14

Artificial Intelligence Techniques

Si al cabo de tres partidas de póquer no sabes todavía quien es el tonto, es que el tonto eres tú.

Manuel Vicent

The training program of an artificial intelligence can certainly include an informant, whether or not children receive negative instances.

E. Mark Gold, in [Gol67]

In the different combinatorial methods described in the previous chapters the main engine to the generalisation process is something like:

> 'If nothing tells me not to generalise, do it.'

For example, in the case where we are learning from an informant, the negative data is providing us with the reason for which one should not generalise (which is usually performed by merging two states). In the case of learning from text, the limitations exercised by the extra bias on the grammar class are what avoids over-generalisation.

But as a principle, the 'do it if you are allowed' idea is surely not the soundest. Since identification in the limit is achieved through elimination of alternatives, and an alternative can only be eliminated if there are facts that prohibit it, the principle is mathematically sound but defies *common sense*.

There are going to be both advantages and risks to consider a less *optimistic* point of view, which could be expressed somehow like:

'If there are good reasons to generalise, then do it.'

On one hand, generalisations will be justified through the fact that there is some positive ground, some good reason to make them. But on the other hand we will most often lose the mathematical performance guarantees. It will then be a matter of expertise to decide if some extra bias has been added to the system, and then to know if this bias is desired or not.

Let us discuss this point a little further. Suppose the task is learning DFA from text, and the algorithm takes as starting point the prefix tree acceptor, then performs different merges between states whenever this seems like a good idea. In this case the 'good idea' might be something like 'if it contributes to diminish the size of the automaton'. Then you will have added such a heavy bias that your learning algorithm will return always the universal automaton that recognises any string! Obviously this is a simple example that would not fool anyone for long. But if we follow on with the idea, it is not difficult to come up with algorithms whose task is to learn context-free grammars, but whose construction rules are such that only grammars that generate regular languages can be effectively learnt!

As, when dealing with these heuristics, the option of identification does not make sense, it is going to be remarkably difficult to decide when a given algorithm has such a hidden bias or not.

On the other hand, there are a number of reasons for which researchers in artificial intelligence have worked thoroughly in searching for new heuristics for grammatical inference problems:

- The sheer size of the search spaces makes the task of interest. When describing the set of admissible solutions as a partition lattice (see Section 6.3.4), the size of this lattice increased in a dramatic way (defined by the Bell formula) with the size of the positive data.
- We also saw in Chapter 6 that the basic operations related with the different classes of grammars and automata were intractable: the equivalence problem, the 'smallest consistent' problem.
- Furthermore the challenging nature of the associated $\mathcal{NP}$-hard problems is also something to be taken into account. Even if $\mathcal{NP}$ is in theory a unique class, there are degrees of approximation that can be different from problem to problem, and it is well known by scientists working on the effective resolution of $\mathcal{NP}$-hard problems that some can be tackled easier than others, at least on particular instances, or that the size of the tractable instances of the problem may vary from one problem to another. In the case of learning DFA, the central problem concerns solving

'minimum consistent DFA', a problem for which only small instances seem to be tractable by exhaustive algorithms, 'small' corresponding to less than about thirty states in the target.

- A fourth reason is that the point of view we have defended since the beginning, that there is a target language to be found, can in many situations not be perfectly adapted: In the case where we are given a sample and the problem is to find the smallest consistent grammar, this corresponds to trying to solve an intractable but combinatorially well defined problem; there are finally cases where the data is noisy or where the target is moving.

The number of possible heuristics is very large, and we will only survey here some of the main ideas that have been tried in the field.

14.1 A survey of some artificial intelligence ideas

We will comment at the end of the chapter that more techniques could be tested; indeed one could almost systematically take an artificial intelligence text-book and choose some meta-heuristic method for solving hard problems, and then try to adapt it to the task of learning grammars or automata. We only give the flavour of some of the better studied ideas here. In the next sections we will describe the use of the following techniques in grammatical inference:

- genetic algorithms,
- Tabu search,
- using the MDL principle,
- heuristic greedy search,
- constraint satisfaction.

We aim here to recall the key ideas of the technique and to show, in each case, through a very brief example, how the technique can be used in grammatical inference. The goal is certainly not to be technical nor to explain the finer tuning explanations necessary in practice, but only to give the idea and to point, in the bibliographical section, to further work with the technique.

14.2 Genetic Algorithms

The principle of genetic algorithms is to simulate biological modifications of genes and hope that via evolutionary mechanisms, nature increases the quality of its population. The fact that both bio-computing and grammatical

inference deal with languages and strings adds a specific flavour to this approach here.

14.2.1 Genetic algorithms: general approach

Genetic algorithms maintain a population of strings that each encode a given solution to the learning problem, then by defining the specific genetic operators allowing this population to evolute and better itself (through an adequacy to a given fitness function).

In the case where the population is made of grammars or automata supposed to somehow better describe a learning sample, a certain number of issues should be addressed:

(i) What is the search space? Does it comprise all strings describing grammars or only those that correspond to correct ones?

(ii) How do we build the first generation?

(iii) What are the genetic operators? Typically some sort of mutation should exist: A symbol in the string can mutate into another symbol. Also a crossing-over operator is usually necessary: This operation takes two strings, mixes them together in some way to obtain the siblings for the next generation.

(iv) What happens when, after an evolution, the given string does not encode a solution any more? One may consider having *stopping sequences* in the string so that non-encoding bits can be blocked between these special sequences. This is quite a nice idea leading to interesting interpretations about what is known as *junk* DNA.

(v) What fitness function should be used? How do we compare the quality of two solutions?

There are also many other parameters that need tuning, such as the number of generations, the number of elements of a generation that should be kept, quantities of genetic operations that should take place etc.

Mechanisms of evolution are essentially of two types (gene level): mutation and crossing-over.

Mutation consists in taking a string and letting one of the letters be modified. For example, in Figure 14.1, the third symbol is substituted. When implemented, the operation consists in randomly selecting a position, and (again randomly) modifying this symbol.

