On Solving a Generalized Constrained Longest Common Subsequence Problem

– Seminar talk –

M. Djukanovic[1], A. Kartelj[3], D. Matic[4], M. Grbic[4], G. Raidl[1], C. Blum[2]

[1]Institute of Logic and Computation, TU Wien, Vienna, Austria,
[2] Artificial Intelligence Research Institute (IIIA), Spanish National Research Council (CSIC), Barcelona, Spain
[3]Faculty of Mathematics, Univeristy of Belgrade, Serbia
[4]Faculty of Science and Mathematics, Univerity of Banja Luka, B & H

March 16, 2021

ALGORITHMS AND COMPLEXITY GROUP

Introduction

A *string* is a finite sequence of characters over some (finite) alphabet Σ.

Strings are commonly as models for presenting DNA and RNA molecules, proteins, texts, etc.

Bioinformatics and strings:

- finding similarities between molecules: understanding of biological processes
- (discrete) optimization problems

Longest Common Subsequence (LCS) Problem

Object to measure similarity: A *subsequence* of string s is any sequence of characters obtained by deleting zero or more characters from s.

LCS Problem:

- Input: set of strings $S = \{s_1, \ldots, s_m\}, m \in \mathbb{N}$, alphabet Σ
- Objective: find a *subsequence* of *maximum* length that is *common* for all strings from S
- $\mathcal{NP}$–hard problem for arbitrary large set S

Constrained Longest Common Subsequence (CLCS)

CLCS Problem (m–CLCS):

- Input: a set of strings $S = \{s_1, \ldots, s_m\}, m \in \mathbb{N}$, and alphabet Σ, and a pattern string p_1.
- Objective: find a *subsequence* of *maximum* length that is *common* for all strings from S and has p_1 as its subsequence.

Constrained Longest Common Subsequence (CLCS)

CLCS Problem (m–CLCS):

- Input: a set of strings $S = \{s_1, \ldots, s_m\}, m \in \mathbb{N}$, and alphabet Σ, and a pattern string p_1.
- Objective: find a *subsequence* of *maximum* length that is *common* for all strings from S and has p_1 as its subsequence.
- **Generalized constrained longest common subsequence problem** ((m, k)–CLCS): apart of m–CLCS problem, it has an arbitrary set of k pattern strings in input.

Example:

s_1	a	b	c	a	c	a	
s_2	a	c	b	c	c	a	a
p_1	a	c					
p_2	b	c					

CLCS: abcaa

Literature Overview

Practical relevance: identifying homology between biological sequences which posses a specific or putative structure in common:

- RNase, Kinase, Protease posses patterns such as KHK, KKH, HKH, etc. in common.

2–CLCS problem:

- Introduced by Tsai (2003)
- *Polynomially* solvable by dynamic programming (DP) in $O(|s_1| \cdot |s_2| \cdot |p_1|)$
  - A few sparse DP approaches

m–CLCS problem:

- $\mathcal{NP}$-hard if m arbitrary, and a fixed pattern
- Approximation algorithm by Gotthilf et al. (2008)
- A Greedy heuristic, Beam Search, and A* proposed by Djukanovic et al. (2020)

Literature approach

(m, k)–**CLCS problem:**

- $(2, k)$–CLCS problem is $\mathcal{NP}$–hard, Gothilf et al. (2011)
- Moreover, approximation algorithms (with guaranteeing ratio) cannot exist for $(2, k)$–CLCS problem
- Interestingly, we were able to prove that the problem of finding at least one feasible solution for (m, k)–CLCS problem is $\mathcal{NP}$–hard
  - was not the case of m–CLCS $= (m, 1)$–CLCS problem
- (m, k)–CLCS solved by Farhana and Rahman (2015), Automaton approach

So, developing algorithms in three different directions makes sense

- feasibility check
- high-quality solutions
- proving optimality

Notation and Data Structures

An instance of (m, k)–CLCS problem is given in the following way:

