ABSTRACT

Title of dissertation:	EFFICIENT NON-DETERMINISTIC SEARCH IN STRUCTURED PREDICTION: A CASE STUDY ON SYNTACTIC PARSING
	Jiarong Jiang, Doctor of Philosophy, 2014
Dissertation directed by:	Professor Hal Daumé III Department of Computer Science

Non-determinism occurs naturally in many search-based machine learning and natural language processing (NLP) problems. For example, in parsing, the goal is to automatically construct the syntactic tree structure of a sentence given a grammar. Computationally, agenda-based parsing is a dynamic programming approach to find the most likely syntactic tree of a sentence according to a probabilistic grammar. A chart is used to maintain all the possible subtrees for different spans in the sentence and a queue (agenda) is used to rank all the candidates. The parser chooses only one candidate (called a partial tree or constituent) to build from the agenda per step and the rest of the candidates on the agenda will remain unchanged until/unless they are selected in later steps. Non-determinism occurs naturally in agenda-based parsing since it is very likely that the new constituent may not depend on the previous two consecutively built constituents. It can be built by combining items from a few steps earlier.

Unfortunately, like most other problems in NLP, the size of the search space in general is huge and exhaustive search is impossible. However, users expect a fast and accurate system. Two major strategies to speed up a system are prioritization and pruning. In this dissertation, I focus on the question of "Why, when, and how shall we take advantage of non-determinism?" and show the efficacy and application of a non-determinism agent (parser) to improve the parsing system in terms of speed and/or accuracy. Many approaches have been applied to make prioritization and pruning more efficient, such as search-based structured prediction methods, imitation learning methods as well as reinforcement learning methods. These all have different limitations when it comes to a large NLP system. The solution proposed in this dissertation is that "We should train the system non-deterministically and apply (test) it deterministically if possible." and I also show that "it is better to learn with oracles than simple heuristics if possible".

To support these major points, we start by solving a generic Markov Decision Process (MDP) with a novel non-deterministic agent. We show theoretical convergence guarantees with non-deterministic learning and verify the efficiency of different non-deterministic algorithms on maze solving problems. Then we focus on the application of agenda-based parsing. To show how to re-prioritize the parser, we model a decoding problem as a Markov Decision Process with a large state and action space. We discuss the advantages/disadvantages of existing techniques: (1) The algorithms have to be well initialized, otherwise the agent could wander in the wrong area of the search space. (2) The parsing trajectories are very long and only a few visited states contribute to the final parse tree. We propose a hybrid reinforcement/apprenticeship learning algorithm to tackle the problem of

initially bad policies and imbalanced examples in trading off speed and accuracy. As re-prioritization involves hard decisions to choose one out of all the items on the agenda, we consider a pruning strategy that restricts the action space to be only pruning and keeping. To show how to prune the parser and make it faster without losing too much on accuracy, we propose to use a dynamic pruner with features that depend on the run-time status of the chart and agenda (dynamic features) and analyze the importance of those features in the pruning classification. Our models show comparable results with respect to state-of-the-art strategies.

EFFICIENT NON-DETMINISTIC SEARCH IN STRUCTURED PREDICTION: A CASE STUDY ON SYNTACTIC PARSING

by

Jiarong Jiang

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2014

Dedication

To my parents and my husband for their continuous support and encouragement.

Acknowledgments

Over the past five and half years I have received help and support from a great number of people without whom my journey in graduate school would have been much more difficult. I owe my gratitude to all the people who have made this dissertation possible and because of whom I sincerely cherish the past five years as one of the most fulfilling spans in my life.

First of all, I would like to show my deepest gratitude to my advisor, Professor Hal Daumé III. I have been incredibly fortunate to have him as my advisor since my first year. I have learned to distinguish valuable ideas in research as well as concertize ideas to problems. His broad interests gave me large exposure to many different areas of machine learning and natural language processing. His patience in helping work out the details of a problem greatly helped my research and made the results more robust. It was truly a pleasure to work with and learn from him.

I am very grateful to Professor Jason Eisner for his practical advice and criticisms. While collaboration over the past three years, the meetings and discussions with him helped me build a better picture of the work as well as implementation details. Some of the projects discussed in this dissertation were conducted under his joint guidance.

I would also like to thank my dissertation committee members Professor Jordan Boyd-Graber, Professor David Jacobs and Professor Norbert Hornstein. They gave me a lot of invaluable advice on the dissertation.

I would like to express my gratitude for the help and support the department staff members, Ms. Fatima Bangura and Ms. Jennifer Story who make the graduation process and requirements clear and efficient. I would also thank Joe Webster and other UMIACS staff for helping me with all sorts of technical issues in the past.

Many friends have helped me in both research and life. I would like to take this opportunity to thank, Jagadeesh Jagarlamudi, Abhishek Kumar, He He, Snigdha Chartuverdi, Gregory Sanders, Yuening Hu, Ke Zhai, Ke Wu, etc. Through the many conversations about research, life and all else, my graduate school experience was immensely richer.

Most importantly, I would like to thank my parents and my husband Taesun Moon. Without their support and encouragement I would not have been able to make it so far in my studies and my career.

Finally, I would like to acknowledge the financial support from National Science Foundation under Grant No. 0964681, “Learned Dynamic Prioritization” that funded most of the research in this dissertation.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

NLP	Natural Language Processing
MDP	Markov Decision Process
MSMDP	Multi-state Markov Decision Process
SEARN	Search-based Structured Prediction
DAgger	Dataset Aggregation Algorithm
UCS	Uniform Cost Search
BP	Belief Propagation
RBP	Residual Belief Propagation

Chapter 1: Introduction

In this dissertation, I tackle the question of when and how to utilize non-determinism in structured prediction problems to improve speed and/or accuracy. Structured prediction is a prediction task: the goal of the task is to learn a prediction function (or model hypothesis) from some input to some output, but it is more general than predicting binary or class labels in that the expected predictions take on specific structures such as (labeled) chains, trees, lattices, etc. Many problems in natural language processing (part-of-speech tagging, parsing, machine translation), vision (articulated body post estimation) and bioinformatics (protein structure prediction) can be categorized as structured prediction problems.

For example, the input space for protein structure prediction [Baker and Sali, 2001] is a set of amino acid sequences and the output space is all the possible 3D structures that can be derived from the sequences. In the problem of natural language parsing[Earley, 1970, Manning and Schütze, 1999], the input space is natural language sentences and the output space is the set of parse trees–syntactic tree representations of sentences. Figure 1 shows an example of input to and output of a parser.

Non-determinism implies revisiting previously explored states so that mistakes

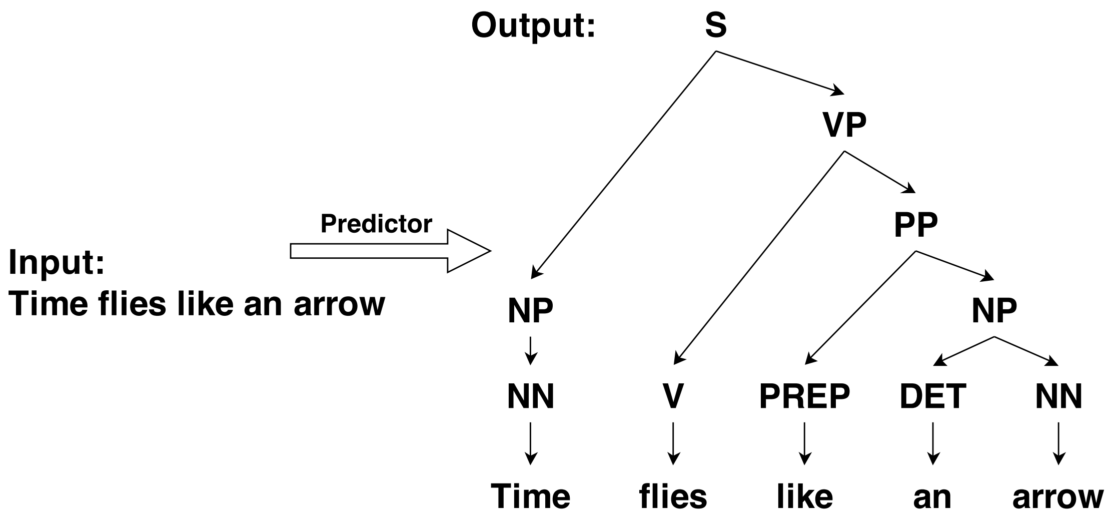

Figure 1.1: Example: Constituency parsing input and output.

made earlier have a chance to be corrected later. Depending on how complex the structures of the output should be, a non-deterministic predictor might be slow and the search space of the output might be too large to explore. In the parsing case, the search space is all the possible trees with each node associated with a tag. Enumeration over all the possibilities for long sentences is intractable. On the other hand, in the case of robot car driving, the goal is to let the robot learn to drive from each origin to destination safely. Under this circumstance, knowledge of more than the canonical ground truth will help. (e.g. if we not only know the solution, but also how to get to the solution for some input.) As said, if we only provide the origin and destination for the robot, it takes a long time for the robot to reach the destination or it might even fail to find it. However, if we provide some instruction on how to drive as well as some example route, it is easier for the robot to learn to generalize from those examples.

Though later chapters in the dissertation will focus on constituency parsing for using non-determinism, the algorithms/frameworks I propose are suitable for any structured prediction algorithm in general.

1.1 Structured Prediction

In machine learning, standard prediction problems (classification/regression) involve three components—an input space $\mathcal{X}$, output space $\mathcal{Y}$ and a mapping from input space to output space $f : \mathcal{X} \rightarrow \mathcal{Y}$. In classification, the output space can be binary $\mathcal{Y} = \{0, 1\}$ or multi-class $\mathcal{Y} = \{1, \ldots, k\}$ where k is the number of classes. For regression, in contrast, the output space is generally real valued $\mathcal{Y} = \mathbb{R}$. Structured prediction can be thought of as a type of classification problem where $\mathcal{Y}$ is a countable categorical variable except that the cardinality of $\mathcal{Y}$ is very large—possibly unbounded—and in general has a complex internal structure. Structured prediction problems naturally occur in the field of computer vision, bioinformatics, natural language processing (NLP), etc.

The quality of a solution (i.e. a predictive function or model hypothesis that generates output given input) to a structured prediction problem is often quantified in terms of loss functions which measure how far off a predicted output is from the true or gold output. For standard classification problems, this often takes the form of 0/1 (zero one) loss which is defined to be the number of instances where the prediction does not match the ground truth. For standard regression problems, a common loss function is squared loss[1] which is defined to be the sum of the squares

[1]equivalently L2 loss

of the deviation between the prediction and the truth.[2] Loss functions for structured prediction problems are more complicated but usually are defined such that the loss function over the structured output can decompose into 0/1 loss over its individual parts.

The traditional solutions to structured prediction problems are composed of model training (learning) and decoding (testing). For example in part-of-speech (POS) tagging, we are given a set of sentences and their POS tags {(He eats an apple, PR V DET NN), (My cat rides roomba, ADJ NN V NN), ...}. One way of learning those tags is to learn a predictor that predicts the tag for a word in the sentence from the word and the tag from the previous word, so the prediction of "PR V DET NN" for "He eats an apple" is decomposed into a sequence of sub-prediction problems: {(current word: `He` and previous POS: `None`, PR), (current word: `eats` and previous POS: `PR`, V), (current word: `an` and previous POS: `V`, DET), ...}. The loss functions are defined over two structured outputs. In this example, it can be 0/1 loss (if there exists one tag that does not match, the loss will be 1) or Hamming loss (where the loss is the number of mismatched tags). The model training part is to find a model hypothesis from input sentences to output tag sequences such that the expected loss is minimized. Once the mapping is found, we can use this mapping to assign tags to new input sentences.

Formally, a training set $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ is a set of input x_i, prediction y_i pairs where $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$, $i = 1, \ldots, n$. Each prediction y_i can be decomposed into a set of sub-predictions of size $\Gamma(|x_i|)$, a function of the num-

[2]for each prediction $\hat{y}_i$ and truth y_i, $i \in \{1, \ldots, n\}$, squared loss or L2 loss is $\sum_i (\hat{y}_i - y_i)^2$.

ber of variables for input x_i. A loss function is defined over a pair of outputs: $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$. Assume the training example (x, y) is drawn from an underlying distribution $\mathcal{P}$ (and the test set will be drawn from the same distribution). The task of learning is to find a mapping $f : \mathcal{X} \to \mathcal{Y}$ such that the expected loss

$$L(f) = \mathbb{E}_{(x,y)\sim\mathcal{P}} l(f(x), y) \tag{1.1}$$

is minimized. If a parameterized scoring function F_θ (such as log-likelihood under a conditional random field model [Lafferty et al., 2001a]) is defined, the goal of training is to find a parameter θ such that $F(y_1, \ldots, y_n | x_1, \ldots, x_n, \theta)$ is maximized. The decoding of an input x can be described as finding the structure $\hat{y}$ such that

$$\hat{y} = \arg\ max_{y \in \text{all possible structures}} F(y|x, \theta) \tag{1.2}$$

It is often computationally expensive to enumerate all possible output structures for a given input. For general problems with a complex structure, like the protein structure prediction problem mentioned before, decoding the secondary 3D structure exactly is NP-hard [Crescenzi et al., 1998]. With a simpler structure such as a chain or tree and K possible labels for each position in the structure, the exact solution is tractable by dynamic programming strategies, but it can still be in the order of $O(n^2)$ or $O(n^3)$ where n is the length of the input (e.g. number of words in a sentence). In sequence labeling for first order history models, the overall cost is K labels $\times$ chain length n for n words $\times O(n)$ cost for looking at other words/labels/other attributes in the sentence $= O(Kn^2)$. In natural language parsing with binary trees, the overall cost is K labels $\times$ tree size $O(n^2)$ for n words (leaves) $\times O(n)$ cost for looking at other words/labels/other attributes $= O(Kn^3)$. Though polynomial complexity in terms of n does not seem a hurdle given the length of natural language sentences, additional computational challenges arise when K is large as is usually the case in natural language where K is the size of the grammar. It is possible to reduce the computational cost ($O(n)$ for looking at other words/labels) of some structured prediction problems to independent sub-predictions or sub-predictions with bounded size—like the POS tagging example discussed earlier. However, for more complex problems or algorithms that work for structured prediction problems in general, this cost is inevitable.

1.2 Non-determinism in Structured Prediction

Many structured prediction algorithms are inherently non-deterministic. When we say "non-deterministic," we mean it in the sense of "non-deterministic Turing machine", "non-deterministic finite state machine," or "non-deterministic algorithm" *not* as a synonym for "stochastic." Some algorithms can be viewed as generating paths from an input to an output. For a non-deterministic algorithm, at some states on the path, multiple actions can become available to choose from at the same time. Or for the same input, multiple paths can be generated to reach the same output.

One typical example is planning problems [Russell and Norvig, 1995]. A planning problem gives a set of initial states, final goals and possible actions for each state. The planning problem is to find a plan (which we also refer to as *policy* later) to generate a sequence of actions that are guaranteed to reach one of the goals. For example, in the maze planning problem, the planner starts with some initial state in the maze and is allowed to move in one of four directions: up, down, left, right as long as there are no blocking walls in the chosen direction. The goal is to find a path from initial state to exit/goal state with minimal cost. Figure 1.2 shows two different paths with minimal cost from the start point to final goal (+10) if the cost

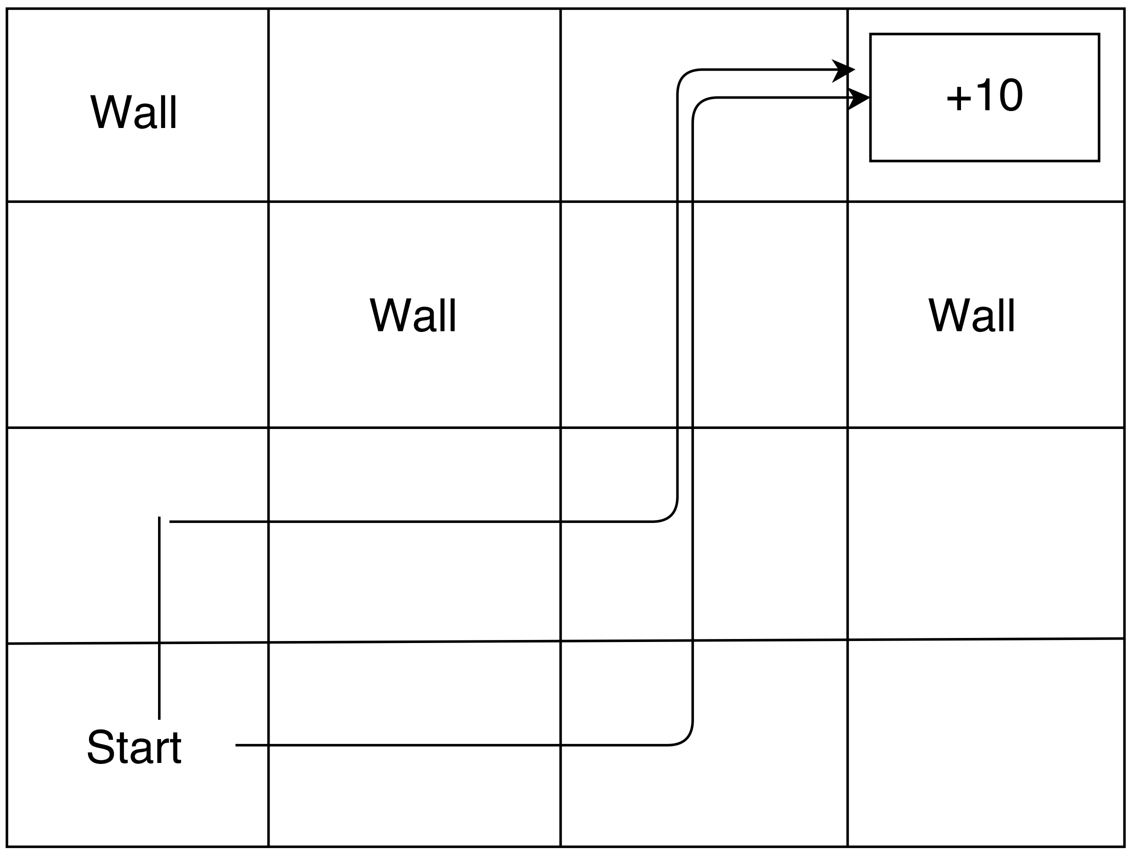

Figure 1.2: Example: Non-deterministic planning on the maze.

of each step is the same. [3]

Another example is conducting inference on a graphical model[4] [Pearl, 1988, Wainwright and Jordan, 2008]. A graphical model is a probabilistic model with a graph representation. The vertices in the graph are random variables and edges are their conditional dependencies. The standard inference tasks are that we query the probability of a combination of random variables in the graph or find the assignment

[3]In a planning problem, the actions can be non-deterministic: the same action can result in more than one outcome.Moreover, if the agent is in a simulated environment, the agent itself can be non-deterministic: the next state might not be a direct result from the immediate previous state. We discuss more about the latter situation in Chapter 3.

[4]Note that this is not a standard approach to inference in probabilistic graphical models.

of each random variable that maximizes the joint probability. Message-passing algorithms are commonly used to infer high probability variable assignments. Each vertex in the graph will receive from its neighbors messages about their quantified beliefs and then subsequently will broadcast a message about its own beliefs to its neighbors. These steps can be done simultaneously (i.e. all the vertices pass messages to their neighbors at once or asynchronously; vertices pass messages to their neighbors one by one). It has been shown that finding a good ordering (which we later refer to as the *policy*) of the set of messages to be passed helps to improve the convergence of the algorithms in practice [Elidan, 2006, Vila Casado et al., 2010].

There are many more examples of non-deterministic algorithms in structured prediction problems (A very detailed parsing example will be in section 1.3). Table 1.1 shows some examples of deterministic v.s. non-deterministic algorithms for different NLP problems.

Table 1.1: List of deterministic/non-deterministic algorithms for NLP problems.

NLP Problem	**Deterministic Algo**	**Non-deterministic Algo**
Part-of-speech Tagging	Greedy Tagger [Brants, 2000]	CRF/HMM Tagger [Lafferty et al., 2001b, Cutting et al., 1992]
Syntactic Parsing	Incremental Parser [Collins and Roark, 2004a]	CKY [Younger, 1967]
Dependency Parsing	Shift-reduce Parser [Aho et al., 2007, Nivre, 2003]	MST Parser [McDonald et al., 2005]

The advantage of choosing a non-deterministic algorithm is that it can tell later in the decoding process whether an earlier decision is good or not. One key question we would like to answer in this dissertation is “When should we use non-determinism?", if we have the choice of changing a deterministic search to a non-deterministic search. Here "when" can refer to either training time or decoding (testing) time. At training time, if the scoring function is parameterized, we can learn the parameters of the model by maximizing the score (if higher is better).

For search problems, search heuristics estimate how far the current state is from the goal state and help to find the goal state faster. A* search is a best-first search algorithm that finds a path from the start point to one goal with minimal cost. Its admissible heuristic is defined to ensure the difference in the heuristics of two nodes on the graph should be less than or equal to the distance between them. For example, for map navigation where the search has to follow the roads, the (admissible) heuristic can be picked as the direct straight line distance from the start to the goal. We can also learn the search heuristics – how to improve a learned model such as learning a pruning parameter or learning admissible heuristics in A* search.

In search problems, a deterministic algorithm always yields a faster solution while a non-deterministic algorithm can better explore the search space. Ideally, if the learner can predict the utility of the states correctly (for each avaiable action at one state, how good is the final result) deterministic training and testing will be very efficient. For example, for part-of-speech tagging, to tag the sentence $s =$"Time flies like an arrow", if it is easy to learn a function that predicts scores for each tag in the sentence: e.g. $f(Time, V, s) = 0$, $f(Time, NN, s) = 1$, $f(flies, V, s) = 1$, $f(flies, NN, s) = 0.8$, etc., then learning and applying this model is good enough.

However, in complicated structured prediction problems like parsing, machine translation or computer vision problems, building a model for all possible actions at all possible states is intractable. Instead, **one should learn how to perform a deterministic search (policy) non-deterministically and run (test) the policy deterministically** if possible so that the final result will be good and quickly obtained. During training time, we want to explore the search space and gather training examples from non-deterministic paths. During test time, we take the deterministic prediction from the learned model. Even if the search algorithm at test time is non-deterministic, we can still learn a good policy to improve the performance by training non-deterministically. Returning to the part-of-speech tagging example "Time flies like an arrow": during training, we would like to generate possible different sequences of the tags and associated scores for the same sentence e.g. ((NN V PREP DET NN), 1.0), ((NN NN V DET NN), 0.5) and learn a model based on those examples. During test time, if we can learn a deterministic model such as a greedy tagger from left to right, it will be very fast. On the other hand, we can also learn a non-deterministic model as CRF and use the examples to speed up the convergence.

Another important question here is "How can we learn a policy non-deterministically?" In this dissertation, we show how to change a deterministic agent to be non-deterministic and create a learning framework with reinforcement learning algorithms and learn policies that give faster and/or better results (Chapter 3). For structured prediction problems like agenda-based parsing, we also show how to learn a better search policy at decoding time by pruning or prioritization (Chapter 4-5).

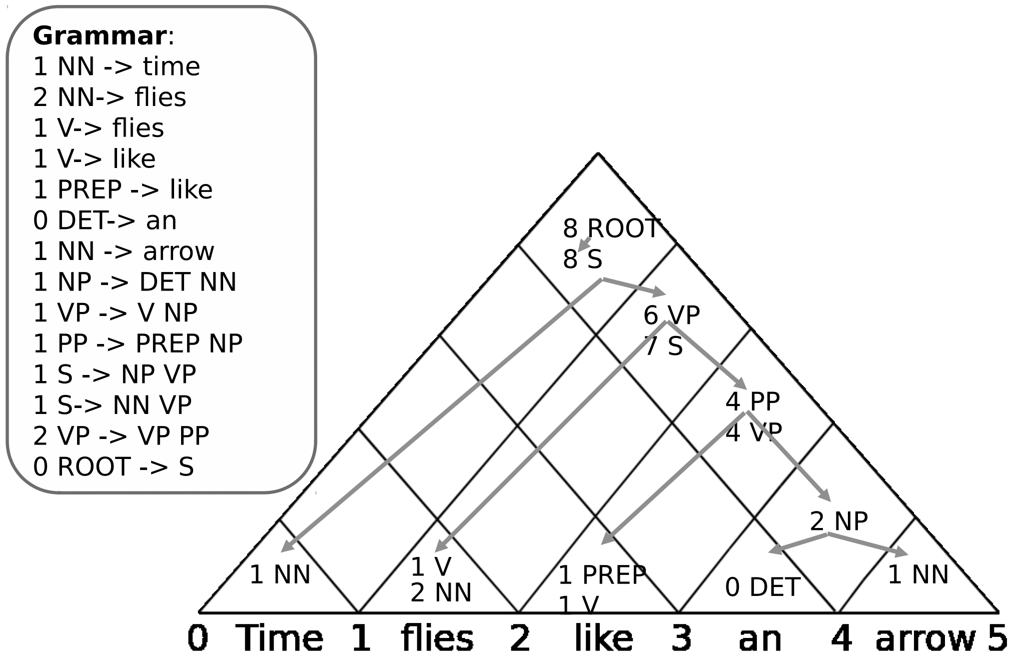

Figure 1.3: Example: Chart parsing example.

1.3 Detailed Example: Agenda-based Parser with Deterministic and Non-deterministic Agent

As an important subject in the dissertation, we now describe the difference between deterministic and non-deterministic agents in parsing [Earley, 1970]. Let us first start with a slightly simpler example – chart parsing. For a visual example, see figure 1.3

For a sentence with length n, a chart is a grid of size $O(n^2)$ that stores the items (constituents) spanning from position i to j in the cell $[i, j]$, where $1 \le i < j \le n$. A grammar describes the rules for combining items from adjacent cells in the chart.

Parse trees can be retrieved from the top of the chart – the cell $[1, n]$. It is an application of dynamic programming. Without getting into too much detail about parsing algorithms, we assume here each rule combines two adjacent cells $[i, j]$ and $[j, k]$ and puts a new item at the cell $[i, k]$.[5]

In the example shown, a grammar is given (or learned) before the decoding. Each grammar rule may be associated with some probabilities or scores. The decoder first fills in some part-of-speech tags of the sentence (in the bottom row), for example "Time" is assigned "NN". Then for each filled cell in the bottom row, it checks all the adjacent cells: e.g. for cell (4, NN, 5), the parser looks up items that can be combined with "NN" on the right and matches any left neighbors. In the grammar, there is NP→DET NN and DET exists as (3, DET, 4), so the rule is applied and we build (3, NP, 5) and fill that into the chart. Its score is the sum of the score of the children and the rule. When the chart is fully populated, we extract the tree with highest likelihood or the best score given the score definition.

We define this chart parsing as a planning problem: the states that we consider are the cells in the chart: e.g. a cell spanning from 3 to 5 is a state. Possible actions are combining two adjacent cells in the chart and adding one more item to a larger span according to the grammar rules: e.g. combining (3, NP, 4) with (4, NN, 5). There is only one initial state: we start with an empty chart. The final states are those states that cannot combine with any other items in the chart. The goal is to

[5]Formally, we assume that we consider a context-free grammar in Chomsky Normal Form with binary rules only as the example here. Note that we will not make this binary assumption in the real application.

find a way of filling the chart and extracting a parse tree.

A deterministic agent in the chart parsing case, e.g. an incremental parser [Collins and Roark, 2004b], you have to use every cell you fill in in the final tree[6]. The following action can only be that of combining two amiable items from the chart. Imaginably, this deterministic parser will not perform well most of the time: once a mistake is made, it will never have a chance to correct it later and it has to continue with some tree containing this mistake. On the other hand, a non-deterministic parser can reconsider past actions and fill the cells in the chart with all possible grammar rules no matter when the children of the rule are built. (For those who are familiar with the area, this is a more flexible version of the standard CKY algorithm [Chappelier et al., 1998, Klein and Manning, 2001, Magerman and Marcus, 1991]).

In this dissertation, we focus on a special case of the chart parser: the agenda-based parser. In agenda-based parsing[7], each grammar rule is associated with a score (log-probability) and instead of directly updating the chart, it pushes all the candidates to the agenda (which is a priority queue) first and pops off the most likely ones and adds it to the chart. 1.3 shows the same example as the chart parser but with the agenda as a priority queue for which item should be popped and added to the chart first.

[6]There might also be a left-to-right restriction in the true incremental parsing setting

[7]For a detailed description of the algorithm, please refer to Chapter 2.2. It is similar to A* search in hypergraphs [Klein and Manning, 2003].

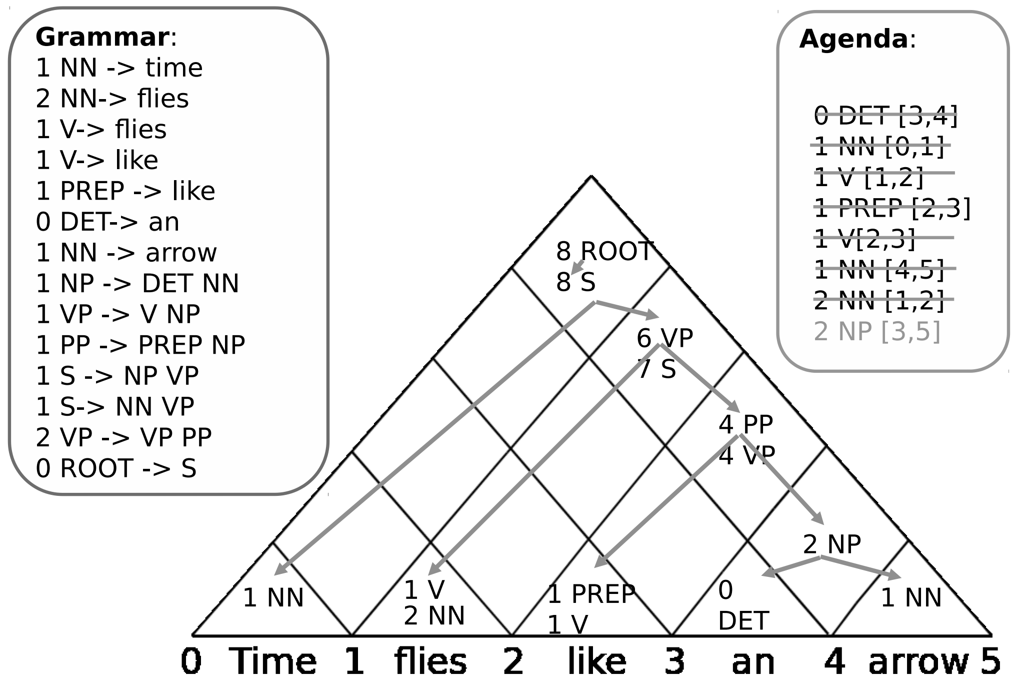

Figure 1.4: Example: Agenda-based parsing example. The crossedout item on the agenda are those already popped.

The agenda-based parsing algorithm is non-deterministic: there exist multiple paths to achieve the same goal – there are no order constraints for which child of a node in the tree should be built first. However, the computational cost for this algorithm is $O(Kn^3)$ where K is the size of the grammar and n is the length of the sentence. K in practice can be very large—some corpora may have more than half a million grammar rules. Parsing a 40-word sentence with this parser may take anywhere from a few seconds to about half a minute given a the grammar with half a million rules.

In order to improve the performance of agenda-based parsers, we can prune or prioritize the agenda. With prioritization (Chapter 3), the same parse tree can be achieved with different paths by alternating the order of the items on the agenda. With pruning (Chapter 4), we enable multiple choices of combining different items to one item according to the grammar.

1.4 Learning with Oracles

For structured prediction problems, knowing the ground truth does not guarantee that a good model can be learned: the ground truth, alone, does not tell the learner how to get to the ground truth. Different from the the standard ground truth (or gold labels) in structured prediction problems, an oracle includes not only the set of gold labels and their structure, but also the process of generating such solutions. The oracle set is built in such a way that it guarantees that a learner can follow it and produce the ground truth.