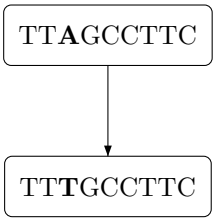


Fig. 14.1. Mutation.

Crossing-over is more complex and involves two strings. They both get cut into two substrings, and then the crossing-over position takes place, with an exchange of the halves. In some cases the position where the strings is cut has to be the same. We give in Figure 14.2 an example of this: Two strings are divided at the same position (here after the fourth symbol), then crossing-over takes place.

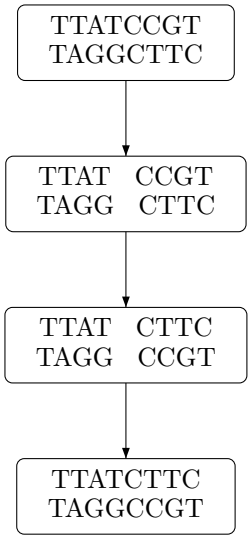


Fig. 14.2. Crossing-over.

14.2.2 A genetic algorithm in grammatical inference

We show here a simple way of implementing the different points put forward in the previous section for the specific task of learning NFA from positive and negative examples. We are given a sample $\langle S_+, S_- \rangle$.

(i) *What is the search space?* We consider the lattice as defined in

Section 6.3, containing all NFA strongly structurally complete with $\langle S_+, S_- \rangle$. Each NFA can therefore be represented by a partition of the states over MCA(S_+). There are several ways to represent partitions as strings. We use the following formalism: let $|E| = n$, and Π be a partition, $\Pi = \{B_1, B_2, \ldots, B_k\}$ of E. We associate with partition Π the string w_Π : $w_\Pi(j) = m \iff j \in B_m$.

For example, partition $\{\{1, 2, 6\}, \{3, 7, 9, 10\}, \{4, 8, 12\}, \{5\}, \{11\}\}$ is encoded by the string w_Π=(112341232253).

(ii) *How do we generate the first generation?* We randomly start from MCA(S_+) (see Definition 6.3.2, page 144), make a certain number of merges, obtaining in that way a population of NFA, all in the lattice and all (weakly) structurally complete.

(iii) *What are the genetic operators?* Structural operators are structural mutation and structural crossing-over over the strings w_Π. We illustrate this in Example 14.2.1.

(iv) *What happens when, after an evolution, the given string does not encode a grammar any more?* This problem does not arise here, as the operators are built to remain inside the lattice.

(v) *What fitness function should be used?* Here the two key issues are the number of strings from S_- accepted and the size of the NFA. We obviously want both as low as possible. One possibility is even to discard any NFA such that $\mathbb{L}(\mathcal{A}) \cap S_- \neq \emptyset$.

Example 14.2.1 *We start with a* MCA *for sample* $S_+ = \{\texttt{aaaa}, \texttt{abba}, \texttt{baa}\}$ *as represented in Figure 14.3 Consider an element of the initial population obtained by using the partition represented by string* $w_\Pi = (112341232253)$. *This* NFA *is drawn in Figure 14.4(a). Then if the second position is selected and the* 1 *is substituted by a* 4 *in string* w_Π*, we obtain string* (142341232253). *Building the corresponding partitions is straightforward.*

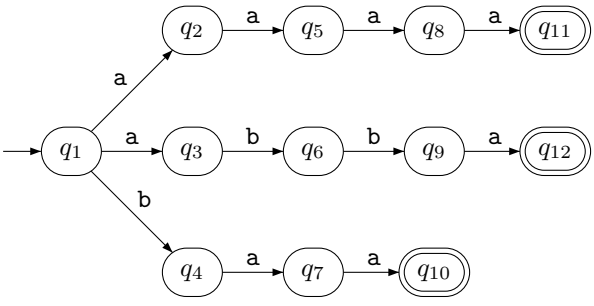
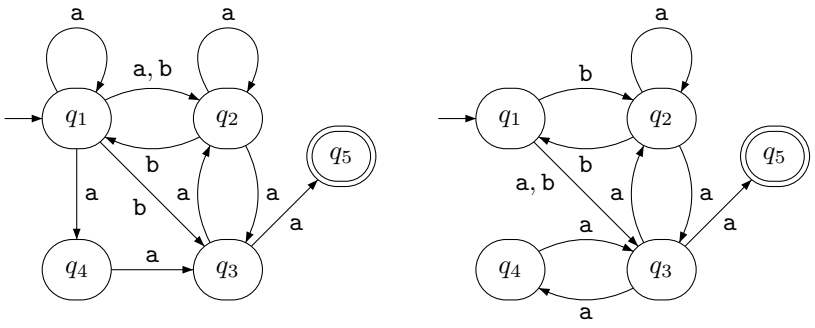


Fig. 14.3. The MCA for $S_+ = \{\texttt{aaaa}, \texttt{abba}, \texttt{baa}\}$.



(a) The NFA for w_Π = (112341232253).

(b) The NFA after the mutation of 1 to 4 in position 2.

Fig. 14.4. General title.

Now a crossing-over between two partitions is described in Figure 14.5. The partitions are encoded as strings, which are cut and mixed, resulting in two different partitions.

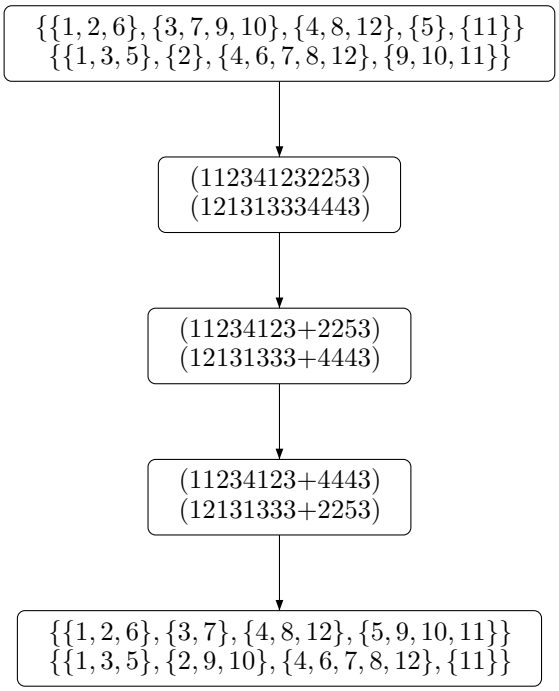


Fig. 14.5. Crossing-over with partitions.

