Lecture 7: Planning and Learning with Tabular Methods

Tianpei Xie

Aug 8th., 2022

Contents

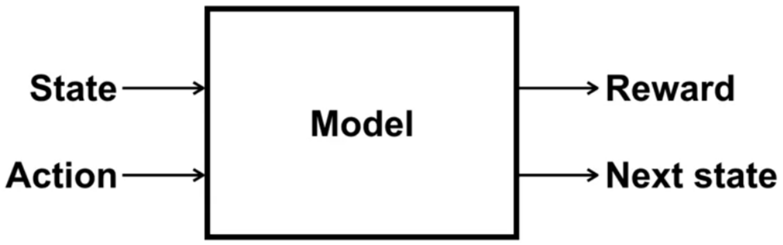


Figure 1: A model in reinforcement learning

1 Introduction

There is a unified view of reinforcement learning methods:

- ***model-based*** reinforcement learning methods: Model-based methods rely on ***planning*** as their primary component. Examples include the *dynamic programming* methods and the *heuristic search* such as A^*.
- ***model-free*** reinforcement learning methods: Model-free methods primarily rely on ***learning***. Examples include *Monte Carlo methods* and *temporal difference (TD) methods*.

The heart of both kinds of methods is the <u>computation of value functions</u>. Moreover, all the methods are based on looking ahead to future events, computing a *backed-up* value, and then using it as an update target for an *approximate* value function.

2 Models and Planning

By a *model* of the environment we mean anything that an agent can use to predict how the environment will respond to its actions. Given a state and an action, a model produces a prediction of the resultant next state and next reward.

There are two type of models:

- ***distribution models***: models produce a description of *all possible* next-state-reward pairs and their probabilities. Dynamic Programming algorithms assume distribution models, since we need to know probability $p(s', r|s, a)$ for all next-state-reward pairs s', r given every state s and action a. Distribution models are *stronger* than sample models in that they can always be used to produce samples.

  A typical class of distribution models is the ***Probabilistic Graphical Model*** [Koller and Friedman, 2009] including the *Bayesian Networks*, *Markov Networks*, *Gaussian Graphical Models*, etc. These models use graph representation to *factorize* the global joint distribution between states, actions and rewards into several connected local factors embedded in subgraphs.

- ***sample models***: models produce just one of the possibilities, *sampled* according to the probabilities. Sample models can be sampling algorithms such as *importance sampling*, sequential

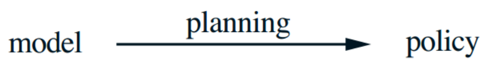


Figure 2: Planning

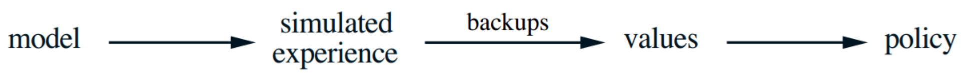


Figure 3: state-space planning structure

sampling like *Markov Chain Monte Carlo*, *Gibbs sampling*, or *Gradient Flow methods* etc.

Note that model is used to *simulate* the environment and produce *simulated experience*. This is in contrast to the actual environment which generates the *real experience*. The word ***planning*** is used in several different ways in different fields. We use the term to refer to any computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment. (Figure 2)

Given probabilistic models, one can use **sampling methods** [Liu and Liu, 2001] such as *Importance Sampling*, *Markov Chain Monte Carlo (MCMC)* such as *Gibs sampling*, *Metropolis Algorithm* etc to obtains samples of actions and states.

There are two type of planning:

- ***State-space planning*** is viewed primarily as a search through the *state space* for an *optimal policy* or an optimal path to a goal. Actions cause transitions from state to state, and value functions are computed over states. In *reinforcement learning*, we mainly concerned about state-space planning.

- ***Plan-space planning***, planning is instead a search through the space of plans. Operators transform one plan into another, and value functions, if any, are defined over the space of plans. Example includes **evolutionary algorithm** and **genetic algorithm**. Plan-space methods are difficult to apply efficiently to the stochastic sequential decision problems that are the focus in reinforcement learning.

All state-space planning methods discussed here share a common structure : there are two basic ideas: (1) all state-space planning methods involve **computing value functions** as a key intermediate step toward **improving the policy**, and (2) they compute value functions by updates or backup operations applied to **simulated experience**. This common structure can be diagrammed as follows: (Figure 3)

The heart of both *learning* and *planning* methods is the **estimation of value functions** by backing-up update operations. The difference is that whereas planning uses *simulated experience* generated by a model, learning methods use *real experience* generated by the environment. In particular, in many cases a learning algorithm can be *substituted* for the key update step of a planning method.