Consider the case of navigating a robot on the road where there will be obstacles in the center of the road. Assume the robot only knows to adjust its angle by adding weights on the left or right side to make turns. If no order is defined (no preference of left or right side in this case), when the robot sees the obstacle, 50% of the time the oracle tells it to turn left and 50% of the time turn right, and then it will not be able to learn which way to go.

In parsing, the oracle can be the sequence of items (constituents) added to the chart to achieve the gold tree rather than the gold tree itself. The ideal oracle should also cover those states that do not result in ground truth and decide what are the best options for all the possible actions given the current state. However, it is very hard to find ideal oracles in structured prediction problems. Even though the ground truth is provided, given a non-deterministic problem there can be multiple paths leading to the ground truth and a measure with total order needs to be defined to choose the oracle.

Moreover, the set of possible actions (new constituents to build at each state) to choose might not be enough in certain cases to achieve the ground truth (e.g. the gold tree in parsing). There are two properties that any oracle construction should follow:

1. The oracle provides a sequence of (ordered) optimal actions. By following those actions, the decoder can recover the ground truth or expert labels efficiently. The order here refers to when multiple actions are available at one state, there is a consistent order to tell which one to pick.

2. For those states that do not lead to the ground truth or expert labels, the oracle provides a way to lead back to some state that will continue to produce optimal actions or find the second best choice from that state.

In this dissertation, we discuss how to construct different oracles and how to approximate the optimal action when the states are not on the oracle path in the case of parsing (Chapter 4).

1.5 Learning Target

Ideally, a NLP system is expected to be both accurate and fast. However, this is usually not the case. Researchers often build complex systems to improve the performance (accuracy measure) of the system at the cost of speed. In applications with a large set of data to process, less accurate but faster solutions are often preferred. For us, prioritizing and pruning refer to two different learning targets: one is to learn to trade off speed and accuracy and the other is to learn to run as fast as possible without losing too much accuracy.

The first target (learning a user-specified trade-off) is achieved by either cascade modeling (building a set of models from the inaccurate but fastest to the most accurate but slowest and choosing the model according to the performance request), or optimizing the model according to some trade-off measurement while the second one is usually approached by doing different types of pruning. In this dissertation, we explore the first target in chapter 3 by using re-prioritization to optimize a speed accuracy trade-off with user defined speed and accuracy measures. We also tackle the second target (learning to be as fast as possible without losing too much accuracy) in chapter 4 by using a pruning classifier for each of the candidates on top of the agenda. Those targets are easily accomplished by allowing non-determinism at training time.

1.6 Learning with Dynamic Features

Feature engineering is important for machine learning and/or NLP problems. In structured prediction, we define three types of features: static, dynamic and trace features. Static features are used in all machine learning/natural language processing problems while dynamic and trace features are specifically designed for structured prediction and planning problem whose output is built by sequences of predictions.

Static features are those features associated with the input. They will not change once the input is decided. *Dynamic features* are associated with the decoding/learning process. Their values depend on local or global context as well as ranks and orderings. *Trace features* records the history of building the structured output so far, not just the current state.

Static features are cheaper to compute while dynamic and trace features require good bookkeeping of the current local/global context and history.

For example, in agenda-based parsing, static features include the width of the cell, starting position, ending position, the punctuation it contains, etc. Dynamic features can look at the competing item locally or globally such as ratio or absolute difference of best in cell, rank of item in cell, etc. Trace features use information like how long an item has been waiting on the agenda or the number of pushes of a constituent, etc.

Computing dynamic and trace features on the run can be very expensive, but a lot of context-based dynamic features (e.g. n-gram features) can be pre-computed and only require simple look-up during learning. We can also use statistics computed from the training data to approximate the on-the-run feature values. As they better describe the “real-time” information of the decoding/search process, dynamic and trace features can improve the efficiency of local decisions. We show the impact of dynamic and trace features in Chapter 5 in the case of learning a dynamic pruner. With dynamic and trace features, the system produces better and faster parse trees.

1.7 Contribution

The main contribution of this dissertation is answering these important questions:

- Can non-determinism improve the accuracy and/or speed of a search/decoding system?
- When shall we take advantage of non-determinism?
- How shall we apply this to NLP systems like parsers?

We claim that we should use non-deterministic algorithms in the training (either learning a model or improving a decoder) stage if the search space is large and deterministic algorithms cannot properly explore different decisions for the visited states. Moreover, if applicable, we should apply a fast deterministic algorithm during decoding.

To answer the first and second question, we study the property of non-determinism in the context of maze solving. We introduce a non-deterministic learning framework that can be applied to all kinds of trajectory-based reinforcement learning algorithms. While learning non-deterministically at training time to produce a deterministic policy at test time if possible, this framework will not sabotage the original convergence condition of the existing algorithms and make those algorithms converge better and/or faster.

For the third question, we take the agenda-based parser as an example. We introduce more non-determinism into the training by allowing agenda prioritizing or pruning. We show different problem formulations with different learning targets. To automate the speed accuracy trade-off, we propose an oracle-infused policy gradient algorithm that re-prioritizes the items on the agenda to trade-off customized speed and accuracy. Besides these algorithms, we show a very detailed analysis of the advantage and disadvantage of some off-the-shelf structured prediction/reinforcement learning algorithms in this application. With similar ideas, we propose a dynamic pruning classifier for the agenda-based parser that achieves state-of-the-art accuracy and speed and show the importance of dynamic features.

1.8 Outline of the dissertation

The structure of this dissertation is as follows:

Chapter 2 In this chapter, we briefly explain some concepts, algorithms and literature overview of Markov decision processes, reinforcement learning, imitation learning and their applications to structured prediction problems. We also discuss the agenda-based parser and different speed-up heuristics.

Chapter 3 To show how a non-deterministic agent can affect the learning of a deterministic policy, we define a non-deterministic agent based on a MDP. Markov Decision Processes (MDPs) are successful abstractions for modeling sequence decision making problems, with decisions being made by a *deterministic, greedy* agent. Reinforcement learning algorithms, like offline Q-learning, use such a deterministic agent to explore an MDP, updating estimates of Q-values incrementally. However, for many problems, restricting one's learning to a deterministic agent is detrimental: this is a major source of difficulty due to the credit assignment problem. We consider a *non-deterministic* variant of MDPs in which the agent can occupy multiple states simultaneously: we call these *multi-state MDPs*. Multi-state MDPs can conceptually be mapped to standard deterministic MDPs, though with an exponential blowup in the size of the state space. We show how to directly take advantage of the non-determinism of the agent in different reinforcement learning algorithms. In all of these cases, we show convergence of the learning algorithms, and empirically demonstrate that the non-deterministic algorithms we propose converge faster with more accurate policies than the other standard learning algorithms.

In Chapter 4 and 5, we study non-determinism in agenda-based parsing.

Chapter 4 Users want inference to be both fast and accurate, but quality often comes at the cost of speed. The field has experimented with approximate inference algorithms that make different speed-accuracy tradeoffs (for particular problems and datasets). We aim to explore this space automatically, focusing here on the case of agenda-based syntactic parsing. Unfortunately, off-the-shelf reinforcement learning techniques fail to learn good policies: the state space is simply too large to explore naively. An attempt to counteract this by applying imitation learning algorithms also fails: the "teacher" follows a far better policy than anything in our learner's policy space, free of the speed-accuracy tradeoff that arises when oracle information is unavailable, and thus largely insensitive to the known reward function. We propose a hybrid reinforcement/apprenticeship learning algorithm that learns to speed up an initial policy, trading off accuracy for speed according to various settings of a speed term in the loss function.

Chapter 5 There are many heuristic based strategies to speed up a parsing algorithm such as prioritization, pruning and coarse-to-fine. In terms of pruning an agenda-based parser, the pruning heuristics decide whether the constituents should be added to or popped off the agenda and speed up the decoding by removing costly or unlikely ones. However, finding a good pruning threshold for a fast parser with reasonable accuracy is hard and this threshold can vary from sentence to sentence.

On the other hand, while using coarse pruning classifiers like [Roark and Hollingshead, 2008] allows larger reductions in the search space before parsing a sentence, it ignores the dynamic features in the decoding process. In this chapter, we propose to use a dynamic pruning classifier at pop time to prune individual candidates from the chart. We show our model can achieve state-of-the-art speed and accuracy by pruning dynamically.

Chapter 6 In this chapter, we summarize the algorithms and results from the dissertation and put a conclusion to the questions we answer in the dissertation. We also discuss the generalization of these methods in different machine learning problems.

Chapter 2: Background and Related Work

The majority of the dissertation is built on ideas and algorithms from reinforcement learning, structured prediction, imitation learning and agenda-based parsing. In this chapter, we will first discuss the basics of Markov decision processes and related reinforcement learning algorithms for discrete and continuous state/feature space. The extension to the non-deterministic agent in chapter 3 is based on these basics. Next, we will introduce the background of imitation learning and structured prediction. We will also provide the necessary background for agenda-based parsing and state-of-the-art speed-up techniques in section 2.2. Our discussion in chapter 4 about the disadvantage of applying off-the-shelf imitation learning methods in structured prediction to the problem of trading off speed and accuracy in parsing is based on the algorithms and ideas that we reveal in section 2.3.

2.1 Markov Decision Process (MDP) and Reinforcement Learning

Markov decision processes (MDP) [Bellman, 1957] provide a coherent model for agents interacting with an environment. An MDP model contains state space S, action space A, transition function T and reward function R. It is a formalization of a memoryless search process — the effects of an action taken in a state depend only on that state, but not the history. Formally, for a deterministic transition function:

$$T : S \times A \rightarrow S \tag{2.1}$$

for each state and action, the subsequent state is deterministically decided.

For stochastic transition function:

$$T : S \times A \rightarrow Prob(S) \tag{2.2}$$

where $Prob(S)$ is a probability distribution over the state space S. In a stochastic setting, for each state and action, a distribution of the possible following states is defined: $T(s'|s, a)$. The reward function $R(s, a)$ can also be deterministic or stochastic according to some distribution.

An agent in an MDP observes the current state $s \in S$ and chooses an action $a \in A$. The environment responds by transitioning to a state $s' \in S$, sampled from the transition distribution $T(s' \mid s, a)$. The agent then receives a reward $R(s' \mid s, a)$, observes its new state and chooses a new action. An agent's *policy* π describes how the (memory-less) agent chooses an action based on its current state, where π is either a deterministic function of the state (i.e., $a = \pi(s)$) or a stochastic distribution over actions (i.e., $a \sim \pi(\cdot \mid s)$), and by giving the agent a reward $r \in \mathbb{R}$, which is sampled from the MDP's reward distribution $R(r \mid s, a, s')$.

We define the cumulative reward of a policy π to be $\sum_{t=0}^{\infty} \gamma^t R(s_t|s_{t+1}a_t$ where $\gamma \in (0, 1]$ is a discount factor. The optimal policy π^* is the one which maximizes the cumulative reward. If the total reward is infinite (in most cases), a few options can be chosen to make the objective function tractable:

1. Discounted reward: a discount factor $0 < \gamma \leq 1$ for the reward for each step. This is the most common one in the literature.

2. Use a finite horizon (time steps) and the total reward for that period of time.

3. Use an averaged reward rate in the limit.

In the dissertation, we will only consider the case with discounting. To make the terminology simpler, whenever we say that we maximize the expected future reward, if the reward is infinite, we implicitly mean that we maximize the cumulative discounted reward.

A value function $V_\pi(s)$ represents the objective value obtained by following policy π from state s. The Bellman equation is a dynamic programming approach for computing the value for each state if both rewards and transition functions are known:

$$V_\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s'|\pi(s), s) V_\pi(s') \tag{2.3}$$

Iteratively, the maximized value at state s is nothing but

$$V^*(s) = \max_a \left[R(s, a) + \gamma \sum_{s' \in S} T(s'|s, a) V^*(s') \right] \tag{2.4}$$

When any of the rewards or transition functions are unknown, reinforcement learning algorithms help the agent learn the model (estimate reward and transition functions) as well as which actions to take at each state according the objective function. In the dissertation, we will consider trajectory-based algorithms where the agent explores and exploits trajectories in the state space to maximize the reward. A trajectory is defined to be a sequence of state and action pairs from some initial state to some terminal state. A terminal state is a state that has no outgoing actions. If the cumulative rewards are infinite, we can use an approach similar to the one mentioned above.

2.1.1 Q-Learning and Variants

Similar to how we define the value for a state $V(s)$, in reinforcement learning we can also define a value function for a state and action pair

$$Q(s,a) = \sum_{s'} P(s|s',a)\left(R(s|s',a) + \gamma V(s')\right) \tag{2.5}$$

which corresponds to the cumulative reward of taking action a at state s and following the current policy.

When the state space is discrete, the Q-learning algorithm can be applied to estimate the value for each state action pair. In Q-learning [Watkins, 1989], the agent starts a trajectory at some start state s_0, chooses an action a using the policy derived from current Q value ($\pi(s) = \max_a Q(s,a)$) with ϵ-greedy exploration [1], and observes reward r and next state s'. The Q-value is then estimated by

$$Q_{t+1}(s_t,a_t) = (1-\alpha_t(s_t,a_t))Q_t(s_t,a_t) + \alpha_t(s_t,a_t)(r(s_t,a_t,s'_t) + \gamma V_t(s_{t+1}))$$

where γ is a discount factor and the estimated value of a state at iteration t is

$$V_t(s) = \max_{a \in A(s)} Q_t(s,a) \tag{2.6}$$

So in equation 2.6, $V_t(s_{t+1}) = \max_a Q_t(s_{t+1},a)$.

Algorithm 1 shows the Q-learning algorithm.

[1]With $1-\epsilon$ probability, the agent takes the action given by the previously learned policy and with ϵ probability, it picks a random action

Algorithm 1 Q Learning Algorithm

```
Input: state space $S$, action space $A$, reward $R$, transition function $T$, discount factor $\gamma$, start state $s_0$.
Initialize Q-values $Q(s,a)$ arbitrarily.
repeat
    repeat
        Choose an action $a$ that has the maximum expected future reward using the current policy: $a = \arg\ max_{a' \in A(s)} Q(s,a')$
        Take action $a$ and observe reward $r$, state $s'$.
        Update $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a')]$.
    until Trajectory termination condition is satisfied.
until The convergence criterion is met.
```

Q learning is a special case of an offline case of the SARSA (state-action-reward-state-action) algorithm [Kaelbling et al., 1996, Sutton and Barto, 1998]. It directly approximates the optimal Q-value Q^* independent of the current policy and only relies on the expected future reward for the following states.

Q value updates can also be policy dependent. Online SARSA is a policy dependent algorithm and it uses the Q value of the following states and its following action given current policy. Specifically,

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma Q(s', \pi(s'))] \tag{2.7}$$

$pi(s')$ is the current policy derived from the current Q value estimates.

In practice, updating the current state as well as some states visited in the past can result in better convergence, such as Dyna Q [Sutton and Barto, 1998] and prioritized sweeping. Prioritized Sweeping in section 2.1.2 is one of the most common ways of updating the value of previous states by backtracking the trajactories.

2.1.2 Queue Dyna

Q learning is guaranteed to eventually converge without error when enough exploration in the space is done. However, when the search space is large or the model is not given (transition/reward needs to be estimated), the actual covergence speed can be slow. For example, if the agent is designed to explore a large maze and it gains a high positive final reward when it finds the exit, it is obvious that the value is high for the states near the exit and decrease with the increasing of distance between the internal state and the exit. In other words, the value of the states further away from the exit are required to wait for the value to be propogated from the exit state. Even for an extremely simple example such as when a maze is a 1000-step straight path, in order to get the Q value properly estimated the agent is required to repeat the same path 1000 times! (For iteration i, only values from 1000-i to 1000 are properly updated.) However, if value backups are allowed – for each value update, the agent is allowed to pick some states in the past and update their values, the convergence speed can be faster. In the 1000-step straight path maze case, if we allow the agent to trace back its trajectory from the exit and update the value of states in the past in order, the agent now only needs one trajectory to reach convergence (although the number of visited states is 2000 when the backup updates are figured in). On the other hand, if the agent selects random states in the past, the convergence will not be as fast. So we need a good strategy to decide the order of the states to be updated in the past.

Queue Dyna [Peng and Williams, 1993] is a trajectory-based reinforcement learning algorithm for propagating the most recent value to the known precedents after the terminal state is reached. Intuitively, if the value of a state differs a lot between updates, it means the value is not close to convergence. If more information is provided (update those states when the terminal state is reached), it is more likely to be useful than updating the states whose values are already close to convergence.

In Queue Dyna, the underlying algorithm is still Q learning and the states to be updated in the history are ranked according to the difference in value updates. Previously visited states that have larger value changes will have higher priority to be updated.

Prioritized Sweeping [Moore and Atkeson, 1993a, Andre et al., 1998] is based on the same core idea with additional model (transition/reward) estimates.

The agent starts with an initial state and chooses the action according to the current policy. The environment responds with state s' and reward r. A priority queue is used for all the visited states with priority as the difference of the value before and after update. The state and action pair is pushed onto the priority queue. N updates are performed by popping one item from the priority queue, updating the Q value and adding all the precedent state-action pairs to the queue. It is shown to have faster convergence speed in practice than unprioritized methods.

Algorithm 2 Queue Dyna Algorithm

```
Initialize the Q value $Q(s,a)$ for all state-action pairs randomly. Initialize the model $M(s,a) : S \times A \rightarrow S \times \mathbb{R}$ for the following state and immediate reward by taking action $a$ from current state $s$. Set priority queue PQ to be empty, priority threshold to be $\epsilon$ and the number of sweeping states to be $N$.
while Values not converged do
    $\hat{s}$ is the current visiting state and $\hat{a} = \arg\max_a Q(s,a)$.
    Move the agent with action $a$, observe the following state $s'$ and the immediate reward $\hat{r}$.
    Update the model $M(\hat{s},\hat{a}) \leftarrow (\hat{s}',\hat{r})$.
    Compute priority $\hat{p} = |\hat{r} + \gamma \max_{a'} Q(\hat{s}',a') - Q(\hat{s},\hat{a})|$.
    if $p > \epsilon$ then
        Add $(\hat{s},\hat{a})$ to PQ with priority $\hat{p}$.
    end if
    while $N$ states have not been updated and PQ is not empty do
        Pop $(s,a)$ from PQ and its following state and immediate reward from model $M$: $(s',r) \leftarrow M(s,a)$.
        Update $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$.
        for all $(\tilde{s},\tilde{a})$ that are precedents of $s$, reward is $\tilde{r}$. do
            Update $\tilde{p} = |\tilde{r} + \gamma \max_a Q(s,a) - Q(\tilde{s},\tilde{a})|$.
            if $p > \epsilon$ then
                Add $(\tilde{s},\tilde{a})$ to PQ with priority $p$.
            end if
        end for
    end while
end while
```

2.1.3 Policy Gradient

When the state space is huge and not many states can be explored during learning, function approximation can be adopted to represent a state s in the state space with its features $\theta = \phi(s)$. So the correspoding policy is also paramterized by θ. Finding the optimal policy is equivalent to finding parameters that yield the highest possible expected reward. We carry out this optimization using a stochastic gradient ascent algorithm known as policy gradient [Baxter and Bartlett, 2001, Williams, 1992, Sutton et al., 2000].

For a parameterized trajectory τ, assume that τ is generated from $p_\theta(\tau)$. Recall the cumulative reward $R(\tau) = \sum_0^T \gamma_t r_k$ where γ is the discount factor and r_k is the reward for each step. The cumulative reward in expectation is

$$\mathbb{E}_\tau[R(\tau)] = \int p_\theta(\tau) r(\tau) \mathrm{d}\tau \tag{2.8}$$

In order to perform gradient descent, we need to compute the gradient of this expectation, $\mathbb{E}_\tau[R(\tau)]$. From basic calculus, we know for any function p_θ,

$$\nabla_{\boldsymbol{\theta}} \log p_\theta(\tau) = \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} \tag{2.9}$$

So we can rewrite this as

$$\nabla_\theta p_\theta(\tau) = p_\theta(\tau) \nabla_{\boldsymbol{\theta}} \log p_\theta(\tau) \tag{2.10}$$

So the derivative of the expectation becomes

$$\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau) = p_{\boldsymbol{\theta}}(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) \tag{2.11}$$

so the gradient of the objective becomes

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}_\tau[R(\tau)] = \mathbb{E}_\tau\Big[R(\tau)\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)\Big] = \mathbb{E}_\tau\Big[R(\tau)\sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi(a_t \mid s_t)\Big] \tag{2.12}$$

which can be approximated by sampling trajectories. The policy gradient algorithm samples one (or several) trajectories according to the current policy $\pi_{\boldsymbol{\theta}}$, and then takes a gradient step according to Eq (2.12). This increases the probability of actions on high-reward trajectories more than actions on low-reward trajectories.

A question in reinforcement learning is the exploration and exploitation trade-off. There are two standard approaches to explore the space. One is ϵ-greedy [Sutton and Barto, 1998] which randomly picks an action with probability ϵ and follows the policy with probability $1-\epsilon$. The other is Boltzmann exploration [Sutton and Barto, 1998]. Our policy is:

$$\pi_{\boldsymbol{\theta}}(a \mid s) = \frac{1}{Z(s)} \exp\left[\frac{1}{\textit{temp}}\ \boldsymbol{\theta} \cdot \boldsymbol{\phi}(a, s)\right] \tag{2.13}$$

That is, the log-probability of action a at state s is an affine function of its priority. The temperature *temp* controls the amount of exploration. As $\textit{temp} \to 0$, π approaches the deterministic policy in Eq (4.2); as $\textit{temp} \to \infty$, π approaches the uniform distribution over available actions. During training, *temp* can be decreased to shift from exploration to exploitation.

2.2 Agenda-based Parsing

The goal of parsing is to obtain a syntax tree from an input sentence given a grammar. Figure 2.1 shows an example of a sentence and its parse tree.

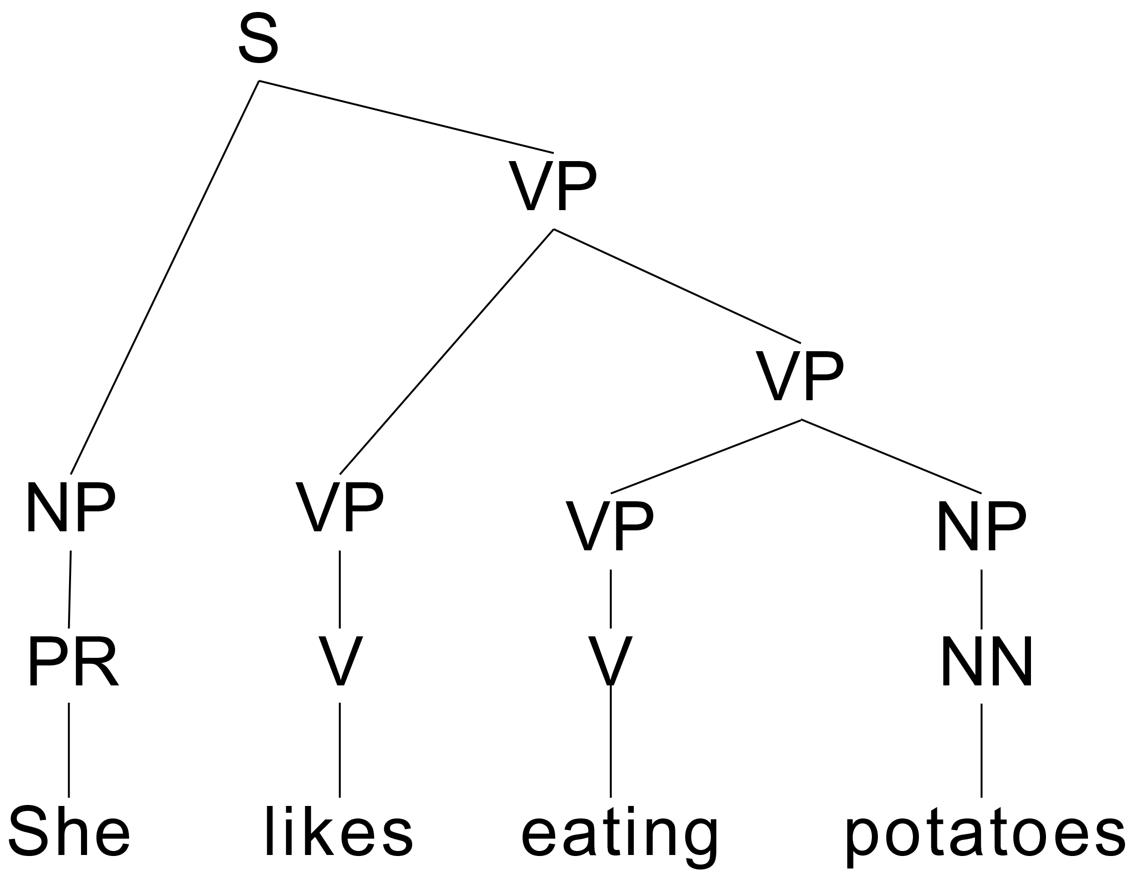

Figure 2.1: Example of the parse tree of "She likes eating potatoes" with given grammar.

Given a grammar, perhaps the simplest approach to inferring the best parse tree for a given sentence is to assemble the parse from the bottom up as in CKY [Younger, 1967](Figure 2.2 shows an example of a filled chart.). When the grammar is probabilistic, a standard extension of the CKY algorithm uses an "agenda" — a priority queue of constituents built so far—to decide what to do next [Kay, 1986]. The priority can be the inside score [Klein and Manning, 2001] or inside score combined with an admissible estimate of the outside score [Klein and Manning, 2003, Pauls and Klein, 2010, Felzenszwalb and McAllester, 2007] so that the first tree built in the chart will be the most likely tree (Viterbi tree) under the current grammar.

Three major strategies are used in the literature to speed up parsing: prioritization, pruning and coarse-to-fine search.

Prioritization heuristics govern the order in which search actions are taken while pruning heuristics explicitly dictate whether particular actions should be taken at all. Examples include A* [Klein and Manning, 2003, Haghighi et al., 2007] and Hierarchical A* [Pauls and Klein, 2010] heuristics, which, in the case of agenda-based parsing, prioritize parse actions by estimating outside scores admissibly so as to reduce work while maintaining the guarantee that the most likely parse is found. Figure-of-merit prioritization [Caraballo and Charniak, 1998, Charniak et al., 1998] can result in even faster inference if a small amount of search error can be tolerated.

Alternatively, pruning heuristics decide if search actions should be taken at all. The maximum number of elements allowed in a cell in the chart can be used to

Grammar:
PR –> She
V –> likes
IN –> likes
V –> eating
NN –> potatoes
NP –> NN
NP –> PR
VP –> V
VP –> VP NP
VP –> NP VP
VP –> VP VP
PP –> PREP VP
S –> NP VP
S –> VP VP

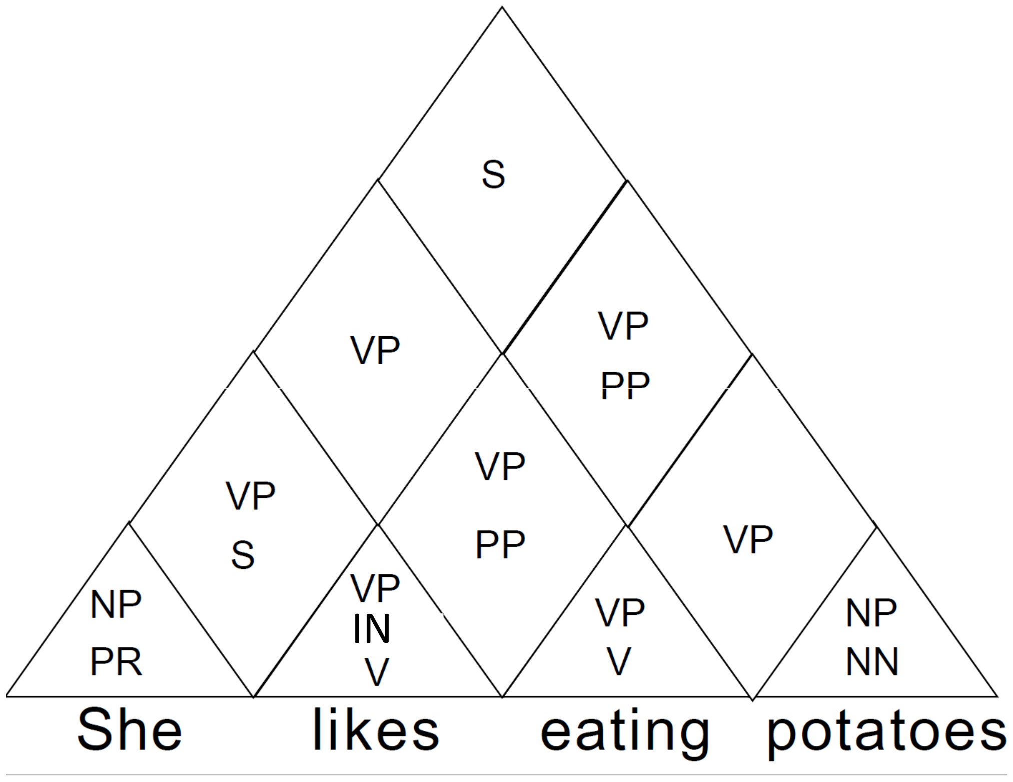

Figure 2.2: Example of a parsing chart of “She likes eating potatoes”.

Algorithm 3 Agenda-based Parsing Algorithm

```
while agenda is not empty do
  dequeue some update according to its priority from the agenda, say $(Y, i, j) \leftarrow$
  75
  update chart$[Y, i, j]$ to 75
  for each constituent adjacent to $(Y, i, j)$, such as $(Z, j, k)$ do
    for each grammar rule $X \rightarrow Y Z$ that can combine $(Y, i, j)$ with $(Z, j, k)$ do
      let $new \leftarrow chart[Y, i, j] + chart[Z, j, k] + score(X \rightarrow Y Z)$
      if $new > chart[X, i, k]$ then
        enqueue $(X, i, k) \leftarrow$ likelihood$([X, i, k])$ on the agenda
      end if
    end for
  end for
end while
```

prune the search space. The difference in probability between the candidate to be filled into the cell and the best built element in the cell is another commonly used pruning threshold. Classifier-based pruning [Roark and Hollingshead, 2008] learns a classifier to disallow spans starting from/ending at a certain position by running a part-of-speech tagger before parsing. Beam-width prediction [Bodenstab et al., 2011] tries to predict the maximum number of items in each cell.

Coarse-to-fine strategies allow multiple passes during decoding. [Pauls and Klein, 2010, Charniak et al., 2006, Petrov and Klein, 2007] uses a multi-level grammar to parse the sentence from the most coarse grammar to finest grammar. The most coarse grammar only uses one instance of each pre-terminal and non-terminal: NP_0 –> DET_0 NN_0, VP_0 –> V_0 (here $\_i$ indicates the ith instance of the symbol). For level-k grammar, $i \leq 2^{k-1}$ instances of symbols are used in the grammar. For high level grammars, different instances of the same symbol can be viewed as latent clusters of the symbol in the examples. The most coarse grammar is the fastest one to decode with but with lower accuracy while the finest grammar is the slowest but most accurate. The parser can stop with coarser grammar if the decoding is good enough.

2.3 Imitation Learning, Reinforcement Learning and Structured Prediction

In reinforcement learning, an agent interacts with an environment and attempts to learn to maximize its reward by repeating actions that led to high rewards in the past. In apprenticeship learning, we assume access to a collection of trajectories taken by an *optimal policy* and attempt to learn to mimic those trajectories. The learner's only goal is to behave like the teacher: it does not have any notion of reward. In fact, the related task of inverse reinforcement learning [Ng and Russell, 2000, Ziebart et al., 2008] (previously called inverse optimal control [Kalman, 1968, Boyd et al., 1994]) attempts to infer a reward function from optimal behavior.