14.3 Tabu search

When searching in large spaces, *hill-climbing* techniques try to explore the space progressively from a starting point to a local optimum, where quality is measured through a fitness function. As the optimum is only local, to try to visit further the space, different ideas have been proposed, one of which corresponds to using *Tabu lists*, or lists of operations that are forbidden, at least for a while.

14.3.1 What is Tabu search?

Tabu search also requires the definition of a search space, then the definition of some local operators to move around this space: Each solution has neighbours and all these neighbours should be measured, the best (for a given fitness function) being kept for the next iteration. The idea is to iteratively try to find a neighbour of the current solution, that betters it. In order to avoid going through the same elements over and over, and getting out of a local optimum, a *Tabu* list of the last few moves is kept, and the algorithm chooses an element outside this list.

There is obviously an issue in reaching a local optimum for the fitness function. To get out of this situation (*i.e.* all the neighbours are worse than the current solution) some sort of major change has to be made.

This sort of heuristic depends strongly on the tuning of a number of parameters. We will not discuss these here as they require to take into account a large number of factors (size of the alphabet, the target,...).

14.3.2 A Tabu search algorithm for grammatical inference

The goal is to learn regular languages, defined here by NFA, from an informant. An inductive bias is proposed: The number of states in the automaton (or at least an upper bound of this number) is fixed. The search space is the set of all NFA with n states, and the neighbour relation is given by the fact that from one NFA to another, the addition or the removal of just one transition.

(i) *What is the search space?* The search space is made of λ-NFA with n states out of which one (q_A) is accepting and all the others are rejecting. Furthermore q_A is reachable by λ-transitions only, and λ-transitions can only be used for this. There is no transition from q_A. An element in this space is represented in Figure 14.6.

(ii) *What are the local operators?* Adding and removing a transition.

If a transition is added it has to comply with the above rules. For example, in the automaton represented in Figure 14.6, any of the transitions could be removed, and transitions could be added connecting states q_1, q_2 and q_2, or λ-transitions leading to state q_A.

(iii) *What fitness function should be used?* We count here the number of strings in S correctly labelled by the NFA. Function v will therefore just parse the sample and return the number of errors.

(iv) *How do we initialise?* In theory, any n state automaton complying with the imposed rules would be acceptable.

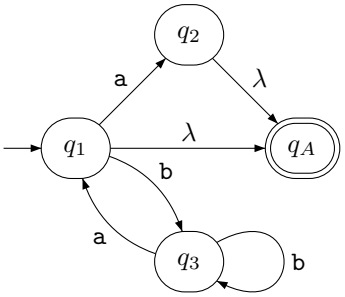


Fig. 14.6. A Tabu automaton.

We denote by :

- $\mathcal{A}*$ is the best solution reached so far;
- T is the Tabu list of transitions that have been added or removed in the last m moves;
- kmax is an integer bounding the number of iterations of the algorithm;
- Q is a set of $n-1$ states, $q_A \notin Q$ is the unique accepting state;
- we denote the rules by triples: $R = Q \times \Sigma \times Q \cup Q \times \{\lambda\} \times \{q_A\}$. Therefore $(q, a, q') \in R \iff q' \in \delta_N(q, a)$.

14.4 MDL principle in grammatical inference

The minimum description length principle states that the best solution is one that minimises the combination of the encoding of the grammar and the encoding of the data when parsed by the grammar.

But the principle obeys to some very strict rules. To be more exact the way both the grammar and the data should be encoded depends on a universal Turing machine, and has to be handled carefully.

Algorithm 14.1: TABU.

Input: a sample $S = \langle S_+, S_- \rangle$, a fitness function v, an integer kmax, an initial NFA $\mathcal{A}$
Output: an NFA
$k \leftarrow 0$;
$T \leftarrow \emptyset$;
$\mathcal{A}* \leftarrow \mathcal{A}$;
while $k \neq$ kmax **do**
  build R the set of admissible transitions that can be added or removed;
  select r in $R \setminus T$, such that the addition or deletion of r to or from $\mathcal{A}$ realizes the maximum of v on S;
  add or delete r from $\mathcal{A}$;
  if $v(\mathcal{A}) > v(\mathcal{A}*)$ **then** $\mathcal{A}* \leftarrow \mathcal{A}$;
  TABU-UPDATE(T, r);
  $k \leftarrow k + 1$
end
return $\mathcal{A}*$

Algorithm 14.2: TABU-UPDATE(T, r).

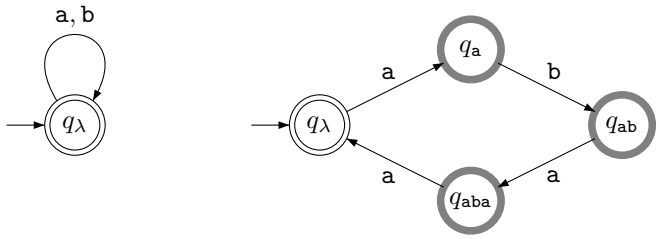
Input: the Tabu list T, its maximal size m, the new element r
Output: T
if CARD(T) $= m$ **then** delete its last element;
Add r as the first element of T;
return T

14.4.1 What is the MDL *principle?*

The MDL principle is a refinement of the Occam principle: Remember that the Occam principle tells us that between various hypothesis, one should choose the simplest. The notion of 'simplest' here refers to some fixed notation system. The MDL principle adds the fact that simplicity should be measured also in the way the data is explained by the hypothesis. That means that the simplicity of a hypothesis (with respect to some data S) is the sum between the size of the encoding of the hypothesis and the size of the encoding of the data where the encoding of the data can be dependent of the hypothesis.

