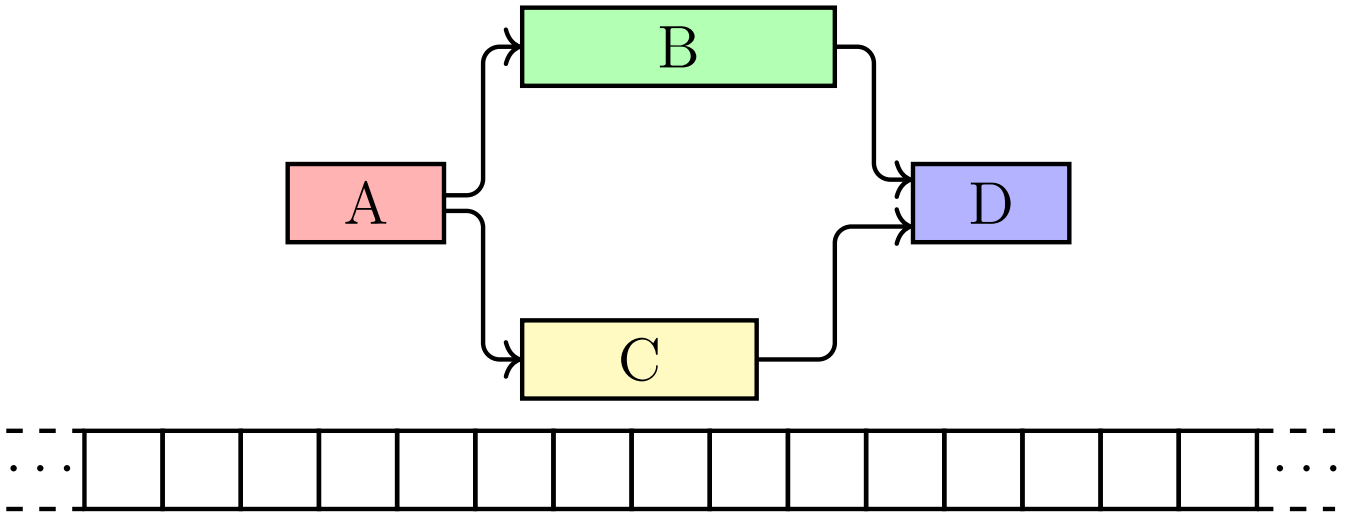
Witnesses

Remembering the witnesses is enough to find partial matches.



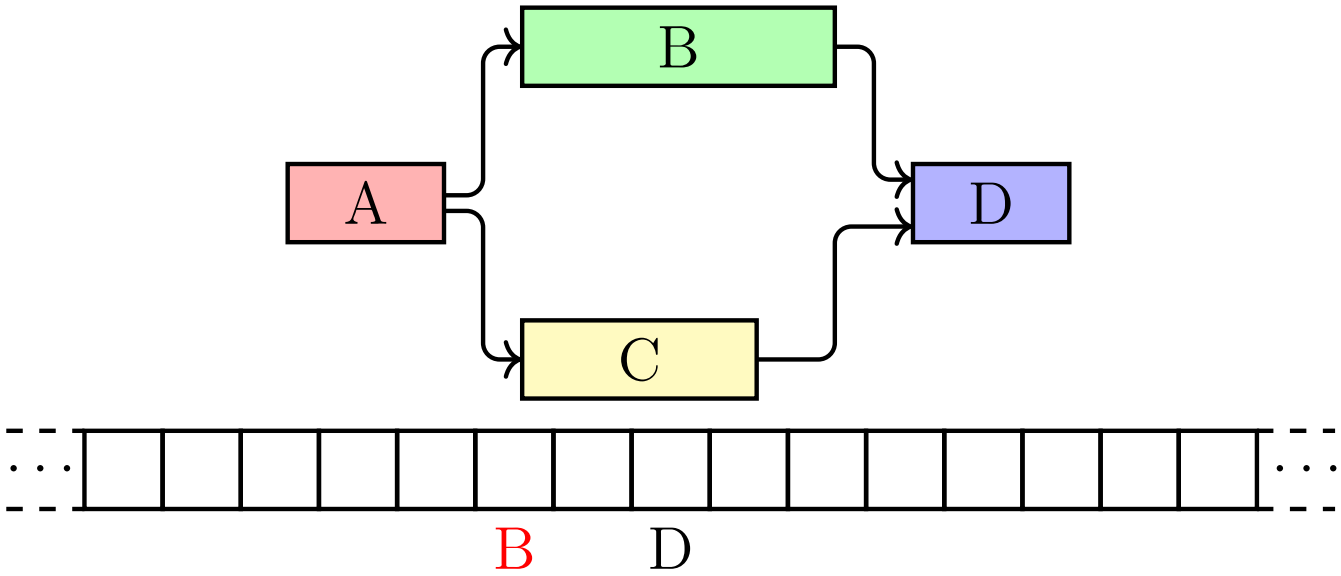
Witnesses

Remembering the witnesses is enough to find partial matches.



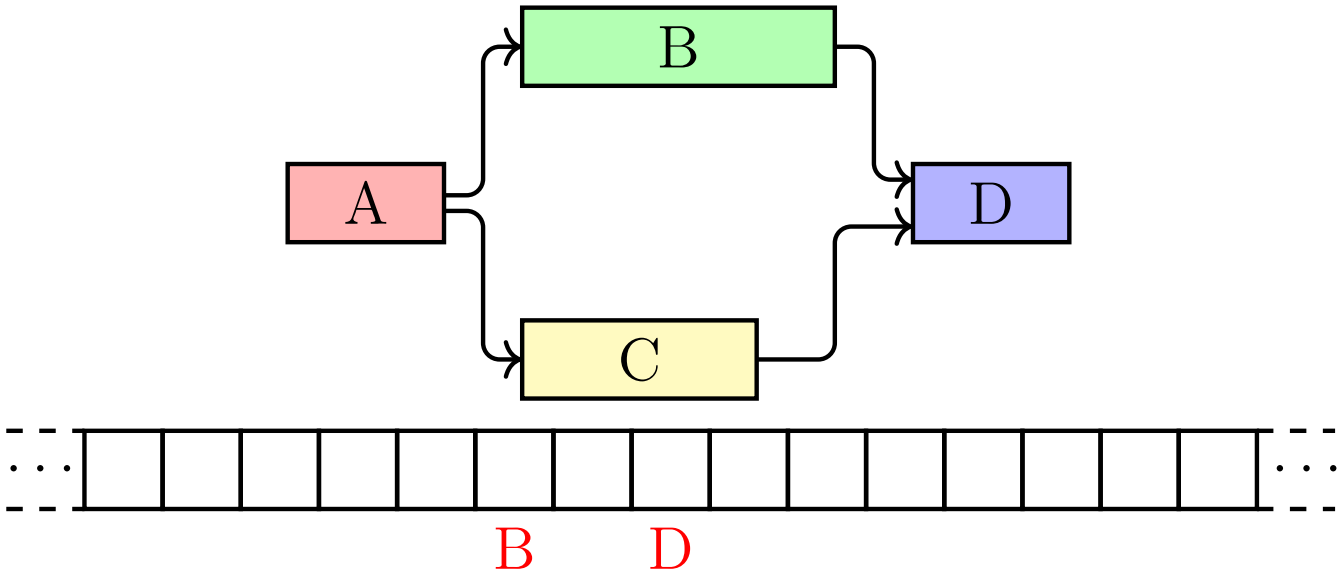
Witnesses

Remembering the witnesses is enough to find partial matches.



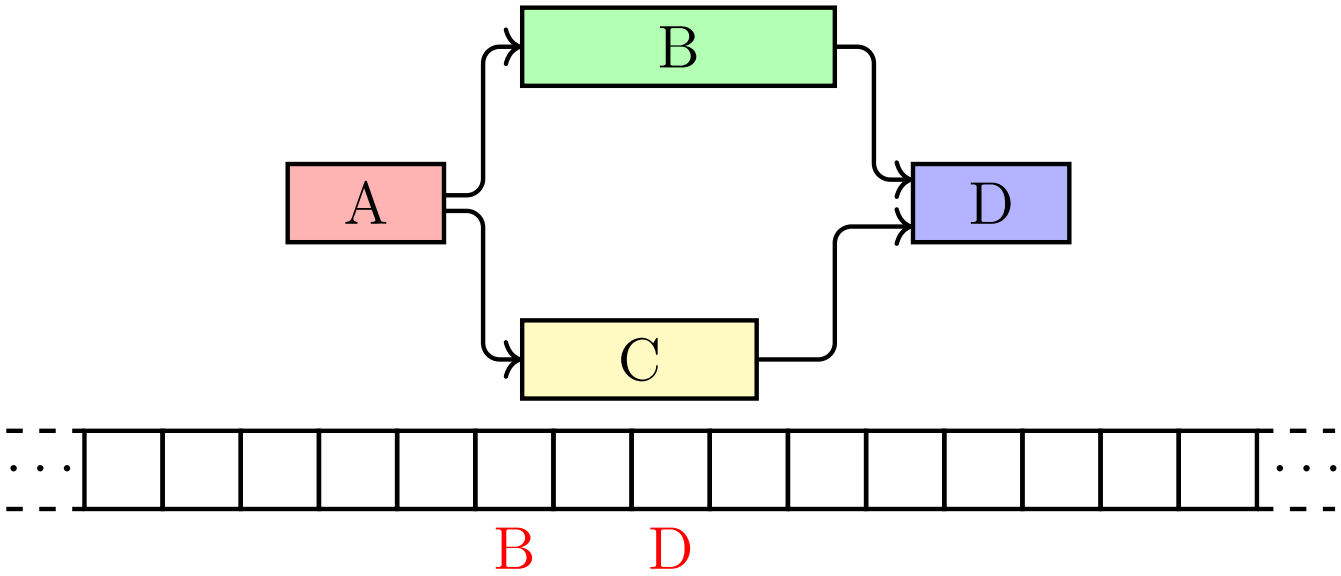
Witnesses

Remembering the witnesses is enough to find partial matches.