- $S = \{s_1 \ldots, s_m\}$
- $P = \{p_1, \ldots, p_k\}$, and
- Σ

Given a position vector $\vec{\theta} \in \mathbb{N}^m$,

- $S[\vec{\theta}] := \{s_i[\theta_i, |s_i|] \mid i = 1, \ldots, m\}$ denotes a subproblem of the original (m, k)–CLCS instance w.r.t. input strings

A cover position vector $\vec{\lambda} \in \mathbb{N}^k$, indicates a subproblem

- $P[\vec{\lambda}] := \{p_j[\lambda_j, |p_j|] \mid j = 1, \ldots, k\}$ concerning set of pattern strings P

Data structures:

- $\mathrm{Succ}[i, j, a] = x$, position $x \geq j$ in string s_i such that $s_i[x] = a$; or -1 otherwise;
- $\texttt{Embed}[i, r, j] = x$ for all $i \in [m]$, $j \in [k]$ and $r \in [|p_j| + 1]$ stores the right-most (largest) position x of s_i such that $p_j[r, |p_j|]$ is a subsequence of $s_i[x, |s_i|]$.

Greedy method

Based on the well-known Best-Next heuristic:

- At each step, a letter with the best greedy value appended (to the end) to current greedy sol. s

Candidates for extension are letters $c \in \Sigma_s^{\mathrm{nd}}$ which fulfill:

- Condition 1: Letter c appears at least once in each of the prefix strings $s_i[\theta_i, |s_i|]$, $i = 1, \ldots, m$,
- where dominated nodes removed from Σ_s^{nd} (w.r.t. positions $\vec{\theta}, \vec{\lambda}$): We say that a dominates by b iff
  - $\mathrm{Succ}[i, \vec{\theta}_i, a] \leq \mathrm{Succ}[i, \vec{\theta}_i, b]$ and
- Condition 2: After appending c to s (and updating $\vec{\theta}, \vec{\lambda}$), we ensure all remaining $p_i[\vec{\lambda}_i, |p_i|], i \in [k]$ may be embedded into each $s_j[\vec{\theta}_i, |s_i|]$, $j \in [m] \Rightarrow$ values of structure Embed not pre-computed, calculated on demand w.r.t. $\vec{\lambda}$

Greedy criterion – additional conditions

To maximize chances for feasibility in our Greedy we add:

- Condition 3. Those letters which contribute to cover at least one not-yet-covered letter of any p_i preferred: $\Sigma_s^{\mathrm{nd,str}}$

Example

Initialize: $\vec{\theta} = (1, ..., 1)$, $\vec{\lambda} = (1, \ldots, 1)$, $s = \varepsilon$, greedy heuristic:

$$g(s, \vec{\theta}, c) = \sum_{i=1}^{m} \frac{\texttt{Succ}[i, \theta_i, c] - \theta_i + 1}{|s_i| - \theta_i + 1} \quad \forall\, c \in \Sigma_s^{\mathrm{nd,str}} \qquad (1)$$

where s is the current greedy sol. $\vec{\theta}$: current position vector.

Example

Initialize: $\vec{\theta} = (1, ..., 1)$, $\vec{\lambda} = (1, \ldots, 1)$, $s = \varepsilon$, greedy heuristic:

$$g(s, \vec{\theta}, c) = \sum_{i=1}^{m} \frac{\mathtt{Succ}[i, \theta_i, c] - \theta_i + 1}{|s_i| - \theta_i + 1} \quad \forall\, c \in \Sigma_s^{\mathrm{nd,str}} \tag{1}$$

where s is the current greedy sol. $\vec{\theta}$: current position vector.

Note that the proposed greedy heuristic can not guarantee the construction of a feasible solution.