As a motivating example take the case of a sample containing strings `abaa`, `abaaabaa`, and `abaaabaaabaaabaa`. A learning algorithm may come up with

either of the two automata depicted in Figure 14.7. Obviously the left-hand side one (Figure 14.7(a)) is easier to encode than the right-hand side one (Figure 14.7(b)). But on the other hand the first automaton does not help us reduce the size of the encoding of the data, whereas using the second one, the data can easily be encoded as something like $\{1, 2, 4\}$, denoting the number of cycles one should make to generate each string.



(a) A very simple DFA. (b) A more complex one.

Fig. 14.7. Two candidates for sample $S_+ = \{\mathtt{abaa}, \mathtt{abaaabaa}, \mathtt{abaaabaaabaaabaa}\}$.

14.4.2 A simple MDL *algorithm for grammatical inference*

To illustrate these ideas let us try to learn a DFA from text. We are given a positive sample S_+.

Let us define the score of an automaton as the number of states of the DFA multiplied by the size of the alphabet. This is of course questionable, and should only be considered for a first approach; Since this size is supposed to be compared with the size of the data, it is essential that the size is fairly computed. Ideally, the size should be that of the smallest Turing machine whose output is the automaton...

Then, given a string w and a DFA $\mathcal{A}$, we can encode the string w depending on the number of choices we have at each stage. For example, using the arguments discussed above for the automata from Figure 14.7, we just have to encode a string by the numbers of its choices every time a choice has to be made. We therefore associate with each state of $\mathcal{A}$ the value $\mathrm{ch}(q)=\log\big(|\{a \in \Sigma : \delta(q,a) \text{ is defined}\}|\big)$ if $q \notin \mathbb{F}_{\mathbb{A}}$. If $q \in \mathbb{F}_{\mathbb{A}}$ then $\mathrm{ch}(q)=\log\big(1 + |\{a \in \Sigma : \delta(q,a) \text{ is defined}\}|\big)$, since one more choice is possible. The value ch corresponds to the size of the encoding of the choices a parser would have in that state. So in the automaton 14.7(b), we have $\mathrm{ch}(q_\lambda)=\log 2$, $\mathrm{ch}(q_{\mathtt{a}})=\mathrm{ch}(q_{\mathtt{ab}})=\mathrm{ch}(q_{\mathtt{aba}})=\log 1=0$. The fact that no cost is counted corresponds to the idea that no choice has to be made and is also consistent with $\log 1 = 0$.

Algorithm 14.3: MDL.

Input: S_+ function CHOOSE
Output: $\mathcal{A} = \langle \Sigma, Q, q_\lambda, \mathbb{F}_\mathbb{A}, \mathbb{F}_\mathbb{R}, \delta \rangle$
$\mathcal{A} \leftarrow$ Build-PTA(S_+);
RED $\leftarrow \{q_\lambda\}$;
current_score $\leftarrow \infty$;
BLUE $\leftarrow \{q_a : a \in \Sigma$ and $S_+ \cap a\,\Sigma^\star \neq \emptyset\}$;
while BLUE $\neq \emptyset$ **do**
  $q_b \leftarrow$ CHOOSE(BLUE);
  BLUE $\leftarrow$ BLUE $\setminus \{q_b\}$;
  if $\exists q_r \in$ RED : sc(MERGE$(q_r, q_b, \mathcal{A})$,S_+)<current_score **then**
    $\mathcal{A} \leftarrow$MERGE$(q_r, q_b, \mathcal{A})$;
    current_score $\leftarrow$ sc($\mathcal{A}$,S_+)
  else
    $\mathcal{A} \leftarrow$ PROMOTE$(q_b, \mathcal{A})$
  end
end
for $q_r \in$ RED **do**
  if $\mathbb{L}(\mathcal{A}_{q_r}) \cap S_+ \neq \emptyset$) **then** $\mathbb{F}_\mathbb{A} \leftarrow \mathbb{F}_\mathbb{A} \cup \{q_r\}$
end
return $\mathcal{A}$

From this we define the associated value ch(w)=ch(q_λ, w) which determines the size of the encoding of the path followed to parse string w, which depends on the recursive definition: ch(q, λ)= ch(q_λ) and ch($q, a \cdot w$)= ch(q)+ch($\delta_\mathcal{A}(q, a), w$).

We can now, given a sample S_+ and a DFA $\mathcal{A}$, measure the score sc of $\mathcal{A}$ and S_+, (denoted by sc($\mathcal{A}, S_+$)) as $\|\mathcal{A}\| \cdot |\Sigma| + \sum_{w \in S_+} \text{ch}(w)$, where $\|\mathcal{A}\|$ is the number of states of $\mathcal{A}$.

We can build a simple state merging algorithm (Algorithm MDL, 14.3) which will merge states until it can no longer lower the score. The operations MERGE and PROMOTE are as in Chapter 12.

The training sample is $S_+ = \{\mathtt{a}, \mathtt{a}^2, \mathtt{b}^2, \mathtt{a}^3, \mathtt{b}^2\mathtt{a}, \mathtt{a}^4, \mathtt{a}\mathtt{b}^2\mathtt{a}, \mathtt{b}^4\}$. From this we build PTA(S_+), depicted in Figure 14.8. We compute the score of the running solution and we get $13\log(2) + 8\log(3)$ for the derivations, and 26 for the PTA. The total is therefore sc($\mathcal{A}, S_+$)=$39 + 8\log(3) \approx 51.68$.