Witnesses

Remembering the witnesses is enough to find partial matches.



Strategy

- remember all witnesses
- for each character read, find all atomic strings for which it's an occurrence
- for each such occurrences, decide whether they are witnesses

Space-Time Tradeoff

The previous strategy is too space-expensive. We are going to trade space efficiency for running time.

Space-Time Tradeoff

The previous strategy is too space-expensive. We are going to trade space efficiency for running time.
Three kinds of occurrences:

Space-Time Tradeoff

The previous strategy is too space-expensive. We are going to trade space efficiency for running time.
Three kinds of occurrences:

- occurrences that happened a long time ago

Space-Time Tradeoff

The previous strategy is too space-expensive. We are going to trade space efficiency for running time.
Three kinds of occurrences:

- occurrences that happened a long time ago, can be forgotten;

Space-Time Tradeoff

The previous strategy is too space-expensive. We are going to trade space efficiency for running time.
Three kinds of occurrences:

- occurrences that happened a long time ago, can be forgotten;
- occurrences that cannot occur too often

Space-Time Tradeoff

The previous strategy is too space-expensive. We are going to trade space efficiency for running time.
Three kinds of occurrences:

- occurrences that happened a long time ago, can be forgotten;
- occurrences that cannot occur too often, we can explicitly remember whether they are witnesses;

Space-Time Tradeoff

The previous strategy is too space-expensive. We are going to trade space efficiency for running time.
Three kinds of occurrences:

- occurrences that happened a long time ago, can be forgotten;
- occurrences that cannot occur too often, we can explicitly remember whether they are witnesses;
- occurrences that could happen often

Space-Time Tradeoff

The previous strategy is too space-expensive. We are going to trade space efficiency for running time.
Three kinds of occurrences:

- occurrences that happened a long time ago, can be forgotten;
- occurrences that cannot occur too often, we can explicitly remember whether they are witnesses;
- occurrences that could happen often, we remember all of them, but not whether they are witnesses.

Space-Time Tradeoff

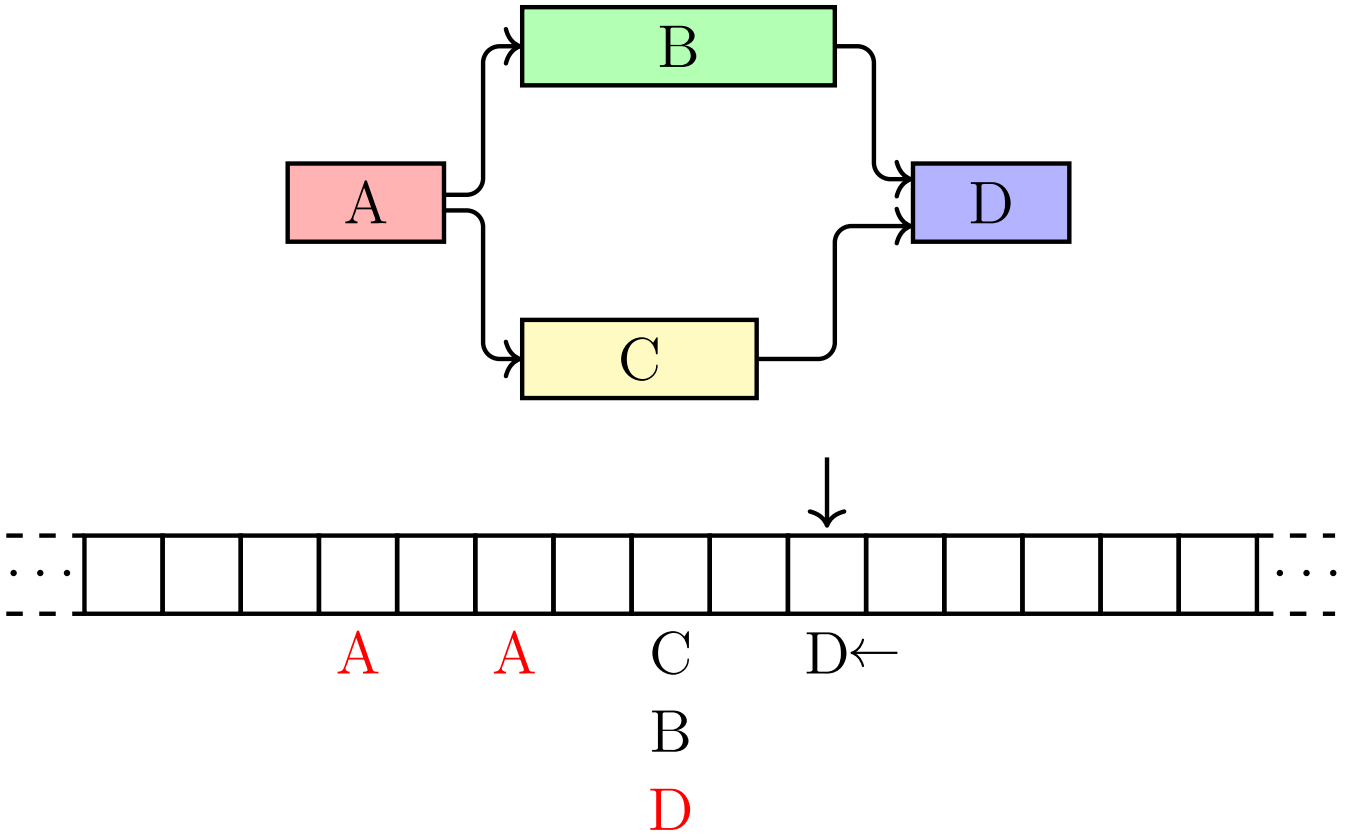
The previous strategy is too space-expensive. We are going to trade space efficiency for running time.
Three kinds of occurrences:

- occurrences that happened a long time ago, can be forgotten;
- occurrences that cannot occur too often, we can explicitly remember whether they are witnesses;
- occurrences that could happen often, we remember all of them, but not whether they are witnesses.

This is still enough information to decide whether a given occurrence is a witness.

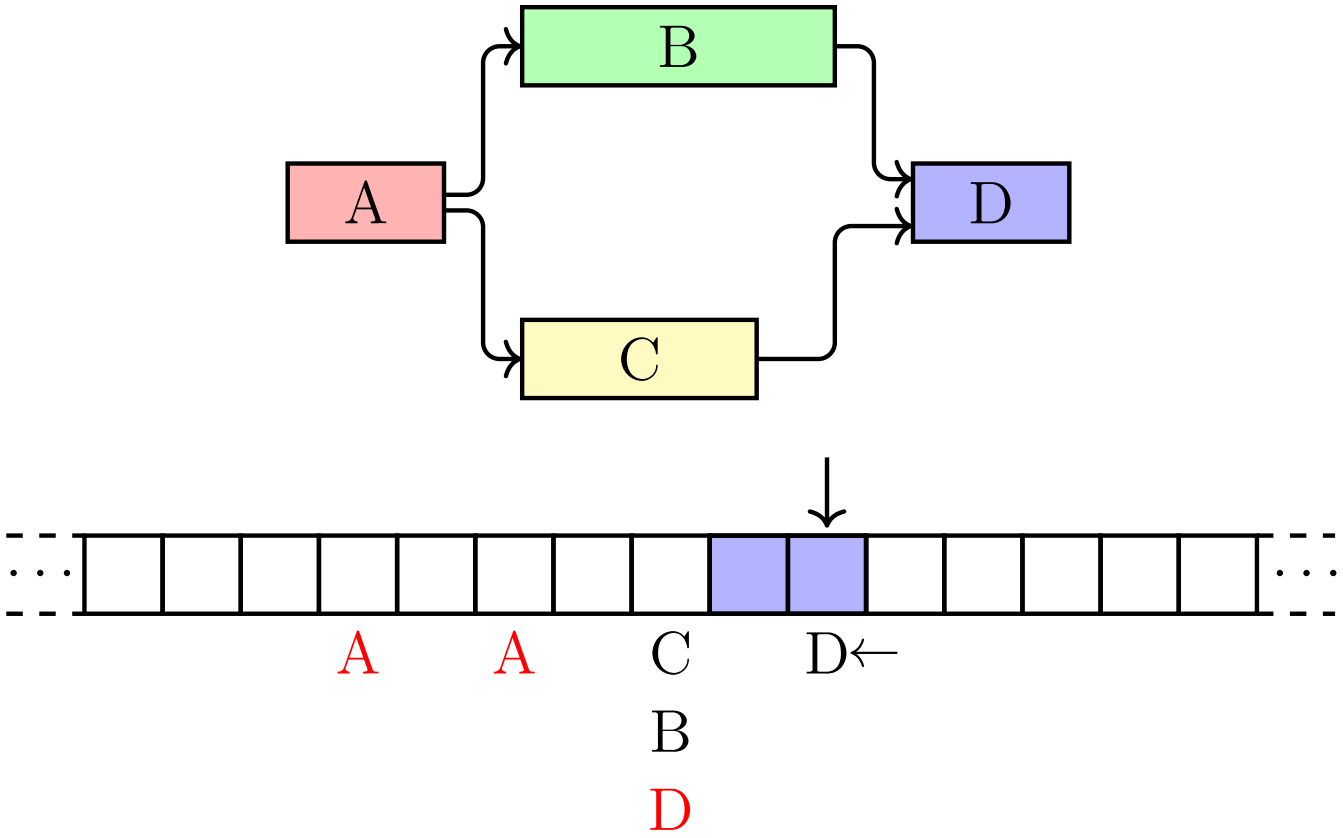
Backtracking Solution

To find whether a given occurrence is a witness, we consider occurrences of preceding string in the automaton.



