Dynamic Programming at ease - with Grammars, Algebras, Products

Stefanie Schirmer

Sept 18, 2014

Combinatorial Optimization Problems

combinatorial "counting" or enumerating *all possible* solutions of a recursive problem

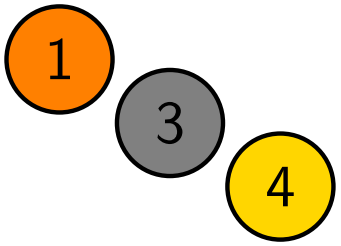
optimization finding *the desired* solution

- Boggle
- Money changing problem
- Text / sequence alignment
- RNA structure prediction with basepair maximisation

Classic Dynamic Programming

- Characterize structure of an optimal solution
- Recursively define value of an optimal solution
  - D[n] = D[n-1]+D[n-3]+D[n-4]
  - D[0] = 0
- Compute value of an optimal solution

n	0	1	2	3	4	5
D	0	1	1	2	4	6



$n = 5$

- Construct optimal solution from computed information (backtracking).

1+1+1+1+1, 1+1 +3, 1 +3 +1, 3 +1+1, 1 +4, 4 +1

DP solves optimization problems

- Over a large (exponential) search space
- in a reasonable (polynomial) time

BUT

The development of sucessful dynamic programming recurrences is "a matter of experience, talent and luck".

Overview

- Life, universe, and all the rest

- Candidates are trees
- Questions are algebras
- Programs are grammars

- Products are fun

Life, the universe and all the rest

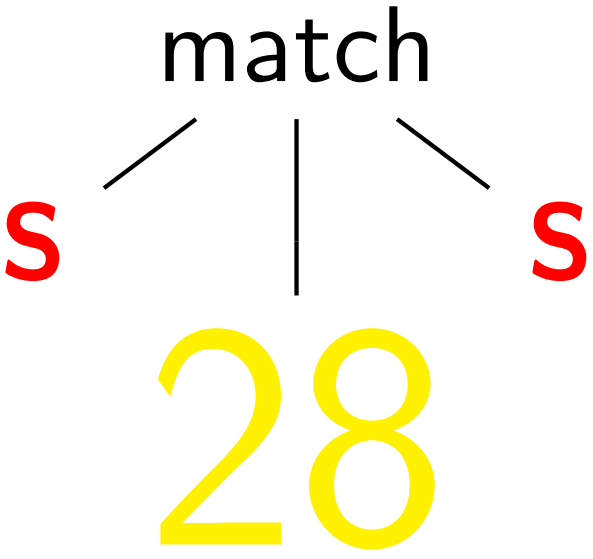
Reverse engineering of DP algorithms

If this is the answer..

42

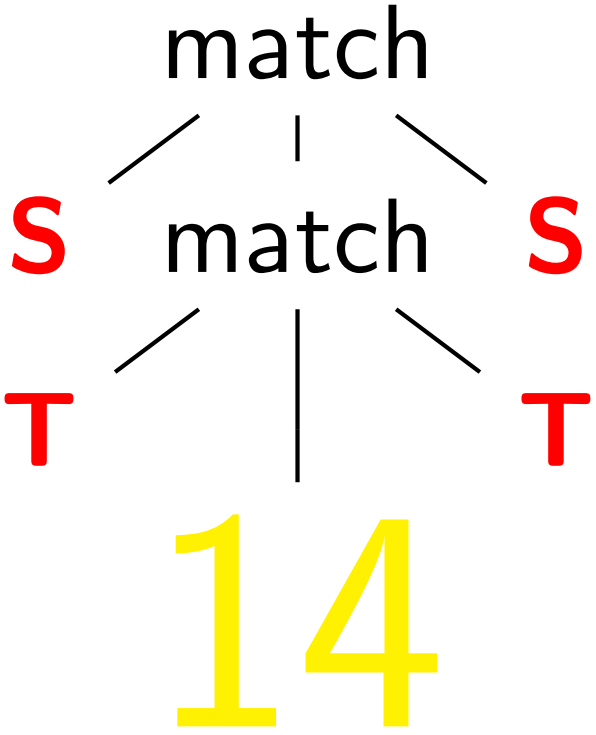
..what was the question?

