Optimizing Statistical Information Extraction Programs Over Evolving Text

Fei Chen [1], Xixuan Feng [2*], Christopher Ré [3], Min Wang [4]

[1,4] *HP Labs China,* [2,3] *University of Wisconsin-Madison*

[1] fei.chen4@hp.com [2] xfeng@cs.wisc.edu [3] chrisre@cs.wisc.edu [4] min.wang6@hp.com

Abstract—**Statistical information extraction (IE) programs are increasingly used to build real-world IE systems such as Alibaba, CiteSeer, Kylin, and YAGO. Current statistical IE approaches consider the text corpora underlying the extraction program to be static. However, many real-world text corpora are dynamic (documents are inserted, modified, and removed). As the corpus evolves, and IE programs must be applied repeatedly to consecutive corpus snapshots to keep extracted information up to date. Applying IE *from scratch* to each snapshot may be inefficient: a pair of consecutive snapshots may change very little, but unaware of this, the program must run again from scratch. In this paper, we present CRFlex, a system that efficiently executes such repeated statistical IE, by recycling previous IE results to enable incremental update. As the first step, CRFlex focuses on statistical IE programs which use a leading statistical model, Conditional Random Fields (CRFs). We show how to model properties of the CRF inference algorithms for incremental update and how to exploit them to correctly recycle previous inference results. Then we show how to efficiently capture and store intermediate results of IE programs for subsequent recycling. We find that there is a tradeoff between the I/O cost spent on reading and writing intermediate results, and CPU cost we can save from recycling those intermediate results. Therefore we present a cost-based solution to determine the most efficient recycling approach for any given CRF-based IE program and an evolving corpus. We conduct extensive experiments with CRF-based IE programs for 3 IE tasks over a real-world data set to demonstrate the utility of our approach.**

I. Introduction

Information extraction (IE) programs extract structured data from unstructured text, such as emails, Webpages and blogs. An important technique to build IE programs is to employ statistical learning models such as Hidden Markov Models, Conditional Random Fields, and Support Vector Machines. Compared to rule-based approaches to IE, *statistical IE programs* can capture complex patterns (e.g., positive/negative opinions) with lower amounts of human interaction and are more robust to noise and variation in unstructured text. As a result, it is not surprising that there are a growing number of real-world systems built by statistical IE, including Kylin [1], Microsoft's Academic Search [2] and YAGO [3].

To keep extracted information up to date, the above systems must apply IE *repeatedly* to snapshots of a corpus; these repeated extractions are necessary because many text corpora are *dynamic*: documents in the corpus are added, deleted, and modified over time. Consider for example DBLife [4], a structured portal for the database community. DBLife operates over a text corpus of 10,000+ URLs. Everyday it recrawls these URLs to generate a 120+ MB corpus snapshot, and then applies IE to this snapshot to find the latest community information (e.g., which database researchers have been mentioned where in the past 24 hours). As another example, the isWiki project at HP Labs China seeks to build a system that manages all information for enterprise IT support through a wiki. This system must regularly re-crawl and then re-apply IE to the wiki, to infer the latest IT-related information. Several efforts (e.g. *freebase.com*) have also focused on converting *Wikipedia* and its wiki "siblings" into structured databases, and hence must regularly recrawl and re-extract information. Other examples of dynamic text corpora can be found in recent work [5], [6].

Despite the pervasiveness of dynamic text corpora, no satisfactory solution has been proposed for statistical IE over them. Given such a corpus, the common solution today is to treat the corpus as *static* and to apply statistical IE to each corpus snapshot *in isolation, from scratch*. This static solution is simple but is highly inefficient. For example, in DBLife reapplying IE from scratch takes 8+ hours each day, leaving little time for higher-level data analysis. Worse still, ignoring the dynamic nature of the corpus precludes applying IE to time-sensitive applications (e.g., stock, auction, intelligence analysis). In such applications, users want the most up-to-date information, which is supported by re-crawling and re-extracting from the corpus at short intervals (e.g., every 30 minutes). In such cases, a static IE solution may take more than 30 minutes simply because it is relabeling many documents which have only changed slightly. Finally, the static solution is ill-suited for *interactive debugging* of IE applications over dynamic corpora, because such debugging often requires applying IE repeatedly to multiple corpus snapshots. Thus, given the growing need for IE over dynamic text corpora, it has now become crucial to develop efficient statistical IE solutions for dynamic settings.

In this paper, we present CRFlex, the first step toward a general solution for efficiently and repeatedly executing statistical IE programs. To the best of our knowledge, we are the first to consider statistical IE over dynamic corpora. CRFlex focuses on a leading statistical learning model, *Condition Random Fields* (CRFs). CRF-based IE is a state-of-the-art IE solution that is a workhorse of the above IE systems, and has been used on many IE tasks, including *named entity extraction* [7], [8], *table extraction* [9], and *citation extraction* [10].

[*] Work done while interning at HP Labs China.

The key technical intuition underlying CRFlex is that consecutive snapshots of a text corpus often differ by only a small amount. For example, suppose that a snapshot contains the text fragment *"We will meet in CS 105 at 3 pm"*, from which we have extracted *"CS 105"* as a room number. Then under certain conditions (see Section III), if a subsequent snapshot also contains this text fragment, we can immediately conclude that *"CS 105"* is a room number, without having to rerun (often expensive) IE programs.

To maximize our opportunities to reuse (or recycle) past results, CRFlex opens up the standard statistical information extraction pipeline to exploit opportunities for reuse at each stage. A statistical information extraction pipeline has three phases: (1) *feature extraction*, (2) *model construction*, and (3) *inference*. In the first phase, *feature extraction*, the document is mapped into a high dimensional vector space called the *feature space*. In a CRF model, there is one feature vector associated with each token in the input document. In the second phase, *model construction*, a structure is built that captures the correlations between the various vectors. Finally, inference is performed on this structure. We find that for some programs, each of these phases can be the bottleneck. So it is critical that CRFlex exploits recycling opportunities at each stage of this pipeline. Our first challenge is how to identify opportunities within the statistical IE pipeline which allow CRFlex to *correctly* recycle inference results when only a small portion of the input text changes, i.e., to produce the exact same results as if the program were run from scratch.

The text corpus to which IE is applied may contain tens of thousands or millions of data pages. So our second challenge is how to develop an efficient recycling solution in the presence of a large amount of disk-resident data. We show that there is a fundamental tradeoff: the more intermediate data that CRFlex saves, the more opportunities CRFlex has for later recycling to improve the runtime. On the other hand, the I/O cost of reading and writing intermediate results may dominate the potential savings. This motivates us to design a spectrum of recycling solutions that range from capturing no results for a particular phase to progressively capture more intermediate results.

Finally, none of these recycling solutions is always optimal. For example, if a CRF-based IE program runs very slowly, the recycling solution that captures many intermediate results and enables many recycling opportunities may be the fastest way to execute the IE program. On the other hand, if a CRF-based IE programs relies on only inexpensive features then capturing too many intermediate results may incur so much overhead that re-running the IE from scratch is even faster. To address this problem, we develop a cost model to select the fastest recycling solution for a given IE program and corpus.

We conduct extensive experiments over several IE programs on 16 snapshots from Wikipedia. Our experiment results show that CRFlex can cut the runtime of re-extraction from scratch by as much as 90%.

In summary, we make the following contributions.

- We establish that it is possible to exploit work done by previous IE runs to significantly speed up CRF-based IE programs over evolving text. As far as we know, CRFlex is the first solution to this important problem.
- We show how to model certain properties of CRF inference algorithms, and how to exploit these properties to recycle the past IE and to guarantee the correctness of our approach.
- We show how to develop a spectrum of recycling solutions, trading off I/O cost spent on reading and writing intermediate results and CPU cost saved from recycling those intermediate results.
- We conduct extensive experiments with several CRF-based IE programs over a real-world data set to demonstrate the utility of our approach. We show in particular that CRFlex can cut the runtime of re-extraction from scratch by as much as 90%.

A. Related Work

Information Extraction: The problem of information extraction has received much attention (see [11], [12], [13] for recent tutorials). Much of the initial work focuses on improving the accuracy and runtime of individual extractors [14]. Recent work has also considered how to combine and manage such extractors in large-scale IE applications [12], [13]. Our work fits into this emerging direction. Optimizing IE programs and developing IE-centric cost models have also been considered in several recent papers [15], [16], [17]. These efforts however have considered only static corpora, not dynamic ones as we do in this paper.

Evolving Text: In terms of IE over evolving text, we have previously developed Cyclex [18] and Delex [19]. Both recycle previous IE efforts to improve extraction time. Cyclex recycles for a single IE blackbox and Delex recycles for multiple IE blackboxes. Those blackboxes are rule-based IE blackboxes, not the statistical programs that we consider here. As we describe, these rule-based solutions ignore many recycling opportunities. Other work (e.g. [20]) considers evolving text data, but not for IE programs: their focus is on incrementally updating an inverted index, as the indexed Web pages change.