Example. Instance $S = \{\mathtt{abbba}, \mathtt{babb}\}$, $P = \{\mathtt{bb}, \mathtt{a}\}$, and $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

- Step I of the greedy heuristic: $\Sigma_s^{\mathrm{nd},str} = \{\mathtt{a}, \mathtt{b}\}$ are the candidates to extend the empty solution. Their greedy heuristic values are equal.
- Choosing $\mathtt{b}$ automatically leads to an unfeasible solution, that is, solution $\mathtt{bbb}$ is not feasible.

Search space of the (m, k)–CLCS problem

Nodes are $v = (\vec{\theta}^v, \lambda^v, l^v)$ where

- $\vec{\theta}^v$ is a position vector,
- λ^v is a cover position vector, and
- l^v is the length of a partial solution represented by node v

We say that partial solution s^v induces node $v = (\vec{\theta}^v, \lambda^v, l^v)$ iff

- $\vec{\theta}^v$ is defined such that $s_i[1, \theta_i^v - 1]$ is the shortest possible prefix string of s_i of which s^v is a subsequence.
- $\vec{\lambda}^v$ is defined such that $p_j[1, \lambda_j^v - 1]$ is the longest prefix string of p_j which is a subsequence of s^v.
- $l^v := |s^v|$

Node Extension

A child node w of v is generated as follows (suppose we extend v by letter $\mathtt{a} \in \Sigma^{\mathrm{nd}}_{s^v} = \Sigma^{\mathrm{nd}}_{v}$):

- $\theta^w_i := \mathtt{Succ}[i, \theta^v_i, a] + 1$, for all $i = 1, \ldots, m$
- If $p_j[\lambda^v_j] = a$ then $\lambda^w_j := \lambda^v_j + 1$; $\lambda^w_j := \lambda^v_j$ otherwise;
- $l^w := l^v + 1$.

The root node: $r = ((1, \ldots, 1), (1, \ldots, 1), 0)$: induced by the empty partial solution ε.

A node v is called non-extensible if $\Sigma^{\mathrm{nd}}_v = \emptyset$.

A node is called *feasible* iff $\lambda^v_j = |p_j| + 1$, for all $j = 1, \ldots, k$.

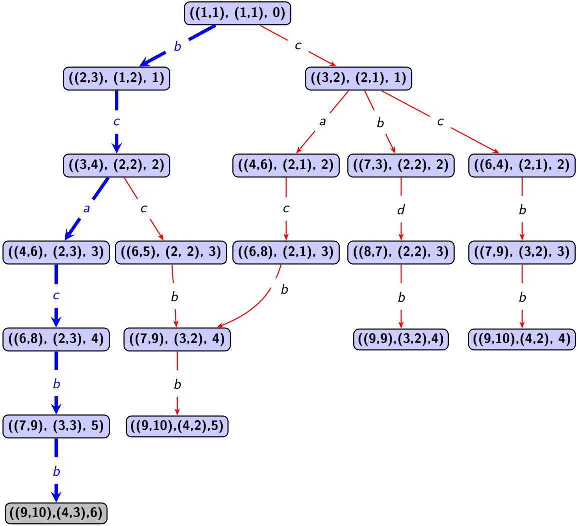


Figure:
($S = \{s_1 = \texttt{bcaacbdba}, s_2 = \texttt{cbccadcbbd}\}, P = \{\texttt{cbb}, \texttt{ba}\}, \Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$).
Only one node in the space (light-gray color) corresponds to a feasible solution $s = \texttt{bcacbb}$

The Concept of Restricted Search Space

- The full state space adapted towards maximizing the chances of finding at least one feasible solution
- Set of child nodes of node v gets restricted: prefer those child nodes over others which improve patterns coverage ($\Sigma_v^{\mathrm{nd}} \Rightarrow \Sigma_v^{\mathrm{nd,str}} \neq \emptyset$)