The exact computations of the ch(x) can be found in Table 14.1.
Merge $q_\mathtt{a}$ with q_λ is tested; This requires recursive merging (for determinisation). The resulting automaton is represented in Figure 14.9. The new

$\|\mathcal{A}\|$	$\mathtt{a}$	$\mathtt{a}^2$	$\mathtt{b}^2$	$\mathtt{a}^3$
12	$1+\log 3$	$2+\log 3$	$1+\log 3$	$3+\log 3$
6	$2\log 3$	$3\log 3$	$2\log 3$	$4\log 3$
2	$2\log 3$	$3\log 3$	$2\log 3$	$4\log 3$
1	2	3	3	4
$\|\mathcal{A}\|$	$\mathtt{b}^2\mathtt{a}$	$\mathtt{a}^4$	$\mathtt{ab}^2\mathtt{a}$	$\mathtt{b}^4$
12	$1+\log 3$	$3+\log 3$	$1+\log 3$	$1+\log 3$
6	$2\log 3$	$5\log 3$	$3\log 3$	$2\log 3$
2	$3\log 3$	$5\log 3$	$4\log 3$	$3\log 3$
1	4	5	5	5

Table 14.1. *Computations of all the* $\mathrm{ch}(x)$*, for the* PTA *and the different automata*

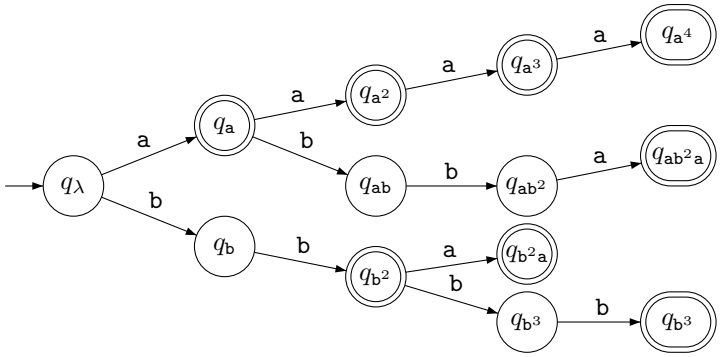


Fig. 14.8. $\mathrm{PTA}(S_+)$ where $S_+ = \{\mathtt{a}, \mathtt{a}^2, \mathtt{b}^2, \mathtt{a}^3, \mathtt{b}^2\mathtt{a}, \mathtt{a}^4, \mathtt{ab}^2\mathtt{a}, \mathtt{b}^4\}$.

automaton has values $\mathrm{ch}(q_\lambda)$=$\mathrm{ch}(q_{\mathtt{b}^2})$=$\log 3$, and $\mathrm{ch}(q_\mathtt{b})$=$\mathrm{ch}(q_{\mathtt{b}^2\mathtt{a}})$=$\mathrm{ch}(q_{\mathtt{b}^3})$=$\mathrm{ch}(q_{\mathtt{b}^4})$=$\log 1$=0. So $\mathrm{sc}(\mathcal{A}, S_+)$ can be computed as:

$6 \cdot 2 + 23\log(3) < 36 + 11\log(3)$ (roughly 48.45 against 51.68), the merge is accepted.

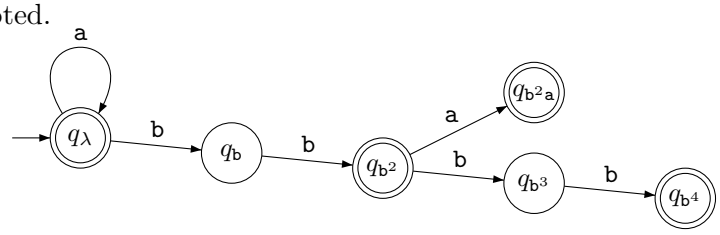


Fig. 14.9. We merge $q_\mathtt{a}$ with q_λ.

We try to merge $q_\mathtt{b}$ with q_λ and obtain the universal automaton whose score is 2 for the DFA + $31\log(3)$. This gives a score of 51.13, which is

more than the score of our current solution (with 6 states). Therefore the merge is rejected. $q_{\mathtt{b}}$ is promoted and we test merging $q_{\mathtt{ab}}$ with q_λ. After determinisations we obtain the 2 state automaton depicted in Figure 14.10. The score of this DFA is 43.68, which is better than the current best. Since no more merges are possible the algorithm halts with this DFA as solution.

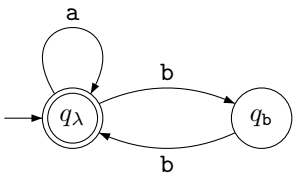


Fig. 14.10. The returned solution.

Note that by taking a different scoring scheme, the result would have been very different (see Exercises 14.7 and 14.8).

14.5 Heuristic Greedy State Merging

When we described RPNI (Section 12.4), we presented it as a deterministic algorithm. Basically, the order in which the compatibilities are checked is defined right from the start. Moreover, as soon as two states are mergeable these are merged. This is clearly an optimistic point of view, and there may be another, based on choosing the *best* merge.

But one should remember that RPNI identifies in the limit. This may well no longer be the case if we use a heuristic to define the best possible merge: One can usually imagine a (luckily counter-intuitive) distribution that will make us explore the lattice the wrong way.

14.5.1 How do greedy state merging algorithms work?

The general idea of a *greedy* state merging algorithm is as follows:

- choose two states,
- perform a cascade of forced merges until the automaton is deterministic,
- if this automaton accepts some sentences from S_-, backtrack and choose another couple,
- if not, loop until no merging is still possible.

Now how are the moves chosen? Consider the current automaton for which RPNI has to make a decision: What moves are allowed?

There are two possibilities:

- merging a BLUE with a RED,
- promoting a BLUE to RED and changing all its successors that are not RED to BLUE.

Promotion takes place when a BLUE state can be merged with no RED state. This means that this event (similar to having a row obviously different in Gold's algorithm) has to be systematically checked.

But once there are no possible promotions, the idea is to do better than RPNI and, instead of greedily checking in order to find the first admissible merge, to check all possible legal merges between a BLUE state and a RED state, compute a score for each merge, and then choose the merge with highest score.