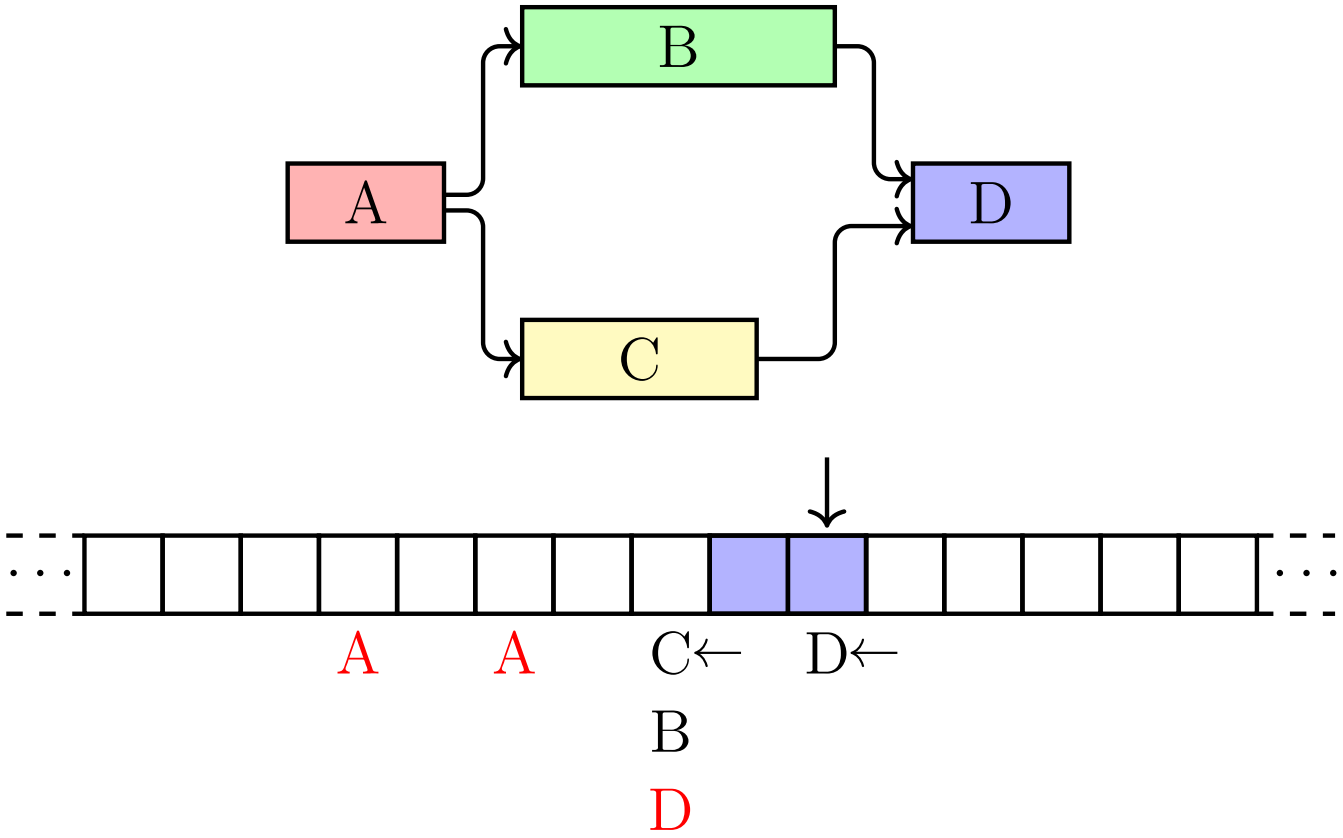
Backtracking Solution

To find whether a given occurrence is a witness, we consider occurrences of preceding string in the automaton.



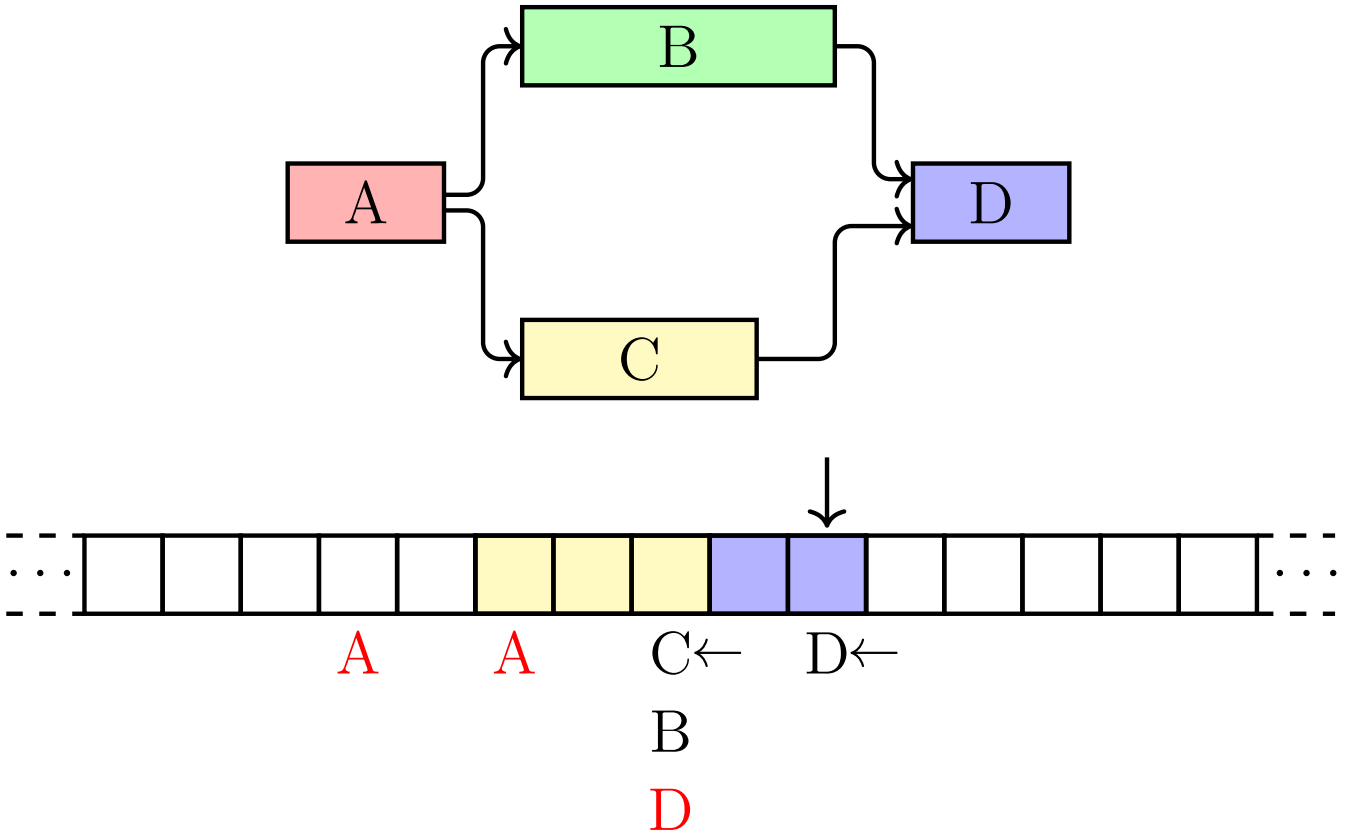
Backtracking Solution

To find whether a given occurrence is a witness, we consider occurrences of preceding string in the automaton.