Reverse engineering of DP algorithms



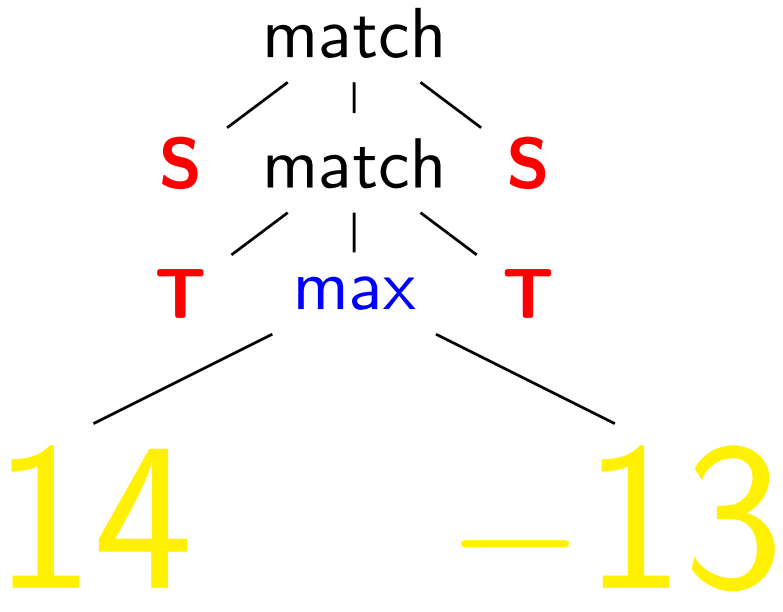
match(a,x,a) = x+14

Reverse engineering of DP algorithms