Figure 4 shows the ***random-sample one-step tabular Q-planning***, converges to the optimal policy for the model under the same conditions that one-step tabular Q-learning converges to the optimal policy for the real environment. Compared to the Q-learning, we need that the only difference is that the samples used to update the Q function is generated by the model. Also the

Random-sample one-step tabular Q-planning

```
Loop forever:
    1. Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(S)$, at random
    2. Send $S, A$ to a sample model, and obtain
           a sample next reward, $R$, and a sample next state, $S'$
    3. Apply one-step tabular Q-learning to $S, A, R, S'$:
           $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S,A)\big]$
```

Figure 4: Tabular Q-planning

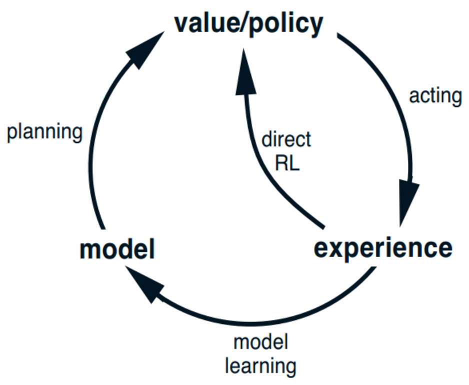


Figure 5: The generic relationship between model learning, direct RL, planning in a planning agent. Note that the value/policy can be updated either by real experience via direct RL or by simulated experience via planning. Meanwhile, the real experience will be used for model update/learning.

starting state and actions are selected at random.

3 Dyna: Integrated Planning, Acting, and Learning

When planning is done online, while interacting with the environment, a number of interesting issues arise. New information gained from the interaction may change the model and thereby interact with planning. It may be desirable to customize the planning process in some way to the states or decisions currently under consideration, or expected in the near future. If decision making and model learning are both computation-intensive processes, then the available computational resources may need to be divided between them.

Within a planning agent, there are at least two roles for **real experience**:

- ***model learning***: i.e. **improve the model** to make it more accurately *match* the real environment;
- ***direct reinforcement learning (direct RL)***: i.e. directly **improve the value function** and **policy** using the kinds of reinforcement learning methods. All RL methods discussed before can be used here.

Figure 5 describes the major components in a planning agent and their interactions. Note how experience can improve value functions and policies either *directly* or *indirectly* via the model. It is

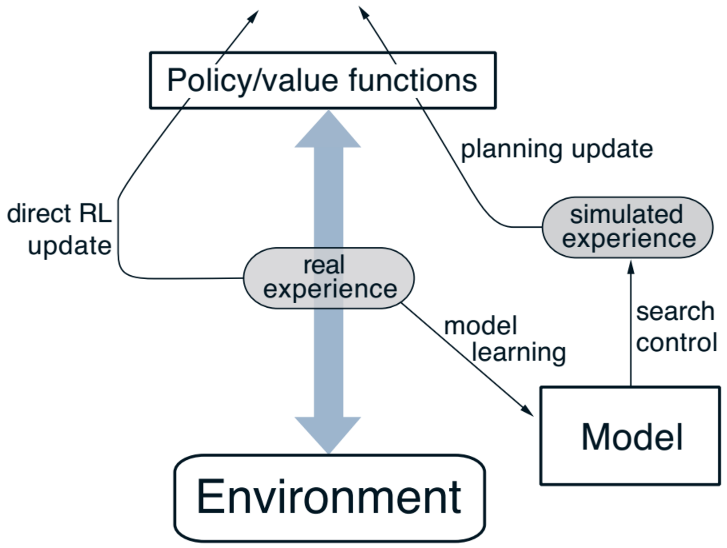


Figure 8.1: The general Dyna Architecture. Real experience, passing back and forth between the environment and the policy, affects policy and value functions in much the same way as does simulated experience generated by the model of the environment.

Figure 6: The Dyna-Q architecture.

the latter, which is sometimes called ***indirect reinforcement learning*** (i.e. learn via simulated experience), that is involved in *planning*.

Both direct and indirect methods have **advantages** and **disadvantages**.

- Indirect methods often make fuller use of a **limited** amount of experience and thus achieve a better policy with **fewer environmental interactions**.
- Direct methods are much *simpler* and are not affected by **biases** in the design of the model

Figure 6 shows the architecture in ***Dyna-Q***, an architecture integrating the major functions needed in an *online planning agent*. It has the following components:

- **planning**: use random-sample one-step tabular Q-planning method in Figure 4.
- **direct RL**: one-step tabular Q-learning discussed in last chapter. Typically, as in Dyna-Q, the *same* reinforcement learning method is used both for learning from real experience and for planning from simulated experience. The reinforcement learning method is thus the "*final* ***common path***" for both learning and planning.
- **model-learning**: ***table-based*** and assumes the environment is ***deterministic***, i.e. given state-action pair, the output of model is **fixed** reward and next-state pair. It can be coded as *dictionary of dictionaries*. Under this assumption, model learning is simply saving the next state S_{t+1} and reward R_{t+1} for each experienced state S_t and action A_t.
- ***search control***: the process that selects the **starting states** and **actions** for the simulated experiences generated by the model.