CRFs: CRF-based IE has received much attention recently. Many have considered how to improve extraction accuracy of CRF-based IE programs [8]. More recent work has consider how to push CRF inference into RDBMS, and then exploit RDBMS to improve extraction time [21], [22], [23]. However, these approaches only consider optimizing CRF-based IE over static text corpora, not over dynamic corpora.

II. Background

CRFlex considers an application that requires repeatedly crawling a set of data sources to retrieve *data pages* (e.g., Web pages). We refer to the set of data pages retrieved at time i as S_i for $i = 1, 2, \ldots,$ as the *i-th snapshot* of the text corpus. A page at a fixed location (e.g., a URL) may change across snapshots, and the central goal of CRFlex is to reduce the repeated time spent on CRF inference by exploiting a simple observation: *many pages only change by a small amount between successive snapshots.*

d: Tom Cruise was born in NYC.

	x_1	x_2	x_3	x_4	x_5	x_6
x:	Tom	Cruise	was	born	in	NYC

	y_1	y_2	y_3	y_4	y_5	y_6
y:	P	P	O	O	O	L

	EntityName	**EntityType**
M:	Tom Cruise	PERSON
	NYC	LOCATION

Fig. 1. An example of using CRFs to extract named entities.

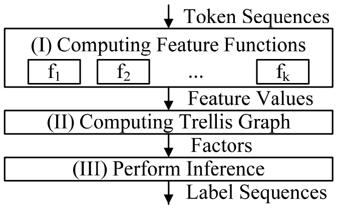


Fig. 2. CRF inference workflow.

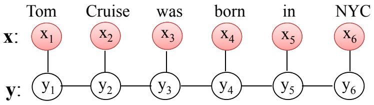


Fig. 3. A 6-token sequence CRF.

The goal of CRFlex is to extract a target relation R from all data pages in each snapshot. Let d be a data page. A *mention* of a relation $R(a_1, \ldots, a_n)$ on a page d is a tuple $(m_1, \ldots, m_n)$ such that for $i = 1, \ldots, n$, m_i is a string in page d that provides a value for a_i or *nil*. For example, consider a relation MEETING(room,time). Suppose a data page d is "We will meet in CS 105 at 3 pm". Then ("CS 105","3 pm") is a mention of MEETING relation, where "CS 105" and "3 pm" are mentions of attributes room and time, respectively.

CRF-based IE Programs: Given a set of data pages in a snapshot, CRF-based IE programs extract mentions from *each data page in isolation*. Since such per-page extraction programs are pervasive, we start with such extraction programs. We leave those programs extracting across multiple pages for future work.

To extract mentions from each data page, CRF-based IE programs transform information extraction to a sequence labeling problem. Given as input a sequence of tokens $\mathbf{x} = x_1 \ldots x_T$ and a set of labels, CRFs are probabilistic models that label each token with one label from a set of labels, denoted $\mathcal{Y}$. We denote the label of x_i by y_i for $i = 1, \ldots, T$. Intuitively, the set of labels, $\mathcal{Y}$, contains the set of entity types to be extracted and a special label *"other"* for tokens that do not belong to any of the entity types.

Example 1. *Fig. 1 illustrates an example of the input and output of a CRF that is used to extract PERSON (P) and LOCATION (L) entities form a page. A CRF-based IE program $\mathcal{P}$ first segments d into a sequence $\boldsymbol{x}$ of tokens. Here, $\mathcal{P}$ picks the most likely labeling for the sequence according to the CRF's probability model (a so-called Maximum Likelihood estimate). Finally, to form the relation, $\mathcal{P}$ outputs a set of name mentions M, where each mention consists of the longest subsequence of tokens with the same labels P or L.*

CRF Inference Workflows: The goal of CRFlex is to exploit recycling opportunities within a single page. The inference step of CRFs is often very expensive even in much larger IE programs, as has been reported in previous work such as the Stanford Named Entity Recognition System [7]. Our experiments also confirm this. Therefore, CRFlex focuses on the CRF inference.

We observe that the CRF inference in many IE applications, including several popular open source CRF packages [24], [25] can be modeled as a workflow which conceptually consists of 3 main steps: (I) feature function computation, (II) computing the trellis graph, and (III) labeling the resulting sequence. Fig. 2 illustrates such CRF inference workflow. we now describe each step in more detail.

(I) Computing Feature Functions. Given a token sequence **x**, a CRF inference workflow outputs the label sequence **y** of the token sequence. The first step is to extract essential properties from the document. To do so, a user first defines a set of *feature functions* $\{f_k\}_{k=1}^K$. In turn, these feature functions are applied to every token. Such feature functions may capture both local features of the document, e.g., whether or not a word is capitalized, and global features of the document, e.g., whether or not the word *"born"* is present anywhere in the text of the page. Additionally, the feature functions may express correlations between labels. In principle, a CRF's feature functions may describe correlations between every label of the a document. However, in the most popular *linear-chain* CRFs used in IE, the feature functions are restricted to correlating the previous y_i with label y_{i-1}, the previous label in the token sequence **x**. Thus, we may denote the set of feature functions as $\{f_k(y_{i-1}, y_i, \mathbf{x}, i)\}_{k=1}^K$. We illustrate by example:

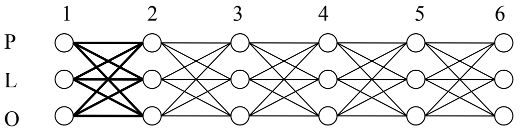
Example 2. *Fig. 3 illustrates a linear CRF model over the 6-token sequence. 2 possible feature functions can be:*

$$\begin{aligned} f_1(y_{i-1}, y_i, \boldsymbol{x}, i) &= [x_i \textit{ starts with a capitalized character}] \cdot \\ &\quad [y_{i-1} = PERSON] \cdot [y_i = PERSON], \\ f_2(y_{i-1}, y_i, \boldsymbol{x}, i) &= [\textit{ "in" appears in the 5-token window} \\ &\quad x_{i-5}x_{i-4}x_{i-3}x_{i-2}x_{i-1}] \\ &\quad \cdot [y_i = LOCATION], \end{aligned}$$

where $[p] = 1$ if the predicate p is true and 0 otherwise.

Given a token sequence **x**, the inference workflow first employs the feature functions $\{f_k(y_{i-1}, y_i, \mathbf{x}, i)\}_{k=1}^K$ to compute the feature values at each position i in the token sequence. This is the first step of inference indicated as (I) in Fig. 2. When the corpus changes, many of the feature functions may need to be recomputed from scratch (as they depend on the text in potentially complex ways). The feature functions may themselves be the output of other CRFs. However, an important special case is when the feature functions are traditional rule-based IE programs. In this case, we can leverage work on incremental execution of rule-based IE programs (we describe the relevant work below).

(II) Computing the Trellis Graph. During the offline training process, each function f_k is associated with a real-valued



(a) Trellis graph

λ: (5,3,4,5,4)

(b) Weights of feature functions

v_2: (1,0,0,0,0)

(c) Feature values at position 2

y_1 \ y_2	P	L	O
P	$1\bullet\lambda_1=5$	$0\bullet\lambda_2+0\bullet\lambda_4=0$	$0\bullet\lambda_5=0$
L	$1\bullet\lambda_3=4$	$0\bullet\lambda_2+0\bullet\lambda_4=0$	$0\bullet\lambda_5=0$
O	$1\bullet\lambda_3=4$	$0\bullet\lambda_2+0\bullet\lambda_4=0$	$0\bullet\lambda_5=0$

(d) Factors at position 2

Fig. 4. An example of computing factors.

weight λ_k that indicates roughly how indicative that feature is of the class it is trying to learn.[1] In the second step, the inference workflow then computes factors $\phi(y_{i-1}, y_i, \mathbf{x}, i)$ which are the dot product between the feature values and the corresponding feature weights. The factors represent the correlations between **x**, the current label y_i and the previous label y_{i-1}. Formally,

$$\phi(y_{i-1}, y_i, \mathbf{x}, i) = \sum_{k=1}^{K} \lambda_k \cdot f_k(y_{i-1}, y_i, \mathbf{x}, i) \qquad (1)$$

One can visualize the factors in the famous *Trellis Graph* which is shown in Fig. 4.(a). In this graph, each row corresponds to a label and each column corresponds to a position. All correlations are confined to cells in adjacent columns and so we visualize the factors ϕ on the edges between such cells.