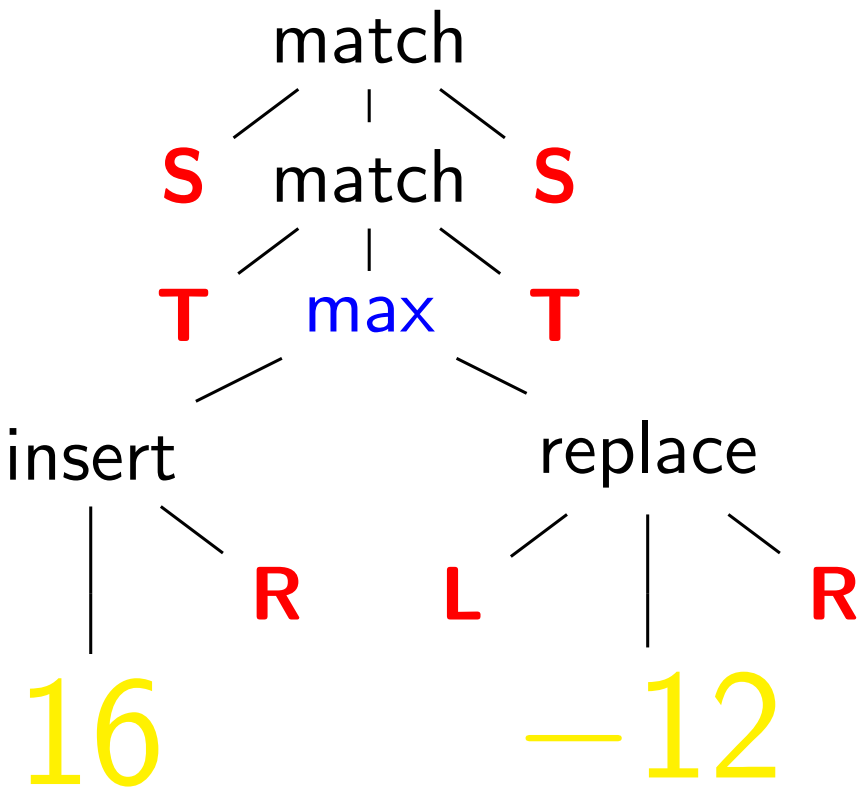
match(a,x,a) = x+14

Reverse engineering of DP algorithms



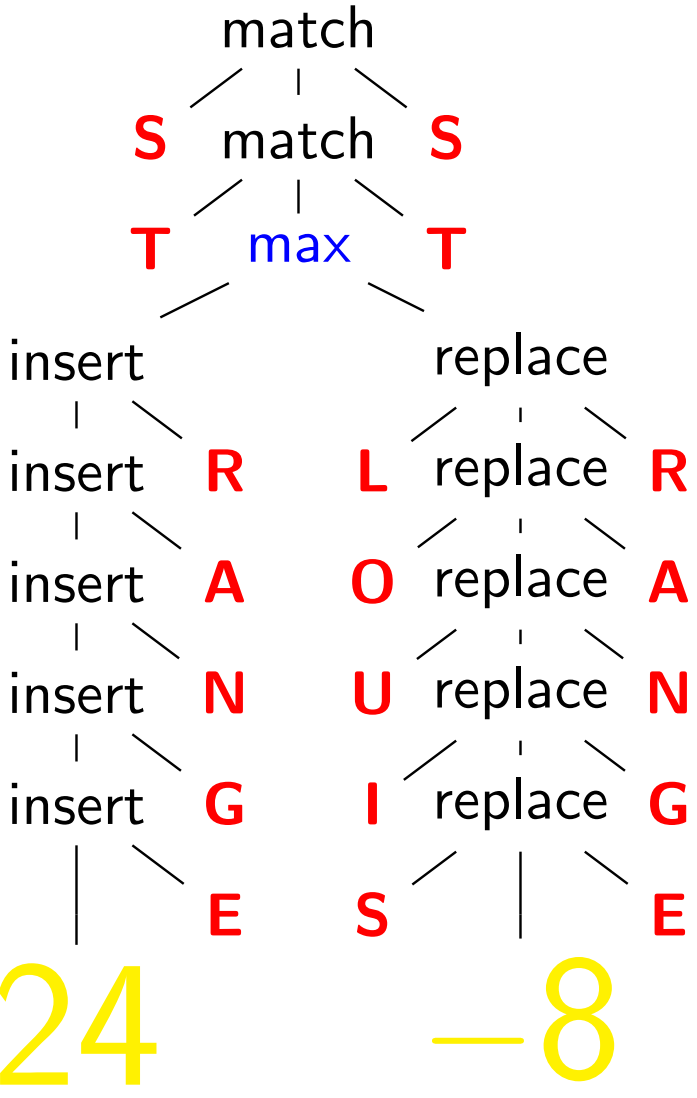
Choice of the maximum score!