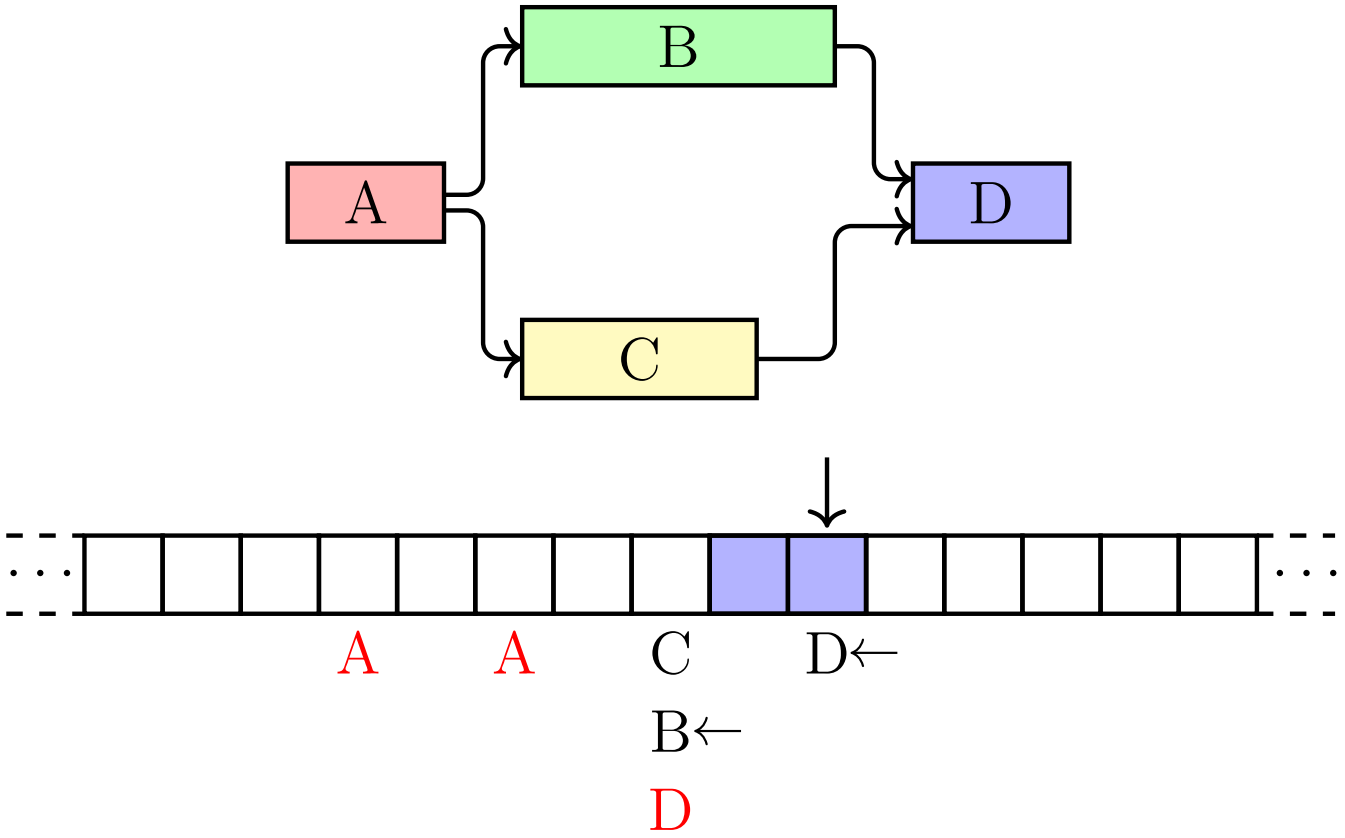
Backtracking Solution

To find whether a given occurrence is a witness, we consider occurrences of preceding string in the automaton.



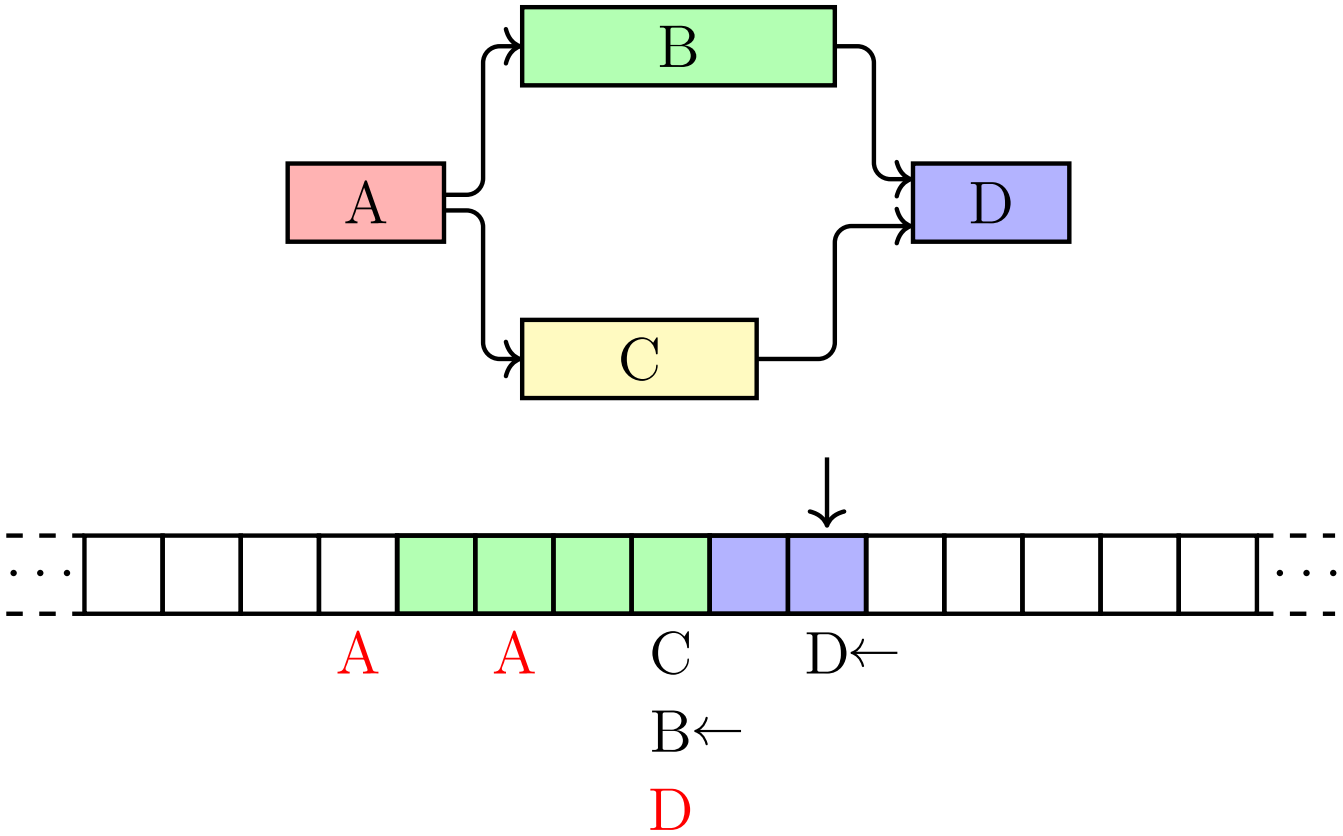
Backtracking Solution

To find whether a given occurrence is a witness, we consider occurrences of preceding string in the automaton.