14.5.2 Evidence driven state merging (EDSM)

The evidence driven approach (see Algorithm 14.4) consists in computing for every pair of states (one BLUE, the other RED) the score of that merge as the number of strings that end in a same state if that merge is done. To do that the strings from S_+ and S_- have to be parsed. If by doing that merge a conflict arises (a negative string is accepted or a positive string is rejected) the score is $-\infty$. The merge with the highest score is chosen.

Algorithm 14.4: EDSM-COUNT.

```
Input: A, S+, S−
Output: the score sc of A
for q ∈ Q do tp[q] ← 0 ; tn[q] ← 0 ;
for w ∈ S+ do tp[δA(qλ, w)] ← tp[δA(qλ, w)] + 1 ;
for w ∈ S− do tn[δA(qλ, w)] ← tn[δA(qλ, w)] + 1 ;
sc← 0;
for q ∈ Q do
    if sc≠ −∞ then
        if tn[q] > 0 then
            if tp[q] > 0 then sc← −∞ else sc← sc+tn[q]-1
        else
            if tp[q] > 0 then sc← sc+tp[q]-1
        end
    end
end
return sc
```

The corresponding algorithm (Algorithm EDSM, 14.5) is given with the

specific counting scheme (Algorithm EDSM-COUNT, 14.4). The merging function is exactly the one (Algorithm 12.11) introduced in Section 12.4.

Algorithm 14.5: EDSM $\mathcal{A}$.

Input: $S = \langle S_+, S_- \rangle$, functions COMPATIBLE, CHOOSE
Output: $\mathcal{A} = \langle \Sigma, Q, q_\lambda, \mathbb{F}_\mathbb{A}, \mathbb{F}_\mathbb{R}, \delta \rangle$
$\mathcal{A} \leftarrow$ BUILD-PTA(S_+); RED $\leftarrow \{q_\lambda\}$; BLUE $\leftarrow \{q_a : a \in \Sigma$ and $S_+ \cap a\,\Sigma^\star \neq \emptyset\}$;
while BLUE $\neq \emptyset$ **do**
  promotion$\leftarrow$ **false**;
  for $q_b \in$ BLUE **do**
    if *not* promotion **then**
      $bs \leftarrow -\infty$;
      atleastonemerge$\leftarrow$ **false**;
      for $q_r \in$ RED **do**
        $s \leftarrow$EDSM-COUNT(MERGE($q_r, q_b, \mathcal{A}$),S_+, S_-);
        if $s > -\infty$ **then** atleastonemerge$\leftarrow$ **true**
        if $s > bs$ **then** $bs \leftarrow s$; $\overline{q_r} \leftarrow q_r$; $\overline{q_b} \leftarrow q_b$
      end
      if *not* atleastonemerge **then** `/* no merge is possible */`
        PROMOTE($q_b, \mathcal{A}$); promotion$\leftarrow$ **true**;
      end
    end
  end
  if *not* promotion **then** `/* we can merge */`
    BLUE $\leftarrow$ BLUE $\setminus \{\overline{q_b}\}$; $\mathcal{A} \leftarrow$MERGE($\overline{q_r}, \overline{q_b}, \mathcal{A}$)
  end
end
for $x \in S_+$ **do** $\mathbb{F}_\mathbb{A} \leftarrow \mathbb{F}_\mathbb{A} \cup \{\delta(q_\lambda, x)\}$;
for $x \in S_-$ **do** $\mathbb{F}_\mathbb{R} \leftarrow \mathbb{F}_\mathbb{R} \cup \{\delta(q_\lambda, x)\}$;
return $\mathcal{A}$

Example 14.5.1

$$\begin{aligned} S_+ &= \{\mathtt{a}, \mathtt{aaa}, \mathtt{bba}, \mathtt{abab}\} \\ S_- &= \{\mathtt{ab}, \mathtt{bb}\} \end{aligned}$$

Consider the DFA *represented in Figure 14.11. States q_λ and $q_\mathtt{a}$ are* RED, *whereas states $q_\mathtt{b}$ and $q_\mathtt{ab}$ are* BLUE. *We compute the different scores for the sample, if we suppose that the* BLUE *state selected for merging is $q_\mathtt{b}$:*

In both cases the merge is actually tested, and counting (via Algorithm EDSM-COUNT *(14.4) is done.*

$$\begin{aligned} S_+ &= \{\mathtt{a}, \mathtt{aaa}, \mathtt{aba}, \mathtt{bba}, \mathtt{abab}\} \\ S_- &= \{\mathtt{ab}, \mathtt{bb}\} \end{aligned}$$

State $q_{\mathtt{ab}}$ *is now selected and the counts are computed:*

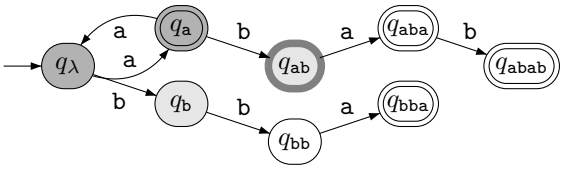


Fig. 14.11. The DFA before attempting a merge.

- EDSM-COUNT(MERGE$(q_\lambda, q_{\mathtt{ab}}, \mathcal{A})$)$=-\infty$, *because this consists in merging* $q_{\mathtt{ab}}$ *with* $q_{\mathtt{abab}}$.
- EDSM-COUNT(MERGE$(q_{\mathtt{a}}, q_{\mathtt{ab}}, \mathcal{A})$)$=-\infty$, *because this consists in merging* $q_{\mathtt{a}}$ *with* $q_{\mathtt{ab}}$.