Reverse engineering of DP algorithms



insert(x,a) = x-2 replace(a,x,b) = x-1

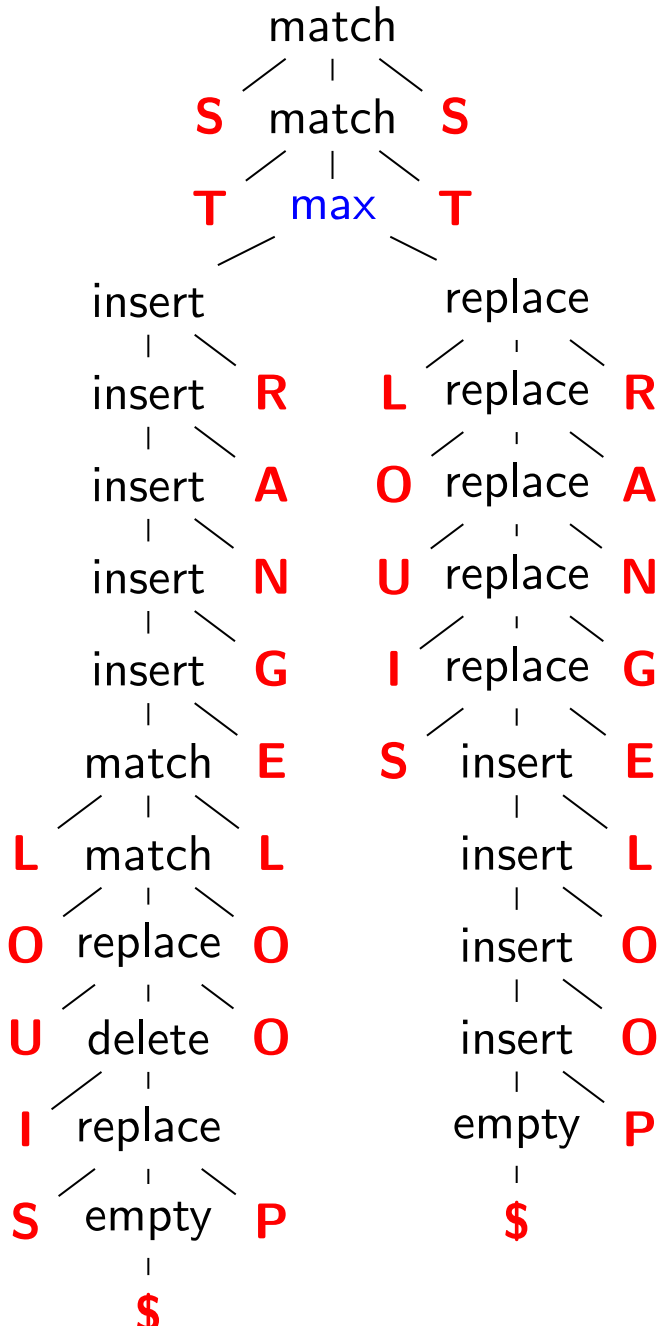
Reverse engineering of DP algorithms



insert(x,a) = x-2

replace(a,x,b) = x-1

Reverse engineering of DP algorithms



We found two alignments of "STLOUIS" and "STRANGELOOP"

```
ST-----LOUIS              STLOUIS----
MMIIIIIMMRRD   versus     MMRRRRRIIII
STRANGELOOP-              STRANGELOOP

score: 42                 score: 15
```

Each alignment & score is represented by the same formulas!

Reverse Engineering Summary

1. Result of a DP algorithm is the value of a formula build from *evaluation* functions, interleaved with applications of a *choice* function
2. All applications of the choice function move to the top
3. Formulas are candidate solutions
4. Input sequence(s) are part of each formula

Reverse engineering - reversed :D :D

Procedure to solve a DP problem:

4. Read the input sequence
3. Construct all candidate solutions (as formulas)
2. Move the applications of the choice function down/inside the formulas
1. Evaluate the formulas to get the desired result

Candidates are trees

Steps **read input**, **apply choice**, and **evaluate** are always the same and can be automated

Talents and experience go into **constructing candidates**:

- which candidates arise for a specific given input? (problem decomposition)
- what does a desired candidate look like? (scoring)

$\Rightarrow$ a language of formulas (trees)

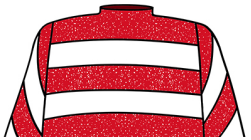
With this language, **constructing candidates** can also be automated!

We get everything for free except for the creative part!! <3

The Signature

```
signature Align(alphabet, answer) {
    answer replace(<alphabet, alphabet>, answer);
    answer delete(<alphabet, void>, answer);
    answer insert(<void, alphabet>, answer);
    answer empty(<void, void>);
    choice [answer] h([answer]);
}
```

The signature is the datatype at the root of every DP program. BUT in the classical style it stays invisible, because the candidates are never represented.



Questions are algebras

Evaluation algebras

Evaluation $=$ scoring candidates $+$ making choices

Evaluation algebra $=$ scoring functions $+$ choice function

Choice Functions

There is a variety of objective functions

h: [Values] $\rightarrow$ [Values]

- most popular: $h = max$, $h = min$
- also popular: $h = max_k$, $h = min_k$
- enumeration: $h = id$ (keep everyone)
- combinatorics: $h = sum$
- sampling: $h =$ random choice

Scoring schemes

"Scores" can be anything:

- distance / similarity between substructures (for alignment)
- probabilities (for predicting based on a probabilistic model)
- free energy (for a thermodynamic folding of molecules)
- candidate representations (as strings / trees / graphics)
- candidate counts

Scoring alingments

```
1 algebra score implements
2   Align(alphabet = char, answer = int) {
3     int replace(<char a, char b>, int x) {
4       if (a == b) return x + 14; else return x - 1; }
5     int delete(<char g, void>, int x) { return x - 2; }
6     int insert(<void, char g>, int x) { return x - 2; }
7     int empty(<void, void>) { return 0; }
8     choice [int] h([int] l) { return list(maximum(l)); }
9 }
```

We evaluate

m(**S**, m(**T**, i(i(i(i(i(m(**L**, m(**O**, r(**U**, d(**I**, r(**S**, e(**$**), **P**), **O**), **O**), **L**), **E**), **G**), **N**), **A**), **R**), **T**), **S**) = 42

Problem variants: Affine gaps and local alignment

For affine gap costs (Gotoh):