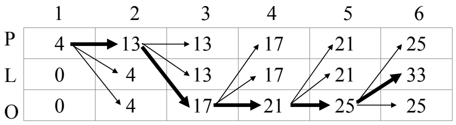
Example 3. *The factors on the highlighted edges in Fig. 4.(a) are computed as follows. Fig. 4.(b) illustrates the 5 λs associated with the 5 feature functions, where f_1 and f_2 are illustrated in Example 2. Given the feature value vector at position 2, which is illustrated in Fig. 4.(c), Fig. 4.(d) illustrates how to compute the factors at position 2. For example, the factor in the first row and first column corresponds to $\phi(P, P, \boldsymbol{x}, 2)$, which is the factor when y_1 is "P" and y_2 is also "P". Since only feature f_1 is triggered for such pair of labels, $\phi(P, P, \boldsymbol{x}, 2)$ is computed as the dot product between the value of f_1 and the λ associated with f_1.*

The reason we compute the factors is that for a given token sequence **x** the factors define a probability distribution over all label sequences. Following previous work [26], we formally define the distribution as follows.

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} \exp\left\{\sum_{i=1}^{T} \phi(y_{i-1}, y_i, \mathbf{x}, i)\right\} \qquad (2)$$

where $Z(\mathbf{x})$ is a normalizing function that guarantees the probability is between 0 and 1.[2]

In step (II), we observe that we may be able to exploit and recycle portion of the factors when the feature value vector of a position remains the same. Thus, there are opportunities for recycling in both of the first two steps.



	1	2	3	4	5	6
P	4	13	13	17	21	25
L	0	4	13	17	21	33
O	0	4	17	21	25	25

Fig. 5. An example of ML inference using Viterbi.

(III) Performing Inference. The last step of the workflow is to perform inference over the Trellis graph. There are two main types of inference: *Marginal Inference* where we try to find the marginal probability of labels, or *Maximum Likelihood (ML) inference* where the goal is to find the labeling sequence **y** that maximizes $p(\mathbf{y}|\mathbf{x})$. In this work, we focus on ML inference, because it is easiest to use by downstream applications.

To perform ML inference, we can use the famous dynamic programming algorithm called Viterbi [27]. It operates in two phases: forward phase and backward phase. In the forward phase, it computes a two dimensional V matrix. Each cell (y, i) of V stores the best partial labeling score of $x_1...x_i$ with the i^{th} token labeled y. The Viterbi algorithm computes scores recursively as follows:

$$V(y,i) = \begin{cases} \max_{y'}\{V(y', i-1) + \phi(y', y, \mathbf{x}, i)\} & \text{if } i > 0 \\ 0 & \text{if } i = 0 \end{cases} \qquad (3)$$

While computing scores, it also keeps track of which previous label y' is used to compute the current score in $V(y, i)$ by adding an edge from cell $V(y', i-1)$ to cell $V(y, i)$. In the end of the forward phase, it fills in all cells of the V matrix and adds all edges indicating which previous labels are used to compute the scores. Then $\mathbf{y}^*$ corresponds to the path traced from the cell that stores $max_y V(y, T)$. In the backward phase, the Viterbi algorithm backtracks by following the edges added in the forward phase to restore $\mathbf{y}^*$.

Example 4. *Fig. 5 illustrates the V matrix computed over the token sequence **x** in Fig. 3. The first row, second row, and third row contain the scores of label P, L, and O respectively. Each column contains the scores for a given position. We also plot all the edges that keep track of which previous labels are used to compute the scores. For example, the edge $V(P, 1) \rightarrow V(L, 2)$ indicates the score in $V(P, 1)$ is used to compute the score in $V(L, 2)$. Finally, the path of the best labeling is highlighted in bold.*

The running time of the Viterbi algorithm is $O(T|\mathcal{Y}|^2)$, where T is the length of the input token sequence and $|\mathcal{Y}|$ is the size of $\mathcal{Y}$.

Background on Multiple Snapshots for Rule-based IE: As we noted above, many feature functions employed in the workflow are similar to traditional rule-based extractors: it takes input as text fragments and outputs a value. Previous work [18], [19] have developed techniques to incrementally produce the results of such extractors when only a small portion of input text fragments changes. They define several properties to localize the extraction of a given extractor. We recall one such property called *context*. Specifically, a feature function f has *context* β iff its output at position i depends only on the small "context windows" of size β to both sides of i. Formally,

[1] We assume the model is learned via an offline process and used repeatedly.

[2] Explicitly, we can write $Z(x) = \sum_{\mathbf{y}} \exp\left\{\sum_{i=1}^{T} \phi(y_{i-1}, y_i, \mathbf{x}, i)\right\}$.

Definition 1 (context). *We say f has* context β *iff for any y_{i-1}, y_i, $\boldsymbol{x}$, and i, $f(y_{i-1}, y_i, \boldsymbol{x}, i) = f(y_{i-1}, y_i, \boldsymbol{x}', j)$, where $\boldsymbol{x}'$ is obtained by perturbing the tokens of $\boldsymbol{x}$ outside $x_{i-\beta}...x_{i+\beta}$ and j is the corresponding position of i in $\boldsymbol{x}'$.*

More details of context can be found in [18].

Therefore, we can apply these techniques to incrementally produce the output of all feature functions, and then re-run the final two stages of the workflow. However, such approach may not be optimal for two reasons. First, in order to recycle the results, the above work requires that we store all output of all feature functions at each snapshot. However, many CRFs often employ hundreds of and even thousands of feature functions [26]. Capturing and storing all their output incurs significant I/O overheads (as we show in our experiments). Second, this approach still requires computing trellis graph and conducting inference from scratch. While incrementally computing trellis graph (mainly conducting dot product) may not be very hard, incrementally conducting the Viterbi inference is challenging. This is because the most likely path is correlated with the entire token sequences (see Equation 3). To address the these limitations, the goal in CRFlex is to make each part of the above workflow incremental.

Problem Definition and the CRFlex Approach: Given a sequence of snapshots, our goal is to compute the most likely labels for the CRF applied to the latest snapshot as efficiently as possible. A straightforward solution is to re-run the entire workflow on the corpus from scratch. This solution is simple, however is often very costly as the workflow often incurs significant runtime. This suggests that we should consider how to *incrementally* produce the latest labels by recycling the labels from the previous snapshots. Of course, it may not be correct to recycle all labels and so our goal is to recycle as many labels as possible.

Our strategy in CRFlex is to develop algorithms that are incremental at each step of the workflow. These incremental algorithms will need to materialize state, which poses a challenge as there is a trade off between the time spent on materializing intermediate state and the amount of time it saves for later execution. Our experiments demonstrate that no single approach for recycling is always optimal, and thus we develop a cost model to choose between approaches. In the following sections, we first discuss the detailed opportunities for recycling employed by CRFlex (section III). We then discuss how to exploit these opportunities to design recycling solutions that CRFlex considers, and CRFlex's optimizer picks the fastest recycling solution using a cost model (Section IV).

III. Opportunities for Recycling

In this section, we will discuss how to *maximize* recycling opportunities in each of the 3 steps of CRF inference workflows.

Scope of Recycling from the Past: Let X_{N+1} and X_N be the set of token sequences on snapshot S_{N+1} an S_N respectively. As discussed earlier, to recycle, we must match each token sequence $\mathbf{x} \in X_{N+1}$ with token sequences in the past snapshots, to find matching regions. Many such matching schemes exist. Currently, we match each token sequence $\mathbf{x}$ from page p only with token sequence $\mathbf{x}'$ from page q at the same URL as p. (If q does not exist then we declare $\mathbf{x}$ to have no overlapping regions.) This simplification is based on the observation that pages with the same URL often change relatively slowly across consecutive snapshots, and hence often share much overlapping data. We will consider how to extend our approach to more general matching schemes in the future.

Given this matching scheme, in the following subsections, we discuss at each step (a) the incremental properties we exploit to guarantee the correctness of recycling, (b) what results we should capture on S_N and how to capture and store them efficiently, and (c) how to recycle the captured results on S_{N+1}.

A. Maximizing Recycling Opportunities in Computing Feature Functions

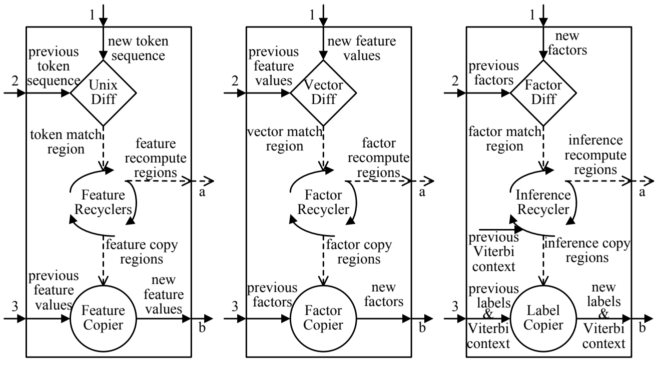
Let V_N and V_{N+1} be the feature values of X_N and X_{N+1} respectively. Our goal is to incrementally compute V_{N+1}. The basic idea is to *match* a token sequence $\mathbf{x} \in X_{N+1}$ with a token sequence $\mathbf{x}' \in X_N$ to find the *matching token regions* between $\mathbf{x}$ and $\mathbf{x}'$. Then we can recycle (i.e. copy) the feature values computed from those matching regions, instead of invoking the feature functions over these regions.