The Concept of Restricted Search Space

- The full state space adapted towards maximizing the chances of finding at least one feasible solution
- Set of child nodes of node v gets restricted: prefer those child nodes over others which improve patterns coverage ($\Sigma_v^{\mathrm{nd}} \Rightarrow \Sigma_v^{\mathrm{nd,str}} \neq \emptyset$)

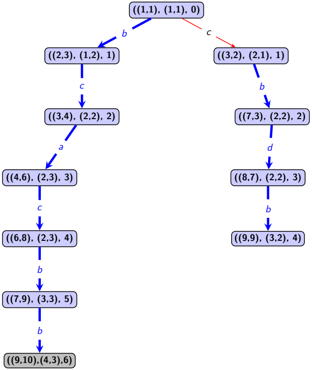


Figure: The restricted search space (same instance from the last slide).

Beam Search

Beam search (BS):

- Works in a restricted Breadth-First-Search manner
- $\beta > 0$ best nodes of each level selected for further expansions acc. to a heuristic guidance h

Beam Search

Beam search (BS):

- Works in a restricted Breadth-First-Search manner
- $\beta > 0$ best nodes of each level selected for further expansions acc. to a heuristic guidance h

Heuristic guidances:

- Reasonably tight UB for LCS problem as the combination of an occurrences–based and a dynamic-programming based upper bound
- A new probability–based heuristic guidance developed (next slide)

Probability–based Heuristic Guidances

From Mousavi and Tabataba (2012), assuming

- independence among the input strings,
- randomness of input strings

the probability that a random string s of length r is a common subsequence of all input strings from S is

$$\text{Prob}(s \prec S) = \prod_{i=1}^{m} \mathrm{Pr}(r, |s_i|), \tag{2}$$

Pr: pre-processed (by DP).

In order to make use of Eq. (2) in the case of the (m,k)–CLCS problem, we assume

- each such string s is extensible towards a feasible (m,k)–CLCS solution (s has at least one feasible completion)

[Cont'd]

Choosing the value of r (length of extension):

- to make fair comparison to all nodes of the same level of BS, r shall be common to all nodes

$$p_j^{\min} = \min_{v \in V_{\mathrm{ext}}} \left(|p_j| - \lambda_j^v + 1 \right), j = 1, \ldots, k. \tag{3}$$

and then summing up all the values (heuristic choice for the number of safe extension w.r.t. our assumption), we get $p^{\min} = \sum_{j=1}^{k} p_j^{\min}$, and finally

$$r = p^{\min} + \min_{v \in V_{\mathrm{ext}}} \left\lfloor \frac{\min_{i=1,\ldots,m} \{|s_i| - \theta_i^v + 1\} - p^{\min}}{|\Sigma|} \right\rfloor. \tag{4}$$

The heuristic guidance is stated for each node v (at same level) by

$$H(v) = \prod_{i=1}^{m} \mathrm{Pr}(r, |s_i| - \theta_i^v + 1).$$

An A* search

- Introduced by Hart et al. (1968)
- Works in best-first-search manner (always most promising nodes expanded first)
- Nodes prioritized acc. to $f(v) = g(v) + h(v)$ where
  - $g(v)$: the length of a longest path from root node r to node v
  - $h(v)$: estimated cost from v to a goal node (dual bound)
- Data structures to set up an A* for (m, k)–CLCS:
  - **Hash map** N of nodes whose keys are pairs $(\vec{\theta}, \vec{\lambda})$ with values l^v which stores the length of longest path to all node assoc. to $\vec{\theta}, \vec{\lambda}$ (nodes: clusters of partial solutions)
  - **Priority queue** $Q \subseteq N$: list of not-yet-expanded nodes
  - UB: the upper bound for LCS known from literature, monotonic
  - **Goal nodes:** non-extendable, feasible nodes
- A problem–specific nodes' filtering:
  $\mathrm{UB}(v) < l^v + \max\{|p_i| - \lambda_i^v + 1 \mid i = 1, \ldots, k\}$