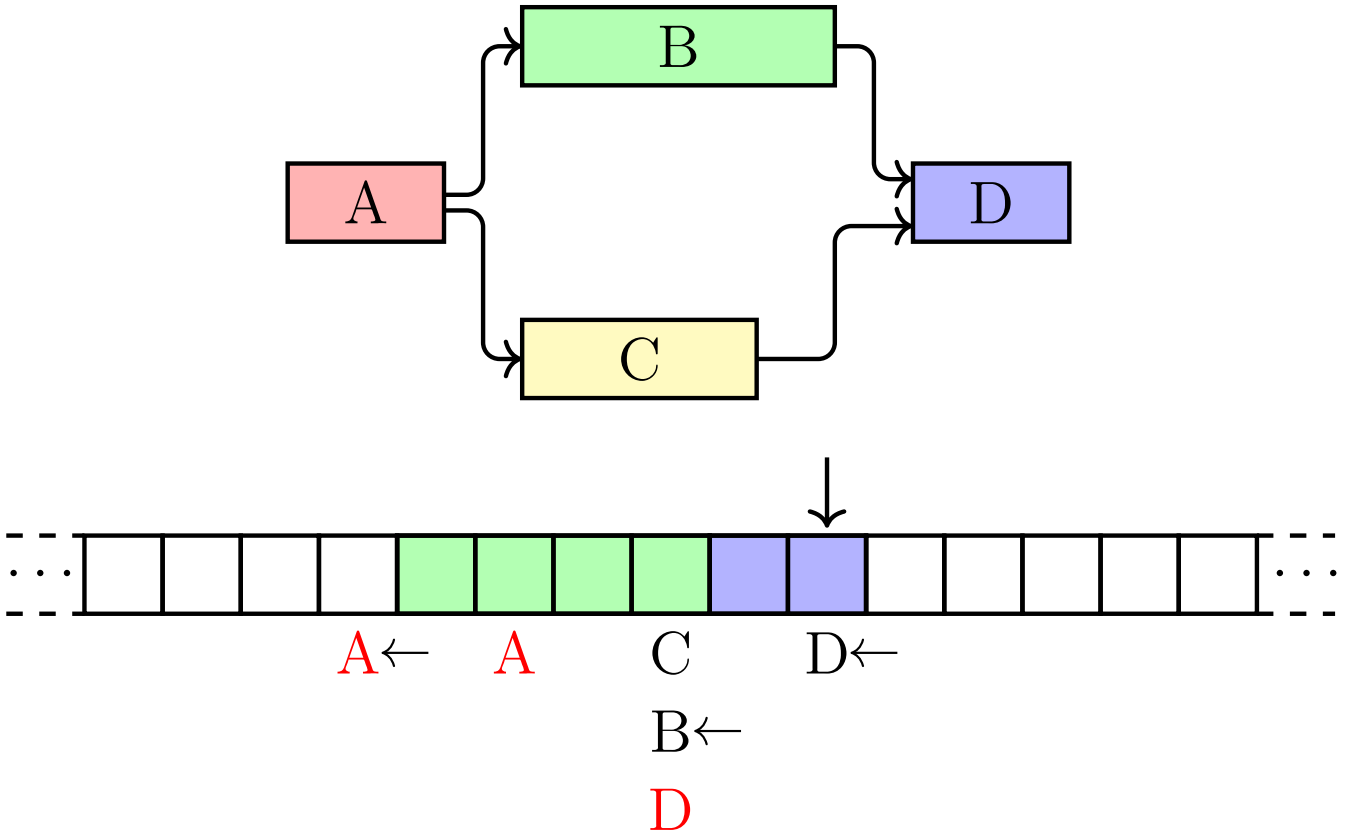
Backtracking Solution

To find whether a given occurrence is a witness, we consider occurrences of preceding string in the automaton.



Backtracking Solution

To find whether a given occurrence is a witness, we consider occurrences of preceding string in the automaton.



Occurrence Structure

To save space, we store more occurrences. Why?

Occurrence Structure

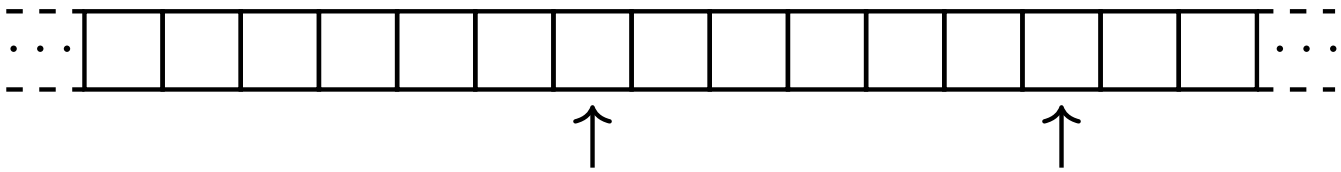
To save space, we store more occurrences. Why?
Sets of occurrences have a more rigid structure, which can be used to compress them.

Occurrence Structure

To save space, we store more occurrences. Why?
Sets of occurrences have a more rigid structure, which can be used to compress them.

Example

Consider the string abababab.

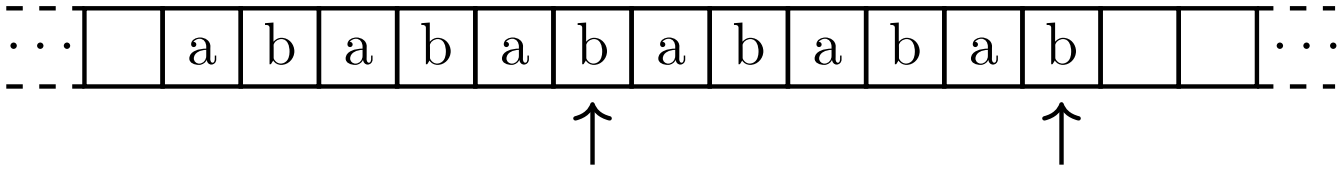


Occurrence Structure

To save space, we store more occurrences. Why?
Sets of occurrences have a more rigid structure, which can be used to compress them.

Example

Consider the string abababab.

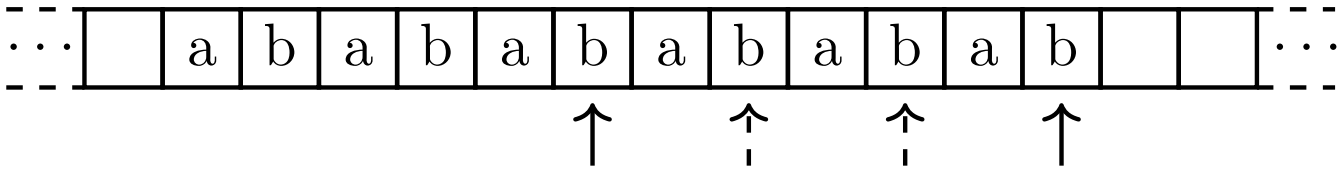


Occurrence Structure

To save space, we store more occurrences. Why?
Sets of occurrences have a more rigid structure, which can be used to compress them.

Example

Consider the string abababab.

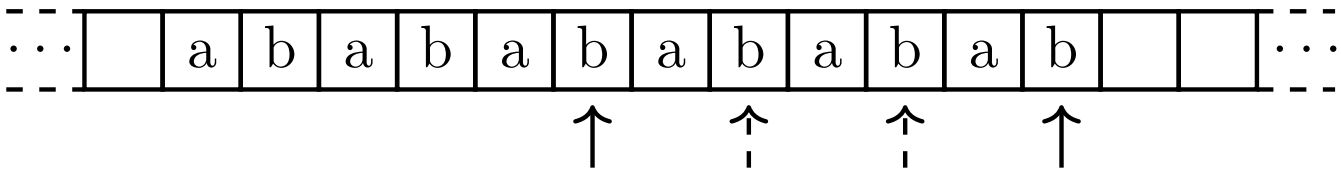


Occurrence Structure

To save space, we store more occurrences. Why?
Sets of occurrences have a more rigid structure, which can be used to compress them.

Example

Consider the string abababab.



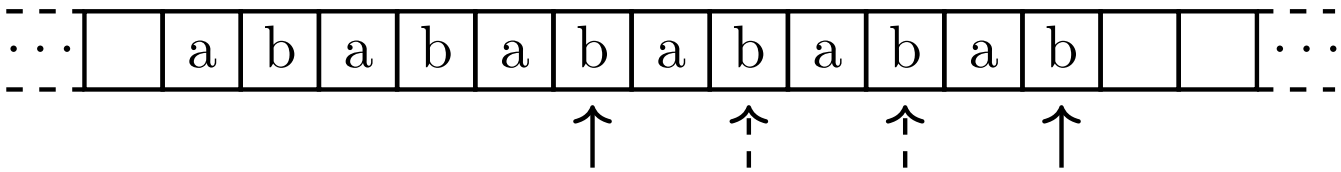
We could not have inferred these occurrences if we were talking about witnesses.

Occurrence Structure

To save space, we store more occurrences. Why?
Sets of occurrences have a more rigid structure, which can be used to compress them.

Example

Consider the string abababab.



We could not have inferred these occurrences if we were talking about witnesses.

In general, close occurrences of periodic strings happen in arithmetic progression, which can be stored efficiently.

Precomputing Backtracking Queries

Quick recap

- for the occurrences that are isolated, we store explicitly whether they're witnesses
- for the occurrences that happen in arithmetic progressions, we store the arithmetic progression and few witnesses at the beginning of the stream

Precomputing Backtracking Queries

Quick recap

- for the occurrences that are isolated, we store explicitly whether they're witnesses
- for the occurrences that happen in arithmetic progressions, we store the arithmetic progression and few witnesses at the beginning of the stream

But backtracking on long arithmetic progressions is (spacewise) expensive.

Precomputing Backtracking Queries

Quick recap

- for the occurrences that are isolated, we store explicitly whether they're witnesses
- for the occurrences that happen in arithmetic progressions, we store the arithmetic progression and few witnesses at the beginning of the stream

But backtracking on long arithmetic progressions is (spacewise) expensive.

Solution

The content of a arithmetic progression is known beforehand, so queries can be precomputed!

Precomputing Backtracking Queries

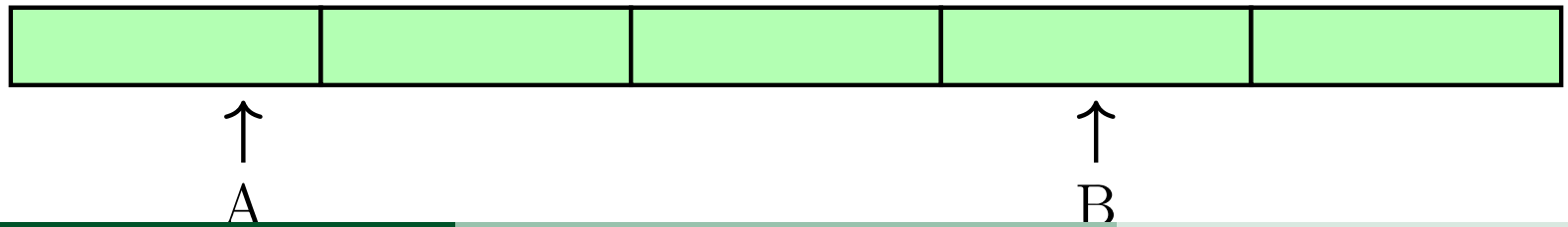
Quick recap

- for the occurrences that are isolated, we store explicitly whether they're witnesses
- for the occurrences that happen in arithmetic progressions, we store the arithmetic progression and few witnesses at the beginning of the stream

But backtracking on long arithmetic progressions is (spacewise) expensive.