Incremental Property of Computing Feature Functions: It is important to notice that it is *not* correct to copy all feature values in the matching token regions. Suppose $\mathbf{x}$ ="Tom Cruise was from NYC" and $\mathbf{x}'$ ="Tom Cruise was born in NYC". Even though "NYC" is a matching token between $\mathbf{x}$ and $\mathbf{x}'$, copying all feature values at this position may lead to errors. Consider the feature function f_2 in Example 2. Its value at "NYC" in $\mathbf{x}'$ is 1, whereas its value at "NYC" in $\mathbf{x}$ is 0. So copying its value in $\mathbf{x}'$ to that in $\mathbf{x}$ leads to an error.

To address this problem, we exploit the context of a feature function (cf Section II) to guarantee that we correctly copy feature values. Given matching token regions, we can exploit the context of each feature function to identify its *copy regions* and *re-computation regions*. Copy regions are regions where we can *safely* copy feature values, and re-computation regions are regions where we must re-apply the feature function. Please refer to previous work [18] on how to exploit context to identify copy and re-computation regions.

Capturing and Storing Feature Vectors: To maximize the recycling opportunities, we need to capture both all input, X_N, and all output, V_N, while we were computing feature functions over X_N. Each token sequence $\mathbf{x}$ in X_N is stored as a sequence of integer triples, one for each token in $\mathbf{x}$. The integers in a triple indicate the ID of the data page where $\mathbf{x}$ is located, and the start and end position of the token in that data page.

Efficiently storing V_N poses a challenge. Because of their great flexibility to include a wide variety of features, CRFs often employ very large feature sets, with millions of features [26]. Therefore, the size of V_N is often very large, and storing it in a straightforward way incurs large I/O



(a) Step I

(b) Step II

(c) Step III

Fig. 6. Recycling workflows at each step.

overheads. To address this problem, we develop the following two techniques to reduce the size of V_N without losing any information.

1. Storing Sparse Feature Vectors. We observe that although the number of features employed by a CRF inference workflow may be large, at a certain position, most feature values are zeros and only a small number of features have non-zero values. Therefore, the feature vectors, which consist of all feature values at each position, are very sparse. We can store such sparse feature vectors as (feature id, feature value) pairs for only those non-zero feature values.

2. Storing Feature Groups. We also observe that many CRFs employ features instantiated from the same "templates". We call such features *feature groups*. For example, in Example 2, the predicate ["in" appears in the 5-token window $x_{i-4}x_{i-3}x_{i-2}x_{i-1}x_i$] is a template. Then we can define a set of features instantiated from this template, one for each possible label. Feature f_2 in Example 2 is the feature instantiated for label "LOCATION".

When we evaluate the feature groups at a position, their values are the same (i.e. are either all 0 or all 1). Therefore, we only need to store one value for such feature groups.

Recycling Captured Feature Values: Fig. 6.(a) illustrates the feature value recycling workflow. Given a token sequence $\mathbf{x} \in X_{N+1}$, we first match it with a token sequence $\mathbf{x}' \in X_N$. There are many possible ways to match two token sequences. As the first step, we use "Unix Diff" in CRFlex, and leave other possible matching algorithms in future work.

Not all feature values remain the same in the token matching regions. So in the next step, the *feature recyclers* exploits the context of each feature function to identify *copy regions* and *re-computation regions* (on edge *a* in Fig. 6.(a)) for each feature function. Then we copy from V_N at those copy regions, and re-apply the feature functions at the re-computation regions. This results in V_{N+1}.

B. Maximizing Recycling Opportunities in Computing Trellis Graph

Let F_N and F_{N+1} be the factors over X_N and X_{N+1} respectively. Our goal is to incrementally compute F_{N+1}.

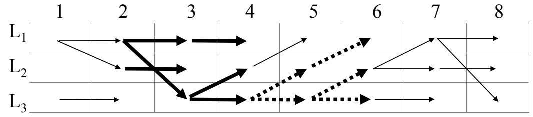


Fig. 7. An illustration of contexts.

Incremental Property of Computing Trellis Graph: Similar to recycling feature values, we first identify *feature vector matching regions*, where feature vectors remain the same at those regions. By the definition of factors (see Equation 1), we can see that if the feature vector at a position i remains the same (which implies that all the feature values at i remain the same), the factors at i remain the same. We exploit this fact to guarantee the factors recycled are correct.

Capturing and Storing Factors: To maximize recycling opportunity, we must capture both the input of this step, V_N, and the output of this step, F_N. V_N is captured and stored as we discussed in Section III-A. As F_N is often linear with the length of token sequences, we can store factors in a straightforward way.

Recycling Captured Factors: Fig. 6.(b) illustrates the factor recycling workflow. The "vector diff" first matches the feature vectors computed from a token sequence $\mathbf{x} \in X_{N+1}$ with those computed from $\mathbf{x}' \in X_N$ to identify feature vector matching regions.

After the vector matching regions are identified, the factor recycler then determines the *factor copy regions* and *factor re-computation regions*. This step is very trivial, as the factor copy regions are exactly the feature vector matching regions, and the factor re-computation regions are the remaining regions. Finally, the copier copies factors from F_N and trellis graph computation is invoked at the re-computation regions.

C. Maximizing Recycling Opportunities in Performing ML Inference

Let Y_N and Y_{N+1} be the labels over X_N and X_{N+1} respectively. Our goal is to incrementally compute Y_{N+1}.

Incremental Properties of Viterbi: To incrementally compute the labels of a token sequence when some factors of the token sequence have changed, We identify a property of Viterbi algorithm, called *Viterbi context*. The Viterbi context specifies small windows surrounding each position i such that, the factors outside those windows are irrelevant for Viterbi to compute y_i^* (the most likely label computed by Viterbi at position i). We first introduce *right context* and *left context*. Then build on them to introduce Viterbi context.

To motivate right contexts, we observe that given the factors from position 1 to position i, the factors of positions far away *after* i are irrelevant to y_i^*, as illustrated by the following example:

Example 5. *Fig. 7 illustrates the paths generated by Viterbi over an eight-token sequence and three possible labels* L_1*,* L_2*, and* L_3*. Notice that cell* $V(L_3, 4)$ *can reach all cells in column 6 by following the paths in dots between column 4 and 6. Since the path of* $\mathbf{y}^*$ *must contain one of those 3 cells in* *column 4, no matter what factors follows position 3, the path of* $\mathbf{y}^*$ *must go through cell* $V(L_3, 4)$*. Therefore, no matter how we perturb the factors after position 6,* $y_4^* = L_3$*. We call the factors in the window* $[5...6]$ *(i.e. a window starting at position 5 and ending at position 6) the right context of* y_4^**.*

We formalize the notion of right context as follows:

Definition 2 (Right context)**.** *Let* Φ_i *denote the factor vector at position* i*, which consists of all factors associated with* i*. So the length of* Φ_i *is* $|\mathcal{Y}|^2$*. The right context of the best label* y_i^* *of* x_i *is the factor vectors within window* $[i+1...i+\nu]$*, denoted as* $\Phi_{i+1}...\Phi_{i+\nu}$*, such that* $i+\nu$ *is the first column of* V *where all cells can be reached by cell* $V(y_i^*, i)$ *through the paths computed by Viterbi.*

The nice property of right context is that the factors outside the right context are irrelevant for Viterbi to compute the label y_i^*. That is, if we perturb the factors after the right context of y_i^*, applying Viterbi to the resulting sequence of factors still produces the same label y_i^*.

To motivate left contexts, we observe that the factors far away *before* the position i also have little influence on y_i^*. Formally,

Definition 3 (Left context)**.** *The left context of the best label* y_i^* *of* x_i *is the factor vectors in window* $[i-\mu...i-1]$*, denoted as* $\Phi_{i-\mu}...\Phi_{i-1}$*, and the label* λ *for token* $x_{i-\mu}$*, such that cell* $V(\lambda, i-\mu)$ *can reach all cells in column* i *through the paths computed by Viterbi, and no cell in a column after* $i-\mu$ *can reach all cells in column* i*. We represent the left context as a tuple* $(\Phi_{i-\mu}...\Phi_{i-1}, \lambda)$*. Furthermore, we call* $\Phi_{i-\mu}...\Phi_{i-1}$ *the left context window of* y_i^**.*

We can show that if we perturb the factors before position i, as long as y_i^*'s left context remains the same, applying Viterbi to the resulting factors still produces the same label y_i^* for token x_i.