Variable Neighborhood Search (VNS)

- Proposed by Mladenovic and Hansen (1997)
- Systematically change of neighborhood structures in order to escape from local minima: diversification
- Intensification: Local search, i.e. small-change-neighbor

Idea of VNS applied on $(m.k)$–CLCS problem based on

- DP for two strings only when necessary
- insert/update/delete operations on current solution
- penalty function counting the number of feasibility violations on a per character basis

VNS: details

- Fitness function:

$$F(\mathbf{sol}) := \begin{cases} \sum_{s_i \in S} (|sol| - |\mathrm{LCS}(sol, s_i)|) + \sum_{p_j \in P} (|p_j| - |\mathrm{LCS}(p_j, sol)|) & \text{if } sol \text{ infeasible,} \\ \frac{n_{\min} - |sol|}{n_{\min} + 1} & \text{if } sol \text{ feasible.} \end{cases} \quad (5)$$

- Motivation for using this function:
  - Until a feasible solution is found, focus is more on reaching feasibility as soon as possible, by updating / removing characters.
  - Once feasibility reached, the fitness function will thrive the algorithm to increase solution, by adding letters.

VNS details: shaking

Two kind of Shaking realized depending on the feasibility of the solution:

- Shaking_Delete(*sol*, κ) applied if *sol* is feasible: randomly removing κ letters from *sol* in order to move away from the current solution;
- Shaking_Change(*sol*, κ) applied if *sol* is unfeasible: it selects κ random positions in *sol* and changes the letters at the chosen positions to randomly chosen letters from Σ.

Purpose of the both shaking:

- Shaking_Delete(*sol*, κ): for the shake of diversification
- Shaking_Change(*sol*, κ): more aiming for solution feasibility

VNS details: local search

It combines two first improvement strategies with a time complexity of $O(|sol| \cdot |\Sigma|)$ per LS iteration.

1. Change-Based-LS: find a pair $(i, \sigma), i \in \{1, \ldots, |sol|\}, \sigma \in \Sigma \cup \{\varepsilon\}$ so that by changing $sol[i] = \sigma$, fitness function $F(sol)$ is improved;
2. Insert-Based-LS: find a pair $(i, \sigma), i \in \{1, \ldots, |sol| + 1\}$, $\sigma \in \Sigma$ so that $F(sol)$ is improved by inserting letter σ before position i in sol.

VNS details: efficiency

The most time consuming part of VNS is fitness calculation in LS:

- Partial fitness function calculate ($\mathrm{LCS}(sol, s_i)$ and $\mathrm{LCS}(p_j, sol)$), $i \in [m], j \in [k]$: $m + k$ DP for two strings
- Fitness score F after operations like insert/update/delete of a single letter in *sol* calculated partially (in linear time) $\Rightarrow$ The two LS–based procedure works without any application of DP

VNS: remarks

Partial fitness calculation is a bit too technical, but based on the concept of

- determining the right–most embedding of $s^* = \mathrm{LCS}(sol, s_i)$ into s_i, $i \in [m]$; and corresponding left-most (linearly)
- detecting middle regions of s_i between left–most and right–most embedding of s^* which give relevant regions of s_i for scanning candidate letters for insertions able to improve F values
- after performing an edit operation (update/delete), in general, a tight upper bound on current F value will be produced

Illustration

Table: Middle regions (shown with a light-gray background) for solution $sol = \texttt{abccada}$ w.r.t. input strings and patterns.

s_1	a	a	b	c	a	a	b	a	a	d
left mapping	a		b	c	a					d
right mapping		a	b	c					a	d
s_2	a	a	a	b	c	a	b	a	d	a
left mapping	a			b	c	a			d	a
right mapping			a	b	c			a	d	a
$p1$	b	c	a	b	a	a				
left mapping	b	c	a		a					
right mapping	b	c			a	a				
$p2$	a	a	b	b	a	a				
left mapping	a		b		a	a				
right mapping		a		b	a	a				