Solution

The content of a arithmetic progression is known beforehand, so queries can be precomputed!



Reducing to a Graph Problem

There's still an infinite number of such queries, we can't store them all.

Reducing to a Graph Problem

There's still an infinite number of such queries, we can't store them all.
But we can precompute all queries of bounded distance.

Reducing to a Graph Problem

There's still an infinite number of such queries, we can't store them all.
But we can precompute all queries of bounded distance.
We can encode this preprocessed information as a graph:

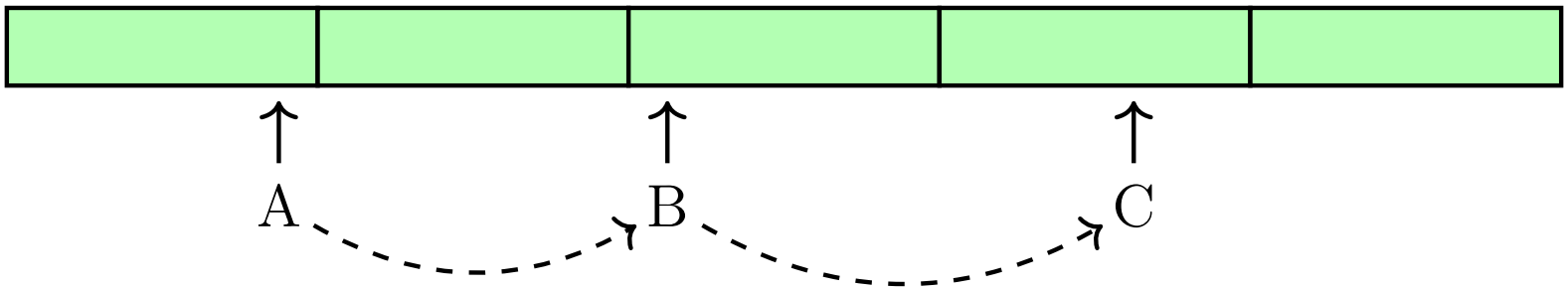
- each node represents an atomic string with an offset;
- each edge of weight w means that there is a walk in the automaton labeled with the letters from the arithmetic progression, of length w.

Reducing to a Graph Problem

There's still an infinite number of such queries, we can't store them all.
But we can precompute all queries of bounded distance.
We can encode this preprocessed information as a graph:

- each node represents an atomic string with an offset;
- each edge of weight w means that there is a walk in the automaton labeled with the letters from the arithmetic progression, of length w.

Each query can be translated in knowning whether there is a path of a given length between two nodes in this graph.

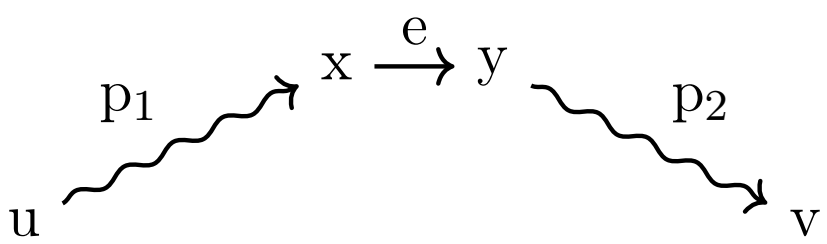


Reducing to a Graph Problem

We solve this problem by computing, for a given k, for all pairs of nodes (u, v), for all weights $w \leq 2^k$, whether there is a path from u to v of weight w.

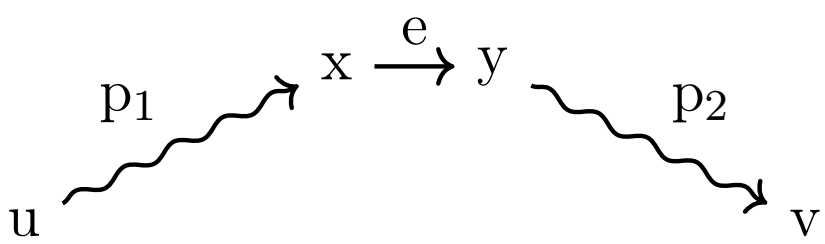
Reducing to a Graph Problem

We solve this problem by computing, for a given k, for all pairs of nodes (u, v), for all weights $w \leq 2^k$, whether there is a path from u to v of weight w.



Reducing to a Graph Problem

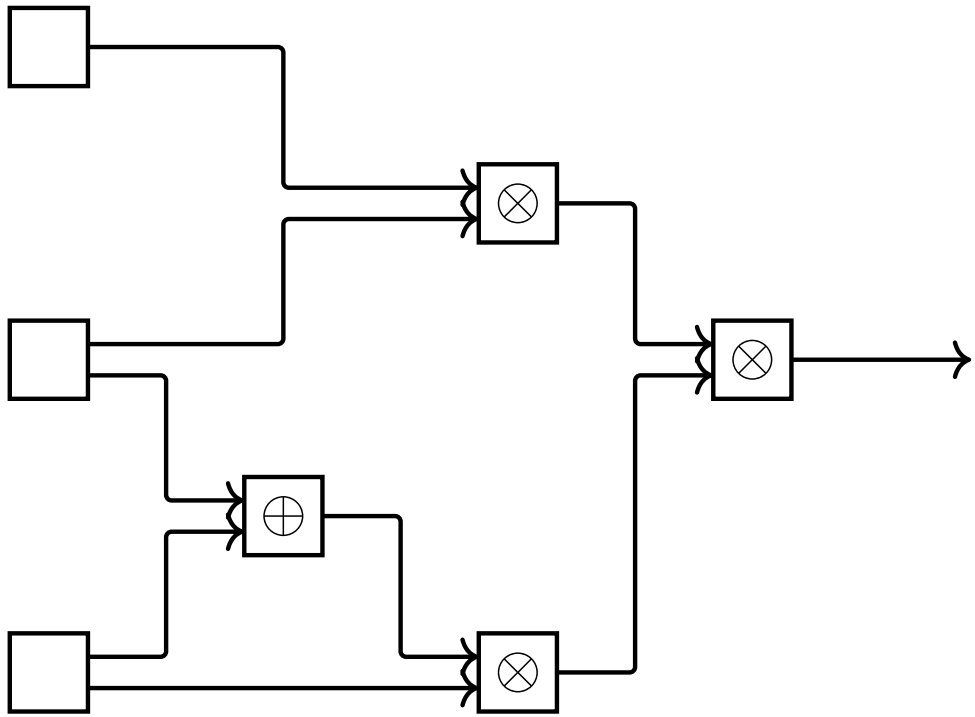
We solve this problem by computing, for a given k, for all pairs of nodes (u, v), for all weights $w \leq 2^k$, whether there is a path from u to v of weight w.



This amounts to iterated convolutions of solutions of the problem for a given k, to get the solutions for $k + 1$.

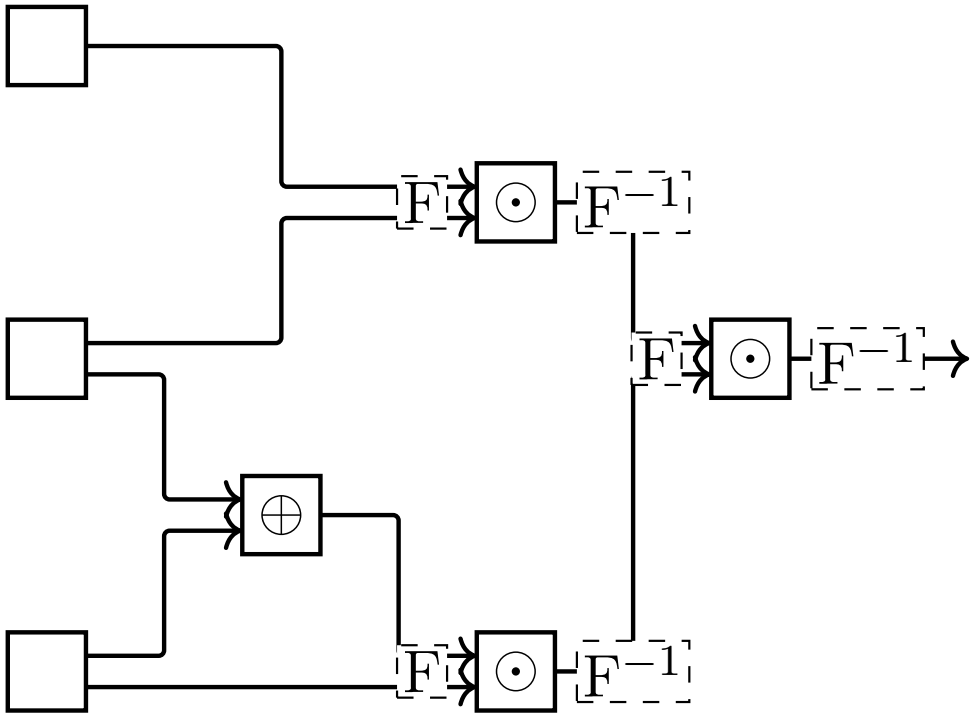
Reducing to a Circuit Problem

We translate the previous problem into a circuit with convolution and addition gates.