Example 6. *In Fig. 7,* $V(L_1, 2)$ *can reach all cells in column 4 by following the highlighted paths between column 2 and 4. Therefore, the left context of* y_4^* *is* $\Phi_2...\Phi_3$ *and label* L_1*.*

We now define Viterbi context. Intuitively, the Viterbi context of y_i^* consists of its left context and its right context. Formally:

Definition 4 (Viterbi context)**.** *The Viterbi context of the best label* y_i^* *of* x_i *is the factors* $\Phi_{i-\mu}...\Phi_{i+\nu}$ *and the label* λ*, such that* $\Phi_{i-\mu}...\Phi_{i-1}$ *is its left context window,* $\Phi_{i+1}...\Phi_{i+\nu}$ *is the right context window and* $y_{i-\mu}^* = \lambda$*. We denote the Viterbi context as a tuple* $(\Phi_{i-\mu}...\Phi_{i+\nu}, \lambda)$*.*

The nice property of Viterbi context is that no matter how we perturb the factors of the token sequence $\mathbf{x}$ outside the window $[i-\mu...i+\nu]$, as long as the label λ of $x_{i-\mu}$ can still reach all possible labels of x_i, applying Viterbi to the perturbed factors still produces the same label y_i^*.

Example 7. *From Example 5 and Example 6, we know that the left context of* y_4^* *is* $(\Phi_2...\Phi_3, L_1)$ *and the right context is* $\Phi_5...\Phi_6$*. Therefore, the Viterbi context of* y_4^* *is* $(\Phi_2...\Phi_6, L_1)$*.*

Capturing and Storing Viterbi Inference Results: To maximize the recycling opportunities, we must capture all input, the factors F_N, and output, Y_N. Furthermore, to guarantee the correctness of recycling, we also need the capturing all the Viterbi contexts of labels in Y_N.

The key step in identifying Viterbi contexts is to identify a cell in each column i of V that can reach all cells in a column after i. This can be done by checking the *reachability* of the cells following all the edges added by the Viterbi algorithm while it is computing the scores in the V matrix. The pseudocode of capturing Viterbi contexts is presented in the full version report [28].

Capturing Viterbi contexts incurs overhead of $\mathrm{O}(TD|\mathcal{Y}|)$ in time, where T is the length of $\mathbf{x}$ and D is the length of the longest right context. In our experiments, we found that D is generally 2-3 tokens.

We have discussed how to store factors in Section III-B. To store the Viterbi context of each y_i^*, we only store μ and ν, which are the lengths of the left and right context window of y_i^*. Finally, the labels output by Viterbi are also stored as integers.

Recycling Captured Results: We now describe how to incrementally produce Y_{N+1} when some factors change. The recycling workflow is illustrated in Fig. 6.(c).

Given the sequences of factor vectors of two token sequences $\mathbf{x}$ in X_{N+1} and $\mathbf{x}'$ in X_N respectively, the "factor diff" first matches the two sequences of factor vectors to identify the *factor matching regions*, where factor vectors remain the same.

Given the factor matching regions, the "inference recycler" then works with the "label copier" and Viterbi to generate the label sequence. An important difference between the recycling workflow in Fig. 6.(c) and those in Fig. 6.(a) and (b) is the Viterbi inference recycling workflow must interleave the processing of copy and re-computation. Specifically, the recycling workflow computes the best labels of $\mathbf{x}$ *sequentially* (by either copying or re-computing) from left to right in the following 3 steps.

1.Re-computing Viterbi Scores, Outputting Labels and Finding Contexts: Starting from the first position, the inference recycler first invokes a Viterbi-like algorithm, *incrementalViterbi*. Like Viterbi, incrementalViterbi computes the scores in the V matrix according to Equation 3. Unlike Viterbi, which outputs the best labels until it finishes computing *all* the scores in the V matrix, incrementalViterbi outputs the best labels for partial token sequence *as soon as it* can. Precisely, incrementalViterbi identifies the Viterbi contexts, as we described above, while computing the scores in V. It then outputs the best labels up to position i as soon as it determines the right context of y_i^*. Take the sequence in Fig. 7 for an example. incrementalViterbi outputs $y_4^* = L_3$ as soon as it finishes computing the scores in column 6 and detects the right context of y_4^* is $\Phi_5...\Phi_6$. In this way, we can exploit the best labels of the partial sequence to determine if the left contexts of subsequent positions in $\mathbf{x}$ remain the same.

2.Locating a Copy Region and Copying Labels: incrementalViterbi is invoked until it enters the first factor matching regions. Now the inference recycler exploits the Viterbi contexts to determine the *inference copy region* within the matching region. An inference copy region is a subsequence of positions in $\mathbf{x}$ where the most likely labeling is the same as its matching part in $\mathbf{x}'$. Exploiting the Viterbi contexts, we can find such copy regions by finding regions where the Viterbi contexts (including both left and right contexts) of all positions in $\mathbf{x}$ are the same as the Viterbi contexts of their matching positions in $\mathbf{x}'$.

3.Re-computing Viterbi Scores After a Copy Region: Let r be the right boundary of the last copy region. incrementalViterbi then resumes computing scores according to Equation 3 and finding Viterbi contexts starting from position $r+1$. However, we did not compute the scores when copying the label y_r^*. How can we compute the scores at position $r+1$ by Equation 3 without the scores at position r?

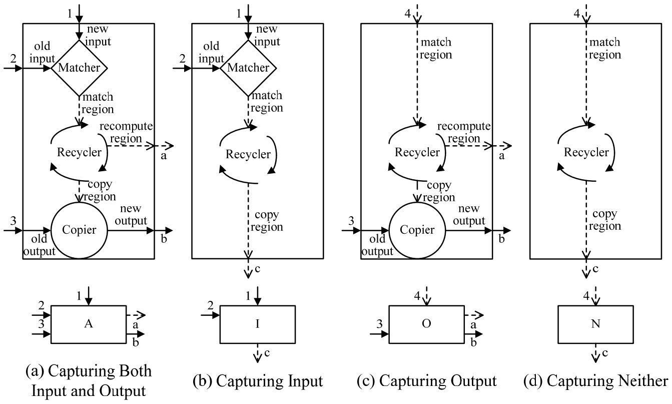
Our solution is based on the observation that we have obtained the most likely label y_r^* at position r by copying. This suggests the path of $\mathbf{y}^*$ must go through the cell $V(y_r^*, r)$. Therefore, we only need to consider the score of $V(y_r^*, r)$ to compute all scores at position $r+1$. Furthermore, we can show that we do not even need the actual score of $V(y_r^*, r)$ for the computation. Any dummy score of $V(y_r^*, r)$ can guarantee that we can obtain the correct best labeling. This suggests that we compute $V(y, r+1)$ for each $y \in \mathcal{Y}$ as $V(y, r+1) = c + \phi(y_r^*, y, \mathbf{x}, r+1)$, where c is the dummy score of $V(y_r^*, r)$. We can then compute the scores of tokens after x_{r+1} by using Equation 3.

The recycling workflow repeats step 1- 3 until it covers the entire sequence. The pseudo-code can be found in the full version report [28]. The following theorem states the correctness of the incremental version of Viterbi.

Theorem 1. *Let* $\boldsymbol{x}$ *be a token sequence, and* $\boldsymbol{\Phi}$ *be the factor sequence of* $\boldsymbol{x}$*. Let* $\mathbf{y}$ *be the label sequence output by Viterbi inference over* $\boldsymbol{\Phi}$*. The incremental version of Viterbi inference is correct in that it will output the exact* $\mathbf{y}$ *when it is applied to* $\boldsymbol{\Phi}$.

IV. Overall Recycling Solutions

In Section III, we have discussed the recycling opportunities at each step of the CRF workflow. We now discuss how to design the overall recycling solutions exploiting these recycling opportunities. It turns out there is a spectrum of recycling solutions, which trade their recycling opportunities with the recycling overheads. We will first discuss the plan space considered by CRFlex in Section IV-A and IV-B. As none of these recycling plans is always the fastest one, we present a cost-based solution in Section IV-C to select the fastest plan for a given corpus and CRF inference flow. Finally, we discuss how to execute the chosen recycling plan in Section IV-D.



(a) Capturing Both Input and Output (b) Capturing Input (c) Capturing Output (d) Capturing Neither

Fig. 8. Generic recycling blocks.

A. Defining Recycling Plans

One way to construct a recycling solution is to stitch together the local recycling workflows in Fig. 6. Although this recycling solution gains the maximal recycling opportunities, it is not always optimal, as it also incurs the maximal recycling overheads in capturing intermediate results.