Experimental Studies

Machine settings:

- C++ using GCC 7.4
- Intel Xeon E5–2640 processor with 2.40 GHz

Time & memory limit:

- 1200 sec.
- Memory:
  - 4 Gb for VNS
  - 16 Gb for BS
  - 32 Gb for A*

Algorithms tested:

1. Greedy algorithm (GREEDY);
2. BS on the full search space, labelled by BS-BASIC;
3. BS on the restricted search space, labelled by RESTRICTED-BS;
4. Variable neighborhood search (VNS);
5. The hybrid BS&VNS in which RESTRICTED-BS provides an initial solution for the VNS.

Benchmark sets

Two different set of instances set up for experiments:

- RANDOM instances where for each
  - length of input strings $n \in \{100, 500, 1000\}$,
  - number of input strings $m \in \{2, 5, 10\}$,
  - alphabet size $|\Sigma| \in \{2, 4, 20\}$,
  - number of pattern strings $k \in \{2, 5, 10\}$, and
  - length of pattern strings descried by a ratio $p = \frac{|p_0|}{n} \in \left\{\frac{1}{50}, \frac{1}{20}\right\}$

  10 instances were generated (ensuring at least one feasible solution), which gives us **1 620 instances**.

- REAL–world benchmark set:
  - 40 different sets of Bacteria where m ranges from 2 to 12 681 which lengths range from around 600 to around 2 000.
  - Number of pattern strings is 15, some of them are:
    - `gtgtagaggtgaaatgcgtagat`
    - `caaacaggattagaaacccaagtagtccacgc`
    - `aaaatcaaaaaaatagacggggacccgcacaag`.

Parameters' tuning:

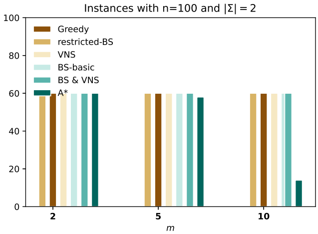
Our algorithms

- BS-BASIC
- RESTRICTED-BS
- VNS

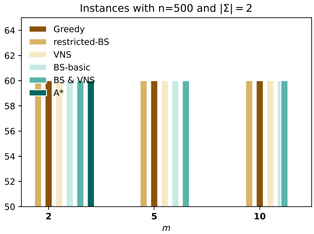
tuned w.r.t solution-quality via *irace*.

- results of RESTRICTED-BS passed to VNS – BS&VNS

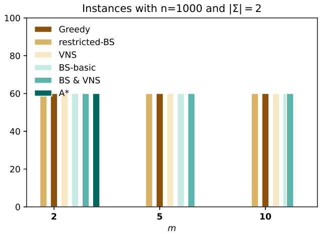
Results: feasibility check for $|\Sigma| = 2$



(a) $n = 100$.

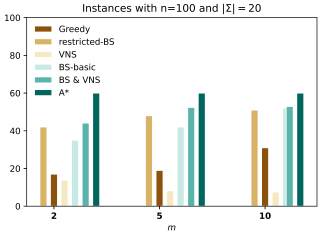


(b) $n = 500$.

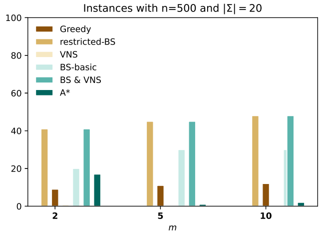


(c) $n = 1000$.

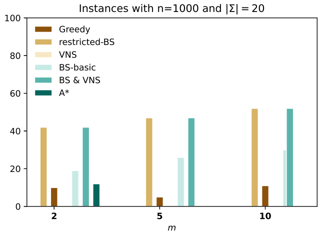
Results: feasibility check for $|\Sigma| = 20$



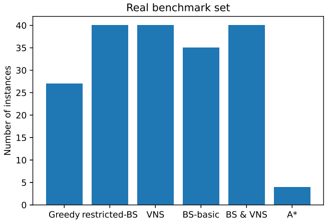
(a) $n = 100$.