Many algorithms exist for apprenticeship learning. Some of them work by first executing inverse reinforcement learning [Ng and Russell, 2000, Ziebart et al., 2008, Kalman, 1968, Boyd et al., 1994] to induce a reward function and then feeding this reward function into an off-the-shelf reinforcement learning algorithm like policy gradient to learn an approximately optimal agent [Abbeel and Ng, 2004, Neu and Szepesvári, 2009]. Alternatively, one can directly learn to mimic an optimal demonstrator, without going through the side task of trying to induce its reward function [Ratliff et al., 2007, Daumé III et al., 2009, Argall et al., 2009, Ross and Bagnell, 2010, Syed and Schapire, 2011, Natarajan et al., 2011, Ross et al., 2011a].

In [Neu and Szepesvri, 2009], inverse reinforcement learning is applied to train a parser. The goal of inverse reinforcement learning is to find a reward function so that if the agent performs optimally according to the reward function, it will follow the optimal trajectories in the training set. As a result, the score of the rules extracted for the parser are trained such that the highest reward decoding for the sentence will match the ground truth.

[Maes et al., 2008a] also discuss reducing structured prediction problems to MDP and use the approximate reinforcement learning algorithm SARSA to solve these and apply this framework to sequence labeling and ranking tasks in [Maes et al., 2008b].

In this section, we introduce two very important algorithms that are widely used in structured prediction: Search-based Structure Prediction (SEARN) [Daumé III et al., 2009] and Dataset Aggregation (DAgger) [Ross et al., 2011a].

2.3.1 Search-based Structure Prediction (SEARN)

In SEARN, the prediction of the final output is decomposed into a sequence of predictions y_i and each prediction y_i depends on the previous $y_1, \ldots, y_{i-1}$ predictions. A cost sensitive classifier is trained on the examples where the cost is estimated by applying the current policy.

As in algorithm 4, SEARN starts with some expert policy π_0 and trains a policy $\bar{\pi}_1$ to minimize the induced loss. The new policy π_1 is a mixture of the expert policy and the learned policy: $h_1 = \alpha\pi_0 + (1-\alpha)\bar{\pi}_1$. In order to train a classifier $\bar{\pi}_i$, a set of cost sensitive examples are generated for each action according to the current policy. The policy at time i, $h_i = (1-\alpha)\pi_{i-1} + \alpha\bar{\pi}_i$. The final policy is returned without π_0. The probability of using earlier classifiers decreases during the learning, so finally, the learned policy will only depend on the expert a little.

SEARN has been applied in different structured prediction problems such as dependency parsing [Choi and Palmer, 2011].

Algorithm 4 SEARN

```
Initialize the policy $\pi = \pi^*$.
while the current policy $\pi_i$ strongly depends on $\pi^*$ do
    Initialize the cost-sensitive dataset $\mathcal{D} = \emptyset$.
    for each training example do
        Compute the current prediction $\hat{y} = \pi(x)$.
        for each component in $x$: $x_i$ do
            Compute features $\phi(x_i)$ based on the input $x$ and output for each component until $i$.
            for each possible following action $a_k$ do
                Compute the cost $c_k$ by taking action $a_k$ and running the current policy $\pi_i$.
            end for
            $\mathcal{D} = (\phi(x_i), a_k, c_k)$.
        end for
        Train a classifier $\bar{\pi}_{i+1}$ on $\mathcal{D}$.
        Update the policy $\pi_{i+1} = \alpha\bar{\pi}_{i+1} + (1 - \alpha)\pi_i$
    end for
end while
Return $\pi$ without $\pi^*$.
```

2.3.2 Dataset Aggregation (DAgger)

DAgger is an iterative dataset aggregating algorithm that collects datasets at each iteration by running the current policy and trains a new policy with the union of all the previously collected datasets. It starts with an expert policy and lets the new policy be a mixture of the expert policy π^* and learned policy. Then it samples trajectories with the current policy and aggregates datasets $\mathcal{D}$ by adding the sampled states and their oracle labels $(s_i, \pi^*(s_i)$. A new classifier trained on the aggregated set $\mathcal{D}$ (See algorithm 5).

Algorithm 5 DAgger Algorithm

```
Initialize the expert policy $\pi^*$, initial policy $\bar{\pi}_0$ and aggregated dataset $\mathcal{D}$.
for each iteration $i$, do
    (optional) Set $\pi_i = \alpha_i \pi_i^* + (1 - \alpha_i)\bar{\pi}_i$.
    Run the agent with policy $\bar{\pi}_i$ for $k$ steps.
    Collect dataset $\mathcal{D}_i = (x, \pi^*(s))$ for all $x$ visited by $\pi_i$.
    Train $\bar{\pi}_{i+1}$ on $\mathcal{D} = \mathcal{D} \cup \mathcal{D}_i$.
end for
Return the best $\bar{\pi}_i$ with cross-validation.
```

DAgger always tries to best mimic the expert behavior on the aggregated dataset. It is a no-regret algorithm: the difference between the cost of the algorithm and optimal solution is zero. Variants of the algorithm have been used in robot remote control [Argall et al., 2009], message-passing algorithms [Ross et al., 2011b], etc.

In the next chapter, we will use the reinforcement learning discussed here and generalize it to a nondeterministic setting. In chapters 4 and 5, we will use agenda-based parsing as an example and discuss some of the structure prediction and reinforcement learning techniques discussed in this chapter to reprioritize and prune the parser.

Chapter 3: Learning with A Non-deterministic Agent

3.1 Overview

Markov decision processes [Bellman, 1957] provide a coherent model for agents interacting with an environment. The standard assumption, appropriate in applications like robotics, is that the learning agent is *deterministic*. That is, it can only occupy one state at any given time. When the agent exists in the "real world", this is sensible. Recently, however, MDPs and reinforcement learning have been applied to a much broader range of problems, particularly ones in which the agent and environment are *simulated* [Silver, 2009]. Here, the assumption of determinism is overly restrictive: when reinforcement learning occurs offline, there is no reason to force ourselves to use a deterministic agent to learn, even if the desired agent will be itself deterministic. In this chapter, we will focus on trajectory-based learning algorithms and show how to use non-deterministic agents to learn deterministic policies (Section 3.2) using computationally efficient variants of a few common learning algorithms (Section 6.2.5), such as Q-learning [Watkins, 1989] and Queue-Dyna [Peng and Williams, 1993]. Our algorithms are provably convergent and achieve improved empirical learning rates compared to other standard reinforcement learning algorithms (Section 3.4).

In order to achieve these goals, we introduce a framework called “multi-state Markov decision processes.” In a standard MDP [Bellman, 1957], one has a state space, an action space, a (possibly stochastic) transition function and a (possibly stochastic) reward function. An agent is a function that maps states to actions, at which point the environment responds by moving the agent to another state (sampled from the MDP’s transition distribution) and by producing a reward. The goal of an agent is to maximize its cumulative long-term reward. When we allow the agent to be *non-deterministic*, we allow it to occupy multiple states simultaneously. At each point in time, it must select which state it wants to move from (from its set of occupied states) and what action to take from that state. Importantly, the agent never “leaves” a state once it visits it: It expands in the state space. Multistate MDPs implicitly encode the credit assignment into the trajectories: if there are two competing states, the agent will explore both of them until some future states show up with a lower reward.

Speed of convergence is a key notion here, since the large state spaces encountered in real world problems can make learning very slow. For example, Dyna-Q learning [Sutton, 1991, Sutton et al., 2008] and Queue-Dyna [Peng and Williams, 1993]/prioritized sweeping [Andre et al., 1998, Moore and Atkeson, 1993b] update the value of the past states during learning. Other methods like experience relay [Cichosz, 1999] use the learning experience in the history to help the current value updates.

Most reinforcement learning algorithms have two major concerns, one is the *exploration-exploitation tradeoff* [Wilson and Strategies, 1996], which we do not address in this chapter. The second is *temporal credit assignment* [Kaelbling et al., 1996, Sutton, 1988]: deciding, after achieving a large reward or suffering a large punishment, what it was that the agent did that led to this outcome. Several approaches exist to deal with credit assignment: TD(λ)-learning [Sutton, 1988] uses a temporal difference operator that updates the prediction of values; while Q-learning [Watkins, 1989] learns the values of state-action pairs without estimating the transition and reward functions explicitly.

As discussed in the previous chapter, if the search space is very large, the deterministic agent will take a long time to wait for the value to be propogated from the terminal state. In order to compare two actions at one state, it requires the agent to wait for multiple trajectories that take either of the actions and update the value for the state action pairs. We show that by using non-deterministic agents, we naturally obtain more control over credit assignment because we can truly answer questions like "what would have happened if we had taken another action at some point back in time."

3.2 Multi-state MDPs

In a standard setting, an MDP is composed of a (typically finite) *state space* S, an *action space* A, a (possibly stochastic) *transition function* T, and a (possibly stochastic) *reward function* R. An agent observes the current state $s \in S$ and chooses an action $a \in A$. The environment responds by moving the agent to another state $s' \in S$, which is sampled from the MDP's transition distribution $T(s \mid a, s')$, and by giving the agent a reward r, which is sampled from the MDP's reward distribution $R(s \mid a, s')$. The agent's goal is to maximize its total cumulative reward over time. An agent's *policy* π describes how the agent chooses an action based on its current state. Formally,

Definition 1 (**MDP**). *A Markov Decision Process can be defined by a tuple $M = (S, A, T, R)$, where*

- *S is a finite state space,*
- *$A(s)$ is a finite set of all available actions at state $s \in S$,*
- *$T(s, a, s')$ is the probability that taking action $a \in A(s)$ in state $s \in S$ will lead to $s' \in S$,*
- *$R(s, a, s')$ is the immediate reward by taking action $a \in A(s)$ in state $s \in S$ and leading to $s' \in S$.*

The Bellman equations for value and Q value are defined as

$$V(s) = \max_a \sum_{s'} T(s, a, s') \left(R(s, a, s') + \gamma V(s')\right) \tag{3.1}$$

$$Q(s, a) = \sum_{s'} T(s, a, s') \left(R(s, a, s') + \gamma V(s')\right) \tag{3.2}$$

A deterministic optimal policy is defined as $\pi(s) = \arg\max_a Q(s, a)$.

We now define a multi-state MDP (MSMDP) on the basis of a given standard MDP M. In an MSMDP, the state space $\mathcal{S}$ is a power set of the original state space S. An agent observes the current state $\tilde{s} \in \mathcal{S}$ where $\tilde{s} \subseteq S$. It chooses a substate $s \in \tilde{s}$, takes action $a \in A(s)$ and expands to state $\tilde{s}'$ where $\tilde{s}' = \tilde{s} \cup \{s'\}$ and s' is the result of taking action a from state s according to M. We will refer to this choice as the *non-determinism of the agent* in this chapter.

Definition 2 (**Power state**). *A state $\tilde{s}$ is called a power state in a MDP $M = (S, A, T, R)$ if*

- $\tilde{s} = \cup\{s_i\}$ *where* $s_i \in S$.
- $\forall s_i, s_j \in \tilde{s},\ s_i \neq s_j,\ \exists s_k \in \tilde{s}$ *and two trajectories* $\{s_{k_0}, a_{k_1}, s_{k_1} \ldots, a_{k_n}, s_{k_n}\}$ *and* $\{s_{k'_0}, a_{k'_1}, s_{k'_1} \ldots, a_{k'_n}, s_{k'_n}\}$, *such that*
  - $s_{k_0} = s_{k'_0} = s_k$, $s_{k_n} = s_i$ *and* $s_{k'_n} = s_j$.
  - $\prod_t T(s_{k_{t-1}}, a_{k_t}, s_{k_t}) \neq 0$ *and* $\prod_t T(s_{k'_{t-1}}, a_{k'_t}, s_{k'_t}) \neq 0$.

The first condition indicates the power state is composed of some single states in the standard MDP. The second condition guarantees those single states connect with each other: in the simple case, if the non-deterministic agent starts with state s_0 and expands to s_1, from s_1 to s_2 and selects s_1 again and expands to s_3, for the powerset that contains $\{s_0, s_1, s_2, s_3\}$,underlyingly, there are two individual paths from s_0: s_0, s_1, s_2 and s_0, s_1, s_3, we need to make sure those paths exist given a power state. The definition for the power state above ensures that the state in a multi-state MDP is composed of a set of connected states in the MDP. Each of the states in the power state can keep backpointers to its previous states. Figure 3.1 shows an example of non-deterministic state versus deterministic state. For simiplicity, we mark the grid axis at lower left corner (0,0) and upper right corner (4,4) (the first dimension is column and second is row and each grid step +1 on the corresponding axis). For the deterministic agent (left), each grid in the maze is a state and the trajectory shown is (0,0), up, (0,1), up, (0,2), up, (0,3), right, (1,3), right, (2,3), right, (3,3), down, (3,2), down, (3,1), up, (3,2), up, (3,3), right, (4,3), up, (4,4). For the non-deterministic agent (right), the trajectory is composed of a set of powerset, so we describe an **expanding** process of the agent by a pair of (picked single state in the power state, action taken from that state): ((0,0), up), ((0,1), up), ((0,1), right), ((1,1), right), ((1,2), right), ((1,2), down), ((0,2), up), ((0,3), right), ((1,3), right), ((2,3), right), ((3,3), right), ((3,4), up). The states in dark color are the single states in the final power state. Note that state sets such as {(0,0), (1,1)} do not satisfy the second condition in the definition of a power state and thus it will not be in the state space of the multi-state MDP.

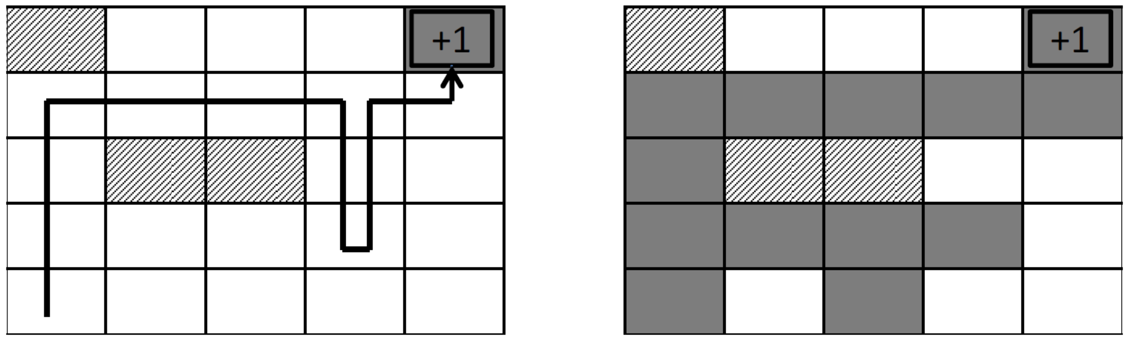

Figure 3.1: A 5×5 grid world example. The cells with diagonal lines are walls. The terminal states are labeled +1. (Right) The full set of dark cells is the final state in a multi-state MDP trajectory. (Left) In contrast, the last state of a trajectory in the standard MDP is the cell at the tip of the arrow below the terminal state.

Define a multi-state MDP as follows:

Definition 3 (Multi-state MDP). *A multi-state MDP (MSMDP)* $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$

on a MDP $M = (S, A, T, R)$ is defined as follows:

- *A state $\tilde{s}$ in $\mathcal{M}$ is a power state in M,*
- *An action $\tilde{a} = (s, a) \in \mathcal{A}(\tilde{s})$, where $s \in S, s \in \tilde{s}, a \in A(s)$,*
- *The transition distribution $\mathcal{T}(\tilde{s} \mid \tilde{a}, \tilde{s}') = T(s \mid a, s')$ where $\tilde{s'} = \{s'\} \cup \tilde{s}$, $\tilde{a} = (s, a)$.*
- *The reward function $\mathcal{R}(\tilde{s} \mid \tilde{a}, \tilde{s}') = R(s \mid a, s')$ where $\tilde{a} = (s, a)$ and $\tilde{s'} = \{s'\} \cup \tilde{s}$.*[1]

Note that in the definition of transition probability in MSMDP, the probability $T(s|\tilde{s})$ cannot be defined according to the standard MDP, so in this chapter, we only consider *model-free (sometimes called direct)* reinforcement learning algorithms that compute the value and policy directly without estimating the model.

Bellman equations for value and Q value: $V(\tilde{s})$ and $Q(\tilde{s}, \tilde{a})$ are defined similarly to the standard MDP:

$$\begin{aligned} V(\tilde{s}) &= \max_{\tilde{a}} \sum_{\tilde{s}'} \mathcal{T}(\tilde{s}, \tilde{a}, \tilde{s}')(\mathcal{R}(\tilde{s}, \tilde{a}, \tilde{s}') + \gamma V(\tilde{s}')) && (3.3) \\ &= \max_{(s,a)} \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V(\tilde{s}')) && (3.4) \end{aligned}$$

and

$$\begin{aligned} Q(\tilde{s}, \tilde{a}) &= \sum_{\tilde{s}'} \mathcal{T}(\tilde{s}, \tilde{a}, \tilde{s}')(\mathcal{R}(\tilde{s}, \tilde{a}, \tilde{s}') + \gamma V(\tilde{s}')) && (3.5) \\ &= \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V(\tilde{s}')) && (3.6) \end{aligned}$$

[1]Transition and reward functions are defined in such a way that the probability of taking a specific single state s in the power state $\tilde{s}$ and taking an action a to expand from the state and adding s' to $\tilde{s}$ to become $\tilde{s}'$ is the same as taking a from s to s'.

If we use $V(\tilde{s}) = \max_s V(s)$ where $s \in \tilde{s}$, in MSMDP, we have

$$V(\tilde{s}) \quad \geq \quad \max_a \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V(s')) \tag{3.7}$$

where $s, s \in S$, $\{s\} \cup \{s'\} = \tilde{s}'$.

We call the agent on MSMDP a non-deterministic agent. The agent starts to expand from a fixed initial state and expands to occupy more states in the standard MDP space. For simplicity, we only consider the case where only *expansion* actions are allowed for the power states: the agent expands to a new state in MDP at each step. So the size of the power set will be increased at least once every $\max_s |A(s)|$ steps.

Definition 4 (**Boundary set**)**.** *The boundary set $B(\tilde{s})$ of a power state $\tilde{s}$ on M is a set of state-action pairs in $(\tilde{s}, \tilde{a})$ that are never explored in a trajectory. Formally,*

- *A history set of* $H(\tilde{s}) = \{\tilde{a} \in \mathcal{A}(\tilde{s}) | \tilde{a} \in \tilde{s}$ *and* $\tilde{a}$ *is already picked by the agent.*$\}$
- $B(\tilde{s}) = \{\tilde{a} = (s,a) \in \mathcal{A}(\tilde{s}) | \exists a \in A(s),\ s.t.\ T(s,a,s') > 0$ *and* $\tilde{a} \notin H(\tilde{s})\}$.
- *Boundary actions* $\tilde{a} = (s,a) \in B(\tilde{s})$.

Instead of allowing all possible actions from the underlying MDP, we restrict the MSMDP to take only the actions from the boundary actions (aka expansion actions). This is a minor restriction: even with it in place, the MSMDP still represents an exponentially large set of possible configurations of the underlying MDP.

Figure 3.2 shows the boundary actions of the gray power state. Same as before, we call the state at lower left corner (0,0) and upper right corner (3,3). Moving horizontally corresponds to the first dimension and vertically the second

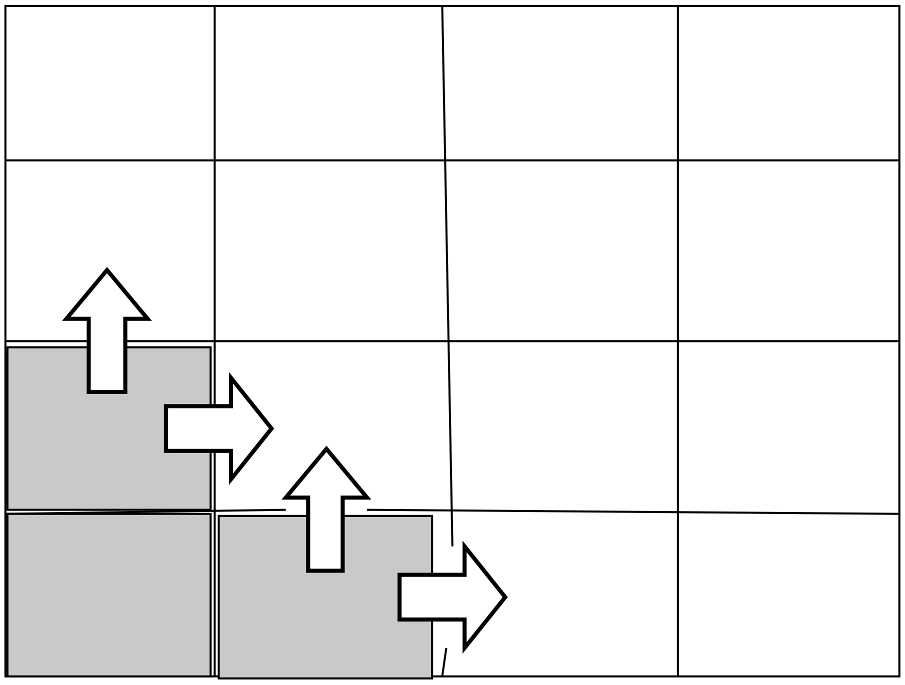

Figure 3.2: Boundary actions of a power state

dimension. In this case, the expanding actions are shown in the arrows. Note that for any determinstic trajectory in the standard MDP, we can find a shorter or equal length non-deterministic trajectory. In the previous example, if the deterministic trajectory is (0,0), up, (0,1), down, (0,0), right (1,0) which is of length 4, by using the non-deterministic agent, it only requires ((0,0), up), ((0,0), right) which is of length 2.

What we will show is that in cases when *we only care about learning the optimal deterministic path and policy* (as opposed to the true value function), we can do so efficiently. Here, the *optimal deterministic path* of a MDP or MSMDP is defined as the sequence of actions from a fixed initial state to a terminal state with maximal reward under the optimal policy. In MSMDP, a terminal state is a power state that contains at least one terminal state in MDP. If the transition function is non-stochastic, then what we don't care about is actions on states that are not visited by the optimal policy. If the transition function is stochastic, we show later in the experiments that a non-determinstic agent converges faster in practice.

Theorem 3.2.1. *In a* ***delayed reward*** *setting*[2]*, for each optimal deterministic path for a standard MDP M which is $p = (s_0, a_0, s_1, a_1, \ldots, s_m, a_m)$ there is an optimal path for a multi-state MDP $\mathcal{M}$ given M is $\tilde{p} = (\tilde{s}_0, \tilde{a}_0, \tilde{s}_1, \tilde{a}_1, \ldots, \tilde{s}_n, \tilde{a}_n)$ such that $\tilde{s}_0 = \{s_0\}$, $\tilde{s}_t = \tilde{s}_{t-1} \cup \{s_t\}$, $\forall i, \tilde{a}_i = (s_i, a_i)$ and $m = n$.*

Proof. Prove by contradiction. Define branches as those paths in a trajectory of a MSMDP that do not lead to a terminal state. (1) There is no branch in $\tilde{p}$. If there is only cost (negative reward) for all the states except the terminal state, pruning those branches in $\tilde{p}$ will increase the accumulated reward. (2) Without branches, an agent in a standard MDP setting can also follow $\tilde{p}$ by the definition of MSMDP. Assume the two paths are different. Choose the one with the higher reward. If it is $\tilde{p}$, construct a new path according to $\tilde{s}_0 = \{s_0\}$, $\tilde{s}_t = \tilde{s}_{t-1} \cup \{s_t\}$. This path will give a higher reward in $\mathcal{M}$, and vice versa. $\square$

By applying theorem 3.2.1 to all the non-terminal states in MDP, their optimal paths in MSMDP are equivalent to those in MDP. Here, the equivalence means that starting from a single state s, the optimal action at s for a non-deterministic agent to expand is the same as the action for a deterministic agent to take.

[2]In a general delayed reward setting, the agent will need to take a long sequence of actions with insignificant rewards until it receives a significant reward. This theorem applies in the case where the immediate rewards for non-terminal states are negative.

3.3 Learning with a Non-deterministic Agent

Theorem 3.2.1 shows that converged results on the MDP and MSMDP will give equivalent optimal paths if running till convergence. Even if the number of power state is exponential to the size of original state space, we are still able to find optimal paths from estimating the value function of the MDP. Performing Bellman updates on this exponential power state space can be expensive. However, we can apply reinforcement learning algorithms with the non-deterministic agent from the MSMDP and find the optimal value and policy for MDP and use MSMDP to answer the question of "what would have happened if we had taken another action at some point back in time."

Algorithm 6 shows a sketch of the general learning framework.[3]

Algorithm 6 Non-deterministic Learning

```
Input: state space $S$, action space $A$, reward $R$, transition function $T$.
Initialize values.
repeat
  repeat
    Choose an action at current power state with non-deterministic agent.
    Update value for certain states in the power state.
    Expand to the new power state.
  until trajectory termination condition is met.
until the convergence criterion is met.
```

[3]We discuss the trajectory termination condition in detail in section 3.3.3.

From now on, we will shift our focus to using the non-deterministic agent to solve MDP more efficiently, not only because the solution to a MSMDP gives the optimal path in the corresponding MDP, but because the agent in MSMDP can make learning on the original MDP converge faster.

This idea of a non-deterministic agent can be applied to most of the trajectory-based reinforcement learning algorithms. In order to obtain a converged and optimal estimate of the value function, one of the most important prerequisites is that no states will starve from updates under exploration [Kaelbling et al., 1996] – if the agent runs forever, every state will be updated infinite times. With a non-deterministic agent, the maximum length of a non-deterministic trajectory is bounded by $|S| \times \max_s |A(s)|$ where $s \in S$. At each step in the MSMDP, the action is chosen from the boundary set — the state-action pairs that have not been updated in the current iteration — with standard exploration policy like ϵ-greedy, these pairs will be updated sufficiently often when the number of iterations goes to infinity. Thus a non-deterministic agent can solve the original MDP exactly.

It turns out that non-deterministic algorithms are more efficient than the original MDP solvers. In fact, **MSMDPs implicitly encode the credit assignment into the trajectories: if there are two competing actions from the same state or two different states, the agent will explore both of them until some future states show up with a lower reward.**

Figure 3.3 shows an example how the credit assignment happens. Algorithmically, assume at iteration t: $Q_t(s, a_1) > Q_t(s, a_2)$ and $V_{t-1}(s_2) > V_{t-1}(s) > V_{t-1}(s_1)$ where s_1 and s_2 are the two following states by taking a_1 and a_2 from state s. Given

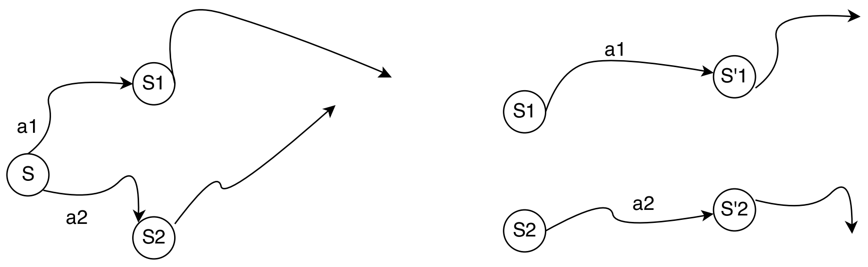

Figure 3.3: Competing actions of states.

a deterministic agent, it will follow a policy that goes from s to s_1 even though s_2 might be potentially better than s_1. On the other hand, a non-deterministic agent will expand from s to s_1 and realise immediately that s_2 has a high value, so it will expand from s to s_1 in the next step. A similar situation happens if there are two competing actions from two different states: $Q_t(s_1, a_1) > Q_t(s_2, a_2) \geq V_{t-1}(s)$ where $s_1, s_2, s \in \tilde{s}$, $s_1 \neq s_2 \neq s$ and for the following states s'_1 and s'_2: $V_{t-1}(s_1) < V_{t-1}(s_2)$. A non-deterministic agent will keep the competition and explore both of the actions until one action turns out later to be worse than the other.

In the experiments (section 3.4), we show that empirically, the non-deterministic framework improves the time to convergence for the learning algorithms.

3.3.1 Non-deterministic Q Learning

In Q-learning [Watkins, 1989], the agent starts a trajectory at some start state s_0, chooses an action a using the policy derived from current Q value with ϵ-greedy exploration, and observes reward r and next state s'. The Q-value is then updated by

$$Q_{t+1}(s_t, a_t) = (1 - \alpha_t(s_t, a_t))Q_t(s_t, a_t) + \alpha_t(s_t, a_t)(r(s_t, a_t, s'_t) + \gamma V_t(s_{t+1})) \quad (3.8)$$

and

$$V_t(s) = \max_a Q_t(s,a) \tag{3.9}$$

Algorithm 7 gives the pseudocode for non-deterministic Q learning. As in the general framework described in the previous section, we can use a non-deterministic agent on a MSMDP to compute a deterministic policy on MDP. The main difference now is how the trajectories are generated.

Algorithm 7 Non-deterministic Q Learning

```
Input: state space $S$, action space $A$, reward $R$, transition function $T$, discount factor $\gamma$, start state $s_0$
Initialize Q-values $Q(s,a)$ arbitrarily and power state $\tilde{s} = \emptyset$.
repeat
    repeat
        With probability $\epsilon$, choose a random state-action pair $\tilde{a} = (s,a) \in B(\tilde{s})$, otherwise choose a state and action pair $\tilde{a} = (s,a)$ from the boundary set $B(\tilde{s})$ under the current policy: $\pi(\tilde{s}) = \arg\max_{(s,a)} Q(s,a)$ where $s \in \tilde{s}$ and $a \in A(s)$.
        Take action $a$ from state $s$ and observe reward $r$, state $s'$.
        Update $Q(s,a) \rightarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma V(s')]$ ($V(s')$ will be also updated correspondingly).
        If $s' \notin \tilde{s}$, add $s'$ into $\tilde{s}$ and add $\{(s',a')\}$ into $B(\tilde{s})$ for all $a' \in A(s')$. Remove $\tilde{a} = (s,a)$ from the boundary set.
    until Trajectory termination condition is satisfied.
until The convergence criterion is met.
```

Definition 5 (Neighbor). *State s_j is a neighbor of state s_i if $\exists a \in A(s_i)$ s.t. $T(s_i, a, s_j) \neq 0$.*

We call a state s in the power state $\tilde{s}$ a *branching state* if it is not the initial state and at least three of its neighbors are in $\tilde{s}$. An initial state is also a branching state if at least two of its neighbors are in $\tilde{s}$. If there is a branching state, this means the current model has over-estimated the value function for some previous states and so early updates should be performed to lower these values. As a result, for those state-action pairs with large uncertainties, we should update the $Q(s, a)$ value (re-propagation) for all seen actions a after the update step in algorithm 7.

The following theorem in [Jaakkola et al., 1994] for the convergence and optimality still hold for the non-deterministic Q-learning algorithms.

Theorem 3.3.1. *The non-deterministic Q-learning algorithm with update*

$$\begin{aligned} Q_{t+1}(s_t, a_t) &= (1 - \alpha_t(s_t, a_t))Q_t(s_t, a_t) \\ &+ \alpha_t(s_t, a_t)(r(s_t, a_t, s'_t) + \gamma V_t(s_{t+1})) \end{aligned} \tag{3.10}$$

converges to the optimal $Q^*(s, a)$ values if

1. *$|S| < \infty$ and $|A| < \infty$.*

2. *$\sum_t \alpha_t(s, a)$ diverges and $\sum_t \alpha_t^2(s, a)$ converges uniformly over s and a with probability one.*

3. *$Var\{r(s, a, s')\} < \infty$.*

4. *If $\gamma = 1$, all policies should end up in a cost free terminal state with probability one.*

The proof is identical to [Jaakkola et al., 1994] as the convergence condition is guaranteed when the exploration policy is non-starving. The boundary set will keep all the unvisited state and action pairs and with ϵ-greedy exploration, the non-deterministic agent will not have any starving state when $t \to \infty$, so the convergence and optimality still hold for learning values and policies on MDP with a non-deterministic agent.