- add functions `del_extend`, `ins_extend`

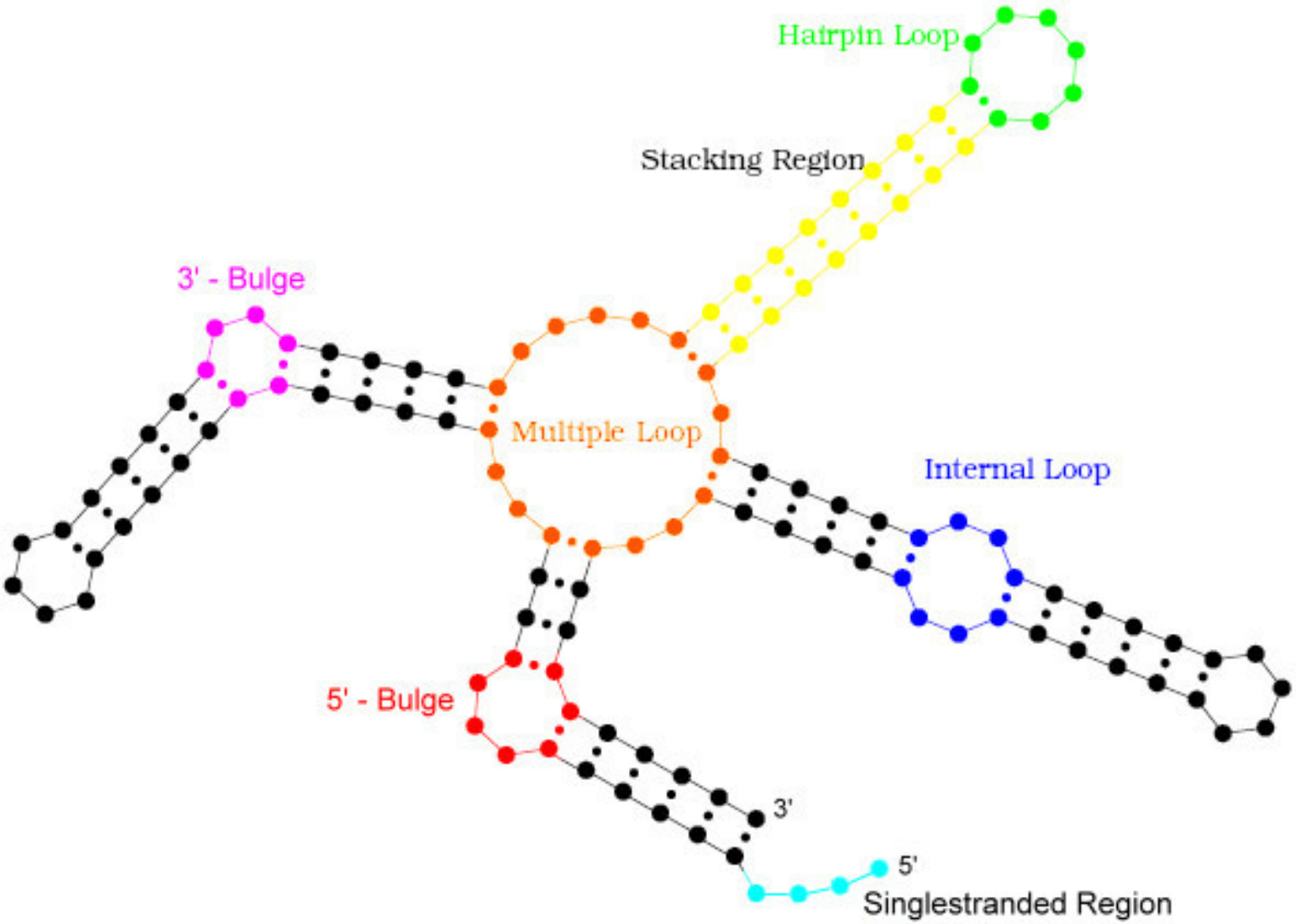
For local alignment (Smith-Waterman):

- add functions `skip_left`, `skip_right`

For local alignment with affine gap costs:

- add both

Building blocks of RNA



```
1  algebra pretty implements
2    FoldRNA(alphabet = char, answer = string) {
3      string sr(Subseq lb,string e,Subseq rb) {
4        string res;
5        append(res, '(');
6        append(res, e);
7        append(res, ')');
8        return res;
9      }
10     string hl(Subseq lb,Subseq region,Subseq rb) {
11       string res;
12       append(res, '(');
13       append(res, '.', size(region));
14       append(res, ')');
15       return res;
16     }
17     ...
18     choice [string] h([string] i) { return i; }
19 }
```

We evaluate

sr(C, sr(C, ml(A, sr(C, hl(C, UUUU, g), G),
sr(C, bl(AUA, hl(C, CCC, G)), G), U), G), G) =
"((((....))(...(...))))"