(b) $n = 500$.

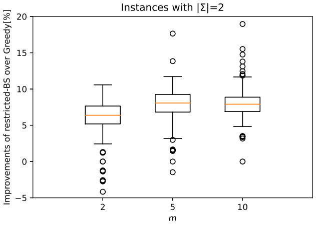


(c) $n = 1000$.

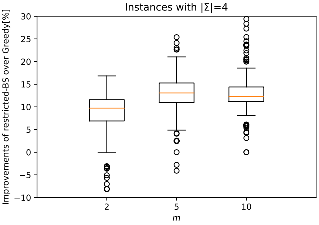


(d) REAL benchmark set.

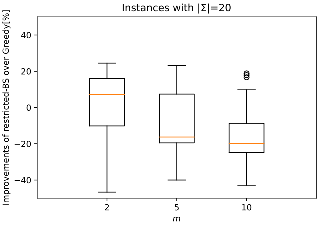
Solution quality: Greedy vs. restricted-BS (common feas.)



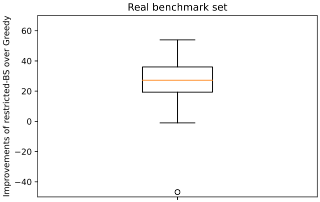
(a) $|\Sigma| = 2$.



(b) $|\Sigma| = 4$.

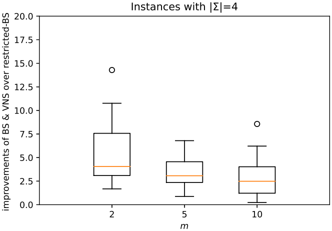


(c) $|\Sigma| = 20$.

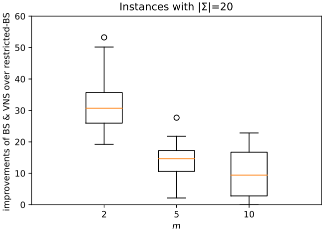


(d) REAL benchmark set.

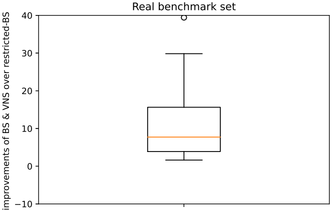
Solution quality: BS & VNS vs. restricted-BS



(a) $|\Sigma| = 4$.

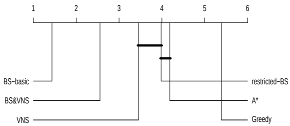


(b) $|\Sigma| = 20$.

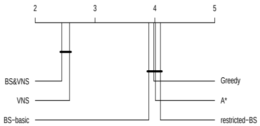


(c) Real benchmark set.

Significance between the algorithms: $|\Sigma| = 4$



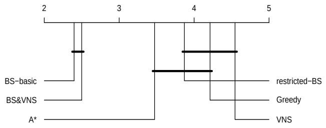
(a) Solution quality comparison



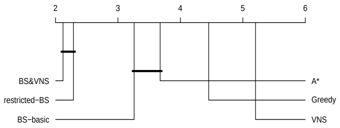
(b) Feasibility comparison

Figure: Instances with $|\Sigma| = 4$.

Significance between the algorithms: $|\Sigma| = 20$



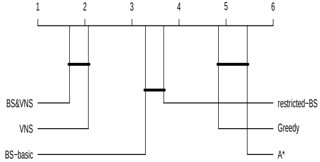
(a) Solution quality comparison



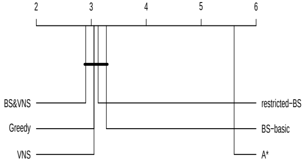
(b) Feasibility comparison

Figure: Instances with $|\Sigma| = 20$.