3.3.2 Non-deterministic Queue-Dyna

A set of common model-free methods to take advantage of previously visited states to perform more updates is Dyna-Q [Sutton, 1991, Sutton et al., 2008] and Queue-Dyna [Peng and Williams, 1993, Zajdel, 2008] (See chapter 2.1.2).[4] In Dyna-Q learning, after updating the value for the current observed state and action pair, it chooses some random states and actions in the space and updates the corresponding values. It performs learning and simulation simultaneously. In Queue-Dyna, after updating the current value, the observed neighbors are added into a priority queue for updates. If the value difference between the current value and the previous value is large (an indicator that the current estimate of the value is uncertain), this state is given a higher priority to be updated again. Similarly, our non-deterministic Queue-Dyna algorithm is described in algorithm 8.

The convergence condition is similar to what was discussed in [Li, 2008, Gullapalli and Barto, 1994].

[4]Similar approaches with model estimation are called Prioritized Sweeping [Andre et al., 1998, Moore and Atkeson, 1993b]

Algorithm 8 Non-deterministic Queue-Dyna

```
Input: state space $S$, action space $A$, reward $R$, transition function $T$, discount factor $\gamma$, start state $s_0$, threshold $\delta$, model $M = \emptyset$ and exploration rate $\epsilon$.
Initialize Q-values $Q(s,a) = 0$ and priority queue $PQ = \emptyset$.
repeat
  repeat
    With probability $\epsilon$, choose a random state-action pair $\tilde{a} = (\hat{s}, \hat{a}) \in B(\tilde{s})$, otherwise choose a state and action pair $\tilde{a} = (\hat{s}, \hat{a})$ from the boundary set $B(\tilde{s})$ under the current policy: $\pi(\tilde{s}) = \arg\max_{(\hat{s},\hat{a})} Q(\hat{s}, \hat{a})$ where $\hat{s} \in \tilde{s}$ and $\hat{a} \in A(\hat{s})$.
    Take the action $\hat{a}$ and observe reward $\hat{r}$, state $\hat{s}'$.
    Update $M(\hat{s}, \hat{a}) \leftarrow (\hat{s}', \hat{r})$.
    If $\hat{s}' \notin \tilde{s}$, add $\hat{s}'$ into $\tilde{s}$ and add $\{(\hat{s}', \hat{a}')\}$ into $B(\tilde{s})$ for all $\hat{a}' \in A(\hat{s}')$. Remove $\tilde{a} = (\hat{s}, \hat{a})$ from the boundary set.
    Compute priority $\hat{p} = |\hat{r} + \gamma V(\hat{s}') - Q(\hat{s}, \hat{a})|$. If $\hat{p} >= \delta$, add $(\hat{s}, \hat{a}, \hat{s}', \hat{r})$ into the priority queue with priority $\hat{p}$.
    repeat
      Pop $(s,a)$ from PQ and $(s', r)$ from $M$.
      Update the Q-value $Q(s,a) \rightarrow (1-\alpha)Q(s,a) + \alpha[r' + \gamma V(s')]$
      for All the preceding state-action pairs $(s'', a'')$ s.t. $M(s'', a'') \leftarrow (s, r'')$ do
        compute their priority by $|r'' + \gamma V(s) - Q(s'', a'')|$ and add into the queue if it is $> \delta$.
      end for
    until queue is empty or maximum number of pops achieved
  until Trajectory termination condition is satisfied.
```

Theorem 3.3.2. *The value function for standard MDP converges to the optimal value with non-determinstic agent if the following conditions hold:*

1. *The priority value of state $s_t \in S$ converges to 0 in the limit:*

$$\lim_{t \to \infty} p_t(s_t) = 0 \tag{3.11}$$

2. *For all the states s that are not chosen to be updated infinitely often and T is the last time state s is updated. For $t > T$, there exists a constant C s.t.*

$$p_t(s) \geq C|E(s, V_t)| \tag{3.12}$$

where $E(s, V_t)$ is the Bellman Error:

$$E(s, V_t) = \max_a \left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V_t(s') - V_t(s) \right) \tag{3.13}$$

The first condition makes sure that states with constant non-zero priorities will be updated frequently enough (without starving). The second condition guarantees for any starving states, the priority of the state will not be too small compared to its Bellman error. So these two conditions show that the Bellman error for the starving states should be 0 and its value already converged to the optimal value. In non-deterministic Queue-Dyna (for deterministic transition function as described in algorithm 8): The priority of a state-action pair (s, a)

$$p_t(s) = \max_a p_t(s,a) = \max_a (R(s,a) + \gamma V_t(s') - Q_t(s,a)) \tag{3.14}$$

$$\geq \max_a (R(s,a) + \gamma V_t(s') - V_t(s)) = E(s, V_t) \tag{3.15}$$

so $C = 1$ and the second condition is satisfied. For the first condition, as for the states that are updated frequently enough, it will converge to its optimal value

[Bertsekas and Tsitsiklis, 1989], so when the time goes to infinity, the Bellman error on those states will be zero.

$$\lim_{t\to\infty} |E(s, V_t)| = \lim_{t\to\infty} |\max_a (R(s,a) + \gamma V_t(s') - V_t(s))| = 0 \tag{3.16}$$

So

$$\lim_{t\to\infty} p_t(s,a) \le \lim_{t\to\infty} |\max_a (R(s,a) + \gamma V_t(s') - V_t(s))| = 0 \tag{3.17}$$

So Dyna-Queue (with deterministic or non-deterministic agent) will converge to the optimal value function.

Though algorithms like Q-Dyna or Dyna-Queue will back-propagate the learned values to update the value of the preceding states, they use change in values as priorities. However, what is more important for the learning is the impact of taking different competing actions. The non-deterministic agent allows the exploration of such state-action pairs which makes learning a better policy faster.

3.3.3 Non-deterministic Agent with Overgenerated Trajectories

In previous non-determinstic algorithms, we have not taken advantage of the fact that there are multiple possible competing paths given a trajectory on the multi-state MDP. Normally, a non-deterministic trajectory ends when the agent expands to a terminal state. However, one advantage of using a non-deterministic agent is that *the agent can keep exploring even after it reaches a terminal state.* This is not applicable in the deterministic setting, because there a trajectory ends whenever a terminal state is found. However, for the non-deterministic agent, it is still possible to expand from other previous states in the power state. This is important for addressing the credit assignment problem. We call those trajectories *overgenerated trajectories.*

Assume the non-deterministic agent expands into one terminal state T_1 with reward R_1. When overgeneration for the trajectory is allowed, the agent will keep expanding from another state s' in the power state until terminal state T_2 with reward R_2 is covered in the power state. Assuming it's reached a new terminal state ($T_2 \neq T_1$) with a higher reward ($R_2 > R_1$), the value of the "better" action (the one that led to T_2) will be updated in the next iteration so that it beats the value of the "worse" action (the one that led to T_1) and finally a policy leading to the better terminal states can be learned. Similarly, when $T_1 = T_2$, the overgenerated trajectories will assign credits to the better actions that reach the terminal state.

The overgenerated trajectories can be terminated when the agent visits terminal states a certain number of times or it reaches the maximum length of a trajectory. Moreover, we can decrease the number of visits to the terminal states during learning, so that the agent can focus on exploiting the non-deterministic trajectories when it has more knowledge of the environment.

3.4 Experiments

In the experiments, we will explore some properties of the proposed non-deterministic learning framework over mazes of different size and show their empirical convergence results: non-determinism in general improves the convergence speed, policy accuracy and value. The baselines we use are: standard Q learning (Q), Dyna-Q (Dyna-Q) and Queue-Dyna (Q-Dyna). In the result tables 3.1 and 3.2, prefix ND refers to non-deterministic algorithms and OG to algorithms with overgenerated trajectories. We use a fixed ϵ-greedy exploration where $\epsilon = 0.2$ and learning rate $\alpha = 0.5$[5]. For Dyna-Q and Q-Dyna, the number of backups are set to be 10. The main convergence measures we use are (1) the number of updates (including backups). (2) the total length of the trajectories, which is the same as the total number of states visited (backups are excluded).

We first focus on detailed analysis of a 19×19 maze (fig. 3.4) with two terminal states: one positive (400) and one negative (-10). There is a fixed cost sample from $c \in U[-1, 0]$ for each step taken in the maze except at the terminal states. The negative reward state is easily accessible from the start state. A deterministic agent will get to the negative reward state very fast and without good strategies to update the value in the history, it is very unlikely for it to find the terminal state far away with a higher reward. However, with overgenerated trajectories, the non-deterministic agent has a second chance to look for another terminal state or a better way to reach the closer terminal state. The expected convergence and policy correctness should be better.

Table 3.1 shows the convergence statistics, the learned values and policy accuracy for the start state[6] over 100 runs. Here policy accuracy is measured over all the

[5]This is a standard setup in the literature even though theorem 3.3.1 requires $\sum_t \alpha_t^2 \to 0$ when $t \to \infty$

[6]We actually tried different random samples of step costs and got similar results.

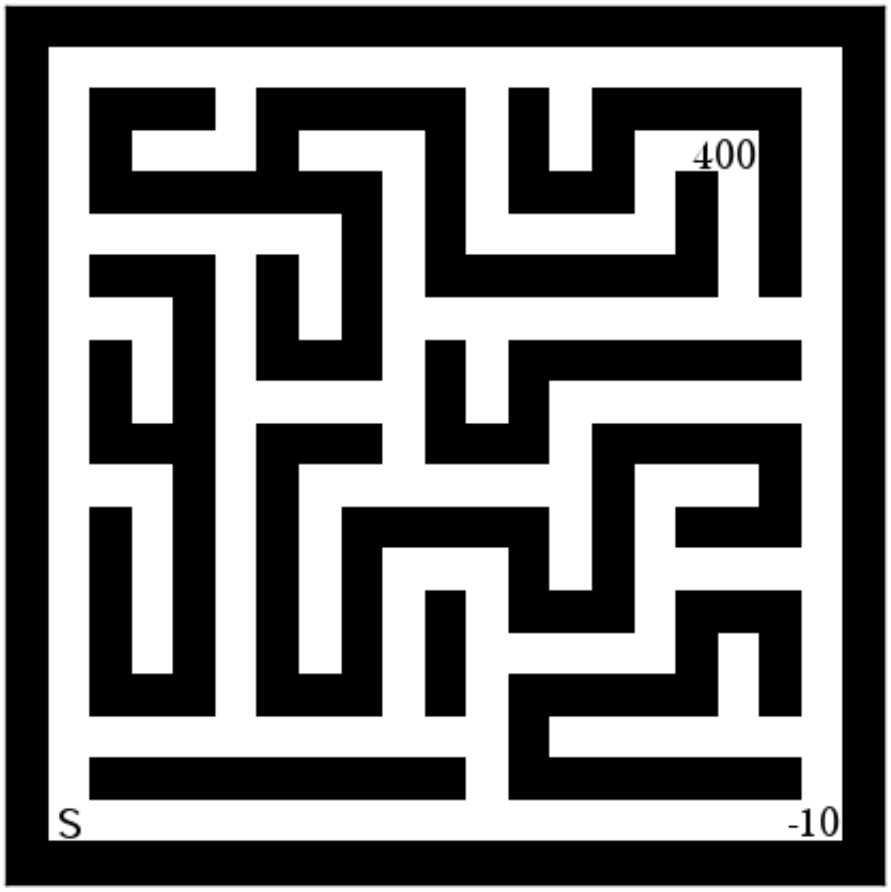

Figure 3.4: The 19 × 19 maze with 2 terminal states. The black cells are the walls. The start state is marked as S.

non-terminal states. Since the maze is not too big, we can compute the value and policy exactly for each state as references for measuring accuracy. We stop learning when the value of states does not change for 3 continuous episodes. [7]

Table 3.1: Convergence and optimality results for 19×19 maze. # Traj: the number of trajectories until convergence. L(Traj) is the total length of trajectories which is the same as the total number of visited states (backups are not included). Updates is the number of Q value updates.

Algorithm	# of traj	L(Traj)	updates ($\times 10^5$)	$V^{\pi_{\text{learned}}}(s_0)$	policy accuracy
Q	106.53	14392	0.1439	0.1217	31.09%
ND-Q	85.74	15063	0.1506	0.1217	32.37%
ND-Q-OG	196.8	123041	1.2304	**293.44**	**86.74%**
Dyna-Q	152.52	8454	0.8454	216.45	81.42%
ND-Dyna-Q	113.43	4967	0.4967	228.03	81.42%
ND-Dyna-Q-OG	22.15	16431	1.6431	**295.65**	**91.68**%
Q-Dyna	272.8	12670	1.268	248.93	99.71%
ND-Q-Dyna	337.97	16333	1.632	248.93	99.62%
ND-Q-Dyna-OG	92.9	25178	2.5161	**295.94**	**100%**

Compared to the standard algorithms, the non-deterministic versions without overgenerated trajectories need to explore roughly the same or smaller amount of states. The learned values for the start state and the final policy accuracy are slightly better than the deterministic version. However, when overgenerated trajectories are allowed, the non-deterministic agent will not stop an episode until it reaches the terminal states twice (which can be the same or different terminal states), it largely increases the learned value for the start state and the final policy accuracy. Moreover, this increase in the learned value not only occurs for the start state, but

[7]We abuse the terminology and refer to this as convergence in the experiments. However, this will cause some states to starve and we will discuss that later.

also for some other non-terminal states.

From the results, it seems that Q-learning converged to a wrong answer, but recall that we terminate the learning when the value does not change for 3 iterations and ϵ-greedy exploration can only guarantee no starving state when $t \to \infty$. Some of the states actually will never be updated enough. For Q-learning, about 33.8% of the non-terminal states starve during the learning. Compared to the non-starving states in Q learning, the values learned by non-deterministic agents increase by 142.96 on average and by 167.87 on average with non-deterministic overgenerated trajectories. For Dyna-Q and Q-Dyna algorithms, the same situation happens.

In order to see the learning curve vs. runtime, we also compare the deterministic and non-deterministic agent with SARSA(λ) algorithm (eligibility trace [Rummery and Niranjan, 1994, Singh and Sutton, 1996]) with $\lambda = 0.9$. SARSA updates the Q value according to the policy:

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha[R(s_t, a_t) + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \qquad (3.18)$$

We try to estimate the value for the initial state as time passes by.

Figure 3.5 shows the learning curves of SARSA and Non-deterministic SARSA (NDSARSA). The y-axis is the value of the path extracted from the initial state to some terminal state. The higher value is the better. If an optimal path cannot be extracted, we set the value to be 0. x-axis is the number of iterations (trajectories). Though at the beginning of learning, SARSA can quickly find a terminal state and update the value along the path from the start state, it does not find the better terminal state. In contrast, NDSARSA spends the beginning episodes to explore the state-action space and finds a better policy later.

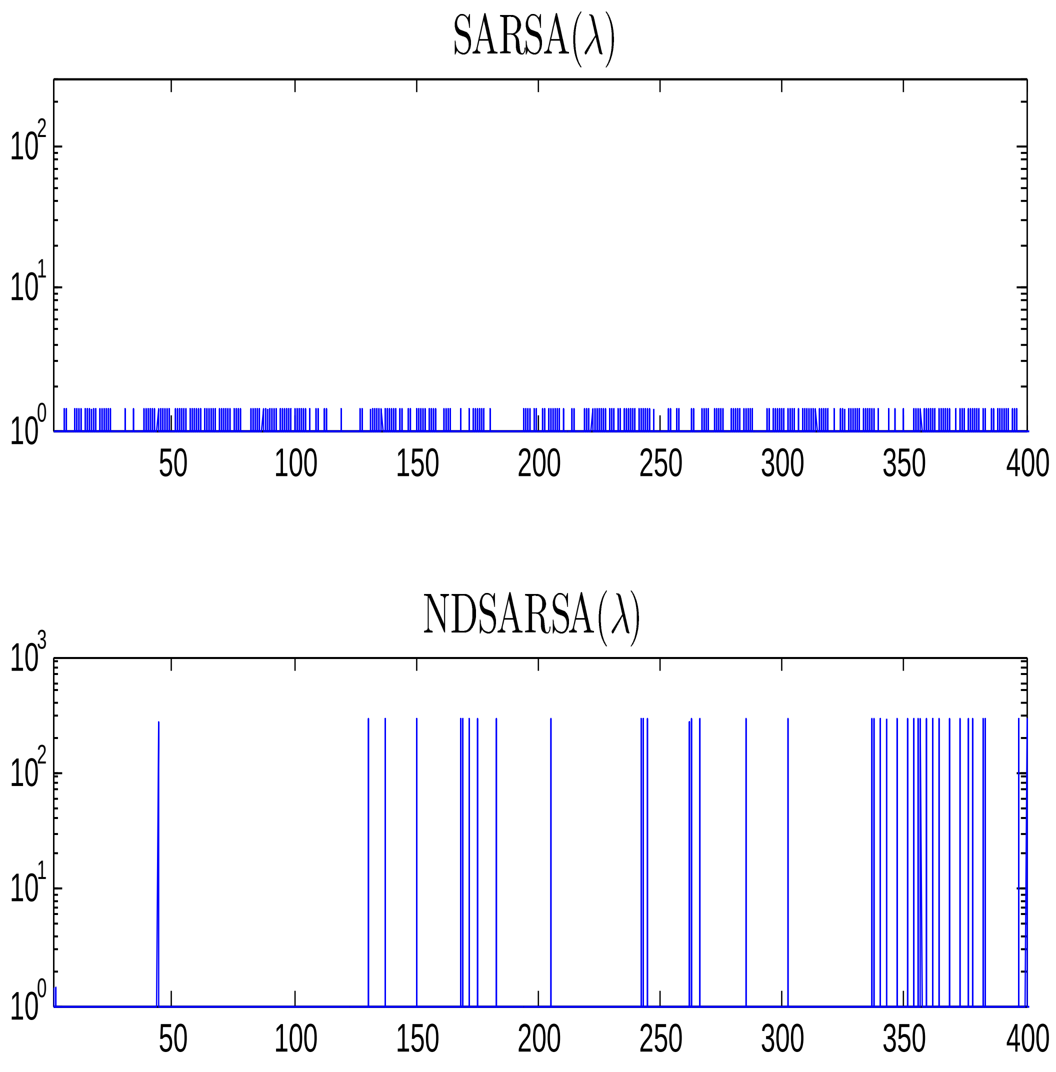

Figure 3.5: Value for the start state during the iterations in online deterministic and non-deterministic SARSA. x-axis: number of iterations. y-axis: value under current policy.

Similarly, we generate larger random mazes. Here we allow the number of overgenerated trajectories to decrease as $\max\{2 - \text{floor}(\log_{100}(\#\text{of traj})), 1\}$ during the learning which improves the convergence. The results are in table 3.2 which shows similar improvements as in the small maze.

Table 3.2: Results on larger mazes of size 900 and 1600. # TRAJ: the number of trajectories until convergence. L(TRAJ): the number of state visits. UPDATES: the number of Q value updates.

SIZE	ALGO	VALUE CONVERGED			POLICY CONVERGED			$V^\pi(s_0)$
		# TRAJ	L(TRAJ)	UPDATES	# TRAJ	L(TRAJ)	UPDATES	
900	Q	898.4	71274.0	71274.0	671.0	61445.2	61445.2	3.85
	ND-Q	184.8	20139.2	20139.2	171.6	19180.4	19180.4	**4.61**
	ND-Q-OG	138.0	14621.2	14621.2	136.8	14602.0	14602.0	**4.61**
	Q-DYNA	152.4	9106.4	100070.4	138.4	8558.0	94138.0	**4.61**
	ND-Q-DYNA	119.0	11070.6	121376.6	113.8	10627.0	116497.0	**4.61**
	ND-Q-DYNA-OG	113.8	11236.4	123209.4	110.8	11025.0	120275.0	**4.61**
1600	Q	1560.0	175232.8	175232.8	1186.0	121154.6	121154.6	0.94
	ND-Q	338.2	54083.0	54083.0	333.0	53579.0	53579.0	1.04
	ND-Q-OG	144.4	31359.4	31359.4	136.75	28970.5	28970.5	**1.34**
	DYNAQ	168.4	16998.4	186982.4	148.2	15887.6	174763.6	1.31
	ND-DYNAQ	197.0	36622.3	402845.3	196.5	37227.0	409497.5	1.31
	Q-DYNA	666.6	44046.0	484306.2	386.0	29473.6	321209.6	1.48
	ND-Q-DYNA	114.8	20905.6	224961.4	113.6	20744.0	226884.3	**1.54**
	ND-Q-DYNA-OG	96.4	17473.6	190209.8	95.8	17418.2	187600.2	**1.54**

3.5 Discussion

In this chapter, we show a non-determinstic learning framework for reinforcement learning algorithms. In a simulated world with a large search space, standard trajectory based reinforment learning algorithms converge slowly. With some additional updates for the value in the past, algorithms with backups like Queue Dyna are proven to be efficient to learn. The non-deterministic agent improves not only on the standard algorithms but also on algorithms with backups in terms of convergence. The overgenerated trajectory with non-deterministic agent implicitly assigns credit to competing actions at each time step – it keeps track of all the possible action candidates and will pick the second best choice back in history if the previous best one is no long competitive. Understanding this non-deterministic agent in reinforcement learning helps us better understand non-determinism in structure prediction (as a search problem).

In the following chapters, we are going to consider another situation for learning with non-determinism — given a non-deterministic algorithm, how can we train heuristics to improve its performance? The heuristics can be deterministic or non-deterministic. In the case of agenda-based parsing, the parsing algorithm is non-deterministic. However, we want to learn some heuristics automatically so that a faster and/or better parse can be obtained using the same non-deterministic algorithm.

Chapter 4: Learned Prioritization for Trading Off Accuracy and Speed

4.1 Overview

The nominal goal of predictive inference is to achieve high accuracy. Unfortunately, high accuracy often comes at the price of slow computation. If you could get an additional 1% accuracy in syntactic parsing, for example, but the parser would take two days to parse a sentence, the result would be significantly weakened. In practice one wants a "reasonable" tradeoff between accuracy and speed. But the definition of "reasonable" varies with the application. Our goal is to optimize a system with respect to a user-specified speed/accuracy tradeoff, on a user-specified data distribution. We formalize our problem in terms of learning priority functions for generic inference algorithms (Section 4.2).

Much research in natural language processing (NLP) has been dedicated to finding speedups for exact or approximate computation in a wide range of inference problems including sequence tagging, constituent parsing, dependency parsing, and machine translation. Many of the speedup strategies in the literature can be expressed as pruning or prioritization heuristics. Prioritization heuristics govern the order in which search actions are taken while pruning heuristics explicitly dictate whether particular actions should be taken at all. Examples of prioritization include

A* [Klein and Manning, 2003] and Hierarchical A* [Pauls and Klein, 2010] heuristics, which, in the case of agenda-based parsing, prioritize parse actions so as to reduce work while maintaining the guarantee that the most likely parse is found. Alternatively, coarse-to-fine pruning [Petrov and Klein, 2007], classifier-based pruning [Roark and Hollingshead, 2008], [Roark et al., 2012] beam-width prediction [Bodenstab et al., 2011], etc can result in even faster inference if a small amount of search error can be tolerated.

Unfortunately, deciding which techniques to use for a specific setting can be difficult: it is impractical to "try everything." In the same way that statistical learning has dramatically improved the *accuracy* of NLP applications, we seek to develop statistical learning technology that can dramatically improve their *speed* while maintaining tolerable accuracy. By combining reinforcement learning and imitation learning methods, we develop an algorithm that can successfully learn such a tradeoff in the context of constituency parsing. Although this chapter focuses on parsing, we expect the approach to transfer to prioritization in other agenda-based algorithms, such as machine translation and residual belief propagation. We give a broader discussion of this setting in [Eisner and Daumé III, 2011].

4.2 Priority-based Inference

Inference algorithms in NLP (e.g. parsers, taggers, or translation systems) as well as more broadly in artificial intelligence (e.g., planners) often rely on prioritized exploration. For concreteness, we describe inference in the context of parsing, though it is well known that this setting captures all the essential structure of a much larger family of "deductive inference" problems [Kay, 1986, Goodman, 1999].

4.2.1 Prioritized Parsing

Given a probabilistic context-free grammar, one approach to inferring the best parse tree for a given sentence is to build the tree from the bottom up by dynamic programming, as in CKY [Younger, 1967]. When a prospective constituent such as "NP from 3 to 8" is built, its *Viterbi inside score* is the log-probability of the best known subparse that matches that description.[1]

A standard extension of the CKY algorithm [Kay, 1986] uses an *agenda*—a priority queue of constituents built so far—to decide which constituent is most promising to extend next, as detailed in section 6.2.4 below. The success of the inference algorithm in terms of speed and accuracy hinge on its ability to prioritize "good" actions before "bad" actions. In our context, a constituent is "good" if it somehow leads to a high accuracy solution, quickly.

Running Example 1. *Either CKY or an agenda-based parser that prioritizes by Viterbi inside score will find the highest-scoring parse. This achieves a percentage accuracy of* 93.3*, given the very large grammar and experimental conditions described in Section 4.6. However, the agenda-based parser is over an order of magnitude faster than CKY (wall clock time) because it stops as soon as it finds a parse, without*

[1]E.g., the maximum log-probability of generating some tree whose fringe is the substring spanning words (3,8], given that NP (noun phrase) is the root nonterminal. This is the total log-probability of rules in the tree.

building further constituents. With mild pruning according to Viterbi inside score, the accuracy remains 93.3 *and the speed triples. With more aggressive pruning, the accuracy drops to* 92.0 *and the speed triples again.*

Our goal is to *learn* a prioritization function that satisfies this condition. In order to operationalize this approach, we need to define the test-time objective function we wish to optimize; we choose a simple linear interpolation of accuracy and speed:

$$\text{quality} = \text{accuracy} - \lambda \times \text{time} \tag{4.1}$$

where we can choose a λ that reflects our true preferences. The goal of λ is to encode "how much more time am I willing to spend to achieve an additional unit of accuracy?" In this chapter, we consider a very simple notion of time: the number of constituents popped from/pushed into the agenda during inference. (A discussion of alternative measures is relegated to Section 4.7.)

When considering how to optimize the expectation of Eq (4.1) over test data, several challenges present themselves. First, this is a *sequential decision process:* the parsing decisions made at a given time may affect both the availability and goodness of future decisions. Second, the parser's total runtime and accuracy on a sentence are unknown until parsing is complete, making this an instance of *delayed reward.* These considerations lead us to formulate this problem as a Markov Decision Process (MDP), a well-studied model of decision processes.

4.2.2 Inference as a Markov Decision Process

A Markov Decision Process (MDP) is a formalization of a memoryless search process. An MDP consists of a *state space* S, an *action space* A, and a *transition function* T. An agent in an MDP observes the current state $s \in S$ and chooses an action $a \in A$. The environment responds by transitioning to a state $s' \in S$, sampled from the transition distribution $T(s' \mid s, a)$. The agent then observes its new state and chooses a new action. An agent's *policy* π describes how the (memory-less) agent chooses an action based on its current state, where π is either a deterministic function of the state (i.e., $\pi(s) \mapsto a$) or a stochastic distribution over actions (i.e., $\pi(a \mid s)$), and by giving the agent a reward $r \in \mathbb{R}$, which is sampled from the MDP's reward distribution $R(r \mid s, a, s')$. The agent's goal is to maximize its total reward over time.

For parsing, the state is the full current chart and agenda (and is astronomically large: roughly 10^{17} states for average sentences). The agent controls which item (constituent) to "pop" from the agenda. The initial state has an agenda consisting of all single-word constituents, and an empty *chart* of previously popped constituents. Possible actions correspond to items currently on the agenda. When the agent chooses to pop item y, the environment *deterministically* adds y to the chart, combines y as licensed by the grammar with adjacent items z in the chart, and places each resulting new item x on the agenda. (Duplicates in the chart or agenda are merged: the one of highest Viterbi inside score is kept.) The only stochasticity is the initial draw of a new sentence to be parsed. The environment also negatively rewards the agent for the time that this took. The agent can alternatively choose a special STOP action, which causes the environment to end the program and return the current best parse (an item of type S that covers the whole input string), if any. Here the environment positively rewards the agent for the accuracy of that parse.[2]

We are interested in learning a deterministic policy that always pops the highest-priority available action. Thus, learning a policy corresponds to learning a priority function. We define the priority of action a in state s as the dot product of a feature vector $\boldsymbol{\phi}(a, s)$ with the weight vector $\boldsymbol{\theta}$; our features are described in Section 6.2.5.1. Formally, our policy is

$$\pi_{\boldsymbol{\theta}}(s) = \arg\max_{a} \ \boldsymbol{\theta} \cdot \boldsymbol{\phi}(a, s) \tag{4.2}$$

An *admissible* policy in the sense of A^* search [Klein and Manning, 2003] would guarantee that we always return the parse of highest Viterbi inside score—but we do not require this, instead aiming to optimize Eq (4.1).

4.2.3 Features for Prioritized Parsing

We use the following simple features to prioritize a possible constituent. (1) Viterbi inside score; (2) constituent touches start of sentence; (3) constituent touches end of sentence; (4) constituent length; (5) $\frac{\text{constituent length}}{\text{sentence length}}$; (6) $\log p(\text{constituent label} \mid \text{prev. word POS tag})$ and $\log p(\text{constituent label} \mid \text{next word POS tag})$, where the part-of-speech (POS) tag of w is taken to be $\arg\max_t p(w \mid t)$ under the grammar;

[2]Thus, all positive reward is delayed till the parse completes. In section 4.3.4 we consider other ways of distributing rewards over time, under training regimens that benefit from early feedback for good actions.

(7) 12 features indicating whether the constituent's {preceding, following, initial} word starts with an {uppercase, lowercase, number, symbol} character; (8) the 5 most positive and 5 most negative punctuation features from [Liang et al., 2008], which consider the placement of punctuation marks within the constituent.

The log-probability features (1), (6) are inspired by work on figures of merit for agenda-based parsing [Caraballo and Charniak, 1998], while case and punctuation patterns (7), (8) are inspired by structure-free parsing [Liang et al., 2008].

4.3 Reinforcement Learning

Reinforcement learning (RL) provides a generic solution to solving learning problems with delayed reward [Sutton and Barto, 1998]. The reward function takes a state of the world s and an agent's chosen action a and returns a real value r that indicates the "immediate reward" the agent receives for taking that action. In general the reward function may be stochastic, but in our case, it is deterministic: $r(s,a) \in \mathbb{R}$. The reward function we consider is: $r(s,a) =$

$$r(s,a) = \begin{cases} \operatorname{acc}(a) - \lambda \cdot \operatorname{time}(s) & \text{if } a \text{ is a full parse tree} \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

Here, $\operatorname{acc}(a)$ measures the accuracy of the full parse tree popped by the action a (against a gold standard) and $\operatorname{time}(s)$ is a user-defined measure of time. In words, when the parser completes parsing, it receives reward given by Eq (4.1); at all other times, it receives no reward.

4.3.1 Boltzmann Exploration

At test time, the transition between states is deterministic: our policy always chooses the action a that has highest priority in the current state s. However, during training, we promote exploration of policy space by running with stochastic policies $\pi_{\boldsymbol{\theta}}(a \mid s)$. Thus, there is some chance of popping a lower-priority action, to find out if it is useful and should be given higher-priority. In particular, we use Boltzmann exploration to construct a stochastic policy with a Gibbs distribution. Our policy is:

$$\pi_{\boldsymbol{\theta}}(a \mid s) = \frac{1}{Z(s)} \exp\left[\frac{1}{\mathit{temp}}\, \boldsymbol{\theta} \cdot \boldsymbol{\phi}(a, s)\right] \text{ with } Z(s) \text{ as the appropriate normalizing constant} \tag{4.4}$$

That is, the log-likelihood of action a at state s is an affine function of its priority. The temperature *temp* controls the amount of exploration. As $\mathit{temp} \to 0$, $\pi_{\boldsymbol{\theta}}$ approaches the deterministic policy in Eq (4.2); as $\mathit{temp} \to \infty$, $\pi_{\boldsymbol{\theta}}$ approaches the uniform distribution over available actions. During training, *temp* can be decreased to shift from exploration to exploitation.