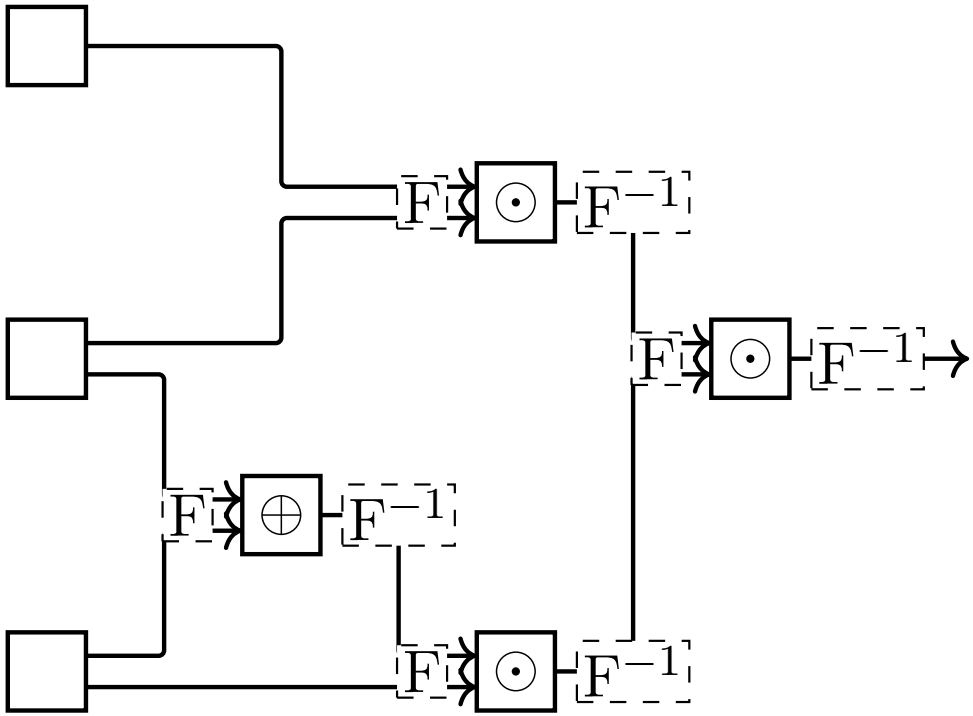
Reducing to a Circuit Problem

We translate the previous problem into a circuit with convolution and addition gates.



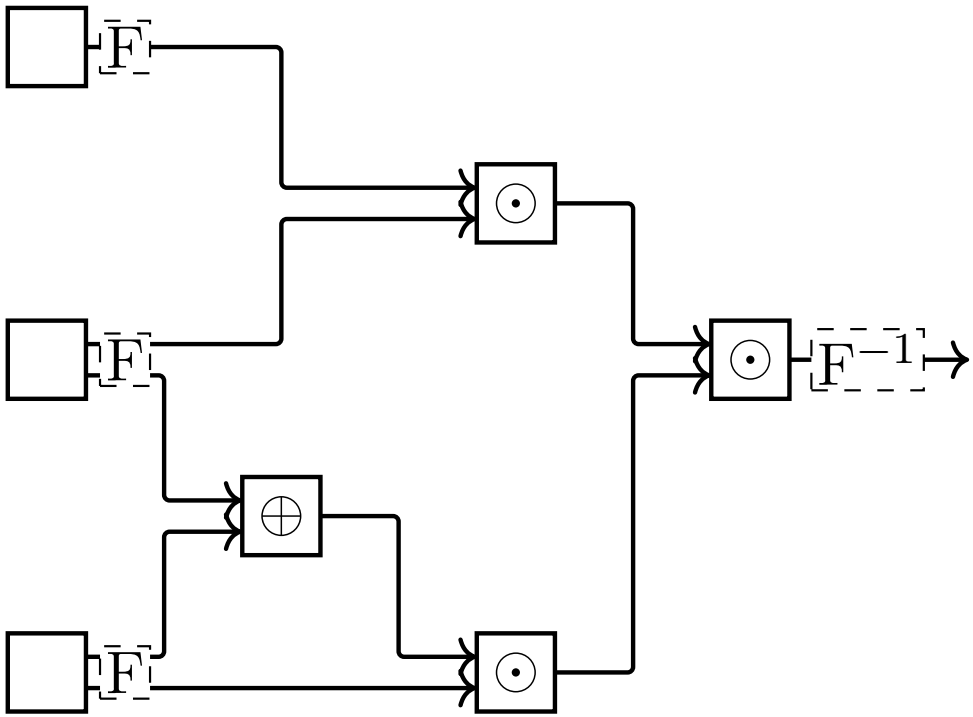
Reducing to a Circuit Problem

We translate the previous problem into a circuit with convolution and addition gates.



Reducing to a Circuit Problem

We translate the previous problem into a circuit with convolution and addition gates.



Where are we going?

1. Exact Regular Expression Matching
   - Solution Sketch
   - Witnesses
   - Space-Time Tradeoff
   - Backtracking Solution
   - Occurrence Structure
   - Precomputing Backtracking Queries
   - Reducing to a Graph Problem
   - Reducing to a Circuit Problem
2. Generalizing to Approximate Matching
   - General Idea
   - Handling Arithmetic Progressions
   - The Stubborn Edge Case

General Idea

To make the algorithm support approximate matching, we apply the following changes:

General Idea

To make the algorithm support approximate matching, we apply the following changes:

- we replace in the exact algorithm the exact string pattern matching routine with an approximate one;

General Idea

To make the algorithm support approximate matching, we apply the following changes:

- we replace in the exact algorithm the exact string pattern matching routine with an approximate one;
- all the witnesses also store the number of mismatches;

General Idea

To make the algorithm support approximate matching, we apply the following changes:

- we replace in the exact algorithm the exact string pattern matching routine with an approximate one;
- all the witnesses also store the number of mismatches;
- the graph-problem solving subroutine can also constrain the number of mismatches along the path;

General Idea

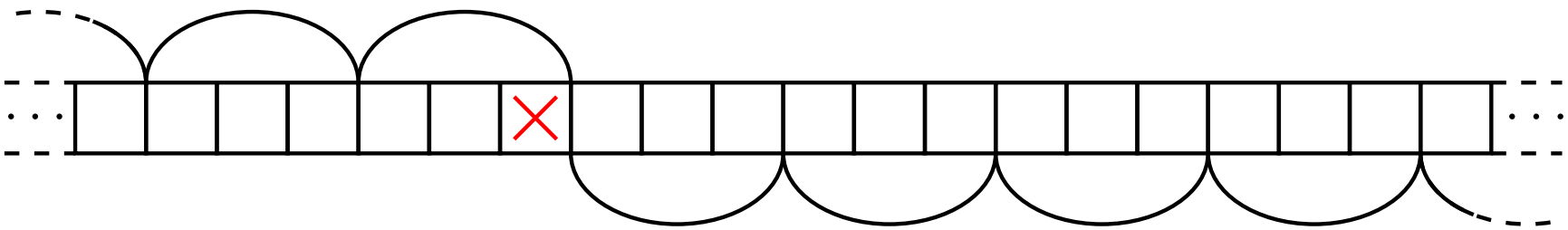
To make the algorithm support approximate matching, we apply the following changes:

- we replace in the exact algorithm the exact string pattern matching routine with an approximate one;
- all the witnesses also store the number of mismatches;
- the graph-problem solving subroutine can also constrain the number of mismatches along the path;

Most of this works for any number of mismatches but, due to a single edge case, we only consider the case of one mismatch.