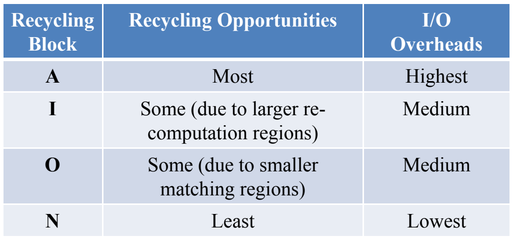
Therefore, we consider alternative recycling solutions which incur less overheads at the cost of losing some recycling opportunities. In this section, we first present 4 types of *generic recycling blocks* for local recycling. They differ in the number of intermediate results they capture. We then discuss how to compose an overall recycling solution using these generic recycling blocks in Section IV-B.

We observe that the local recycling workflows in Fig. 6 are very similar: they all consist of 3 operators, a matcher, a recycler, and a copier. These operators are recycling operators, in contrast to the actual computation operators. A *recycling block* encapsulates these recycling operators. It may contain all 3 or some of these recycling operators. We categorize recycling blocks into 4 types of *generic recycling blocks*, depending on the operators and their workflows encapsulated. We now introduce the 4 generic recycling blocks.

Capturing-All (A) Recycling Block: In sections III, to enable maximal recycling opportunities, we capture all input and output for each step in the CRF inference workflow, as showed in Fig. 6. These local recycling workflows can be treated as instantiations from the same generic recycling block, illustrated by Fig. 8.(a).

This generic recycling solution consists of all 3 recycling operators. As it captures both the input and output of a step in the CRF workflow, we call it "capturing-all" recycling block and denote it as *A* recycling block. The box in the bottom of Fig. 8.(a) illustrates the recycling block. In particular, all the arrows with numbers (i.e. arrow "1", "2" and "3") are input to A and all the arrows with characters (i.e. arrow "a" and "b") are output from A.

The A recycling block incurs the most I/O overheads among the 4 generic recycling blocks (in capturing both input and output). But it also gains the maximal recycling opportunities .



Recycling Block	Recycling Opportunities	I/O Overheads
A	Most	Highest
I	Some (due to larger re-computation regions)	Medium
O	Some (due to smaller matching regions)	Medium
N	Least	Lowest

Fig. 9. Recycling blocks comparison.

Capturing-Input (I) Recycling Block: Fig. 8.(b) illustrates a generic recycling block that only captures the input (e.g., a token sequence, feature vectors or factors) to a step in the CRF inference workflow. We call it *I* recycling block.

In contrast to the A recycling block, it does not capture the output of a step in the CRF inference workflow. Therefore, it does not need to copy any previous output, and thus does not include a copier (or the arrow "3" and "b"). Even though it does not copy anything, it still outputs the copy regions (on arrow "c") identified by the recycler, so that the downstream recycling blocks can exploit them (as we will discuss in the following paragraphs).

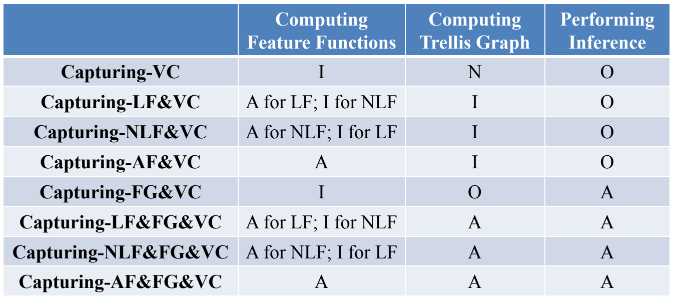
Another difference between I and A recycling blocks is how to determine the re-computation regions. In A recycling block, re-computation regions are solely determined by the recycler in the recycling block for the current step. In contrast, in B recycling block, the recycler of the current recycling block *cannot* determine the re-computation regions. This is because in CRF inference workflows, the re-computation regions of a downstream step often overlap with the copy regions of the current step. For example, the inference re-computation regions may overlap with the factor copy regions because the Viterbi contexts of some factor copy regions have changed.

As a result, for the re-computation of the downstream step at those regions, we need to obtain the output of the current step. For A recycling block, the output can be obtained by copying from the output captured at the last snapshot. For I recycling block, the output has to be obtained by re-computation. Hence, the re-computation regions of the current step is determined by the re-computation regions of the downstream steps.

In summary, compared with A recycling block, I recycling block incurs less I/O overheads in capturing the output. However, it also loses some recycling opportunities in that it may have to invoke re-computation even at some copy regions.

Capturing-Output (O) Recycling Block: Fig. 8.(c) illustrates a generic recycling block that only captures the output (e.g., feature vectors, factors or labels) of a step in the CRF inference workflow. We call it *O* recycling block.

In contrast to the A and I recycling blocks, it does not capture the input of a step in the CRF inference workflow. Therefore, it cannot use a matcher to identify matching regions. Instead, it uses the copy regions of the last upstream step (on arrow "4") as the matching regions, because the output from the last step (which is the input to the current step) remains the same in those regions.



	Computing Feature Functions	Computing Trellis Graph	Performing Inference
Capturing-VC	I	N	O
Capturing-LF&VC	A for LF; I for NLF	I	O
Capturing-NLF&VC	A for NLF; I for LF	I	O
Capturing-AF&VC	A	I	O
Capturing-FG&VC	I	O	A
Capturing-LF&FG&VC	A for LF; I for NLF	A	A
Capturing-NLF&FG&VC	A for NLF; I for LF	A	A
Capturing-AF&FG&VC	A	A	A

Fig. 10. Composing global recycling plans.

Compared with the A recycling block, O recycling block incurs less I/O overheads in capturing the input. However, it also loses some recycling opportunities in those matching regions not contained in the copy regions of an upstream step.

Capturing-None (N) Recycling Block: Fig. 8.(d) illustrates a generic recycling block which does not capture either the input or the output of a step in the CRF inference workflow. We call it "capturing-nothing", denoted as *N*, recycling block.

The processing of N recycling block is a mix of that of O and I recycling blocks. Like the O recycling block, it does not use a matcher to identify matching regions. Instead, it uses the copy regions from an upstream step. Like the I recycling block, it does not use a copier to copy any output. Furthermore, the re-computation regions are also determined by the re-computation regions of the downstream steps.

Compared with the A, I and O recycling blocks, N recycling block incurs the least I/O overheads. However, it also loses the most recycling opportunities in that it may have to invoke re-computation even at some copy regions and it cannot recycle in those matching regions not contained in the copy regions of the upstream step. Fig. 9 lists the comparison of the generic recycling blocks.

B. Composing Overall Recycling Plans

We can compose an overall recycling plan by choosing one of the 4 generic recycling blocks for each step in the CRF inference workflow and instantiating the chosen recycling blocks with the matcher, recycler and copier for the corresponding step. Furthermore, at the feature computation step, we can choose different recycling blocks for different types of features.

We distinguish two types of features when choosing recycling blocks: *local features* and *non-local features*. Local features are features whose context β is zero, and nonlocal features are those features whose context β is non-zero.

Given that we can choose from 4 generic recycling blocks for local feature computation, non-local feature computation, trellis graph computation and Viterbi inference, there are totally 16 possible recycling plans. The input/output dependency constraint further rules out 8 plans. The remaining 8 plans are listed in Fig. 10. These are the recycling plans considered by CRFlex. We now explain them in detail.

The capturing-VC recycling solution captures no intermediate results of the inference workflow except the Viterbi

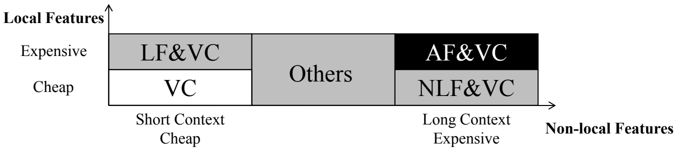


Fig. 11. Comparisons of recycling plans.

contexts. The next three recycling plans capture values of local features, non-local features and all features respectively, besides the Viterbi contexts. From Fig. 10, we can see that both capturing-LF&VC plan and capturing-NLF&VC use I recycling block for some feature computations.

The next three recycling plans in Fig. 10 capture the factors, among other intermediate results such as feature values and Viterbi contexts. The last recycling plan captures all intermediate results. This recycling plan is the one composed by stitching together the three local recycling flows in Fig. 6.

C. Cost-Based Selection of the Best Recycling Plan

We first discuss the pros and cons of various recycling solutions. Each of the recycling solution could be the fastest solution given different characteristics of the CRF workflow and corpus. We then present the cost model that captures these characteristics.

Pros and Cons of Various Recycling Solutions: The tradeoff between I/O cost and recycling opportunities can be roughly quantified by how expensive the CPU cost of computing a result is relative to the I/O cost needed to recycle the result. As such ratio is different for different CRF inference workflows, the plan space considered by CRFlex covers different ratios. In particular, we find that the plan space considered by CRFlex can be easily categorized by various characteristic of feature functions, as plotted in Fig. 11.