As we have a fixed-horizon episodic task (i.e., the agent does not live forever), a stochastic policy π gives rise to a separate total reward R for each episode, i.e., each time the parser parses a sentence. We want our policy to maximize the *expected total reward*, which is fully defined by π together with the distribution over the random behavior of the environment (transitions and rewards). We can measure this by simulation.

A trajectory τ is the complete sequence of state/action/reward triples from parsing a single sentence. As is common, we denote $\tau = \langle s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_T, a_T, r_T \rangle$, where: s_0 is the starting state; a_t is chosen by the agent by $\pi_{\boldsymbol{\theta}}(a_t \mid s_t)$; $r_t = r(s_t, a_t)$; and s_{t+1} is drawn by the environment from $T(s_{t+1} \mid s_t, a_t)$, deterministically in our case. At a given temperature, the weight vector $\boldsymbol{\theta}$ gives rise to a distribution over trajectories and hence to an expected total reward:

$$R = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} [R(\tau)] = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} \left[\sum_{t=0}^{T} r_t \right]. \tag{4.5}$$

where τ is a random trajectory chosen by policy $\pi_{\boldsymbol{\theta}}$, and r_t is the reward at step t of τ.

4.3.2 Policy Gradient

Given our features, we wish to find parameters that yield the highest possible expected reward. We carry out this optimization using a stochastic gradient ascent algorithm known as policy gradient [Williams, 1992, Sutton et al., 2000]. This operates by taking steps in the direction of $\nabla_{\boldsymbol{\theta}} R$:

$$\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau) = p_{\boldsymbol{\theta}}(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) \tag{4.6}$$

so the gradient of the objective becomes

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}_\tau [R(\tau)] = \mathbb{E}_\tau \left[\frac{\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau)}{p_{\boldsymbol{\theta}}(\tau)} R(\tau) \right] = \mathbb{E}_\tau \Big[R(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) \Big] = \mathbb{E}_\tau \Big[R(\tau) \sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi(a_t \mid s_t) \Big] \tag{4.7}$$

The expectation can be approximated by sampling trajectories. It also requires computing the gradient of each policy decision, which, by Eq (4.4), is:

$$\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) = \frac{1}{temp} \left(\boldsymbol{\phi}(a_t, s_t) - \sum_{a' \in A} \pi_{\boldsymbol{\theta}}(a' \mid s_t) \boldsymbol{\phi}(a', s_t) \right) \tag{4.8}$$

Combining Eq (4.7) and Eq (4.8) gives the form of the gradient with respect to a single trajectory. The policy gradient algorithm samples one trajectory (or several) according to the current $\pi_{\boldsymbol{\theta}}$, and then takes a gradient step according to Eq (4.7). This increases the probability of actions on high-reward trajectories more than actions on low-reward trajectories.

Running Example 2. *The baseline system from Running Example 1 always returns the target parse (the complete parse with maximum Viterbi inside score). This achieves an accuracy of* 93.3 *(percent recall) and speed of* 1.5 *mpops (million pops) on training data. Unfortunately, running policy gradient from this starting point degrades speed* and *accuracy. Training is not practically feasible: even the first pass over 100 training sentences (sampling 5 trajectories per sentence) takes over a day.*

4.3.3 Analysis

One might wonder *why* policy gradient performed so poorly on this problem. One hypothesis is that it is the fault of stochastic gradient descent: the optimization problem was too hard or our step sizes were chosen poorly. To address this, we attempted an experiment where we added a "cheating" feature to the model, which had a value of one for constituents that should be in the final parse, and zero otherwise. Under almost every condition, policy gradient was able to learn a near-optimal policy by placing high weight on this cheating feature.

An alternative hypothesis is overfitting to the training data. However, we were unable to achieve significantly higher accuracy even when evaluating on our training data—indeed, even for a single train/test sentence.

The main difficulty with policy gradient is *credit assignment*: it has no way to determine which actions were "responsible" for a trajectory's reward. Without causal reasoning, we need to sample many trajectories in order to distinguish which actions are reliably associated with higher-reward. This is a significant problem for us, since the average trajectory length of an A_0^* parser on a 15 word sentence is about 30,000 steps, only about 40 of which (less than 0.15%) are actually needed to successfully complete the parse optimally.

4.3.4 Reward Shaping

Like other kinds of machine learning, reinforcement learning must learn which features are responsible for high performance. This can be particularly difficult on long trajectories (e.g. in the parsing case, hundred thousands of pops is typical), since it is not clear which of the many actions contributed positively or negatively to the total reward. It may take a long time to learn the correlation.

However, as in other kinds of learning, we can bias the learner toward likely-good solutions in order to reduce the variance of learning. A simple option in our setting is to exploit the traditional discount parameter γ that is used in *infinite-*

horizon reinforcement learning. When $\gamma < 1$, the learner will more greatly increase the probability of actions that were rapidly followed by a reward. This is a bias toward assuming that temporal proximity implies causality.

A classic approach to attenuating the credit assignment problem when one has some knowledge about the domain is *reward shaping* [Gullapalli and Barto, 1992]. The goal of reward shaping is to heuristically associate portions of the total reward with specific time steps, and to favor actions that are observed to be soon followed by a reward, on the assumption that they caused that reward.

If speed is measured by the number of popped items and accuracy is measured by labeled constituent recall of the first-popped complete parse (compared to the gold-standard parse), one natural way to shape rewards is to give an immediate penalty for the time incurred in performing the action while giving an immediate positive reward for actions that build constituents of the gold parse. Since only some of the correct constituents built may actually make it into the returned tree, we can correct for having "incorrectly" rewarded the others by penalizing the final action. Thus, the shaped reward:

$$\tilde{r}(s,a) = \begin{cases} 1 - \Delta(s,a) - \lambda & \text{if } a \text{ pops a complete parse (causing the parser to halt and return } a\text{)} \\ 1 - \lambda & \text{if } a \text{ pops a labeled constituent that appears in the gold parse} \\ -\lambda & \text{otherwise} \end{cases} \tag{4.9}$$

λ is from Eq (4.1), penalizing the runtime of each step. 1 rewards a correct constituent. The correction $\Delta(s,a)$ is the number of correct constituents popped into the chart of s that were not in the first-popped parse a. It is easy to see that for any trajectory ending in a complete parse, the total shaped and unshaped rewards along a trajectory are equal (i.e. $r(\tau) = \tilde{r}(\tau)$).

We now modify the total reward to use temporal discounting. Let $0 \leq \gamma \leq 1$ be a discount factor. When rewards are discounted over time, the policy gradient becomes

$$\mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[\tilde{R}_\gamma(\tau)] = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}\left[\sum_{t=0}^{T}\left(\sum_{t'=t}^{T} \gamma^{t'-t}\tilde{r}_{t'}\right)\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t)\right] \tag{4.10}$$

where $\tilde{r}_{t'} = \tilde{r}(s_{t'}, a_{t'})$. When $\gamma = 1$, the gradient of the above turns out to be equivalent to Eq (4.7) [Peters and Schaal, 2008, section 3.1], and therefore following the gradient is equivalent to policy gradient. When $\gamma = 0$, the parser gets only immediate reward—and in general, a small γ assigns the credit for a local reward $\tilde{r}_{t'}$ mainly to actions a_t at closely preceding times.

This gradient step can now achieve some credit assignment. If an action is on a good trajectory but occurs *after* most of the useful actions (pops of correct constituents), then it does *not* receive credit for those previously occurring actions. However, if it occurs *before* useful actions, it still does receive credit because we do not know (without additional simulation) whether it was a necessary step toward those actions.

Running Example 3. *Reward shaping helps significantly, but not enough to be competitive. As the parser speeds up, training is about 10 times faster than before. The best setting ($\gamma = 0, \lambda = 10^{-6}$) achieves an accuracy in the mid-70's with* *only about* 0.2 *mpops. No settings were able to achieve higher accuracy.*

4.4 Apprenticeship Learning

In reinforcement learning, an agent interacts with an environment and attempts to learn to maximize its reward by repeating actions that led to high reward in the past. In apprenticeship learning, we assume access to a collection of trajectories taken by an *optimal policy* and attempt to learn to mimic those trajectories. The learner's only goal is to behave like the teacher at every step: it does not have any notion of reward. In contrast, the related task of inverse reinforcement learning/optimal control [Ng and Russell, 2000, Kalman, 1968] (previously called inverse optimal control [Kalman, 1968]) attempts to infer a reward function from the teacher's optimal behavior.

Many algorithms exist for apprenticeship learning. Some of them work by first executing inverse reinforcement learning [Ng and Russell, 2000, Kalman, 1968] to induce a reward function and then feeding this reward function into an off-the-shelf reinforcement learning algorithm like policy gradient to learn an approximately optimal agent [Abbeel and Ng, 2004]. Alternatively, one can directly learn to mimic an optimal demonstrator, without going through the side task of trying to induce its reward function [Daumé III et al., 2009, Ross et al., 2011a].

4.4.1 Oracle Actions

With a teacher to help guide the learning process, we would like to explore more intelligently than Boltzmann exploration, in particular, focusing on high-reward regions of policy space. We introduce *oracle actions* as a guidance for areas to explore.

Ideally, oracle actions should lead to a maximum-reward tree. In training, we will identify oracle actions to be those that build items in the maximum likelihood parse consistent with the gold parse. When multiple oracle actions are available on the agenda, we will break ties according to the priority assigned by the current policy (i.e., we train the learner to choose the oracle action that it currently likes best).

where reward is defined in terms of accuracy *and* speed (Eq (4.3)). However, several oracle actions may be available at once. First, our agenda-based algorithm has some freedom in how it orders its actions: e.g., it can arrive at the optimal tree by building either the subject noun-phrase first or the main verb-phrase first. Second, there may be multiple optimal trees with equal reward: e.g., for state-of-the-art grammars like the ones we use [Petrov and Klein, 2008], the grammar produces fine-grained parses, but accuracy is only measured against the coarse-grained parses provided in labeled data. Thus, two fine-grained parses are equally accurate if they disagree only on latent variables.

There are many different ways to get oracle trees. For simplicity, in the experiments, we use the ground truth of a sentence as the oracle tree (Details in experiment section). Alternatively, if the full set of possible parses of a sentence is accessible, we can find out the exact parse with the best speed-accuracy tradeoff and use that as the oracle tree. Or, we can run the agenda-based parser but with the pushed-back reward as the priority to get the oracle tree. However, these initial choices should not influence the results at the end of reinforcement learning if we gradually increase the ability of stochastic exploration for the parser.

We address this problem by letting the current policy decide which of the many possible oracle actions it likes the best. Practically, this works as follows. During the process of parsing, when an item is pushed onto the agenda, it is *flagged* if it belongs to the maximum-reward parse tree. However, this flag does *not* affect its position on the priority queue. When a learning algorithm requests an oracle action, we pop the highest-priority flagged item. (We implement this as simply two queues: one on which we push all actions and one on which we only push oracle actions as they come). This favors oracle actions that the current policy is already somewhat fond of, somewhat akin to easy-first parsing [Goldberg and Elhadad, 2010]. We let our oracle actions be those that build constituents in the gold parse tree since this tends to result in both fast and accurate parsing in most cases.

In order to use the oracle trees, we build two different agendas: one is the standard for agenda-based parsing and the other for oracle constituents. There two agendas are synchronized, in the sense that if the popped item exists in both of the agendas, it will be removed from them after it is popped. When a constituent is pushed into the standard agenda, it is checked against the oracle tree. If it is present in the oracle tree, it is also pushed into the oracle agenda.

In the following, we will describe two approaches to take the advantage of oracle trees to speed up the training process.

4.4.2 Apprenticeship Learning via Classification

Given a notion of oracle actions, a straightforward approach to policy learning is to simply train a classifier to follow the oracle—a popular approach in incremental parsing [Collins and Roark, 2004a, Charniak, 2010]. Indeed, this serves as the initial iteration of the state-of-the-art apprenticeship learning algorithm, DAgger [Ross et al., 2011a].

Note that, although it was not originally described this way, using a shaped reward has also been used to turn reinforcement learning into classification in the task of learning to follow instructions [Branavan et al., 2009].

We train a classifier as follows. Trajectories are generated by following oracle actions, breaking ties using the initial policy (Viterbi inside score) when multiple oracle actions are available. These trajectories are incredibly short (roughly double the number of words in the sentence). At each step in the trajectory, (s_t, a_t), a classification example is generated, where the action taken by the oracle (a_t) is considered the correct class and all other available actions are considered incorrect. The classifier that we train on these examples is a maximum entropy classifier, so it has *exactly* the same form as the Boltzmann exploration model (Eq (4.4)) but without the temperature control. In fact, the *gradient* of this classifier (Eq (4.11)) is nearly identical to the policy gradient (Eq (4.7)) except that τ is distributed differently and the total reward $R(\tau)$ does not appear: instead of mimicking high-reward trajectories we now try to mimic oracle trajectories.

$$\mathbb{E}_{\tau\sim\pi^*}\left[\sum_{t=0}^{T}\left(\boldsymbol{\phi}(a_t,s_t)-\sum_{a'\in A}\pi_{\boldsymbol{\theta}}(a'\mid s_t)\boldsymbol{\phi}(a',s_t)\right)\right] \tag{4.11}$$

where π^* denotes the oracle policy so a_t is the oracle action. The potential benefit of the classifier-based approach over policy gradient with shaped rewards is increased credit assignment. In policy gradient with reward shaping, an action gets (possibly discounted) credit for all future reward (though no past reward). In the classifier-based approach, it gets credit for exactly whether or not it builds an item that is in the true parse.

Running Example 4. *The classifier-based approach performs only marginally better than policy gradient with shaped rewards. The best accuracy we can obtain is* 76.5 *with* 0.19 *mpops.*

To execute the DAGGER algorithm, we would continue in the next iteration by following the trajectories learned by the classifier and generating new classification examples on those states. Unfortunately, this is not computationally feasible due to the poor quality of the policy learned in the first iteration. Attempting to follow the learned policy essentially tries to build *all* possible constituents licensed by the grammar, which can be prohibitively expensive. We will remedy this in section 4.5.

4.4.3 What's Wrong With Apprenticeship Learning

An obvious practical issue with the classifier-based approach is that it trains the classifier only at states visited by the oracle. This leads to the well-known problem that it is unable to learn to recover from past errors [Daumé III et al., 2009, Ross et al., 2011a, Bagnell, 2005, Xu and Fern, 2007]. Even though our current feature set depends *only* on the action and not on the state, making action scores independent of the current state, there is still an issue since the set of actions to choose from does depend on the state. That is, the classifier is trained to discriminate only among the small set of agenda items available on the oracle trajectory (which are always combinations of *correct* constituents). But the action sets the parser faces at test time are much larger and more diverse. Thus, the notion that the model gets itself stuck in states it has not yet seen before cannot be the only issue.

An additional objection to classifiers is that not all errors are created equal. As far as the classifier-based approach is concerned, an action is either correct (the one taken by the oracle) or incorrect (any other action). Some incorrect actions are more expensive than others, if they create constituents that can be combined in many locally-attractive ways and hence slow the parser down or result in errors. Our classification problem does not distinguish among incorrect actions. The Searn algorithm [Daumé III et al., 2009] would distinguish them by explicitly evaluating the future reward of each possible action (instead of using a teacher) and incorporating this into the classification problem. But explicit evaluation is computationally infeasible in our setting (at each time step, it must roll out a full future trajectory for *each* possible action from the agenda). Policy gradient provides another approach by observing which actions are good or bad across many random trajectories, but recall that we found it impractical as well. We do not further address this problem in this chpater, but in [Eisner and Daumé III, 2011] we suggested explicit causality analysis.

A final issue has to do with the nature of the oracle. Recall that the oracle is "supposed to" choose optimal actions for the given reward. Also recall that our oracle always picks correct constituents. There seems to be a contradiction here: our oracle action selector *ignores* λ, the tradeoff between accuracy and speed, and only focuses on accuracy.

This happens because for any reasonable setting of λ, the optimal thing to do is always to just build the correct tree without building any extra constituents. For fully binary trees, the minimum number of pops required to compute *any* tree is a constant, so regardless of λ, the optimal trajectory is the same. Our trees, of course, occasionally have unary rules. Even so, in order for the oracle to "want" to exclude a unary rule, λ would need to be quite large. In an otherwise-binary tree, the exclusion of a unary rule will lose one point in recall. In order for it to be worth not popping this unary constituent, λ would have to be one! Even for trees that are not otherwise binary, the oracle trajectories only change for values of λ on the order of 0.1 or greater.

However, at test time, a λ of 0.1 or greater is completely unreasonable. Even attempting to overfit the data as much as possible (by training and testing on a *single* sentence at a time), the smallest number of pops we are able to achieve is, on average, $15k$ (compared to the oracle, which takes 30 pops). Even at this (unrealistically good) number of pops, having a value of λ more than about 10^{-3} would mean that the best thing the parser could do is give up immediately (which our parser is not allowed to do).

This means that under the apprenticeship learning setting, we are actually *never* going to be able to learn to tradeoff accuracy and speed: as far as the oracle is concerned, you can easily achieve both! The only reason the tradeoff appears is because our model cannot come remotely close to mimicking the oracle. In fact, if we add a cheating feature (akin to Section 4.3.3), the model *is* almost always able to learn to mimic the oracle and once again λ does not matter.

4.5 Oracle-Infused Policy Gradient

The failure of both standard reinforcement learning algorithms and standard apprenticeship learning algorithms on our problem leads us to develop a new approach. We start with the policy gradient algorithm (Section 4.3.2) and use ideas from apprenticeship learning to improve it. Our formulation preserves the reinforcement learning flavor of our overall setting, which involves delayed reward for a *known* reward function.

Our approach is specifically designed for the non-deterministic nature of the agenda-based parsing setting [Eisner and Daumé III, 2011]: once some action a becomes available (appears on the agenda), it never goes away until it is taken. This makes the notion of "interleaving" oracle actions with policy actions both feasible and sensible. Like policy gradient, we draw trajectories from a policy and take gradient steps that favor actions with high reward under reward shaping. Like Searn and DAgger, we begin by exploring the space around the optimal policy and slowly explore out from there.

To achieve this, we define the notion of an *oracle-infused policy.* Let π be an arbitrary policy and let $\delta \in [0, 1]$. We define the oracle-infused policy π_δ^+ as follows:

$$\pi_\delta^+(a \mid s) = \delta\pi^*(a \mid s) + (1 - \delta)\pi(a \mid s) \tag{4.12}$$

In other words, when choosing an action, π_δ^+ explores the policy space with probability $1 - \delta$ (according to its current model), but with probability δ, we force it to take an oracle action.

Our algorithm takes policy gradient steps with reward shaping (Eqs (4.10) and (4.8)), but with respect to trajectories drawn from π_δ^+ rather than π. If $\delta = 0$, it reduces to policy gradient, with reward shaping if $\gamma < 1$ and immediate reward if $\gamma = 0$. For $\delta = 1$, the $\gamma = 0$ case reduces to the classifier-based approach with π^* (which in turn breaks ties by choosing the best action under π).

Similar to DAgger and Searn, we do not stay at $\delta = 1$, but wean our learner off the oracle supervision as it starts to find a good policy π that imitates the classifier reasonably well. We use $\delta = 0.8^{\text{epoch}}$, where epoch is the total number of passes made through the training set at that point (so $\delta = 0.8^0 = 1$ on the initial pass). Over time, $\delta \to 0$, so that eventually we are training the policy to do well on the same distribution of states that it will pass through at test time (as in policy gradient). With intermediate values of δ (and $\gamma \approx 1$), an iteration behaves similarly to an iteration of Searn, except that it "rolls out" the consequences of an action chosen randomly from (4.12) instead of evaluating all possible actions in parallel.

Running Example 5. *Oracle-infusion gives a competitive speed and accuracy tradeoff. A typical result is* 91.2 *with* 0.68 *mpops.*

4.6 Experiments

All of our experiments (including those discussed earlier) are based on the Wall Street Journal portion of the Penn Treebank [Marcus et al., 1993a]. We use a probabilistic context-free grammar with 370,396 rules—enough to make the baseline system accurate but slow. We obtained it as a latent-variable grammar [Matsuzaki et al., 2005] using 5 split-merge iterations [Petrov and Klein, 2007] on sections 2–20 of the Treebank, reserving section 22 for learning the parameters of our policy. All approaches to trading off speed and accuracy are trained on section 22; in particular, for the running example and Section 4.6.2, the same 100 sentences of at most 15 words from that section were used for training and test. We measure accuracy in terms of labeled recall (including preterminals) and measure speed in terms of the number of pops from on the agenda. The limitation to relatively short sentences is purely for improved efficiency at training time.

4.6.1 Baseline Approaches

Our baseline approaches trade off speed and accuracy not by learning to prioritize, but by varying the pruning level Δ. A constituent is pruned if its Viterbi inside score is more than Δ worse than that of some other constituent that covers the same substring.

Our baselines are: **(HA*)** a Hierarchical A* parser [Pauls and Klein, 2009] with the same pruning threshold at each hierarchy level; **($\mathbf{A}_0^*$)** an A* parser with a 0 heuristic function plus pruning; **($\mathbf{IDA}_0^*$)** an iterative deepening A* algorithm, on which a failure to find any parse causes us to increase Δ and try again with less aggressive pruning (note that this is not the traditional meaning of IDA*); and **(CTF)** the default coarse-to-fine parser in the Berkeley parser [Petrov and Klein, 2007]. Several of these algorithms can make multiple passes, in which case the runtime (number of pops) is assessed *cumulatively*.

4.6.2 Learned Prioritization Approaches

Table 4.1: Performance on 100 sentences on dev set.

Model	# of pops	Recall	F1
A_0^* (no pruning)	1496080	93.34	93.19
`D-`	686641	56.35	58.74
`I-`	187403	76.48	76.92
`D+`	1275292	84.17	83.38
`I+`	682540	91.16	91.33

We explored four variants of our oracle-infused policy gradient with with $\lambda = 10^{-6}$. Figure 4.1 shows the result on the 100 training sentences. The "-" tests are the degenerate case of $\delta = 1$, or apprenticeship learning (section 4.4.2), while the "+" tests use $\delta = 0.8^{\text{epoch}}$ as recommended in section 4.5. Temperature matters for the "+" tests and we use *temp* $= 1$. We performed stochastic gradient descent for 25 passes over the data, sampling 5 trajectories in a row for each sentence (when $\delta < 1$ so that trajectories are random).

We can see that the classifier-based approaches "-" perform poorly: when training trajectories consist of *only* oracle actions, learning is severely biased. Yet we saw in section 4.3.2 that without *any* help from the oracle actions, we suffer from such large variance in the training trajectories that performance degrades rapidly and learning does not converge even after days of training. Our “oracle-infused” compromise “+” uses *some* oracle actions: after several passes through the data, the parser learns to make good decisions without help from the oracle.

The other axis of variation is that the “**D**” tests (delayed reward) use $\gamma = 1$, while the “**I**” tests (immediate reward) use $\gamma = 0$. Note that **I+** attempts a form of credit assignment and works better than **D+**.[3] We were not able to get better results with intermediate values of γ, presumably because this crudely assigns credit for a reward (correct constituent) to the actions that closely preceded it, whereas in our agenda-based parser, the causes of the reward (correct subconstituents) related actions may have happened much earlier [Eisner and Daumé III, 2011].

Our final evaluation is on the held-out test set (length-limited sentences from Section 23). A 5-split grammar trained on section 2-21 is used. Given our previous results in Table 4.1, we only consider the `I+` model: immediate reward with oracle infusion. To investigate trading off speed and accuracy, we learn and then evaluate a policy for each of several settings of the tradeoff parameter: λ. We train our policy using sentences of at most 15 words from Section 22 and evaluate the learned policy

[3]The **D-** and **I-** approaches are quite similar to each other. Both train on oracle trajectories where all actions receive a reward of $1 - \lambda$, and simply try to make these oracle actions probable. However, **D-** trains more aggressively on long trajectories, since (4.10) implies that it weights a given training action by $T - t + 1$, the number of future actions on that trajectory. The difference between **D+** and **I+** is more interesting because the trajectory includes non-oracle actions as well.

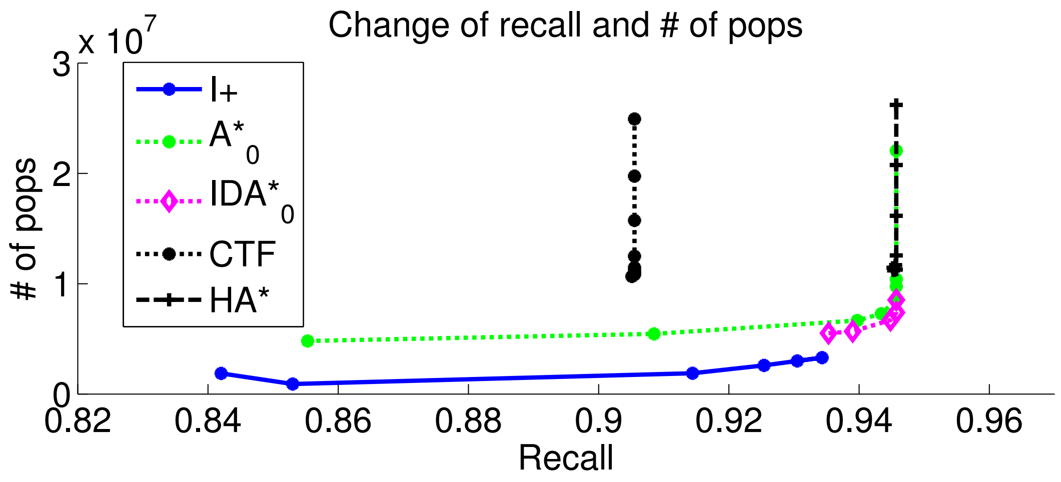

Figure 4.1: Pareto frontiers: Our `I+` parser at different values of λ, against the baselines at different pruning levels.

on the held out data (from Section 23). We measure accuracy as labeled constituent recall and evaluate speed in terms of the number of pops (or pushes) performed on the agenda.

Figure 4.1 shows the baselines at different pruning thresholds as well as the performance of our policies trained using `I+` for $\lambda \in \{10^{-3}, 10^{-4}, \ldots, 10^{-8}\}$, using agenda pops as the measure of time. `I+` is about 3 times as fast as unpruned $\mathbf{A}_0^*$ at the cost of about 1% drop in accuracy (F-score from 94.58 to 93.56). Thus, `I+` achieves the same accuracy as the pruned version of $\mathbf{A}_0^*$ while still being twice as fast. `I+` also improves upon **HA*** and $\mathbf{IDA}_0^*$ with respect to speed at 60% of the pops. `I+` always does better than the coarse-to-fine parser (**CTF**) in terms of **both** speed and accuracy, though using the number of agenda pops as our measure of speed puts both of our hierarchical baselines at a disadvantage.

We also ran experiments using the number of agenda *pushes* as a more accurate measure of time, again sweeping over settings of λ. Since our reward shaping was crafted with agenda pops in mind, perhaps it is not surprising that learning performs relatively poorly in this setting. Still, we do manage to learn to trade off speed and accuracy. With a 1% drop in recall (F-score from 94.58 to 93.54), we speed up from A_0^* by a factor of 4 (from around 8 billion pushes to 2 billion). Note that known pruning methods could also be employed in conjunction with learned prioritization.

4.7 Conclusions and Future Work

In this chapter, we considered the application of both reinforcement learning and apprenticeship learning to prioritize search in a way that is sensitive to a user-defined tradeoff between speed and accuracy. We found that a novel oracle-infused variant of the policy gradient algorithm for reinforcement learning is effective for learning a fast and accurate parser with only a simple set of features. In addition, we uncovered many properties of this problem that separate it from more standard learning scenarios, and designed experiments to determine the *reasons* off-the-shelf learning algorithms fail.

An important avenue for future work is to consider better credit assignment. We are also very interested in designing richer feature sets, including "dynamic" features that depend on both the action *and* the state of the chart and agenda. One role for dynamic features is to decide when to halt. The parser might decide to continue working past the first complete parse, or give up (returning a partial or default parse) before any complete parse is found.

Chapter 5: Learned Dynamic Pruning for Agenda-based Parser

5.1 Overview

While prioritization heuristics govern the order in which search actions are taken, pruning heuristics explicitly dictate whether particular actions should be taken at all. Pruning is widely used in all kinds of search based algorithms to reduce the search space.

In the case of syntactic parsing, pruning is extremely important for parsing algorithms to scale to large grammars and long sentences. The parsing accuracy increases with a rich grammar: [Petrov et al., 2006] increases the size of the grammar from a few thousand rules to half a million rules and boosts the F-1 score from $\sim 75\%$ to $\sim 90\%$. Though the standard CKY algorithm only takes $O(n^3)$ running time in theory where n is the length of the sentence, it is more realistic to factor in the size of the grammar $|G|$ explicitly due to its size in natural language parsing, and the complexity of the CKY algorithm becomes $O(|G|n^3)$.

In this case, the grammar constant $|G|$ plays an important role in parsing – to parse a set of sentences with length less than or equal to 40 words, $|G| \geq 40^3$, so the size of grammar $|G|$ has a large impact on the real parsing time. Approximate inference methods like [Charniak et al., 2006, Petrov and Klein, 2007, Weiss et al., 2012] prune the large search space based on an accurate (but slow) model to speed up the decoding.

For agenda-based parsers, the pruning can be applied to chart cells as well as agenda candidates (constituents). Coarse grained methods prune multiple cells or constituents at each step. Fine grained methods only consider one cell candidate or one constituent each time. For example, a simple fine grained pruning heuristic is to compare the difference in score between the best candidate in the cell so far and the current candidate. New candidates with scores underperforming the best candidate by more than a certain threshold get pruned. For coarse grained pruning, [Roark and Hollingshead, 2008] tags the sentence with a part-of-speech tagger and trains an averaged perceptron to prune (close) the chart cells before actually parsing the sentence. Starting or ending a non-terminal at certain positions is not allowed after pruning. [Roark and Hollingshead, 2008] also shows the complexity of the chart parsing algorithm can be reduced from $O(n^3)$ to $O(n^2)$ or even approximately $O(n)$. Moreover, [Bodenstab et al., 2011] applies beam-width prediction to limit the potential search space for each cell in the chart.

In this chapter, we will focus on automatically learning a fine grained pruning strategy to make inference faster. In contrast with the focus of re-prioritization, the goal here is to decode the sentences as fast as possible without losing too much on accuracy. In contrast with [Roark and Hollingshead, 2008] and [Bodenstab et al., 2011], we do not learn a pruner to prune the chart before parsing the sentence. We instead aim to learn a pruner that prunes while parsing the sentences. More specifically, we learn a pruner for chart cells as well as constituents on the agenda during parsing. Using both a beam-width threshold to restrict the number of elements in the cell as a cell pruner and the difference of inside score of the current candidate and the best candidate as a constituent pruner is a simple case (without learning the heuristics) for our setting. We learn pruning decisions for a cell when the elements are updated in the cell and for a constituent whether it should be added to the chart.