Benchmark set Real



(a) Solution quality comparison



(b) Feasibility comparison

Figure: Benchmark set Real.

A^* vs. Automaton approach

Instance group	#inst	A*			Automaton		
		$\overline{t}[s]$	ub	opt[%]	$\overline{\lvert s \rvert}$	$\overline{t}[s]$	opt[%]
Rnase	3	0.12	68.33	100	68.33	4.78	100
Protease	15	0.7	55.6	100	55.6	4.71	100
Kinase	3	0.1	111	100	111	13.4	100
Globin	10	0.11	84.1	100	84.1	7.8	100
Input100	1	0.06	2	100	2	48.38	100

Table: Results on real-world benchmark set used for Automaton approach.

A^* vs. Automaton approach

Instance group	#inst	A*			Automaton		
		$\overline{t}[s]$	ub	opt[%]	$\overline{\lvert s\rvert}$	$\overline{t}[s]$	opt[%]
Rnase	3	0.12	68.33	100	68.33	4.78	100
Protease	15	0.7	55.6	100	55.6	4.71	100
Kinase	3	0.1	111	100	111	13.4	100
Globin	10	0.11	84.1	100	84.1	7.8	100
Input100	1	0.06	2	100	2	48.38	100

Table: Results on real-world benchmark set used for Automaton approach.

Instance					A*			Automaton		
m	n	$\lvert p_i\rvert$	k	#inst	$\overline{t}[s]$	ub	opt[%]	$\overline{\lvert s\rvert}$	$\overline{t}[s]$	opt[%]
2	100	40	1	10	0.02	44.5	100	44.5	1.70	100
2	250	45	2	10	14.2	107.6	70	106.9	4.71	100
2	250	8	3	10	120.1	88.4	30	87.4	27.2	100
2	250	6	4	10	154.9	87.1	80	87.0	82.6	100

Table: Results on random instances used for Automaton approach, $\lvert\Sigma\rvert = 20$.

Conclusions & Future Work

Conclusions:

- a few heuristic approaches proposed to deal with large–sized instances:
  - efficient in various aspects such as finding high-quality solutions (BS-BASIC and BS&VNS) as well as proving feasibility (RESTRICTED-BS)
  - the search guided by a probability–based heuristic guidance
  - BS & VNS works best on benchmark set REAL
- proposed an A* search to deal with the instances of moderate size:
  - ≈ 35% random instances solved to proven optimality
  - 4 real–world instances solved to optimality

Conclusions & Future Work

Conclusions:

- a few heuristic approaches proposed to deal with large–sized instances:
  - efficient in various aspects such as finding high-quality solutions (BS-BASIC and BS&VNS) as well as proving feasibility (RESTRICTED-BS)
  - the search guided by a probability–based heuristic guidance
  - BS & VNS works best on benchmark set REAL
- proposed an A* search to deal with the instances of moderate size:
  - $\approx$ 35% random instances solved to proven optimality
  - 4 real–world instances solved to optimality

Future work:

- develop anytime algorithms for the large-sized instances (gaps)
- develop more sophisticated search guidances
- prove feasibility of remaining instances where our algorithms fail ($\approx 7 - 8\%$) random instances, $|\Sigma| = 20$ (why not MCTS?)

Thank you for your attention!

Edit operations and partial fitness calculation

Example. Consider the change operation:

$$sol = \texttt{abcca}\boxed{\texttt{d}}\texttt{a} \text{ to } s^{new} = \texttt{abcca}\boxed{\texttt{a}}\texttt{a}.$$

- It is never considered in the partial LCS calculation w.r.t. s_1 and s_2, since d is not part of a middle region.
- Note that changing d to character a would produce $\mathrm{LCS}(s_1, s^{new}) = \texttt{abcaaa}$, which has length 6
- But, the result of partial calculation would be 5