Like defining recycling plans, the plan space is determined by two types of feature functions: local features and non-local features. Both types of features are categorized as “cheap” and “expensive”. The features are cheap if the CPU cost of computing a feature value is on average smaller than the I/O cost of reading and writing a feature value, and are expensive otherwise.

For non-local features, the I/O cost of recycling a feature value is also determined by the contexts. The longer the contexts are, the less feature values we can safely recycle from a given length of text fragment, and thereby increasing the I/O overheads in recycling feature values.

Using the characteristics of local and non-local features, we can divide the plan space into 5 parts, as in Fig. 11, with 4 extreme cases in the 4 corners. The left bottom corner (in white) is where both local and non-local features are relatively cheap to compute in comparison to the I/O overheads. Therefore, we expect the capturing-VC plan, which incurs the least I/O overheads, is the fastest plan in this region. The left upper corner is where local features become expensive. Therefore, the plan captures local features, capturing-LF&VC plan, is expected to be the fastest.

On the other hand, the right upper corner (in black) is where both local and non-local features are relatively expensive to compute. Therefore, we expect capturing-AF&VC, which captures all feature values is the fastest plan in this region. The right bottom corner is where only non-local features are expensive. Therefore, we expect capturing-NLF&VC, the plan only captures non-local features, to be the fastest.

Cost Model: The cost model captures the above characteristics. It models the runtime time of each plan as the summation of the costs of the four computation components in a CRF inference workflow: the local feature computation, non-local feature computation, trellis graph computation and the Viterbi inference. Each component consists of both the CPU cost and I/O cost. For space reasons we do not discuss the cost model further, but the full version report [28] describes it in detail.

Selecting the Best Recycling Plan: As the first step, CRFlex assumes that the evolving rate of a corpus is constant. Many real-life corpora have such characteristics [18], [19]. Therefore, we only need to estimate the parameters using a small sample on each of the first k snapshots, for a pre-specified k. Then we evaluate the cost and pick the recycling plan with the smallest estimated cost. From snapshot $k+1$, we will execute the same recycling plan on all subsequent snapshots. For space reasons, we do not discuss parameter estimation further. Section V demonstrates empirically that a small sample size and k are sufficient for CRFlex, meaning that parameter estimation and cost-based plan selection adds very little overhead to the overall cost.

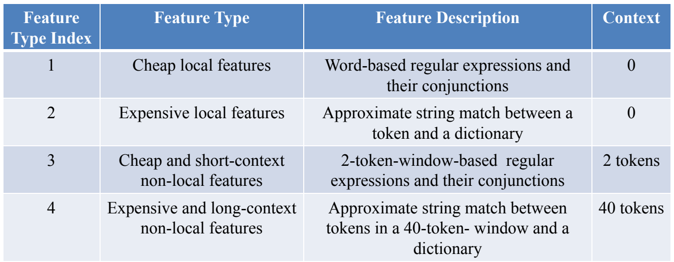
D. Executing A Selected Recycling Plan

There are two cases of executing a selected plan. The first case is at S_{k+1}, the first snapshot over which CRFlex executes the selected plan. When CRFlex re-ran the CRF workflow on S_k, it did not capture any intermediate results. Therefore, it cannot recycle at S_{k+1}, and has to re-run the entire CRF inference workflow. The second case is after S_{k+1}. Now CRFlex can recycle the results from the last snapshot. At the same time, it also produces intermediate results for recycling at the next snapshot. As the intermediate result file, R_{k+1}, is often very large and cannot fit into memory, we must guarantee that CRFlex will access it sequentially.

To do so, CRFlex processes data pages at each snapshot in the same order determined by their URLs. Specifically, let $q_1, q_2, \ldots, q_N$ be the order CRFlex processed the data pages at S_{k+1}. Therefore, the results in R_{k+1} were also stored in that order. Then at S_{k+2}, CRFlex processes the data pages *in the same order*. That is, let p_i be the page with same URL as $q_i, i = 1, \ldots, N$. Then CRFlex processes p_1, then p_2, and so on. (If a page $p \in S_{k+2}$ does not have a corresponding page in S_{k+1}, then we can process it at any time, by simply re-running the CRF inference workflow.) By processing in the same order, we only need to scan R_{k+1} sequentially once.

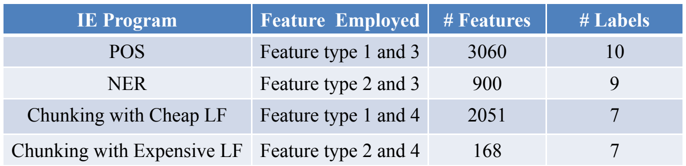
V. Empirical Evaluation

Dataset: We now empirically evaluate the utility of CRFlex. We chose a set of Wikipedia.com URLs in the entertainment domain, including Wiki articles of actors, actress, movies, etc. We then obtained 16 snapshots from these URLs, with 21 days



Feature Type Index	Feature Type	Feature Description	Context
1	Cheap local features	Word-based regular expressions and their conjunctions	0
2	Expensive local features	Approximate string match between a token and a dictionary	0
3	Cheap and short-context non-local features	2-token-window-based regular expressions and their conjunctions	2 tokens
4	Expensive and long-context non-local features	Approximate string match between tokens in a 40-token- window and a dictionary	40 tokens

Fig. 12. Feature description.



IE Program	Feature Employed	# Features	# Labels
POS	Feature type 1 and 3	3060	10
NER	Feature type 2 and 3	900	9
Chunking with Cheap LF	Feature type 1 and 4	2051	7
Chunking with Expensive LF	Feature type 2 and 4	168	7

Fig. 13. IE programs description.

between consecutive snapshots. On average, each snapshot contains 3038 data pages, and its total size is about 35M.

Features: To construct CRF-based IE programs for our experiments, we consider 4 types of feature functions. Fig. 12 lists the description of those features. Most of these features have been used in previous work. For example, the word-based regular expressions are default features in the CRF open source package [24]. Conjunctions of regular expressions have been considered in CRF-based Named Entity Recognition (NER) and Noun Phrase Segmentation [26]. Approximate-string-matching-based features have also been used in CRF-based NER [29].

IE Programs: Based on these 4 types of features, we construct 4 IE programs. Fig. 13 illustrates the statistics of these IE programs. Each IE program first uses the same tokenizer to segment a document into token sequences, and then employs a different CRF with different types of features chosen from the features described above to label the token sequences. The 4 IE programs represent the 4 extreme cases in the plan space illustrated in Fig. 11. All the IE programs were developed using the CRF open source package [24].

1. Part-of-speech (POS) Tagging: POS tagging is to identify tokens such as nouns, verbs, adjectives, and adverbs. There are totally 10 possible labels considered by the CRF. As this task is relatively easy, the IE program we construct employs the cheap local features and cheap non-local features.

2. NER: NER is to extract entities such as “Person”, “Location”, and “Organization” from documents. There are totally 9 possible labels considered by the CRF. Similar to previous work [29], this IE program employs expensive local features and cheap non-local features.

3&4. Chunking: Text chunking is to divide a document into syntactically correlated parts of words, such as noun phrase words and verb phrase words. There are totally 7 possible labels considered by the CRF. We consider two IE programs for this task. Both employs expensive non-local features with long contexts. The first one considers cheap local features, while the second one considers expensive local features.

Runtime Comparison: We considered 2 baselines: Rerun and Cyclex. Rerun re-executes IE programs over all pages in a snapshot. Cyclex is a rule-based recycling solution treating the entire IE programs as a blackbox to recycle. Fig. 14 plots the runtime curves of Rerun, Cyclex and CRFlex.

CRFlex used the first 2 snapshots to collect statistics for its cost model while re-running the IE programs from scratch. The statistics were collected over 150 data pages on each snapshot. So the runtime of CRFlex on these snapshots included both the runtime of re-running the programs and the runtime of collecting statistics. On the third snapshot, CRFlex still reran the programs. But it also captured intermediate results according to the picked recycling plan.

We observe that, on the first 3 snapshots, the runtime curves of Rerun Cyclex and CRFlex were very close, indicating a small overhead of collecting statistics and capturing results for CRFlex. From snapshot 3 and on, CRFlex performed significantly better than Rerun and Shortcut, cutting runtime by as much as 90%. These results suggest that CRFlex was able to exploit the properties specific to the CRF workflows to recycle more IE results, thereby significantly speeding up execution.

As CRFlex considered a set of recycling plans, we also executed all recycling plans from snapshot 3 to the last snapshot, and plotted the runtimes of three fastest recycling plans for each task in Fig. 15.