Counting solutions: RNA structures

```
1 algebra mycount auto count
```

We evaluate

sr(C, sr(C, ml(A, sr(C, hl(C, UUUU, g), G),
sr(C, bl(AUA, hl(C, CCC, G)), G), U), G), G) = 1

Programs are grammars

Where do we stand?

We know now

- how to represent candidates
- how to score and choose

We still need to know:

- which are the candidates for given input (problem istance)

We do this using tree grammars

string grammar: describes a language of **strings**.

A parser, derived automatically from the grammar, recognizes the strings that are in the language.

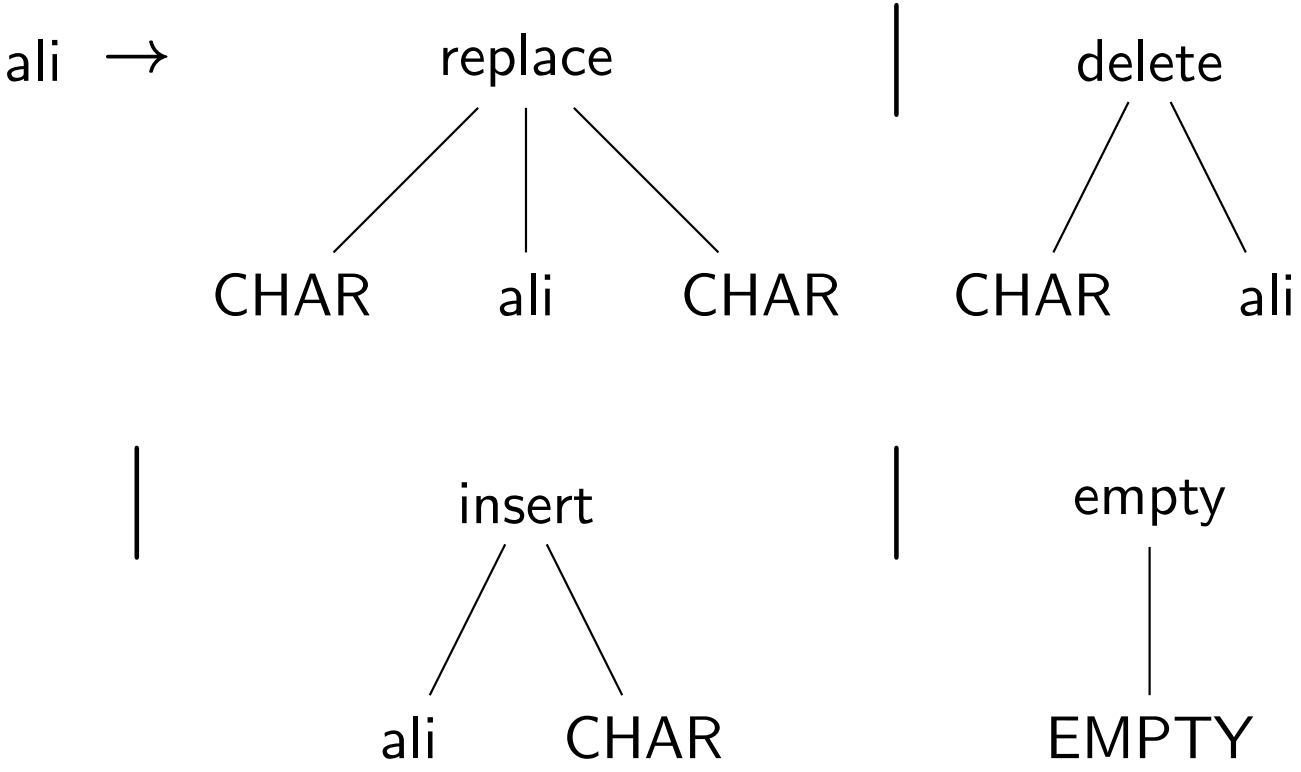
tree grammar: describes a language of **trees** (candidates).