From now on, we call the pruner designated to prune cells in the *agenda-based parser* as the *cell pruner* or *cell classifier* and the pruner to prune individual constituents as the *constituent pruner* or *constituent classifier*. In the agenda-based parser, a cell is updated when a constituent is popped from the agenda and filled into the corresponding cell in the chart. The cell pruner is learned such that every time an open cell in the chart is updated, it will decide if the cell can continue to accept more constituents (open) or no more (close). If the cell is already closed, the cell pruner will not re-classify it and mark it for re-opening later. The constituent pruner only applies when removing constituents from the agenda and when updating the chart. If the constituent is pruned, it will be gone immediately without any cell updates. For more details of the algorithm, please refer to section 5.3.3 and algorithm 9. Both of the pruners are applied at parsing time – unlike the beam-width prediction or cell complete closure approach (background section 5.2.2) whose pruning is done before parsing and no pruning is required during parsing.

In the experiments, we show that compared with different baselines, the learned pruners can be comparable with some state-of-the-art approaches in speed and accuracy.

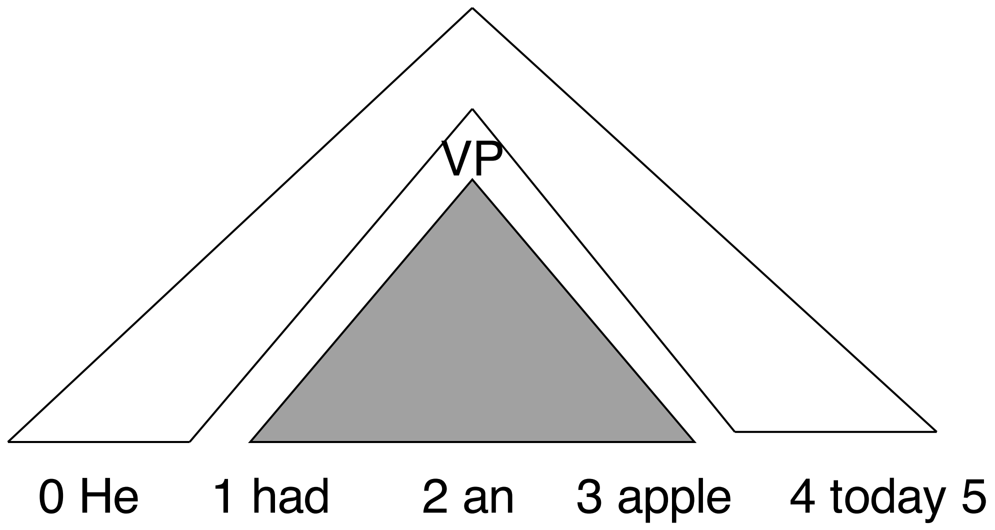

Figure 5.1: Grey area is the inside tree and white area is the outside tree of non-terminal NP[1,3].

5.2 Background

5.2.1 Agenda-based Parser with Figure-of-merit

One way to speed up agenda-based parsing is to compute a better search heuristic for constituents instead of using estimated inside scores from the grammar. Figure-of-merit is an estimated score of the product of inside and outside score. We use figure 5.1 to demonstrate the definition of inside, outside and figure-of-merit scores.

Each word $w[i, j]$ in a sentence of length n is labeled by its span – from i to j. Each edge/symbol is also associated with a span, in the figure, NP[1,4] is a noun phrase spanning from 1 to 4. The inside score of an edge $N[i, j]$ (subtree) is defined as

$$\beta(N[i, j]) = P(w[i, j]|N[i, j]) \tag{5.1}$$

e.g. $\beta(N[1,4]) = P(had\ an\ apple|NP[1,4])$ which is the likelihood of the word sequence given the non-terminal (in the grey area of figure). On the other hand, the outside score estimates how likely a sequence is if the subsequence covered by inside tree of a sentence is replaced by the edge. For agenda-based parsing, inside score is usually given with the grammar.

$$\alpha(N[i,j]) = P(w[0,i], N[i,j], w[j,n]) \tag{5.2}$$

where n is the length of the sentence. In the figure, $\alpha(NP[1,4]) = P(he[0,1], NP[1,4], today[4,5])$. The outside score is not available directly from the grammar and there is a lot of work done in the field to give good estimates. Figure-of-merit estimates the merit of the edge in the final tree as the product of the inside and outside score:

$$FOM(N[i,j]) = \alpha(N[i,j])\beta(N[i,j]) \tag{5.3}$$

In this chapter, we will use the same FOM as what is implemented in [Bodenstab et al., 2011, 2010] for comparison (which uses a modified version of [Caraballo and Charniak, 1998]). This Boundary FOM approximates the outside score of an edge with POS forward-backward scores and transition probabilities of part-of-speech to nonterminal (constituent) boundary. By using this FOM as a heuristic, an agenda-based parser can be more than 20 times faster compared to using inside score only.

5.2.2 Cell Closing Algorithms and Beam-width prediction

The dynamic pruner that we propose shares some commonality with different cell closure methods and beam-width prediction methods from [Roark and Hollingshead, 2008, Bodenstab et al., 2011]. [Roark and Hollingshead, 2008] uses so-called

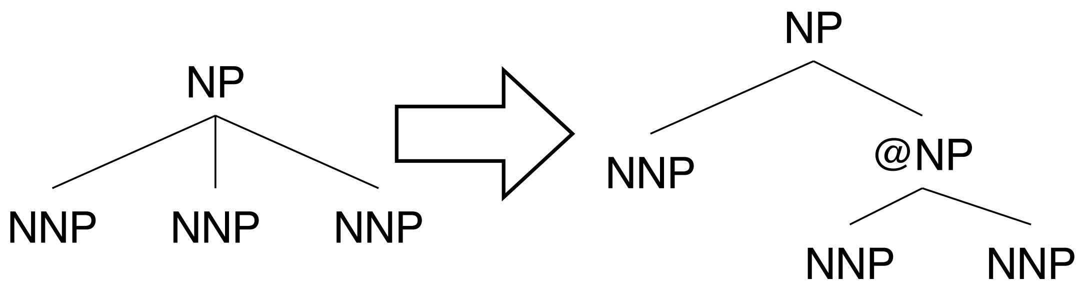

Figure 5.2: Left: a non-binary parse tree in the training data. Right: a binarized tree with partial constituent @NP.

chart contraints that train two classifiers to decide if any multiword edge (constituent) can start from a word or end at a word. They convert training sentences to part-of-speech and lexical features and train the classifiers.

[Bodenstab et al., 2011] propose three different cell closing algorithms: consituent closure, complete closure and beam-width prediction. The constituent closure algorithm learns a classifier to open or close cells for full consitutents while complete closure makes the same decision for any constituent, partial or full. Partial constituent here refers to a set of special factored nonterminals generated when binarizing the grammar. In a given training set, a parse tree is not necessarily a binary tree. Without giving too much detail, figure 5.2 shows an example of a non-binary tree and the tree after binarization[1]. The grammar is already trained on the input with the same binarization.

Constitutent or complete closure trains a binary classifier for whether the cell contains a full or partial edge while beam-width prediction trains a classifier to tell

[1]There are many ways of binarizing a tree, we just show one example of what a binarized tree looks like and what the partial constituent looks like here.

how many constituents one cell can have. It is done by using a few binary classifiers to classify if the cell is open for 0, 1, 2, 3, 4 constituents, etc.

In order to train those classifiers, [Roark and Hollingshead, 2008, Bodenstab et al., 2011] use an averaged perceptron with an asymetric loss function to penalize false negatives during training – if some cells that should be open got pruned, the overall accuracy would definitely be hurt.

The major differences of our approach in this chapter are:

1. We prune both at cell and constituent level.
2. The pruning is done dynamically at runtime, **not** before parsing.
3. The feature set we use contains not only static features, but also dynamic/trace features.

5.3 Constituent and Cell Pruner

As mentioned in the previous sections, we aim to learn two different classifiers at training time, one for constituents and one for cells. Before introducing the training methods for those pruners, we first show how and when we prune with those classifiers.

5.3.1 Constituent-based Pruning

In agenda-based parsing, the chart changes when a constituent is popped from the agenda. The agenda is re-organized when new constituents are built and pushed.

Pruning can be involved at pop time or push time or both. Most state-of-the-art parsers do not use exhaustive parsing. Either the first popped tree is extracted or additional overparsing is allowed [Hall and Johnson, 2003, Charniak and Johnson, 2005]. In these cases, the number of constituents pushed is greater than the number of those popped. Given a grammar of around 500,000 rules and the Viterbi inside score as the order of the agenda, when the first tree is popped, less than 1% of the constituents pushed into the agenda are processed and for a sentence of length 15, more than 20,000 items are already popped. So fewer pruning decisions are required to be made at pop time than at push time. Even if cheating is allowed (each pruning decision is made by looking up the ground truth) pruning at push time has a more significant delay on the wallclock time speed.

As a result, a constituent classifier is learned at pop time. If a constituent is marked to be pruned by the classifier, the constituent will not be added into the chart and the next constituent at the top of the agenda will be popped. This allows the parser to avoid the extra computation of combining with adjacent constituents and significantly reduces the number of candidates to be pushed when the number of pops is large.

5.3.2 Cell-based Pruning

There are two different situations where one cell can be pruned. The first one is before parsing happens. This is exactly the same as what complete closure does. However, we do not try to make this decision before parsing, we only need to check if a cell is open right before when we start to fill in constituents. Another chance to close a cell will be after when the cell is updated, i.e. when a constituent is just added to the cell, the classifier can make a decision for whether to close the cell to additional constituents. If a cell is closed before adding any constituent, the cell will remain empty during parsing and any candidate of that cell will be removed directly. When a cell is closed after being populated with some constituents, the constituents in the cell will still be available in the later steps, but no new consituents can be added to the cell.

5.3.3 Constituent and Cell Dynamic Pruning Algorithm

By combining the cell pruner and constituent pruner at test time, the parser works as follows: when a constituent is popped from the agenda, if the target cell of the constituent is never considered before, the cell pruner will be run and decide if the cell will be open or closed. If the cell is closed, no further action is required. The current constituent will be discarded and the next one on top of the agenda will be popped.

If the cell pruner keeps the cell open, the constituent pruner will classify if the constituent is pruned or not, if the constituent is pruned, then the parser moves on to the next candidate. Otherwise, the constituent is filled into the chart. After that, the cell classifier will be run again and check if it is ok to close the cell after adding this constituent. The added constituent will combine with its neighbors and add more candidates to the agenda.

Algorithm 9 Classifier-based Cell and Constituent Pruner at Test Time

```
Initialize the classifier.
while Agenda is not empty and no tree is returned do
  1. Dequeue a constituent from the agenda, say $(Y, i, j) \rightarrow 75$
  2. If cell has never been visited before, classify whether the cell is open. Otherwise check whether cell is open. If the cell is closed, discard the constituent and go to 1.
  3. Predict if $(Y, i, j)$ should be pruned. If it should be pruned: Skip the following and go back to step 1. If not, continue to step 4.
  4. Update the chart if necessary.
  5. Run the cell classifier again and check if the cell can be closed now.
  for each constituent adjacent to $(Y, i, j)$, such as $(Z, j, k)$ do
    for each grammar rule, e.g. $X \rightarrow Y\ Z$, that combines $(Y, i, j)$ with $(Z, j, k)$ do
      a. Let $new \leftarrow chart[Y, i, j] + chart[Z, j, k] + score(X \rightarrow YZ)$.
      b. Enqueue new constituent $(X, i, k)$ with priority $new$.
    end for
  end for
end while
Return the first complete parse that is popped without overparsing.
```

Algorithm 9 shows a sketch of the test algorithm.

5.4 Training classifiers

In order to train a cell and constituent pruner, we must define the three key components for any supervised learning algorithm: (1) training examples (2) gold labels/ground truth (3) a classification algorithm.

5.4.1 Ground truth and oracles

The ideal pruner will be one that keeps only gold constituents and prunes everything else. This is nearly impossible, but a good pruner should at least mimic this behavior as much as possible. So how do we decide if a constituent is "gold" or not or if the cell should be open or not? Here, trees that are used to label constituents and cells are called oracles.

Two very obvious choices are according to the Viterbi tree and the gold tree. The Viterbi tree is the most likely tree (tree with maximum likelihood) under the given grammar while the gold tree gives the best accuracy. The Viterbi tree requires that we parse the training set once ahead of time with the inside score as priority which might be relatively slow to obtain, but it can be applied to the training set without the ground truth and can be adapted to other domains.

On the other hand, gold trees are hard to get: the training data can be unlabeled or some sentences might not be re-constructible under nonsmooth grammars or the gold trees can be very unlikely given the grammar. With a properly smoothed grammar, it is possible to extract some trees that are compatible with gold trees.

We now show how to find a labeled tree for a sentence that is compatible with its gold tree. Assume we are given a grammar G, a sentence $w_1, \ldots, w_n$ with length n and a gold tree T=$\{N[i,j] \mid i,j \in [0,n]\}$. Here we represent the gold tree as a collection of constituents / edges from the gold tree. For latent grammars like the Berkeley grammar, each nonterminal is associated with a latent tag while in the ground truth, there is no such tag, e.g. a grammar rule will look like NP_3 $->$ DET_0 NN_3 with latent tags but the gold tree looks like (ROOT (S (PR He) (VP (V has) (NP (DET a) (NN cat))))). (For a real example of latent grammar and ground truth, please refer to Appendix B). The extraction process is as follows: we take an agenda-based parser with inside score as the priority for the agenda and parse the sentence without pruning. (This is a UCS parser which we use as a baseline in Chapter 4 and later in this Chapter.) We parse the sentence with the UCS parser, but only allow the constituents that are compatible with the gold tree to be filled into the chart. For example, in the previous example, we allow NP_? $->$ DET_? NN_? with arbitrary latent tags as long as the nonterminals/preterminals in the grammar match the gold tree. We then extract the first complete tree with the UCS parser under this restriction. We call this tree most probable compatible tree. The most probable compatible trees are not necessarily gold: it might skip some unary rules to make the trees more likely under the grammar. (For examples of latent grammar and gold tree, please refer to Appendix B).

Once we pick one type of oracle, either most probable compatible trees or Viterbi trees, the ground truth of a cell is easy to define: if the cell contains all the edges that it should have to get the ground truth tree, the cell should be closed. Otherwise, it should remain open.

5.4.2 Discussion: Dynamic Oracle v.s. Fixed Oracle

One question one might ask is what if the oracle action gets pruned by the pruner during learning. More specifically, shall we change the current oracle if the constituents that should be kept get pruned? Our simple answer here is that a "dynamic oracle" is not necessary in the parsing case. This sounds counter-intuitive at first glance since if the current oracle items are pruned, it might be better to find the second best oracle in the original setting and continue with that. In the cases of robot control or driving cars, this is exactly the case. However, in parsing, the goal is not to successfully finish the parsing process, but to obtain a tree with high accuracy.

If one oracle item is pruned, the second best option to continue will be finding a tree that is closest to the gold tree without that pruned item. If the grammar is large enough: almost every non-terminal can combine with others, we can safely assume that the next oracle item can be constructed in some other way that does not contain the pruned item. So the trajectory after pruning will not change too much. It is possible that the previous kept gold constituents will not be used in the final tree, but keeping those around is the best choice with respect to accuracy. When the rest of the oracle constituents cannot be achieved after pruning, for example, in rule VP $\rightarrow$ NP VP, if the right child VP is pruned by mistake and NP is the current right child, rule VP $\rightarrow$ NP NP does not exist in the grammar, so it is not possible to build the remaining oracles.

It is possible that once an oracle item is pruned during training, another parser can be run from that point to obtain the second best tree. If the learned non-zero weight feature sets are disjoint for different oracles, using dynamic oracles is a good choice. Assume the true objective is to predict a one-sided branching parse tree: left-branching tree or right-branching tree. The training data is a set of parse trees that are either left-branching or right-branching. Two indicator features are used to mark whether the constituent touches the beginning or the end of the sentence. By learning towards this oracle, the learner increases the weights on both features and results in one-sided trees for most of the sentences.

However, when the learned feature sets are not disjoint for different oracles, dynamic features can be troublesome: We use the same left-branching and right-branching tree example here, but now, our feature set contains two features: one is still the indicator feature of whether the constituents touch the beginning of the sentence or not, but the other is now the length of the span. By alternating the two features, the right branching tree oracle can only add weights on the length of the span while the left-branching tree oracle adds more weight to the beginning-of-sentence indicator. The resulting learner now is very likely to produce neither a left-branching nor right-branching tree in this case.

We run preliminary experiments with pruners with dynamic oracles on real parse data where we observe that the pruners actually get confused and produces a worse model compared to a fixed oracle in terms of both speed and accuracy (lost more than 10% in accuracy and no faster) given our feature set. So in our experiments, we only use a fixed oracle for each sentence.

5.4.3 Training examples

Gathering training examples for the cell classifier and constituent classifier is not trivial here as the examples for the cell classifier are strongly correlated with the behavior of the constituent classifier and vice versa. Here is a simplified example. Assume we are collecting training examples for the cell pruner for a cell C=[NP, VP, PP] and the only feature we use here is the number of elements in the cell. Also assume VP here is in the ground truth. If there is no constituent pruner, we can generate four examples for the cell pruner: { empty_cell: (0, open), NP: (1, open), VP: (2, close), PP: (3, close) } and the cell pruner can learn to close after 2 constituents in the cell. However, if there is a constituent classifier to be learned at the same time. The constituent pruner may decide to prune or not to prune the NP before the cell pruning happens and this results in another set of training examples: { empty_cell: (0, open), VP: (1, close), PP: (2, close) }. Learning on these two sets of examples for different iterations in the training will confuse the learner.

On the other hand, ideally, what we want are two classifiers that can mimic what perfect pruners do: if a popped constituent is not gold, it is pruned. If after the cell is updated, all the gold edges are in the cell, then the cell should be closed. However, these perfect pruners are not always easy to achieve. If we use the perfect pruners to generate examples, during test time, if there are some examples cannot be classified perfectly, the data from the training set and test set will be generated from two different distributions: training from perfect pruners and test from non-perfect pruners. So it is hard for the learned pruners to guarantee a good performance on the test set.

In order to tackle these two problems, we propose two different training algorithms. The first option is iterative training. Assume we start with an initial cell pruner (it will also work out when starting with a constituent classifier). For iteration $2i-1$ where $i \geq 1$, we fix the cell pruner and train a constituent pruner based on parsing results with the fixed cell pruner only. For iteration $2i$, we then fix the constituent pruner as what we learned from the previous iteration and train a cell pruner. This learning procedure is not guaranteed to converge to a pair of classifiers that is globally optimal, but after training for a few iterations, the examples for the cell pruner are generated by the learned constituent pruner which we will use during test time and vice versa.

The second option is joint training. Algorithm 10 describes this training process. Let us see what exactly we want to fix in the second problem. Assume our classifiers after training T iterations is h_T, we hope the classifiers in iteration $T-1$ will be close to h_T. Recall DAgger[Ross et al., 2011a] (section 2.3.2) is an imitation learning algorithm that encourages learning from past errors. For each iteration, it aggregates instances by running a hybrid classifier that combines a previously learned classifier with the ground truth (or oracles as we called them in the previous chapter). Then the classifier is re-trained on the aggregated dataset from the beginning until this iteration.

Algorithm 10 DAgger joint learning for dynamic pruners

```
Initialize oracle probability $p_0$, cell classifier $e$, training set $\mathcal{D}_e = \emptyset$ and oracle cell classifier
$h_{e-oracle}$, constituent classifier $o$, training set $\mathcal{D}_o = \emptyset$ and its oracle classifier $h_{o-oracle}$.
for iteration $t = 1, \ldots, n$ do
    Set the current hybrid cell classifier $e'_t = (1 - p_t) \times e_{t-1} + p_t \times h_{e-oracle}$.
    Set the current hybrid const classifier $o'_t = (1 - p_t) \times o_{t-1} + p_t \times h_{o-oracle}$.
    for each training sentence $i$ do
        while Agenda is not empty and no tree is returned do
            1. Dequeue a constituent from the agenda, say $x[j,k]$.
            2. Add this constituent and its oracle label to the constituent dataset $\mathcal{D}_o = \mathcal{D}_o \cup \{x[j,k]\}$.
            3. If the cell is never visited, generate the features for the cell $c[j,k]$ and add its oracle
            label to the cell dataset $\mathcal{D}_e = \mathcal{D}_e \cup \{c[j,k]\}$.
            4. If the cell is closed, discard $x[j,k]$ and back to step 1.
            5. Predict the pruning action by $y_{x[j,k]} = o_t(x[j,k])$.
            if $y_{x[j,k]}$ =prune then
                Return to step 1.
            end if
            if $y_{x[j,k]}$ =keep then
                Add $x[j,k]$ to the chart and push the new constituents to the agenda according to
                the grammar.
                6. generate the features for the cell $c[j,k]$ and add its oracle label to the cell dataset
                $\mathcal{D}_e = \mathcal{D}_e \cup \{c[j,k]\}$.
                7. Predict the cell closing by $e_t$.
            end if
        end while
    end for
    Train new cell and constituent classifier $e_t$ and $o_t$ based on $\mathcal{D}_e$ and $\mathcal{D}_o$ and decrease the oracle
    probability to $p_{t+1} = C\exp(-f(i,t))$ where $C$ is a constant and $f(i,t) > 0$ is a monotonic
    function with sentence number $i$ and iteration number $t$.
end for

Return $e_n$ and $o_n$.
```

In this work, we use DAgger with the perfect pruner as the initial classifier to train the pruner. The initial classifier is set to prune everything that is not gold. In each training iteration, a hybrid classifier of oracle and classifier learned from previous iterations is used to make decisions. A new data instance (x_i, l_i) is added to the dataset where x_i is the set of features of the constituent at the top of the agenda or the cell features and l_i is its label according to the oracle. The data instance is pruned or not pruned according to the hybrid classifier. When an iteration is complete, two new classifiers will be trained on the two accumulated datasets.

5.4.4 Classification algorithm

Pruning a correct constituent can drastically reduce the accuracy with cascade effects; keeping incorrect candidates around will add to the parsing time without too much impact on accuracy. During agenda-based parsing, most of the constituents on the agenda or chart will not be used to build the final tree, so the set of pruned constituents is larger than that of kept constituents by at least three orders of magnitude given a reasonable size of the grammar. For a 20-word sentence, it has only a little more than 40 gold (positive) constituents while the agenda-based parser will pop more than 60,000 constituents from the agenda. It is possible for the pruner to decide to prune all the constituents and achieve an (unweighted) accuracy of 99.93% for the classification, but return no parse tree. For a balanced dataset, predicting all the examples to be in a single class should return an accuracy of 50%.

If the accuracy is weighted, by simple math, the ratio of the weights on positive examples and negative examples should be around 1500 : 1.

Given this skewed and cost-sensitive dataset in the training, assume the cost for pruning a constituent that should be kept (positive class) is C_{keep} and the cost for retaining a constituent that should be pruned (negative class) is C_{prune}. We choose the costs given n_{keep}, n_{prune} constituents from each class such that

$$C_{keep}/n_{keep} = C_{prune}/n_{prune} \tag{5.4}$$

If the pruner is trained by a support vector machine (SVM) with soft margin, for each instance x_i, we want to find the weight vector such that

$$\min_{w} \frac{1}{2} \mid w \mid ^2 + C \sum_i \max(0, 1 - y_i(w^T x_i + b)) \tag{5.5}$$

where w is the weight vector, y_i is the gold label for x_i and b is a bias. The total misclassification error $\lambda \sum_{all} \xi_i$ can be decomposed into $\lambda_{prune} \sum_{prune} \xi_{prune} + \lambda_{keep} \sum_{keep} \xi_{keep}$. The soft margin parameters $\lambda_{prune} = \frac{1}{C_{prune}}$ and $\lambda_{keep} = \frac{1}{C_{keep}}$. This reweighting scheme is applied in LibSVM [Chang and Lin, 2011], PyML and etc. [Ben-Hur and Weston, 2010]

[Roark and Hollingshead, 2008, Bodenstab et al., 2011] similarly define an asymmetric loss function for cell pruning with 100 for the false negative and 1 for the false positive with averaged percetron without data rebalancing.

In the experiments, when comparing with Bubsparser [Bodenstab et al., 2011], we will use its implementation of an averaged perceptron. For the study with constituent only pruner, we use the LibSVM package (from WEKA) to try out different algorithms.[2]

5.5 Features

In this work, we consider two sets of features: static features and dynamic features. The static features are decided whenever the constituent is built and they will not change in the learning process. The static features we use are listed below. They are mostly identical to the previous chapter. The dynamic features monitor the real-time condition of the agenda and chart and change at each time step. In this disseration, we have about 60-70 different static feature templates and about 10-20 different dynamic feature templates. For some of the feature templates, the size can be large: e.g. for lexical feature templates, the unigram feature is of the size of the vocabulary and unigram feature is the square of that.

In the following, we use e (edge) as the constituent whose features are going to be extracted. $S(e)$ is the starting position of e in the sentence and $E(e)$ is the ending position. $L(e)$ is the length of the constituent and $N_c(e) = L(e)+1$ is used as a normalization constant for inside scores: for a constituent of length $l > 1$, if only binary rules are used, it requires a total of $l+1$ rules. L_s is the length of the current sentence. Some of the features are computed on the run, such as Viterbi inside scores, figure of merit (FOM), etc. Values for conditional likelihood features for all tags are approximated from the training data, precomputed and hashed before

[2]We use WEKA here to try out different classification systems and pick LibSVM due to its speed and accuracy. We also try other classifiers like random forest. It is fast but prunes very conservatively to achieve a good accuracy.

training, so they can be retrieved very quickly.

The word at position i is w_i.

5.5.1 Static Features

Values of static features do not change during parsing.

1. Lexical and POS features: we take the unigram and bigram of lexicons and POS tags from [Roark and Hollingshead, 2008, Bodenstab et al., 2011].

2. Position features:

   - Ratio between the length of the constituent and the length of the sentence: $L(e)/L_s$.

   - The starting/ending position of the span: We use the relative position of the starting or ending position for the span here given the length of the sentence, $S(e)/L_s$ and $E(e)/L_s$.

   - Binary indicators for whether the constituents touch the beginning or end of the sentence.

   - Punctuation patterns of the constituent span: We use the same type of punctuation patterns as in [Liang et al., 2008]. The punctuation patterns only retain the punctuation tokens in a sentence and replace non-punctuation tokens with a constant: if the token is punctuation matching `$, . `` '' -LRB- -RRB- : #`, it is retained. Otherwise a token is replaced with "x". In the experiments, we only use indicator features for the 25 top rated, most likely patterns for a nonterminal and 19 bottom rated, most unlikely patterns. For the list of punctuation patterns used in the experiments, please refer to Appendix A.

- Case patterns. The case pattern of a word here is defined as the case of the first letter in the word. This feature indicates whether the first letter of the word is in the form of uppercase, lowercase, number or punctuation. We compute the case pattern of the first word in the span and the words just before/after the span.
- Number of times ROOT is popped from the agenda. This is supposed to be a dynamic feature during parsing, however, we only return the first complete tree, so it is actually an indicator for whether the constituent is ROOT.

3. Conditional likelihood features:

   - The log likelihood of the span given the part-of-speech tags for the previous or following word. The conditional probabilities are computed and hashed according to the ground truth dataset that the grammar is trained on. And only coarse labels are used to be memory efficient. The span labels include pre-terminals and non-terminals. When using the constituent boundary FOM as priority, we need to train a POS tagger ahead of time, so we use the tags of words in the sentences tagged by this learned tagger. When using inside score as priority, instead of using the most likely tag sequence for the sentence built so far (which can be very expensive), we only take the top three most likely tags for the cell and take the highest conditional log likelihood.

- Similar to the previous one, but the previous or following two tags are used. We also use the conditional likelihood depending on the previous and following tag.
- The maximum log likelihood of the span $[i, j]$ given its neighboring constituents that end at position i or start from position j. If multiple neighbors are found, we use the maximum log likelihood from all the possible neighbors.

5.5.2 Dynamic/Trace Features

Here is the list of dynamic features that we use in the experiments:

1. Scores:

    - Viterbi inside score $V(e)$ which is a sum of the log-likelihood of all the grammar rules used to build the constituent e.
    - Normalized Viterbi inside score which is $V(e)/N_c(e)$.

2. Margins:

    $V^*(i, j)$ indicates the maximal Viterbi inside score in the cell starting at i and ending at j. Similar for $F^*(i, j)$.

    - Margin between the best candidates with inside score in the cell and the current constituent: $V^*(S(e), E(e)) - V(e))$. This computes the difference in the inside score between the current candidate and the up-to-date best candidate in the same cell.

  - Normalized inside margin: margin normalized by the number of rules: $(V^*(S(e), E(e)) - V(e))/(L(e) + 1)$.

  - FOM margin: $F^*(S(e), E(e)) - F(e)$.

3. Crossing bracket feature. We use additional array to keep track of the current best competitor for each cell. This feature is one of our most expensive features in our experiments.

  - The margin between the normalized inside score of the current span and the maximal normalized inside score of all the constituents in the chart that cross brackets with the span.

  - The margin between the FOM of current span and maximal FOM of its crossing bracket competitors.

4. Cell features:

  - Last time step (as indexed by number of pops) when the cell is updated. More specifically, we use the difference between the current time and the time when the cell is updated.

  - Relative rank of the constituent with regard to all the current constituents in the cell.

  - Number of constituents in the same cell.

5.6 Experiment

In the experiments, we first study how the parser works with only one pruner in section 5.6.1. As cell pruning is somewhat well studied in Bubsparser [Bodenstab et al., 2011], we only focus on learning a constituent parser. In fact, if we train a cell pruner (only) with lexical and POS features (which are the same from Bubsparser), relative and absolute length of the span, this is identical to training the complete closure method in Bubsparser. Next, in section 5.6.3, we will train our dynamic constituent and cell parser at the same time and show results compared to state-of-the-art parsers.

We use the Wall Street Journal section of the Penn Tree Bank[Marcus et al., 1993b]. The grammar we use is the level-6 grammar[Petrov et al., 2006] trained on section 2-21. Section 22 is used to train the dynamic pruner, section 24 is the development set and section 23 is the final test set. For the first part, we train and test on sentences with length less than or equal to 15 with Berkeley parser. For the second part, we use the sentences with length less than or equal to 25 with Bubsparser.

5.6.1 Constituent pruner only

For training the pruner at pop time, we use LibLinear [Fan et al., 2008] with weight 1000 on the positive examples (We pick the weight given tuning on the dev set) and 1 on the negative examples.

Training on the whole development section multiple times is time-consuming, so we divide the training section into sets of 10 sentences and re-train the classifier every 10 sentences. The time measures in the experiments are the number of pops and number of pushes. We measure the time consumed to compute the features for each edge and the average difference in the wall clock time for building a constituent with and without features is less than $O(10^{-5})$ seconds (which is less than 10% of the total parsing time.[3]).

The first set of baselines we compare to are the agenda parser with inside score as priority and pruning on difference of inside scores (UCS), A* parser (* parser) with different pruning thresholds. [4]

The parameter for choosing an oracle classifier at iteration t is defined as $p_t = exp(-(t-1))$, so the probability of choosing an oracle action decreases exponentially during training. We call the models learned according to Viterbi trees as LDP-V and gold trees as LDP-G.

The baselines we compared to in this part are implemented in the Berkeley parser. Due to the implementation efficiency restriction, we use all sets of features excluding the lexical and POS features. While computing conditional likelihood features, we use the top 3 locally best candidates in the adjacent cells to avoid heavy computation overhead.

NLDP model is a model trained with examples generated by agenda-based

[3]This is just an analysis with constituent pruning only. We will compare to a state-of-the-art parser with both constituent and cell pruning in the next part.

[4]For the A* parser, we use a coarse grammar with at 2 labels for each non-terminal or pre-terminal (Berkeley level 1 grammar) to parse the sentence and estimate outside scores.