From Fig. 15, we observe that none of the recycling plan was always optimal. For the POS program which employed cheap local and non-local features, capturing-VC was the fastest plan. For the NER program which employed expensive local and cheap non-local features, capturing-LF&VC was the fastest plan. The two Chunking IE programs both employed expensive non-local features. For the one employing cheap local features, capturing-NLF&VC was the fastest plan, while for the one employing expensive local features, capturing-AF&VC was the fastest plan.

This underscores the importance for the CRFlex optimizer to consider various recycling plans and select the best for a particular setting. Furthermore, we also observe that the optimizer of CRFlex was able to pick the fastest plan in all cases, indicating the effectiveness of the optimizer in CRFlex.

Contributions of Components: Fig. 16 shows the runtime decomposition of the three fastest recycling plans and Rerun for each IE program (numbers in the figure were averaged over all snapshots). “Match” is the total CPU time of applying matchers. “Extraction” is the total CPU time on extraction, including the time used for running both the tokenizer and the CRF inference workflow. “IO” is the total time of reading and writing intermediate results produced by the CRF inference workflows. Finally, “Others” is the remaining time (e.g., to read and write data pages, and to load the CRF model into memory etc.)

The results showed that extraction time and IO time dominated runtimes of all recycling plans. Furthermore, the runtime of the CRFlex inference workflow also dominated the extraction time, taking more than 96% of extraction time.

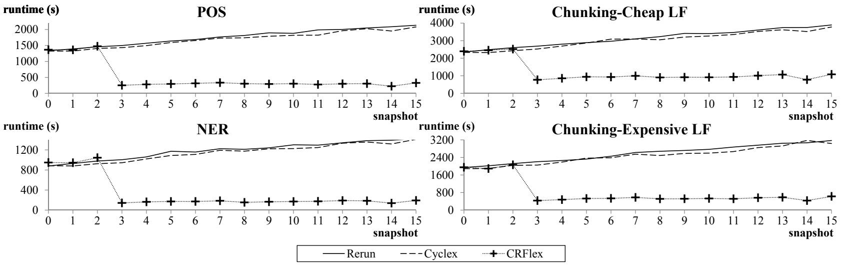


Fig. 14. Runtime of Rerun, Cyclex, and CRFlex.

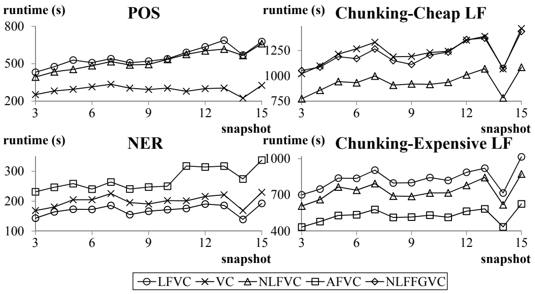


Fig. 15. Runtime comparison of top-3 plans.

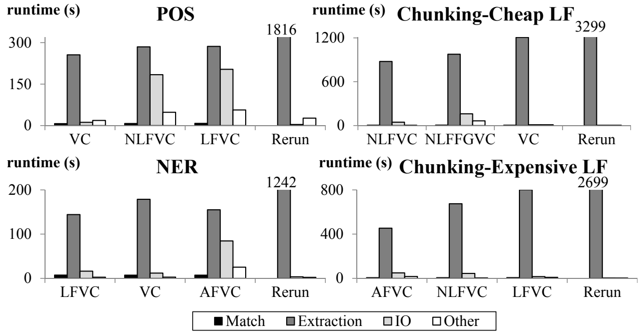


Fig. 16. Runtime decomposition.

This indicates that we should focus on optimizing these two components, as CRFlex does.

VI. Conclusions and Future Work

A growing number of real-world applications must deal with statistical IE over dynamic text corpora. Executing such IE programs from scratch at each snapshot is very time-consuming. To address this problem, we have developed CRFlex, a solution that efficiently executes CRF-based IE programs over evolving text by recycling previous inference results. As far as we know, CRFlex is the first in-depth solution for this important problem. CRFlex also opens up several interesting directions that we are planning to pursue, including (a) how to handle IE programs based on statistical models other than CRFs, and (b) how to handle evolving IE programs as well as evolving text.

Acknowledgments: We thank AnHai Doan and Raghu Ramakrishnan for their valuable comments on the early version of this work. Christopher Ré is generously supported by the Air Force Research Laboratory under prime contract No. FA8750-09-C-0181, Office of Naval Research Award under No. N000141210041, the NSF CAREER Award under IIS-1054009, the University of Wisconsin-Madison, and gifts or research awards from Google, LogicBlox, and Johnson Controls, Inc. Any opinions, findings, and conclusion or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of any of the above sponsors.

References

[1] F. Wu, R. Hoffmann, and D. S. Weld, “Information extraction from Wikipedia: moving down the long tail,” *SIGKDD-08*.

[2] Z. Nie, J.-R. Wen, and W.-Y. Ma, “Object-level vertical search,” *CIDR-07*.

[3] G. Kasneci, M. Ramanath, F. Suchanek, and G. Weikum, “The YAGO-NAGA approach to knowledge discovery,” *SIGMOD Record*, vol. 37, no. 4, 2008.

[4] P. DeRose, W. Shen, F. Chen, Y. Lee, D. Burdick, A. Doan, and R. Ramakrishnan, “DBLife: A community information management platform for the database research community (demo),” *CIDR-07*.

[5] M. Mathioudakis and N. Koudas, “TwitterMonitor : Trend detection over the twitter stream,” *SIGMOD-10*.

[6] M. Yang, H. Wang, L. Lim, and M. Wang, “Optimizing content freshness of relations extracted from web using keyword search,” *SIGMOD-10*.

[7] J. R. Finkel, T. Grenager, and C. Manning, “Incorporating non-local information into information extraction systems by gibbs sampling,” *ACL-05*.

[8] A. McCallum and W. Li, “Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons,” *CoNLL-03*.

[9] D. Pinto, A. McCallum, X. Wei, and W. B. Croft, “Table extraction using conditional random fields,” *SIGIR-03*.

[10] F. Peng and A. McCallum, “Accurate information extraction from research papers using conditional random fields,” in *HLT-NAACL-04*.

[11] W. Cohen and A. McCallum, “Information extraction from the world wide web(tutorial),” *KDD-03*.

[12] E. Agichtein and S. Sarawagi, “Scalable information extraction and integration (tutorial),” *KDD-06*.

[13] A. Doan, R. Ramakrishnan, and S. Vaithyanathan, “Managing information extraction: state of the art and research directions (tutorial),” *SIGMOD-06*.

[14] S. Sarawagi, “Information extraction,” *Foundations and Trends in Databases*, vol. 1, no. 3, pp. 261–377, 2008.

[15] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan, “Declarative information extraction using datalog with embedded extraction predicates,” *VLDB-07*.

[16] P. G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano, “To search or to crawl? Towards a query optimizer for text-centric tasks,” *SIGMOD-06*.

[17] A. Jain, A. Doan, and L. Gravano, “SQL queries over unstructured text batabases,” *ICDE-07*.

[18] F. Chen, B. J. Gao, A. Doan, J. Yang, and R. Ramakrishnan, “Optimizing complex extraction programs over evolving text data,” *SIGMOD-09*.

[19] F. Chen, A. Doan, J. Yang, and R. Ramakrishnan, “Efficient information extraction over evolving text data,” *ICDE-08*.

[20] L. Lim, M. Wang, J. Vitter, and R. Agarwal, “Dynamic maintenance of web indexes using landmarks,” *WWW-03*.

[21] M. Wick, A. McCallum, and G. Miklau, “Scalable probabilistic databases with factor graphs and MCMC,” *PVLDB-10*.

[22] D. Z. Wang, M. J. Franklin, M. N. Garofalakis, and J. M. Hellerstein, “Querying probabilistic information extraction,” *PVLDB*, vol. 3, 2010.

[23] D. Z. Wang, M. J. Franklin, M. Garofalakis, J. M. Hellerstein, and M. L. Wick, “Hybrid in-database inference for declarative information extraction,” *SIGMOD-11*.

[24] CRF, http://crf.sourceforge.net/.

[25] CRF++, http://crfpp.sourceforge.net/.

[26] A. McCallum, “Efficiently inducing features of conditional random fields,” *UAI-03*.

[27] G. D. Forney, “The Viterbi algorithm,” *IEEE*, vol. 61, 1973.

[28] F. Chen, X. Feng, C. Ré, and M. Wang, “Optimizing statistical information extraction programs over evolving text,” http://pages.cs.wisc.edu/~fchen/papers/crflex-tr.pdf.

[29] A. Chandel, P. Nagesh, and S. Sarawagi, “Efficient batch top-k search for dictionary-based entity recognition,” *ICDE-06*.