These trees have "input strings" as their yield sequences.
A yield parser, derived automatically from the grammar, reads the yields and generates the corresponding candidates.

Programs are grammars

```
1 grammar alignment uses Align(axiom = ali) {
2     ali = replace(<CHAR, CHAR>, ali) |
3           delete(<CHAR, EMPTY>, ali) |
4           insert(<EMPTY, CHAR>, ali) |
5           empty(<EMPTY, EMPTY>) # h;
6 }
```

Programs are grammars



Problem specification

Definition An Algebraic DP algorithm is specified by

- an evaluation signature Σ
- a tree grammar $\mathcal{G}$ over Σ
- a concrete evalutaion algebra $\mathcal{A}$ with an objective function h satisfying *Bellman's Principle*

Bellman's Principle of Optimality

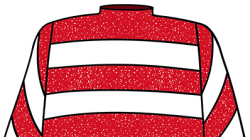
Richard Bellman (1964):
"An optimal solution can be composed solely from optimal solutions to sub-problems."

That's a requirement, not a theorem!!

We can prove this by proving distributivity of **choice** over **scoring**:

$$h(f(X, Y)) = h(f(h(X), h(Y)))$$

Moving the choice function around in the formula should not affect the final result list.



Phase amagalmation

$$\underbrace{\textbf{rnafold}}_{\textit{grammar}}\,(\,\underbrace{\textbf{basepair}}_{\textit{algebra}},\ \underbrace{\texttt{"ACAGGUUGU"}}_{\textit{input}}\,) \Rightarrow 3$$

Conceptual view:

Phase 1 = yield parsing

Phase 2 = evaluation & choice

Reality: Both phases are merged

Products are fun!

Where do we stand? (revisited)

We can

- describe algorithms on an abstract level
- generate correct and efficient code
- independently vary tree grammar or evaluation algebra
- run one analysis at a time

How about doing several analyses at a time?

- computing the best score AND printing the best scoring candidate
- computing the best RNA structure for each different abstract shape of the molecule

Products of algebras

Product algebras $\mathcal{A} := \mathcal{A}_1 * \mathcal{A}_2$

- compute answer-value pairs
- using functions **f** and **h**
  1. $f_1 * f_2$ component wise
  2. $h_1 * h_2$ dependent

Semantics of $*$

Intuitive understanding:

Phase 1 computes all candidates via $f_1 * f_2$

Phase 2 applies $h_1 * h_2$ once in the end

Reality: everything is interleaved! (like phase amalgamation of DP)

No programming, no debugging, but proof obligation with $*$:
$\mathcal{A}_1 * \mathcal{A}_2$ must satisfy Bellman's Principle

Fun things to do with products

- Number of co-optimal solutions **basepair∗count**
- Easy candidate output (backtracking) **basepair∗pretty**
- Classified dynamic programming **shape∗count**, **shape∗bpmax**
- Ambiguity checking **pretty∗count**
- Sampling **A** | **B**
- Products of products...

Tools developed with ADP

Tools

- RNAhybrid
- pknotsRG
- RNAshapes
- Locomotif
- KnotInFrame
- RNAsifter

Problems solved

- miRNA target prediction
- pseudoknot folding
- abstract shape analyis
- consensus structure prediction
- probabilistic shape analysis
- RNA motif search description and search
- programmed ribosomal frame shift detection
- filtering out unproductive Rfam searches

What's cool about Algebraic Dynamic Programming?

Advantages:

- our work is reduced to the creative apects
- we explore ideas rather than debug code
- we create re-usable and reliable components
- we turn tricks into techniques
- we make DP easier to learn

Disadvantages:

- textbooks use old-fashioned recurrences ;)
- limited to sequence-like data, decomposition into subwords

Thanks to Robert, Peter, Jens, Christian, Stefan...

Remember reverse engineering of 42!!
Talk to me about your favorite DP problem!
sschirme@gmail.com
@linse on twitter

Thanks for your attention! <3

Resources

- Bellman's Gap Cafe — http://gapc.eu
- The compiler — http://gapc.eu/compiler.html
- Literature — http://gapc.eu/literature.html