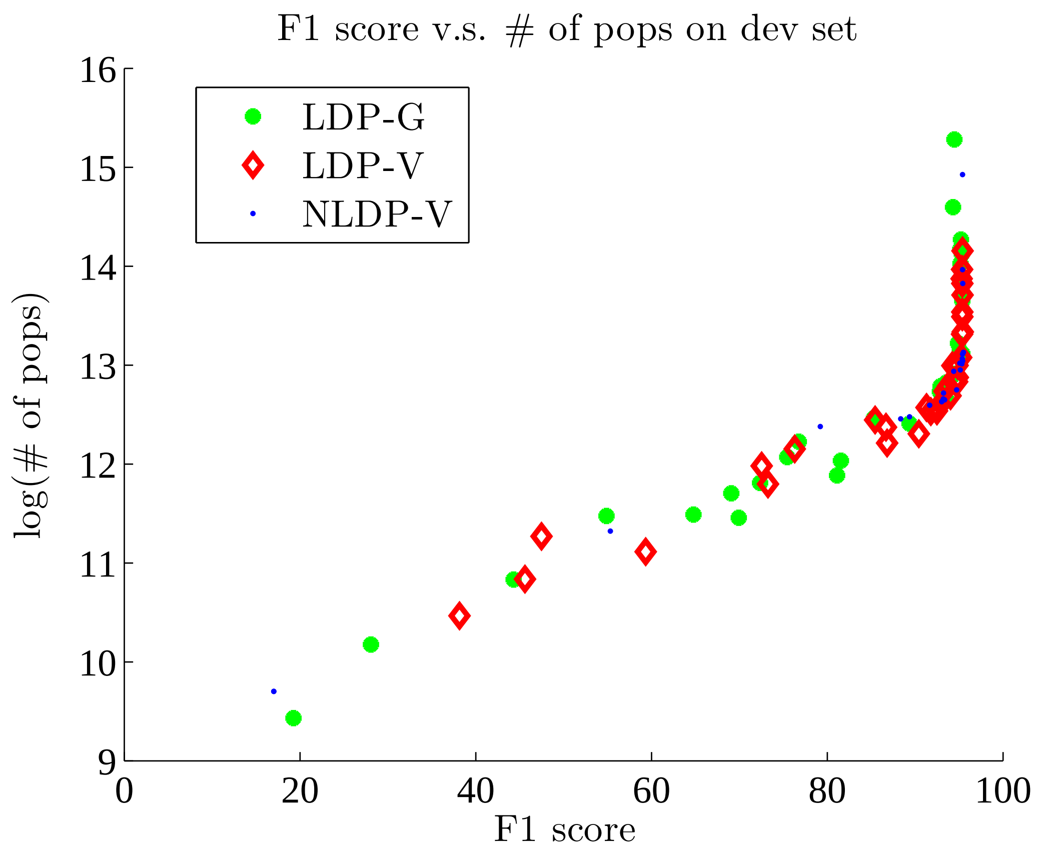

Figure 5.3: The graph of accuracy vs. number of pops on dev set. “LDP-V” is the dynamic pruning model with Viterbi trees as reference and ‘LDP-G” is the one with the most probable compatible trees as reference. “NLDP-V” does not use data aggregation (DAgger).

parsing without pruning and then labeling the examples with oracles. The other models LDP-V and LDP-G are trained with the DAgger algorithm with constituent pruning only (Algorithm 10). LDP-V uses the Viterbi tree as ground truth / oracle while LDP-G uses most probable compatible trees. Each DAgger iteration, we train on a batch of 10 sentences and pick the final model on the dev set with desired speed and accuracy. Figure 5.4 plots all the models learned by LDP-V, LDP-G and NLDP-V.

With data aggregation, the learned models are spread out on a pareto frontier.

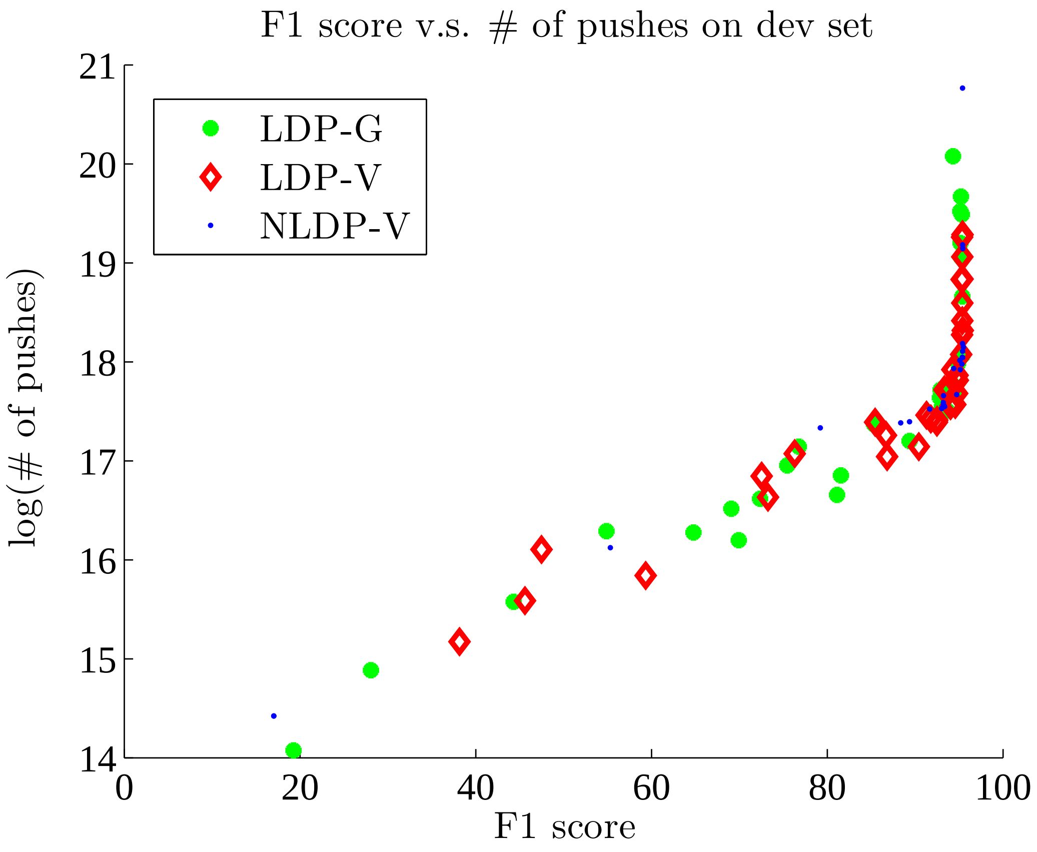

Figure 5.4: The graph of accuracy vs. number of pushes on dev set. "LDP-V" is the dynamic pruning model with Viterbi trees as reference and 'LDP-G" is the one with the most probable compatible trees as reference. "NLDP-V" does not use data aggregation (DAgger).

The reason there is such a pareto frontier curve is due to the limit of the consitutent pruner. If it is possible to learn a really good pruner, the pruner prefers a fast and accurate parser with no such trade-off. Without that, the only difference among the learned models is the size of the training data and most of the models fit the training set well. However, in order to choose a single model in this situation, the definition of a "best" pruner is vague: a good pruner should be both fast and accurate. So we pick a set of trade-off parameters λ and measure the efficiency of a model by

$$\textsc{model efficiency} = \text{F1 } \textsc{score} - \lambda C \times \# \textsc{ of pushes} \tag{5.6}$$

and plot the curve with increasing λ.

Figure 5.5 and 5.6 show the accuracy vs. speed plots with respect to the number of pops and number of pushes. Ideally, a fast and accurate parser will show up in the bottom right corner while a slow but accurate one will appear in the upper right corner.

NLDP-V without data aggregation overfits the training data and does not perform as well on the test set. We showed the models that use both Viterbi trees (LDP-V) and most probable compatible trees (LDP-G). Since dynamic pruning is done at pop time, our model does a much better job when measuring number of pops (figure 5.5) compared with measuring the number of pushes (figure 5.6) (Note that those numbers in the graph are in log-scale). Compared with LDP-G, LDP-V does slightly better in terms of speed. When doing the model selection on the development set, LDP-V produces more models with a decent accuracy (F1>

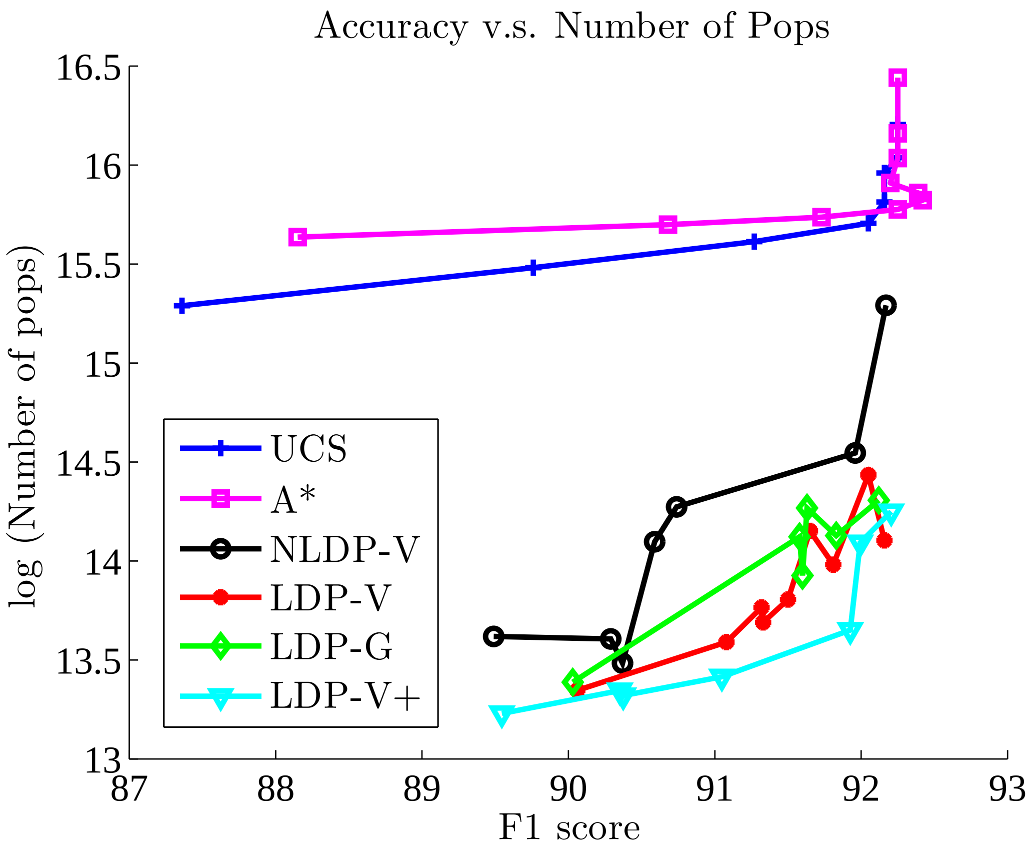

Figure 5.5: The graph of accuracy vs. number of pops on final test set with different trade-off. "UCS" is the standard agenda-based parser (with uniform cost search) with different fixed pruning thresholds. "A*" uses outside scores estimated from a coarse level grammar (level-1) to speed up parsing. "LDP-V" is the dynamic pruning model with Viterbi trees as reference and 'LDP-G" is the one with the most probable compatible trees as reference. NLDP-V does not use data aggregation. LDP-V+ uses more dynamic features.

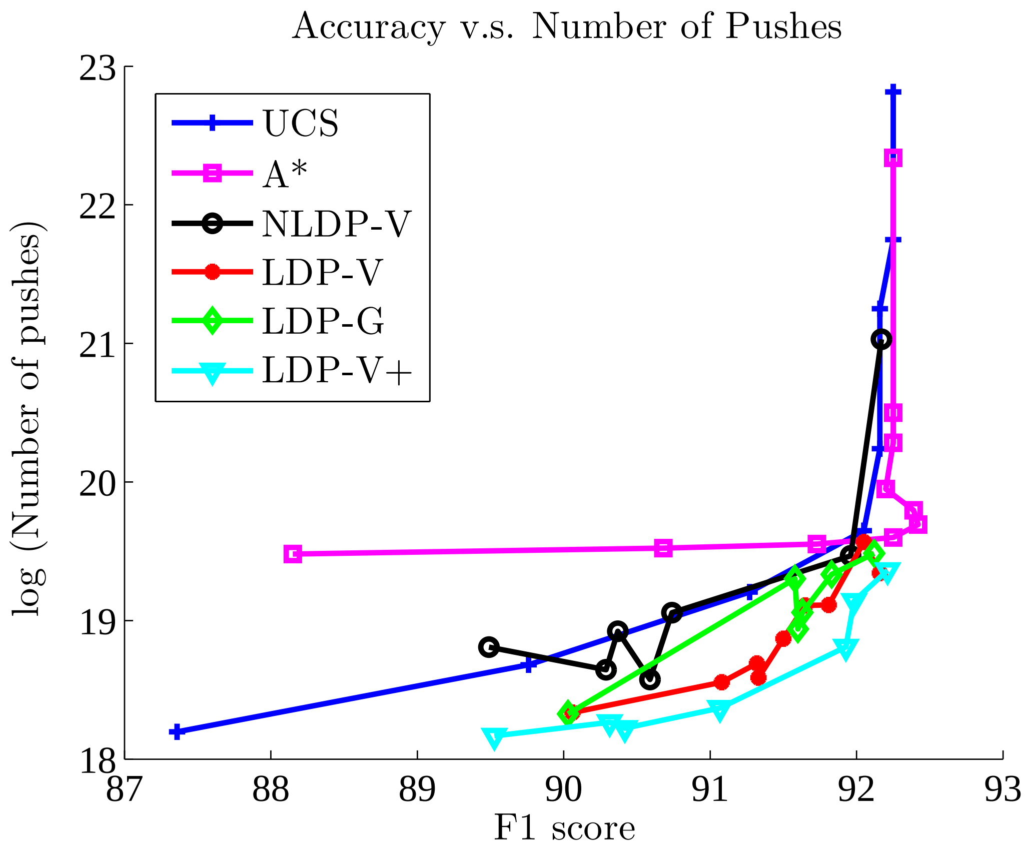

Figure 5.6: The graph of accuracy vs. number of pushes on final test set with different trade-off. "UCS" is the standard agenda-based parser (with uniform cost search) with different fixed pruning thresholds. "A*" uses outside scores estimated from a coarse level grammar (level-1) to speed up parsing. "LDP-V" is the dynamic pruning model with Viterbi trees as reference and 'LDP-G" is the one with the most probable compatible trees as reference. NLDP-V does not use data aggregation. LDP-V only uses unigram conditional dependency information while LDP-V+ uses additional bigram features as well as cross-bracket features.

90%). With more dynamic features, model LDP-V+ improves slightly more on the accuracy. By looking at the weights of LDP-V and LDP-V+ from LibSVM, we find the LDP-V+ pruner learns to put high weights at margin features and cross-bracket features to prune more constituents dynamically.

Table 5.1: Relative comparison of accuracy and speed for dynamic pruner. UCS-Prune-1 is used as a reference. Smaller number is faster.

Model	F1 difference	# of pops ratio	# of pushes ratio	Runtime
UCS-No-Prune	0	0.85	0.34	2.44x
UCS-Prune-1	0	1	1	1x
UCS-Prune-2	-0.09	1.25	4.5	0.31x
A*-No-Prune	0	0.67	0.55	1.61x
A*-Prune-1	+0.17	1.2	7.8	0.19x
A*-Prune-2	0	1.3	8.6	0.17x
A*-Prune-3	-0.52	1.4	9.0	0.14x
LDP-V	-0.09	6.9	11.1	0.13x
LDP-G	-0.13	5.6	15.0	0.13x
LDP-V+	-0.02	7.0	13.4	0.12x

In table 5.1, we pick a few points from figure 5.5 and 5.6: UCS-No-Prune is uniform cost search without pruning. UCS-Prune-1 is UCS with one pruning threshold that maintains the same F1 score but improves the speed. UCS-Prune-2 is picked so that the F1 score after pruning happens to be the same as the best of the LDP-V models. A*-No-Prune is the A* search without pruning. A*-Prune-1 model is the one with highest F1 score. A*-Prune-2 is a fast model with the same accuracy as UCS. A*-Prune-3 is a model that loses on accuracy but is faster than A*-Prune-2. LDP-V and LDP-G are the dynamic pruner models with their best F1 score.

In terms of accuracy, the best model that LDP-V has is about 0.09% lower than the accuracy of the Viterbi tree and LDP-G is about 0.13% lower. As A* uses a reasonably good heuristic, pruning happens to help the F1 score. A* has better and faster models when the pruning threshold is not too large and with larger thresholds, the speed curve becomes flat. The dynamic pruning models are learned to prune a lot more compared to other baseline models. The overhead of feature extraction and classification takes about 18% of the overall time, so the gain in the runtime is not as much as the numbers of pops and pushes, but they are still better or comparable with baseline models.

5.6.2 Heuristic Pruning at Push Time

The learned dynamic pruner only performs pruning at pop time while computing rich features at push time will slow down the parser. To speed up at push time, we can use the standard heuristic pruner directly. Here, we use the difference between inside scores as a pruning heuristic and apply various thresholds to our

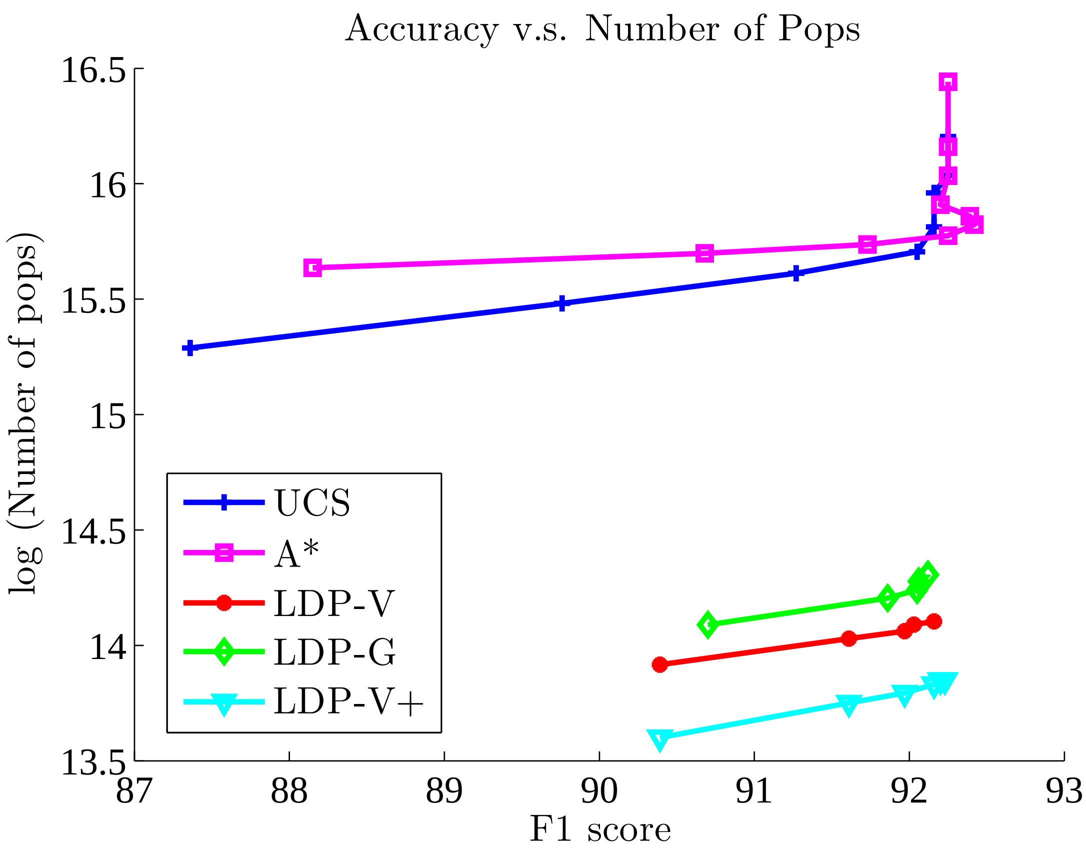

Figure 5.7: The graph of accuracy vs. number of pops on final test set with additional heuristic pruning at push time. "UCS" is the standard agenda-based parser (with uniform cost search) with different fixed pruning thresholds."A*" uses admissible outside scores to speed up parsing. "LDP-V" is the dynamic pruning model with Viterbi trees as reference and 'LDP-G" is the one with the most probable compatible trees as reference. LDP-V+ is LDP-V model with more dynamic features from 5.5 and 5.6.

learned models. To show the performance improvement by this additional pruning, we pick the best models from LDP-V, LDP-G, LDP-V+ curves in the previous section and set the pruning threshold at push time to be the same as UCS. We also use the same threshold to generate the Pareto frontier for A$\star$. Figure 5.7 and 5.8 show the curve after pruning for the picked models.

If we only train a constituent parser at pop time, the pruner learns it's best to prune constituents at pop time. However, there are constituents built at push time

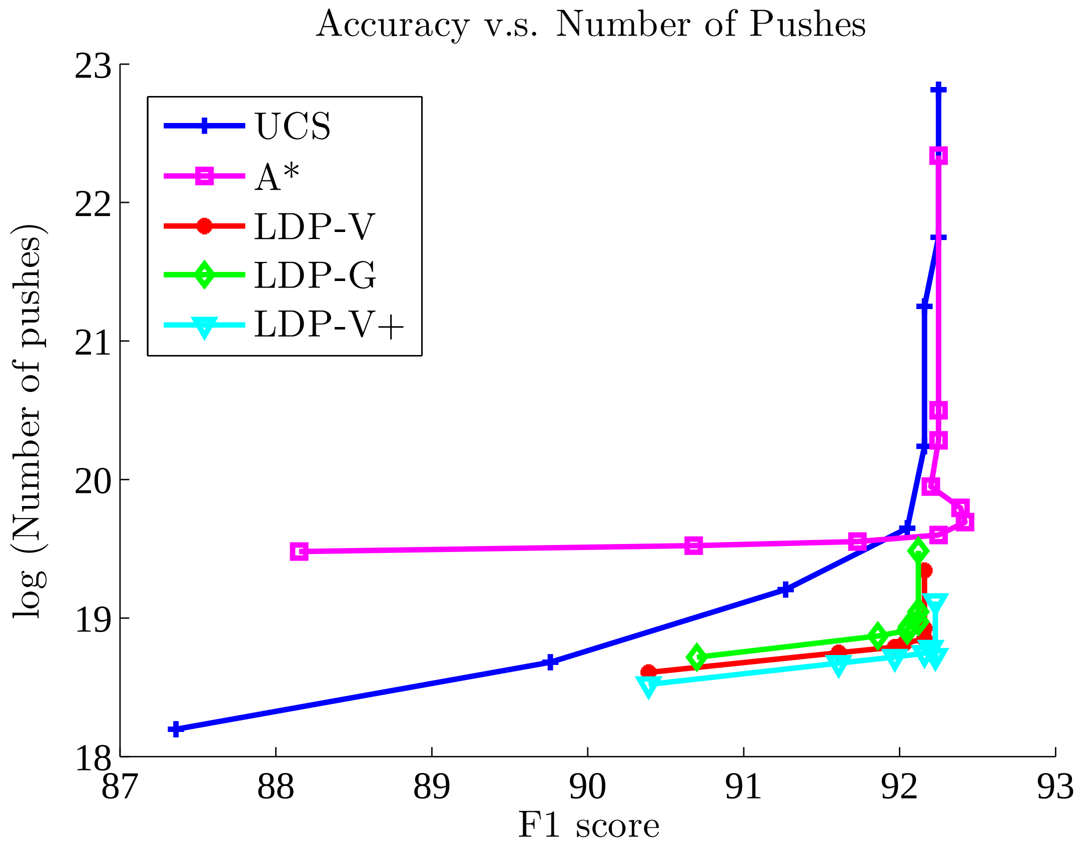

Figure 5.8: The graph of accuracy vs. number of pushes on final test set with additional heuristic pruning at push time. "UCS" is the standard agenda-based parser (with uniform cost search) with different fixed pruning thresholds. "A*" uses admissible outside scores to speed up parsing. "LDP-V" is the dynamic pruning model with Viterbi trees as reference and 'LDP-G" is the one with the most probable compatible trees as reference. LDP-V+ is LDP-V model with more dynamic features from 5.5 and 5.6.

that are too low in inside score compared to what is currently available in the cell, so they should not even be added to the agenda. By pruning those constituents at push time with heuristics, we save the time spent on reordering things available on the agenda. Many constituents with low scores will not be popped from the agenda when parsing with a pruner at pop time, pruning these will not affect the number of pops too much. When comparing LDP-V and LDP-G models (different oracles) with the same pruning thresholds, the drop in accuracy of models trained with most probable compatible trees is smaller than models trained with Viterbi trees. The number of pushes has a sharp drop without hurting accuracy at the beginning and the curves become flat when the thresholds are large. Our learned model behaves very similar to the pruned A* while the learned pruner still maintains a faster speed.

5.6.3 Joint constituent and cell pruner

As shown in the previous part, we can achieve some speedup by using a constituent pruner. However, the constituent pruner can only remove candidates when it is popped from the agenda and the runtime is still much slower compared to a state-of-the-art parser. Here we will add our cell pruner which prunes even more candidates. For the implementation and baselines, we implemented our model on top of bubsparser [Bodenstab et al., 2011] and use their averaged perceptron to train cell and constituent classifiers.

For the features, we use all the features mentioned above except the score features: inside score, FOM score and outside score (we run some preliminary experiments with these and they do not help in this case). To compute the conditional likelihood features, we use the POS tags labeled by a POS tagger trained for FOM estimations in Bubsparser. (This is necessary to compute FOM for Bubsparser, so we just take it for free. In the inside score case, for A*, we can use the POS tags from the coarse level grammar estimation, but we do not use it so the features for all the models in the previous sections are extracted in the same way.)

We first compare a model trained by static features only, with static only features. On the development set, it can learn a model that is slightly slower than the complete closure model but faster than the FOM model while losing about 0.2% on F-1 score. On the other hand, with dynamic features, we can get comparable results with adaptive beam prediction.

Table 5.2: Relative comparison of accuracy and speed for dynamic pruner to parse sentences of length 25 and less. Figure-of-merit parser is used as reference. Smaller number is faster.

Model	F1	Runtime
FOM	89.22	1x
Adaptive Beam	90.1	0.41x
Complete Closure	89.22	0.59x
LDP2-Joint	89.22	0.47x
LDP2-Iterative	89.20	0.53x

Table 5.2 shows the results on the test set. FOM is the figure-of-merit best-first-parser and the rest of the algorithms all use figure-of-merit as priorities instead of inside scores. Adaptive Beam is the beam-width prediction algorithm implemented in Bubsparser. Complete Closure is the algorithm that closes individual cells before parsing in Bubsparser. LDP2-Joint is the cell and constituent pruner trained with joint DAgger algorithm and LDP2-Iterative is the algorithm where we fix a pruner and train for the other. We use the development set to select our models with highest accuracy then pick the fastest one (in the pruning case, we only want something that is fast and accurate) and report the results on the test set. We assume the time to parse the test set is 1x for the FOM model and compare the relative speed in terms of wall clock time. Iterative training results have a large variance in the final models – some models tend to prune more cells and some tend to prune more constituents. It is very likely that we get different local optimum by using this method. On the other hand, joint training is fairly stable and achieves a decent speed compare to other baselines. We also observe that our cell pruner prunes about 80% about the constituents overall and the constituent pruner prunes the rest. This also supports the fact that training a consitutent pruner is not enough to achieve good accuracy and speed. However, in general, when using both cell and constituent pruners, we need to compute more features as well as classify more candidates compared to the adaptive beam model. Excluding the variance in different runs, our models are still slightly slower than the adaptive beam method.

Compared to the complete closure model, we add a constituent pruner and close cells dynamically during runtime with speed-up and no loss in accuracy. However, compared to the adaptive beam model, which trains multiple classifiers for different beam-widths, we are slightly slower. Moreover the beam-width prediction model wins in accuracy due to its beneficial search error. A simple preliminary test shows that training with features from the beam-width prediction model with one additional feature for number of elements in the cell does not prune as many constituents as multiple independent classifiers. This is an indicator that classifiers of different beam-widths prefer different weights on the features. In this case, we need to properly divide decisions for hard pruning problems to a few easier ones and combine the results efficiently. Using a fast classifier with a non-linear decision boundary that can beat an averaged perceptron might be helpful here. We will leave investigation of this issue for future work.

5.7 Conclusion

In this chapter, we discussed another way to cooperate with dynamic features to speed up an agenda-based parser – dynamic pruning. We proposed two different learning algorithms to learn pruning classifiers for both constituents and cells and make pruning decisions for each of the constituents at runtime. We show that by using dynamic features, we get comparable performance compared to the state-of-art bubs-parser or standard Berkeley parser. Meanwhile, our training framework can be generalized to other pruning problems where we want to learn two pruners at the same time.

So far in chapter 3-5, we discussed nondeterministic agents in reinforcement learning, reprioritization and pruning for agenda based parsing. In the next chapter, we will conclude with a general framework of how can we use nondeterminism in learning and show short examples of other applications.

Chapter 6: Conclusion and Future Work

In this dissertation, I answer the question of "Why, when and how shall we take advantage of the non-determinism to speed up a search system?" by showing an abstract example of a non-deterministic agent in reinforcement learning as well as a detailed case study in constituency parsing. We also show how to use the ground truth as oracles in agenda-based parsing. The simple answers to these questions are:

- We should use non-deterministic training.
- Do not let the agent explore the space randomly if the search space is huge. We should exploit the search space near oracles first and explore the space around them later. In supervised learning, good oracles may be constructed according to ground truth.
- Carefully use as many dynamic features as possible.

By following these rules, we propose an oracle-infused policy gradient algorithm to automatically trade off speed and accuracy in agenda-based parsing. We also apply it to learning a dynamic pruner. As shown in the experiments, our models achieve performance comparable to the state-of-the-art.

In the main chapters of the discussion, we gave some demonstrations of how to do the training when oracles can be constructed in structured prediction problems. In this final chapter, we are going to summarize the framework such that it may be applied to other problems as well as suggesting a procedure for solving problems where oracles cannot be found. As preliminary future work, we will discuss an application in asynchronous belief propagation.

6.1 General Solution for Training with Non-deterministic Algorithms

Summarizing the work in this dissertation, to optimize a non-deterministic structured prediction problem, we can take the following steps:

- Step 1: Choose an optimization target and prioritizing heuristic.
- Step 2: Construct oracles.
- Step 3: Train iteratively with a mixture of policies from the previous iterations and oracles.
- Step 4: Incorporate as many dynamic features as possible when the overall impact of the feature extraction on speed is low.

In step 1, we have to select the target that we want to optimize: whether to pursue a speed and accuracy trade-off by prioritizing search actions or pruning search space. Pruning can be done directly on the search space or action space. Prioritizing can be formulated as optimizing an objective of speed and accuracy tradeoff or running a cascade model. Optimization heuristics are the strategies that improve the performance, e.g. prioritization or pruning in the case of parsing. This decides which learning algorithms we can use to solve the problem.

In step 2, we need to construct oracles given ground truth or other expert knowledge. Ideally, the oracles or expert knowledge should be fully aware of which action to take at any possible state in the search space and be optimal in terms of the learning target. In practice, we can try to construct the oracles by mimicking or enforcing the behavior to produce ground truth. [Daumé III et al., 2009, Ross and Bagnell, 2010]

In step 3, during training, we can define the running policy/classifier/ranker to be a mixture of previously learned policy and oracle policy. This allows the learner to start searching in a reasonably good policy area. The weight on the oracle policy can be reduced so that the final learned policy does not have to depend on the oracles. [Daumé III et al., 2009, Ross and Bagnell, 2010]

In step 4, as shown in the dissertation, dynamic features play an important role in improving the performance of the system in both speed and accuracy. However, we also need to take the feature extraction and computation time into consideration. For a lot of dynamic features, we can pre-compute or approximate the values and store it in hashmaps to speed up the search. Multi-level dynamic features can be used when a coarse-to-fine style algorithm is used for structured prediction.