Therefore a promotion takes place: Since we have a BLUE *which can be merged, the* DFA *is updated with state* $q_{\mathtt{ab}}$ *promoted to* RED *(see Figure 14.12).*

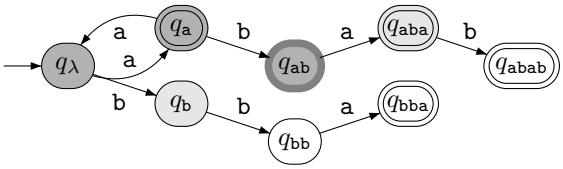


Fig. 14.12. The DFA after the promotion of $q_{\mathtt{ab}}$.

Suppose instead state b *was selected. The counts are now different*

- EDSM-COUNT(MERGE$(q_\lambda, q_{\mathtt{b}}, \mathcal{A})$)$=2$
- EDSM-COUNT(MERGE$(q_{\mathtt{a}}, q_{\mathtt{b}}, \mathcal{A})$)$=3$

In this case, the merge between $q_{\mathtt{a}}$ *and* $q_{\mathtt{b}}$ *would be selected.*

There are different ways to perform the possible operations but what characterises the evidence driven state merging techniques is that one should be careful to always check first if some promotion is possible.

A different idea is to use a heuristic to decide before testing consistency in what order the merges should be checked. This is of course cheaper, but the problem is that promotion is then hard to detect. In practice this approach (called ***data driven***) has not proven to be successful, at least in the deterministic setting. When trying to learn probabilistic automata, it seems that data driven state merging is a good option.

14.6 Graph colouring and constraint satisfaction

The PTA can be seen as a graph for which the goal is to obtain a colouring of the nodes respecting a certain number of conditions. These conditions can be described as constraints, and again the nodes of the graph have to be valued in a way satisfying a set of constraints. Alas these constraints are dynamic (some constraints will depend on others).

There are too many options to mention them all here, we just describe briefly how we can convert the problem of learning a DFA from an informed sample $\langle S_+, S_- \rangle$ into a constraint satisfaction question.

We first build the complete prefix tree acceptor PTA(S_+, S_-) using Algorithm 12.1, page 281. Let us suppose the PTA has m states.

Now consider the graph whose nodes are the states of the PTA and where there is an edge between two nodes/states q and q' if they are incompatible, *i.e.* they cannot be merged. Then the problem is to find a colouring of the graph (no two adjacent nodes can take the same colour) with a minimum number of colours.

An alternative problem whose resolution can provide us with a partition is that of building cliques in the graph of consistency.

Technically things are a little more complex, since the constraints are dynamic: Choosing to colour two nodes with a given colour corresponds to merging the states, with the usual problems relating to determinism.

Nevertheless there are many heuristics that have been tested for these particular and well known problems.

Let us model further the problem. We consider (given the PTA) m variables $X_1, \ldots, X_m$, and n possible values 1,..., n, corresponding to the n states of the target automaton (which supposes we take an initial gamble on the size of the intended target).

One can describe three types of constraints:

- global constraints: $q_i \in \mathbb{F}_{\mathbb{A}}$, $q_j \in \mathbb{F}_{\mathbb{R}} \implies X_i \neq X_j$;
- propagation constraints: $X_k \neq X_l \wedge \delta(q_i, a) = q_k \wedge \delta(q_j, a) = q_l \implies X_i \neq X_j$;
- deterministic constraints: $\delta(q_i, a) = q_j \wedge \delta(q_k, a) = q_l \implies \big[X_i = X_k \implies X_j = X_l\big]$.

Note that the deterministic constraints are dynamic: They will only be used when the algorithm starts deciding to effectively colour some states.

Between the different systems used to solve such constraints, conflict diagnosis (using intelligent backtracking) has been used.

Example 14.6.1 *Consider the* PTA *represented in Figure 14.13. Then a*

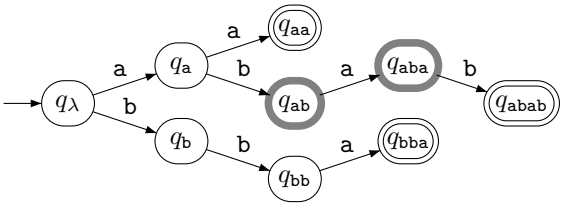


Fig. 14.13. PTA($\{(\mathtt{aa}, 1)\,(\mathtt{aba}, 0)\,(\mathtt{bba}, 1)\,(\mathtt{ab}, 0)\,(\mathtt{abab}, 1)\}$).

certain number of initial global constraints can be established. If we denote by $\langle X_i, X_j\rangle$ *the constraint: "*q_i *and* q_j *cannot be merged", we have by taking all pairs of states, one being accepting and the other rejecting:*

Initial constraints: $\langle X_{\mathtt{aa}}, X_{\mathtt{ab}}\rangle$, $\langle X_{\mathtt{aa}}, X_{\mathtt{aba}}\rangle$, $\langle X_{\mathtt{abab}}, X_{\mathtt{ab}}\rangle$, $\langle X_{\mathtt{abab}}, X_{\mathtt{aba}}\rangle$, $\langle X_{\mathtt{bba}}, X_{\mathtt{ab}}\rangle$, $\langle X_{\mathtt{bba}}, X_{\mathtt{aba}}\rangle$.

We can now represent some of the propagation constraints, where we use the rule $\langle X_{ua}, X_{va}\rangle \implies \langle X_u, X_v\rangle$.

$\langle X_{ua}, X_{va}\rangle \implies \langle X_u, X_v\rangle$

This, when propagated, gives us $\langle X_{\mathtt{a}}, X_{\mathtt{ab}}\rangle$, $\langle X_{\mathtt{ab}}, X_{\mathtt{bb}}\rangle$, $\langle X_{\mathtt{a}}, X_{\mathtt{b}}\rangle$, $\langle X_{\mathtt{a}}, q_{\mathtt{aba}}\rangle$, $\langle X_\lambda, X_{\mathtt{ab}}\rangle$.