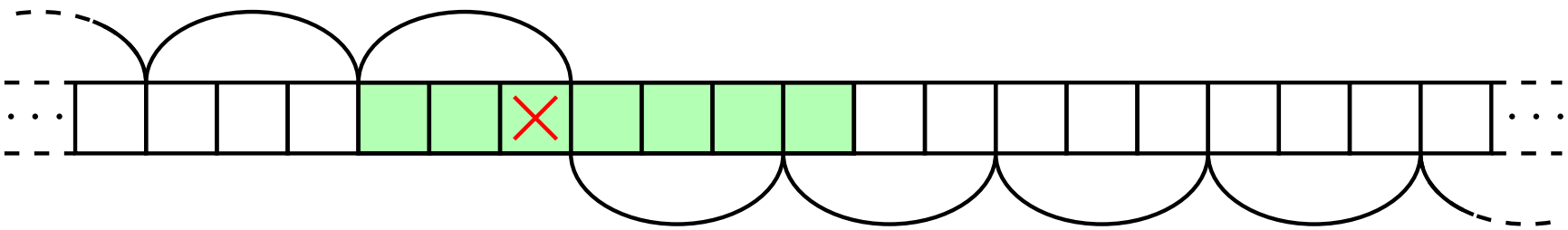
Handling Arithmetic Progressions

In the exact setting, the invariant is that all occurrences of an atomic string happen in an arithmetic progression streak. This is not true anymore:



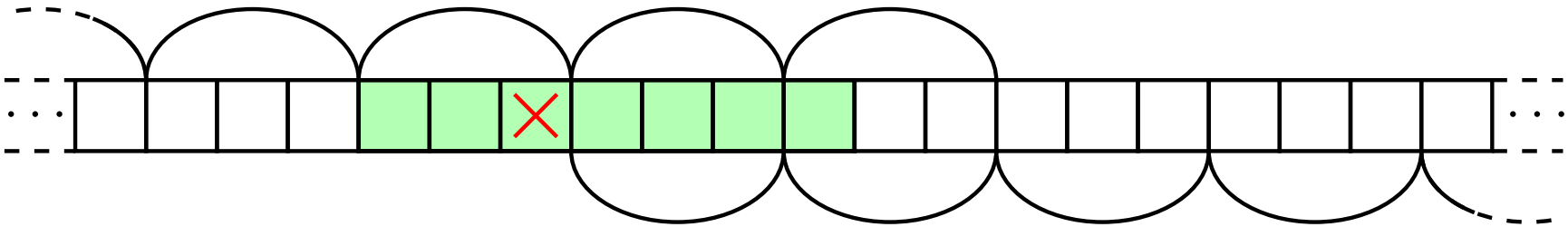
Handling Arithmetic Progressions

In the exact setting, the invariant is that all occurrences of an atomic string happen in an arithmetic progression streak. This is not true anymore:



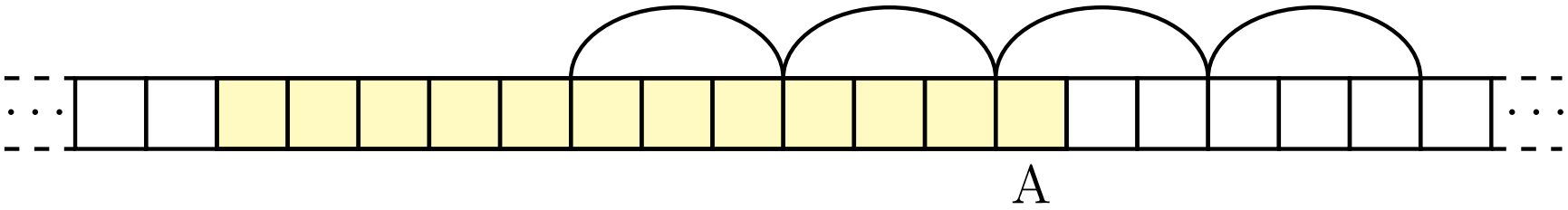
Handling Arithmetic Progressions

In the exact setting, the invariant is that all occurrences of an atomic string happen in an arithmetic progression streak. This is not true anymore:



The Stubborn Edge Case

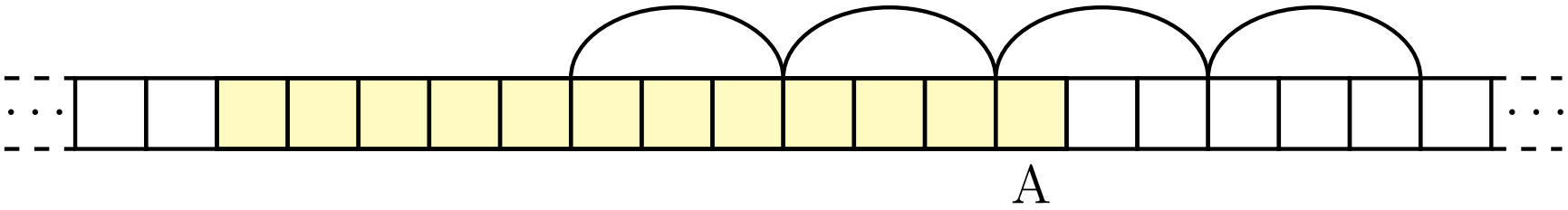
The annoying edge case is that of a long atomic string A that overlaps with an arithmetic progression.



In the exact setting, there can be at most one such overlap.

The Stubborn Edge Case

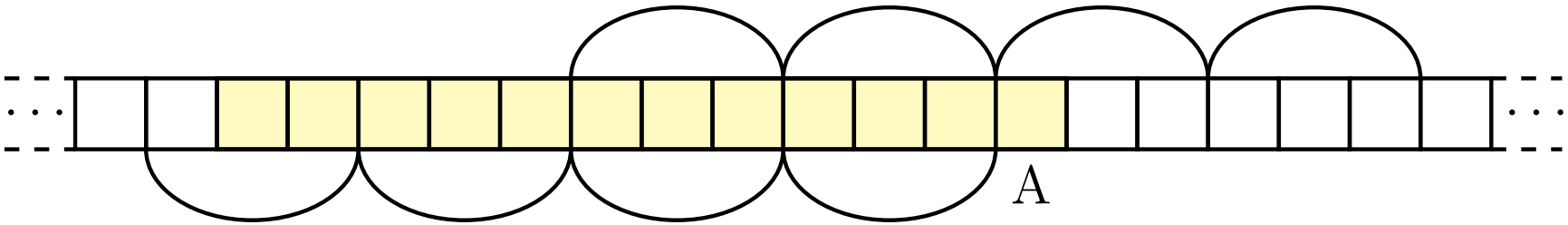
The annoying edge case is that of a long atomic string A that overlaps with an arithmetic progression.



In the exact setting, there can be at most one such overlap.
In the approximate setting, in some very particular situations, there can be a lot of such overlaps.

The Stubborn Edge Case

The annoying edge case is that of a long atomic string A that overlaps with an arithmetic progression.



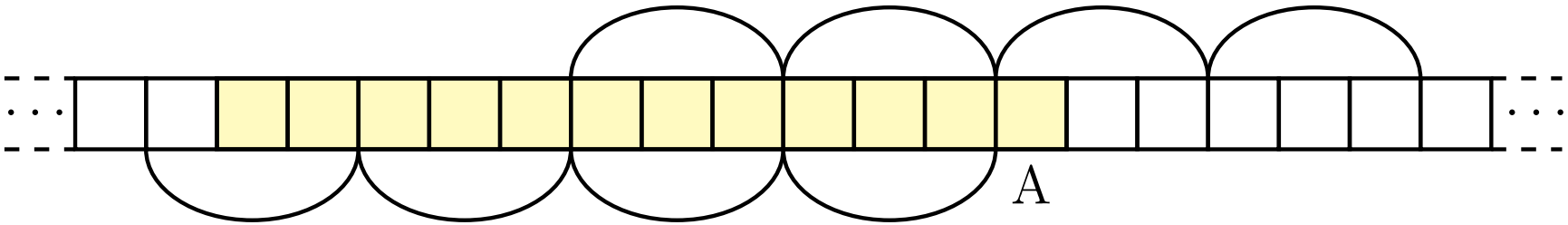
In the exact setting, there can be at most one such overlap.
In the approximate setting, in some very particular situations, there can be a lot of such overlaps.

Solution

- we store the arithmetic progression of occurrences of the first half of A that are close to the beginning of the overlap;

The Stubborn Edge Case

The annoying edge case is that of a long atomic string A that overlaps with an arithmetic progression.



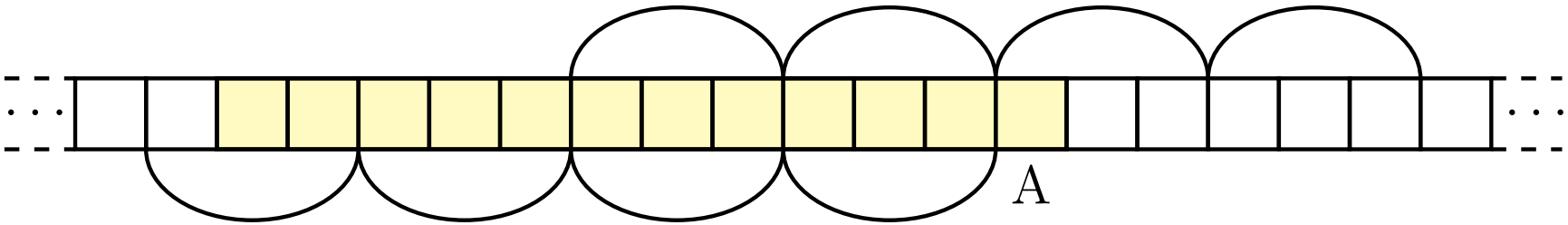
In the exact setting, there can be at most one such overlap.
In the approximate setting, in some very particular situations, there can be a lot of such overlaps.

Solution

- we store the arithmetic progression of occurrences of the first half of A that are close to the beginning of the overlap;
- we "guess" an occurrence of A;

The Stubborn Edge Case

The annoying edge case is that of a long atomic string A that overlaps with an arithmetic progression.



In the exact setting, there can be at most one such overlap.
In the approximate setting, in some very particular situations, there can be a lot of such overlaps.

Solution

- we store the arithmetic progression of occurrences of the first half of A that are close to the beginning of the overlap;
- we "guess" an occurrence of A;
- we check whether it's a witness.