The above case considers when oracles can be found somehow. When oracles cannot be built at all, we can make modifications to steps 2 and 3 as follows:

In step 2, we need to find a way to estimate or approximate the cumulative future reward for each possible choice. For example, [Daumé III et al., 2009, Huang et al., 2012] provide ways of doing roll-out or inexact search; or we can use reinforcement learning algorithms in the delayed reward setting to update once a goal (terminal state) is reached. This estimates how good an action is in the long run which is similar to outside score estimation in the parsing setting, and it is the learning target for prioritizing search candidates.

In step 3, instead of training with a mixed policy, we need to allow certain stochasticity in the learning so that the search space can be well explored. Moreover, if a measurement of the difference between the ground truth and the current learned policy is defined, we can properly evaluate the policy and choose a better region to explore.

In the next section, we will give a preliminary example of optimizing belief propagation where oracles are hard to define by following our framework.

6.2 Example: Asynchronous Belief Propagation

Message scheduling is shown to be very effective in belief propagation (BP) algorithms. However, most existing scheduling algorithms use fixed heuristics regardless of the structure of the graphs or properties of the distribution. On the other hand, designing different scheduling heuristics for all graph structures are not feasible. In this section, we propose a reinforcement learning based message scheduling framework (RLBP) to learn the heuristics automatically which generalizes to any graph structures and distributions. In the experiments, we show that the learned problem-specific heuristics largely outperform other baselines in speed.

6.2.1 Problem Overview

Probabilistic graphical models [Pearl, 1988, Wainwright and Jordan, 2008, Jordan, 1999] play an important role in representing complex distributions and dependencies between random variables for many real world applications. Many approximate inference algorithms have been proposed to solve the problem efficiently. One of the most common methods is message-passing algorithms such as loopy belief propagation [Murphy et al., 1999, Ihler et al., 2005, Yedidia et al., 2000]. The idea behind this is to pass messages between adjacent nodes until the fixed points of the beliefs are achieved.

It is shown in [Elidan, 2006] that asynchronous propagation can achieve better convergence compared to synchronous methods and thus initiated interest in studying message scheduling schemes. However, most existing research only focuses on manually designed heuristics. For example, [Wainwright et al., 2001] proposed a tree reparameterization algorithm as message scheduling for asynchronous propagation. [Elidan, 2006] orders the messages in the order of the differences between two consecutive values of the message. [Vila Casado et al., 2010, Yedidia et al., 2005] also design specific scheduling for LDPC decoding problems.

In contrast with standard scheduling algorithms, in this chapter, we explore *automatically* scheduling messages for any graph structure and distribution. To achieve this, we formalized the learning of the ordering as a Markov Decision Process (MDP) and associate the final reward with the convergence speed. The ordering can be viewed as a linear policy of the MDP. The goal is trying to learn a good policy such that the corresponding scheduling can achieve a better convergence speed. In the experiments, we show that that for the graphs that the heuristics are automatically learned, the learned heuristics performs better than the baselines. We also notice that the learned weights reveal some useful features to consider in the scheduling.

6.2.2 Background

In this chapter, we consider the problem of marginal inference on undirected and discrete graphical models. Formally, we denote a graph $G = (X, E)$ and let $x = \{x_1, ..., x_m\}$ to be the random variables associated with nodes in X. The exponential family distribution p over the random variables is defined as follows:

$$p_\theta(x) = \exp[\langle \theta, \phi(x) \rangle - A(\theta)] \tag{6.1}$$

where $\phi(x)$ is the sufficient statistics of x and θ is the canonical or exponential parameters. $A(\theta) = \log \sum_x \exp[\langle \theta, \phi(x) \rangle]$ is the log-partition function. In this section, we will focus on solving the marginal problem which is inferring marginal distribution $p(x)$ for all x.

6.2.2.1 Belief Propagation

Belief propagation [Murphy et al., 1999, Ihler et al., 2005, Yedidia et al., 2000] (or sum-product) algorithm is the standard message-passing algorithm for inferring marginal distributions over random variables. It is a fixed point iteration algorithm where the fixed point is the exact solution to trees or polytrees structures [Pearl, 1988]. On loopy graphs, they are not guaranteed to converge, but if they do, the final estimates are shown to be reasonably good [Yedidia et al., 2000].

The message M_{ts} passed from node t to one of its neighbors s is defined as:

$$M_{ts}(x_s) \leftarrow \kappa \sum_{x'_t \in \mathcal{X}_t} \Big\{ \exp[\theta_{st}(x_s, x_{t'}) + \theta_t(x_{t'})] \prod_{u \in N(t) \backslash s} M_{ut}(x_{t'}) \Big\} \tag{6.2}$$

where κ is the normalization constant. $N(t) \backslash s$ are the neighbors of t excluding s. When the messages converge, the belief/psuedomarginal distribution for the node s is given by

$$\mu_s(x_s) = \kappa \exp\{\theta_s(x_s)\} \prod_{t \in N(s)} M_{ts}(x_s) \tag{6.3}$$

For synchronous message-passing, only the old messages from the previous iteration are used to compute the messages for the current iteration. However, asynchronous message-passing uses the latest messages for updates. In [Elidan, 2006], it is shown that any reasonable asynchronous BP algorithm converges to a unique fixed point under the assumptions that are similar to the synchronous version. They use the residues (differences in beliefs between updates) as a priority to schedule the messages. In this chapter, instead of using the greedy residual schedule, we learn a proper schedule automatically by a reinforcement learner.

6.2.3 Message Scheduling with Reinforcement Learner

We formulate the schedule learning problem as a MDP and the goal is to learn a policy for the MDP such that the corresponding ordering can improve the speed of convergence. Note that in this chapter, we will only consider ordering the nodes in the graph, but the learning framework can be easily generalized to ordering each message in the graph.

6.2.4 MDP Formulation

A Markov Decision Process (MDP) [Bellman, 1957, Puterman, 2009] can be viewed as a memoryless search process. It consists of a state space S, an action space A, a transition function T, a reward function R and the environment $\mathcal{E}$. An agent repeatedly observes the current state $s \in S$ and takes an action $a \in A$. The environment $\mathcal{E}$ then samples the new state s' from the transition function $T(s'|s, a)$ and gives a reward $R(s'|a, s)$. A policy $\pi(a|s)$ describes how the agent chooses an action based on its current state.

More specifically, for the schedule learning problem, the state space S is the graph and its messages/beliefs. The action to take at each step is choosing a node x in the graph and computing its outgoing messages. Assume the features associated with node x is $\phi(x)$ and the transition function T is deterministic. The reward is defined with regard to the convergence speed. Here we consider a linear policy (priority) which is

$$\pi_\omega(s) = \arg\max_x \omega \cdot \phi(x) \tag{6.4}$$

where ω is the feature weight vector. The goal is to maximize the expected reward $R = \mathbb{E}_{r \sim \pi_\omega}[R(\tau)] = \mathbb{E}_{r \sim \pi_\omega}[\sum_{t=0}^{T} r_t]$ where $\tau = (s_0, a_0, r_0, ..., s_T, a_T, r_T)$ is a trajectory. In the current situation, this is equivalent to minimizing the number of messages passed.

At test time, the transition function is deterministic so that we always choose the node that has the highest priority in the current state s. However, at the training time, we allow the agent to explore the space by a stochastic policy:

$$\pi_\omega(x_t) = \frac{1}{Z} \exp\left(\frac{1}{T}\omega \cdot \phi(x)\right) \tag{6.5}$$

where T is temperature and Z is a normalization constant. When $T \to \infty$, π_ω achieves a random choice over all the nodes while $T \to 0$ only exploits the deterministic policy.

To solve this, we apply the standard policy gradient algorithm as discussed in Chapter 4.

6.2.5 Algorithm

A detailed training algorithm is described in Algorithm 11. In the training, assume node and edge potentials are given for a set of training graphs. The goal is to learn the policy (feature weights) so that the ordering of the nodes will help the model convergence. One run on a graph is treated as a trajectory for the learner.

A priority queue is maintained to order the nodes of the graph. For each iteration, a node is popped from the queue according to a Boltzmann exploration policy. Then the relevant messages are updated for the node and the features (and their priorities) of the neighboring nodes are also updated. One iteration is over when there are no more nodes in the priority queue. If the beliefs are not converged, then push all the nodes to the priority queue and repeat. Continue this process until the gradient converges and update the weight vector ω. Return the final weight vector ω when the gradient $\to 0$.

During the test time, the learned deterministic policy will be used to prioritize the nodes and the node with the highest priority is updated for each step.

6.2.5.1 Features

The features we use here include both static features and dynamic features. The static features are mostly extracted according to the graph properties once per inference run. Dynamic features are associated with real-time updates and a change in the values of dynamic features for a node can impact the feature values for other nodes in the graph.

The static features are:

- the degree of the node,
- the dimension of the node,
- the max/min degrees of the neighboring nodes,
- the max/min dimension of the neighboring nodes

Dynamic features are those features that change between iterations or even within a single iteration. Here we use

- KL divergence and residual between the current learned belief and previous belief
- the difference between the current and last incoming messages
- the difference between the outgoing messages.

6.2.6 Experiments

To empirically evaluate the model, we randomly generate Ising models according to the experiment setups in [Elidan, 2006][2] with degree 4 and variable dimension 2: The graphical representation of an Ising model is a $N \times N$ random grid with a corresponding number of binary variables. The node potentials are drawn from $U[0, 1]$ and the pairwised potentials

$$\phi_{s,t}(X_s, X_t) = \begin{cases} \exp(\lambda C) & \text{if } x_s = x_t \\ \\ \exp(-\lambda C) & \text{if } x_s \neq x_t \end{cases} \tag{6.6}$$

where $\lambda \sim U[-0.5, 0.5]$ and C is a constant.

We evaluate our reinforcement learning based message scheduling framework (RLBP) against the standard loopy belief propagation (LBP) and residual belief prorogation (RBP). We also propose and evaluate against a novel variant of the RBP where instead of measuring the L2 differences between belief updates, we use a KL divergence type of difference (KLBP).

We generate 40 5×5 grid graphs with $C = 5$. We train our scheduling policy on the same graphs for which we report convergence results. LBP converges on 19 graphs and the remaining 3 algorithms converge on 27 graphs. We only try to learn the scheduling on the 27 graphs where either RBP or KLBP converges. In table 6.1, we show results in terms of time reduction over LBP per model. For clarity, LBP is shown as having 0% time reduction over itself. It is clear that RLBP outperforms all other models by a wide margin.

[2]As ongoing work, we only show the preliminary results here.

LBP	0%	RBP	34.70%
KLBP	39.62%	RLBP	55.10%

Table 6.1: Time reduction in percentage over LBP by propagation framework

Figure 6.2.6 shows the number of iterations taken for each of the 27 graphs. Note some of the results for LBP are missing because it does not converge. In order to make the plot easy to read, we order the graphs on the x-axis by the number of iterations RLBP (red line) takes.

It can be seen that RBP and KLBP also show large reductions in the time to convergence and sometimes one scheme is better than the other. However, the learned heuristics (RLBP) performs the best of all most of time. One interesting observation in the final feature weights is that the degrees of the node here is a very indicative feature besides the difference or divergence of the beliefs, which supports the fact that different graph structures should have different scheduling for the messages.

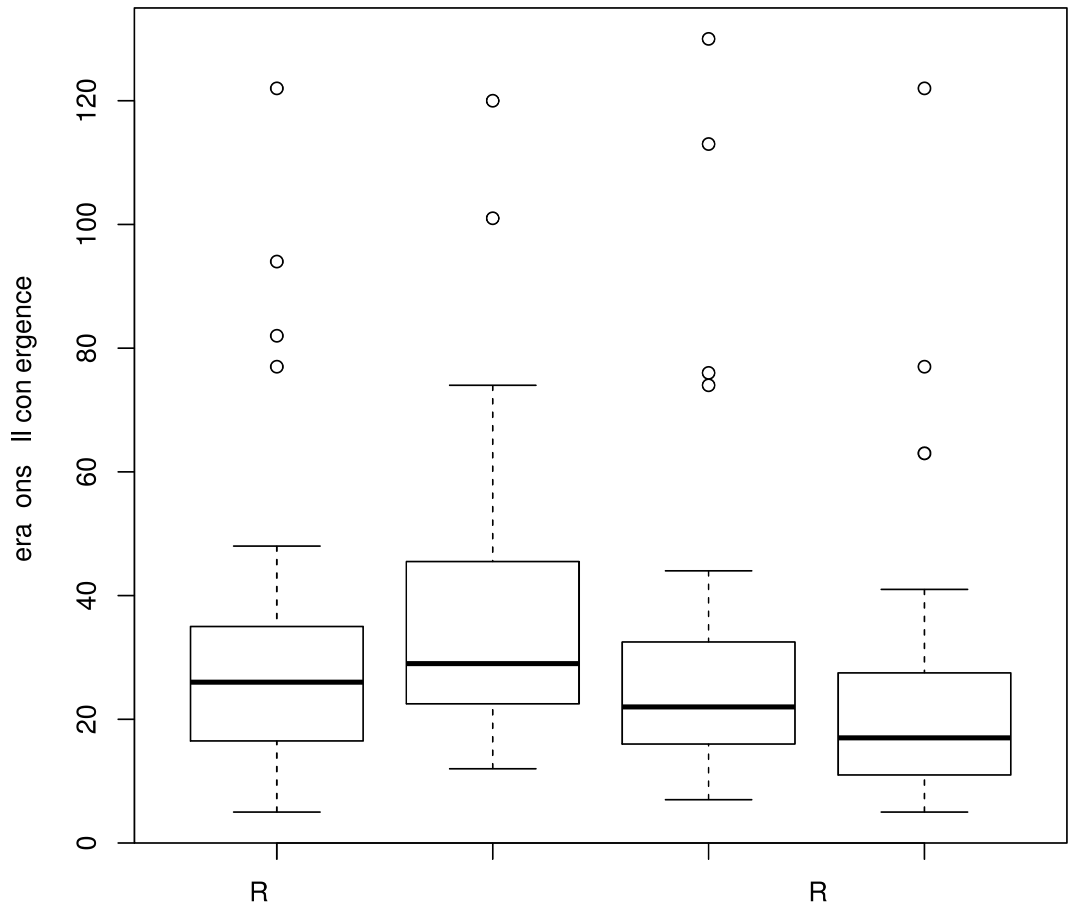

Figure 6.1: Plot of iterations per model until convergence.

Algorithm 11 Training Algorithm

```
Input: Training set with graphs and given potentials.
Initialize the policy parameter $\omega$.
for each graph in the training set do
    Initialize the messages and priority queue $Q$ with all the nodes in the graph.
    while the beliefs are not converged or the maximum number of iterations is not reached do
        Pop a node $x$ from $Q$ according to (6.5).
        Compute its messages by (6.3).
        Compute the derivative $\nabla_\omega \log \pi(x_t|Q_t)$ as in policy gradient (Eq.2.12).
        Update the dynamic features of its neighbors.
        if the priority queue $Q$ is empty and algorithm not converged then
            Enqueue all the nodes[1].
        end if
    end while
    Compute the reward of the current trajectory.
    Compute the baseline.
    Update the policy when the gradient converges.
end for
```

Chapter A: List of Punctuation Patterns

As we mentioned in the prioritization and pruning chapters, we use the same type of punctuation features as in Liang et al. [2008]. For a span, regular words are replace with symbol “x” and only punctuation: $, . “ ” -LRB- -RRB- : # are saved. Here is a list of top 25 most likely punctuation patterns. The list is ranked in the order of decreasing likelihood.

x

x , x ,

x $ x

$ x

x , x , x

, x ,

x : x :

x “ x ” x

x -LRB- x -RRB-

“ x , ” x

x -LRB- x -RRB- x

-LRB- x

x $ x $ x

: x

x -LRB- $ x -RRB-

‘ x , x , ”

x “

x , x , x ,

“ x , ”

x # x

x : x , x :

“ x , x , ” x

x : “

x $

x

Following is the list of bottom 19 punctuation patterns. The list is in the order of increasing likelihood.

“ x

x , ”

x ,

x -RRB-

x : x

x -LRB- x

, x , x

x , x “ x

x , ” x

x , x : x

“ x ,

x :

x $ x ,

x , ” x , x

x , x

x .

x , x $ x

x “ x .

$ x ,

Chapter B: Examples of Latent Grammar and Ground Truth

Grammar rules in Berkeley parser is in the form of

```
S_8 -> VP_12 NP_3 3.5515382782419514E-9

S_9 -> VP_12 NP_3 3.818597576454279E-11

S_10 -> VP_12 NP_3 4.4037681483615745E-8

S_11 -> VP_12 NP_3 7.531926043833406E-11
```

and groud truth looks like:

```
(ROOT (S (NP (DT The)

             (NNP Ways)

             (CC and)

             (NNP Means)

             (NNP Committee))

         (VP (MD will)

             (VP (VB hold)

                 (NP (NP (DT a)

                         (NN hearing))

                     (PP (IN on)

                         (NP (DT the)
                    (NN bill))))
        (NP (IN next)
            (NNP Tuesday))))
(. .)))
```

Bibliography

David Baker and Andrej Sali. Protein structure prediction and structural genomics. *Science*, 294(5540):93–96, 2001.

Jay Earley. An efficient context-free parsing algorithm. In *Communications of the ACM*, volume 13, pages 94–102. ACM, 1970.

Christopher D Manning and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT press, 1999.

John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 282–289, 2001a.

Pierluigi Crescenzi, Deborah Goldman, Christos Papadimitriou, Antonio Piccolboni, and Mihalis Yannakakis. On the complexity of protein folding. *Journal of computational biology*, 5(3):423–465, 1998.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, New Jersey, 1995.

Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausble Inference*. Morgan Kaufmann Pub, 1988.

Martin J Wainwright and Michael I Jordan. Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, 1(1-2): 1–305, 2008.

Gal Elidan. Residual belief propagation: Informed scheduling for asynchronous message passing. In *Proceedings of the Twenty-second Conference on Uncertainty in AI (UAI*, 2006.

Andres I Vila Casado, Miguel Griot, and Richard D Wesel. Ldpc decoders with informed dynamic scheduling. *Communications, IEEE Transactions on*, 58(12): 3470–3479, 2010.

Thorsten Brants. Tnt: a statistical part-of-speech tagger. In *Proceedings of the sixth conference on Applied natural language processing*, pages 224–231. Association for Computational Linguistics, 2000.

John Lafferty, Andrew McCallum, and Fernando CN Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. *Proceedings of the International Conference on Machine Learning (ICML)*, 2001b.

Doug Cutting, Julian Kupiec, Jan Pedersen, and Penelope Sibun. A practical part-of-speech tagger. In *Proceedings of the third conference on Applied natural language processing*, pages 133–140. Association for Computational Linguistics, 1992.

Michael Collins and Brian Roark. Incremental parsing with the perceptron algorithm. In *Proceedings of the Conference of the Association for Computational Linguistics (ACL)*, 2004a.

D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, February 1967.

Alfred V Aho et al. *Compilers: principles, techniques, & tools.* Pearson Education India, 2007.

Joakim Nivre. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT*. Citeseer, 2003.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 523–530. Association for Computational Linguistics, 2005.

Michael Collins and Brian Roark. Incremental parsing with the perceptron algorithm. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, page 111. Association for Computational Linguistics, 2004b.

J.-C. Chappelier, M. Rajman, and Ch-Lausanne. A generalized cyk algorithm for parsing stochastic cfg, 1998.

Dan Klein and Christopher D. Manning. An o(n3) agenda-based chart parser for arbitrary probabilistic context-free grammars, 2001.

David M. Magerman and Mitchell P. Marcus. Pearl: A probabilistic chart parser, 1991.

Dan Klein and Chris Manning. A* parsing: Fast exact Viterbi parse selection. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics and Human Language Technology (NAACL/HLT)*, 2003.

Brian Roark and Kristy Hollingshead. Classifying chart cells for quadratic complexity context-free inference. In *Proceedings of the International Conference on Computational Linguistics (COLING)*, pages 745–752, Manchester, UK, August 2008. Coling 2008 Organizing Committee. URL `http://www.aclweb.org/anthology/C08-1094`.

R. Bellman. A markovian decision process. In *Journal of Mathematics and Mechanics*, 1957.

C.J.C.H Watkins. Learning from delayed rewards. In *Cambridge University*, 1989.

Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

Jing Peng and Ronald J Williams. Efficient learning and planning within the dyna framework. *Adaptive Behavior*, 1(4):437–454, 1993.

Andrew W Moore and Christopher G Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13(1):103–130, 1993a.

David Andre, Nir Friedman, and Ronald Parr. Generalized prioritized sweeping. In *Advances in Neural Information Processing Systems.* MIT Press, 1998.

J. Baxter and P. Bartlett. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research (JAIR)*, 15:319–350, 2001.

R.J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement elarning. *Machine Learning*, 8(23), 1992.

Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1057–1063. MIT Press, 2000.

Martin Kay. Algorithm schemata and data structures in syntactic processing. In B. J. Grosz, K. Sparck Jones, and B. L. Webber, editors, *Readings in Natural Language Processing*, pages 35–70. Kaufmann, 1986. First published (1980) as Xerox PARC TR CSL-80-12.

A. Pauls and D. Klein. Hierarchical A* parsing with bridge outside scores. In *Proceedings of the Conference of the Association for Computational Linguistics (ACL)*, pages 348–352. Association for Computational Linguistics, 2010.

P. F. Felzenszwalb and D. McAllester. The generalized A* architecture. *Journal of Artificial Intelligence Research (JAIR)*, 29:153–190, 2007. URL `http://www.jair.org/papers/paper2187.html`.

Aria Haghighi, John DeNero, and Dan Klein. Approximate factoring for A* search. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics and Human Language Technology (NAACL/HLT)*, pages 412–419, Rochester, New York, April 2007. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/N/N07/N07-1052`.

Sharon A. Caraballo and Eugene Charniak. New figures of merit for best-first probabilistic chart parsing. *Computational Linguistics*, 24(2):275–298, 1998. URL `http://www.cs.brown.edu/people/sc/NewFiguresofMerit.ps.Z`.

E. Charniak, S. Goldwater, and M. Johnson. Edge-based best-first chart parsing. In *Proceedings of the Workshop on Very Large Corpora*, pages 127–133, 1998.

Nathan Bodenstab, Aaron Dunlop, Keith Hall, and Brian Roark. Beam-width prediction for efficient CYK parsing. In *Proceedings of the Conference of the Association for Computational Linguistics (ACL)*, 2011.

E. Charniak, M. Johnson, M. Elsner, J. Austerweil, D. Ellis, I. Haxton, C. Hill, R. Shrivaths, J. Moore, M. Pozar, et al. Multilevel coarse-to-fine PCFG parsing. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics and Human Language Technology (NAACL/HLT)*, pages 168–175. Association for Computational Linguistics, 2006.

S. Petrov and D. Klein. Improved inference for unlexicalized parsing. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics and Human Language Technology (NAACL/HLT)*, pages 404–411, 2007.

Andrew Ng and Stuart Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2000.

Brian Ziebart, Andrew Maas, J. Andrew Bagnell, and Anind Dey. Maximum entropy inverse reinforcement learning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2008.

R. Kalman. Contributions to the theory of optimal control. *Bol. Soc. Mat. Mexicana*, 5:558–563, 1968.

S. Boyd, L. El Ghaoui, E. Feron, and V. Balakrishnan. Linear matrix inequalities in system and control theory. *SIAM*, 15, 1994.

Pieter Abbeel and Andrew Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2004.

G. Neu and Cs. Szepesvári. Training parsers by inverse reinforcement learning. *Machine Learning*, 77, 2009.

Nathan Ratliff, David Bradley, J. Andrew Bagnell, and Joel Chestnutt. Boosting structured prediction for imitation learning. In *Advances in Neural Information Processing Systems (NIPS)*, 2007.

Hal Daumé III, John Langford, and Daniel Marcu. Search-based structured prediction. *Machine Learning*, 75(3):297–325, 2009. ISSN 0885-6125.

B. D. Argall, S. Chernova, M. Veloso, and B. Browning. A survey of robot learning from demonstration. In *Robotics and Autonomous Systems*, 2009.

Stephane Ross and J. Andrew Bagnell. Efficient reductions for imitation learning. In *Proceedings of the Workshop on Artificial Intelligence and Statistics (AI-Stats)*, 2010.

Umar Syed and Robert E. Schapire. A reduction from apprenticeship learning to classification. In *Advances in Neural Information Processing Systems (NIPS)*, 2011.

S. Natarajan, S. Joshi, P. Tadepalli, K. Kersting, and J. Shavlik. Imitation learning in relational domains: A functional-gradient boosting approach. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2011.

Stephane Ross, Geoff J. Gordon, and J. Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the Workshop on Artificial Intelligence and Statistics (AI-Stats)*, 2011a.

Gergely Neu and Csaba Szepesvri. Training parsers by inverse reinforcement learning, 2009.

Francis Maes, Ludovic Denoyer, and Patrick Gallinari. Structured prediction with reinforcement learning, 2008a.

Francis Maes, Ludovic Denoyer, and Patrick Gallinari. Sequence labeling with reinforcement learning and ranking algorithms. In *ECML*, 2008b.

Jinho D Choi and Martha Palmer. Getting the most out of transition-based dependency parsing. In *ACL (Short Papers)*, pages 687–692, 2011.

Stephane Ross, Daniel Munoz, Martial Hebert, and J Andrew Bagnell. Learning message-passing inference machines for structured prediction. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 2737–2744. IEEE, 2011b.

David Silver. Reinforcement learning and simulation-based search. *Doctor of philosophy, University of Alberta*, 2009.

Richard S. Sutton. Dyna, an integrated architecture for learning, planning, and reacting, 1991.

Richard S. Sutton, Csaba Szepesvri, Alborz Geramifard, and Michael Bowling. Dyna-style planning with linear function approximation and prioritized sweeping. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, 2008.

Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. In *Machine Learning*, pages 103–130, 1993b.

Pawel Cichosz. An analysis of experience replay in temporal difference learning. *Cybernetics and Systems*, (5):341–363, 1999.

S. W. Wilson and S. W. Wilson Explore/exploit Strategies. Explore/exploit strategies in autonomy, 1996.

Richard S. Sutton. Learning to predict by the methods of temporal differences. In *MACHINE LEARNING*, pages 9–44. Kluwer Academic Publishers, 1988.

Tommi Jaakkola, Michael I. Jordan, and Satinder P. Singh. Convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6:1185–1201, 1994.

Roman Zajdel. Epoch-incremental queue-dyna algorithm. In *Artificial Intelligence and Soft Computing–ICAISC 2008*, pages 1160–1170. Springer, 2008.

Littman Li. Prioritized sweeping converges to the optimal value function. *Technical Report DCS-TR-631*, 2008.

Vijaykumar Gullapalli and Andrew G Barto. Convergence of indirect adaptive asynchronous value iteration algorithms. *Advances in neural information processing systems*, pages 695–695, 1994.

Dimitri P Bertsekas and John N Tsitsiklis. *Parallel and distributed computation*. Old Tappan, NJ (USA); Prentice Hall Inc., 1989.

Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994.

Satinder P Singh and Richard S Sutton. Reinforcement learning with replacing eligibility traces. *Machine learning*, 22(1-3):123–158, 1996.

Brian Roark, Kristy Hollingshead, and Nathan Bodenstab. Finite-state chart constraints for reduced complexity context-free parsing pipelines. *Computational Linguistics*, 38(4):719–753, 2012.

Jason Eisner and Hal Daumé III. Learning speed-accuracy tradeoffs in nondeterministic inference algorithms. In *COST: NIPS Workshop on Computational Trade-offs in Statistical Learning*, Sierra Nevada, Spain, December 2011. 5 pages.

Joshua Goodman. Semiring parsing. *Computational Linguistics*, 25(4):573–605, December 1999. URL `http://research.microsoft.com/ joshuago/finalring.ps`.

Percy Liang, Hal Daumé III, and Dan Klein. Structure compilation: Trading structure for features. In *Proceedings of the International Conference on Machine Learning (ICML)*, Helsinki, Finland, 2008.

V. Gullapalli and A. G. Barto. Shaping as a method for accelerating reinforcement learning. In *Proceedings of the IEEE International Symposium on Intelligent Control*, 1992.

Jan Peters and Stefan Schaal. Reinforcement learning of motor skills with policy gradients. *Neural networks*, 21(4):682–697, 2008.

Slav Petrov and Dan Klein. Sparse multi-scale grammars for discriminative latent variable parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 867–876, Honolulu, Hawaii, October 2008. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/D08-1091`.

Y. Goldberg and M. Elhadad. An efficient algorithm for easy-first non-directional dependency parsing. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics and Human Language Technology (NAACL/HLT)*, pages 742–750. Association for Computational Linguistics, 2010. ISBN 1932432655.

Eugene Charniak. Top-down nearly-context-sensitive parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2010.

S.R.K. Branavan, Harr Chen, Luke Zettlemoyer, and Regina Barzilay. Reinforcement learning for mapping instructions to actions. In *Proceedings of the Conference of the Association for Computational Linguistics (ACL)*, 2009.

J. Andrew Bagnell. Robust supervised learning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2005.

Yuehua Xu and Alan Fern. On learning linear ranking functions for beam search. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 1047–1054, 2007.

M.P. Marcus, M.A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*, 19(2):330, 1993a. ISSN 0891-2017.

Takuya Matsuzaki, Yusuke Miyao, and Jun'ichi Tsujii. Probabilistic cfg with latent annotations. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 75–82. Association for Computational Linguistics, 2005.

A. Pauls and D. Klein. Hierarchical search for parsing. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics and Human Language Technology (NAACL/HLT)*, pages 557–565. Association for Computational Linguistics, 2009.

Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 433–440. Association for Computational Linguistics, 2006.

David Weiss, Benjamin Sapp, and Ben Taskar. Structured prediction cascades. *arXiv preprint arXiv:1208.3279*, 2012.

Nathan Bodenstab, Aaron Dunlop, Brian Roark, and Keith Hall. Exponential decay pruning for bottom-up beam-search parsing. In *Pacific Northwest Regional NLP Workshop (NW-NLP 2010)*, 2010.

Keith Hall and Mark Johnson. Language modeling using efficient best-first bottom-up parsing. In *Automatic Speech Recognition and Understanding, 2003. ASRU'03. 2003 IEEE Workshop on*, pages 507–512. IEEE, 2003.

Eugene Charniak and Mark Johnson. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of the Conference of the Association for Computational Linguistics (ACL)*, pages 173–180, Ann Arbor, Michigan, June 2005. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P/P05/P05-1022`.

Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3): 27, 2011.

Asa Ben-Hur and Jason Weston. A users guide to support vector machines. *Data mining techniques for the life sciences*, pages 223–239, 2010.

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993b.

Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *The Journal of Machine Learning Research*, 9:1871–1874, 2008.

Liang Huang, Suphan Fayong, and Yang Guo. Structured perceptron with inexact search. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 142–151. Association for Computational Linguistics, 2012.

Michael I. Jordan. An introduction to variational methods for graphical models. In *Machine Learning*, pages 183–233. MIT Press, 1999.

Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of Uncertainty in AI*, pages 467–475, 1999.

Alexander T. Ihler, John W. Fisher Iii, Alan S. Willsky, and Maxwell Chickering. Loopy belief propagation: Convergence and effects of message errors. *Journal of Machine Learning Research*, 6:905–936, 2005.

Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Generalized belief propagation. In *IN NIPS 13*, pages 689–695. MIT Press, 2000.

MJ Wainwright, T Jaakkola, and AS Willsky. Tree-based reparameterization for approximate estimation on loopy graphs. *Advances in Neural Information Processing Systems*, 14, 2001.

Jonathan S. Yedidia, Yige Wang, Yige Wang, Juntan Zhang, Juntan Zhang, Marc Fossorier, and Marc Fossorier. Reduced latency iterative decoding of ldpc codes. In *Proc. of the IEEE Global Communications Conf*, 2005.

Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*, volume 414. Wiley-Interscience, 2009.