14.7 Exercises

14.1 Write the different missing algorithms for the genetic algorithms.

14.2 Find a difficult language to identify with a genetic algorithm.

14.3 The definition of the value function v for the Tabu search method is very naive. Can we do better?

14.4 Find a difficult language to identify with the Tabu search algorithm, using a fixed k.

14.5 If the target is a complete automaton, then the MDL algorithm will perform poorly. Why?

14.6 What would be a good class of DFA that for the MDL algorithm? Hint: one may want to have large alphabets but only very few transitions. A definition in the spirit of Definition 4.4.1 (page 96 may be a good idea.

14.7 Using the same data as in Section 14.4, run the MDL algorithm with a score function that ignores the size of the alphabet, *i.e.* $\mathrm{sc}(\mathcal{A}, S_+))=\|\mathcal{A}\| + \sum_{w\in S_+}\mathrm{ch}(w)$.

14.8 Conversely, let us suppose we intend to represent the automaton as a table with three entries (one for the alphabet and two for the states). Therefore we could choose $\mathrm{sc}(\mathcal{A}, S_+))=\|\mathcal{A}\|^2 \cdot |\Sigma| + \sum_{w\in S_+}\mathrm{ch}(w)$

14.9 In Algorithm EDSM, the computation of the scores is very expensive. Can we combine the data driven and the evidence driven approaches in order to not have to compute all the scores but to be able to discover the promotion situations?

14.10 In Algorithm EDSM, once $\mathrm{sc}(q,q',\mathcal{A})$ is computed as $-\infty$, does it need to be recomputed? Is there any way to avoid such expensive recomputations?

14.11 Build the set of constraints corresponding to the PTA represented in Figure 14.14

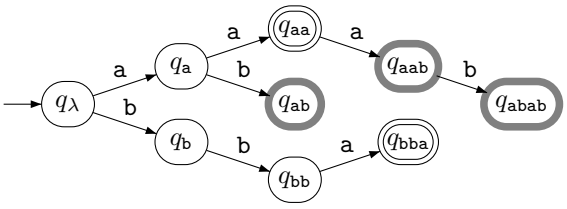


Fig. 14.14. PTA($\{(\mathtt{aa}, 1)\,(\mathtt{aab}, 0)\,(\mathtt{bba}, 0)\,(\mathtt{ab}, 0)\,(\mathtt{abab}, 1)\}$).

14.8 Conclusions of the chapter and further reading

14.8.1 Bibliographical background

Some of the key ideas presented in the first section of this chapter have been discussed in [dlH06a].

The question of the size of the DFA is a real issue. Whereas following on the line of the ABBADINGO competition [LP97], many authors were keen on finding algorithms working with targets of a few hundred states, there

has been also a group of researchers intersected in considering the purely combinatoric problem of finding the smallest DFA and coming up with some heuristics for this case [dOS01].

Presentation of Section 14.2 on genetic algorithms is based on work by many researchers but specially [Dup94, SK99, SM00]. Research in this area has taken place both from the grammatical inference perspective and from the genetic algorithms one. It is therefore essential, when wanting to study this particular topic, to look at the bibliography from both areas. An attempts to use genetic algorithms (using the term coevolutionary learning) in an active setting was made by Josh Bongard and Hod Lipson [BL05].

The MDL principle is known under different names [WB68, Ris78]. In grammatical inference some key ideas were introduced by Gerard Wolff [Wol78]. More recently his work was pursued by Pat Langley and Sean Stromstean [Lan95, LS00], George Petasis *et al.*[PPK$^+$04]. New ideas, in the case of learning DFA with MDL are by Pieter Adriaans and Ceriel Jacobs [AJ06].

Presentation of Section 14.3, concerning Tabu search, is based on work by Jean-Yves Giordano [Gio96]. A more general presentation of Tabu search, can be found in Fred Glover's book [GL97].

The main results concerning algorithm EDSM correspond to work done by Nick Price, Hugues Juillé and Kevin Lang during or after the ABBADINGO competition [LPP98, Lan99]. The cheaper (but also worse) data driven approach was used by Colin de la Higuera *et al.* [dlHOV96]

Presentation of Section 14.6 is based on work by Alan Biermann [Bie71], Arlindo de Oliveira and João Marques Silva[dOS98], François Coste [CN98b, CN98a].

Pure heuristics are problematic in that they can introduce an unwanted, undeclared added bias. Typically if the intended class of grammars is different from the one that is really going to be learnt, something is wrong.

An interesting alternative is to base a heuristic on a provably convergent algorithm. There is still the necessity to study what is really happening in this case, but a prudent guess is that somehow one is keeping control of the bias.

14.8.2 Some alternative lines of research

- A very different approach to learn context-free grammars has been followed in system SYNAPSE by Katsuhiko Nakamura and his colleagues [NM05]: The goal there is to learn directly and inductively the CKY parser. The system is in part incremental. Along similar lines have

been tried genetic algorithms, with the same goal of learning the parser [SK99, SM00].

- Neural networks have been used in grammatical inference with varying degrees of success. Among the best known papers are those by René Alquezar and Alberto Sanfeliu, Lee Giles, Mikel Forcada, and their colleagues [AS94, CFS96, GLT01]. In some cases a mixture of numerical and symbolic techniques were used: The symbolic grammatical inference allowing to better find the parameters for the recurrent network and conversely the network allowing to decide compatibility of states. An important question is that of extracting an automaton from a learnt neural network, so as to avoid the black-box effect. In this case, we can be facing an interesting task of interactive learning in which the neural network can play the part of the Oracle.

14.8.3 Open problems and possible new lines of research

There is obviously a lot of work possible, once the limits of provable methods are well understood. Let us discuss a certain number of elements of reflection:

- The GOLD algorithm gives an interesting basis for learning as it redefines the search space. This should be considered as a good place to start from. Moreover, the complexity of the algorithm can be considerably reduced through a careful use of good data structures.
- Use of semantic information in real tasks should be encouraged. This semantic information needs to be translated into syntactic constraints that in turn could improve the algorithms.
- EDSM is a good example of what should be done: The basis is a provable algorithm (RPNI) in which the greediness is controlled by a *common sense* function instead of by an arbitrary order.