Figure 7 describes the algorithm. In Dyna-Q, the acting, model-learning, and direct RL processes require little computation, and we assume they consume just a fraction of the time. The remaining

```
Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
    (a) $S \leftarrow$ current (nonterminal) state
    (b) $A \leftarrow \varepsilon$-greedy$(S, Q)$                                                    Q-learning
    (c) Take action $A$; observe resultant reward, $R$, and state, $S'$
    (d) $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
    (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)  ← Model Update
    (f) Loop repeat $n$ times:
            $S \leftarrow$ random previously observed state
            $A \leftarrow$ random action previously taken in $S$                        Planning Step
            $R, S' \leftarrow Model(S,A)$
            $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
```

Figure 7: The Tabular Dyna-Q algorithm.

time in each step can be devoted to the *planning* process, which is inherently **computation-intensive**.

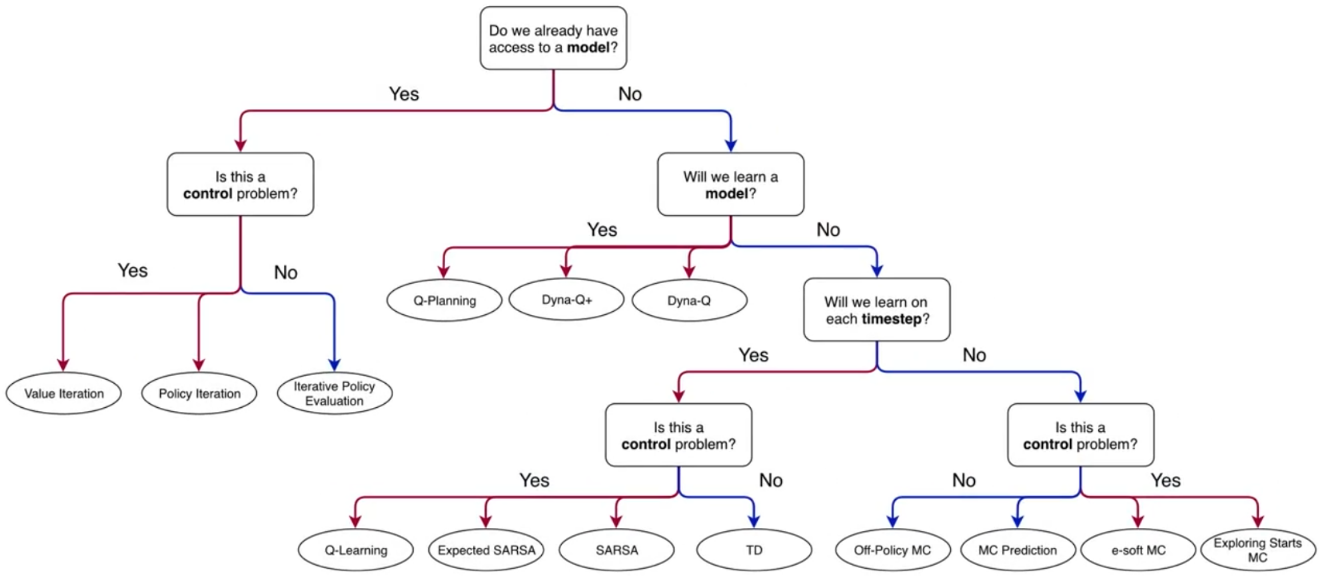
In Dyna-Q, learning and planning are accomplished by exactly the same algorithm, operating on real experience for learning and on simulated experience for planning. Because planning proceeds incrementally, it is trivial to intermix planning and acting. Both proceed as fast as they can. The agent is always **reactive** and always **deliberative**, responding *instantly* to the latest sensory information and yet always planning in the background. Also ongoing in the background is the model-learning process. As new information is gained, the model is updated to better **match reality**. As the model changes, the ongoing planning process will *gradually* compute a different way of behaving to match the new model.

3.1 When the Model Is Wrong

Models may be **incorrect** because the environment is stochastic and only a *limited* number of samples have been observed, or because the model was learned using function approximation that has **generalized imperfectly**, or simply because the **environment has changed** and its new behavior has not yet been observed. When the model is incorrect, the planning process is likely to compute a *suboptimal* policy.

The general problem here is another version of the conflict between **exploration** and **exploitation**. In a planning context, exploration means trying actions that ***improve the model***, whereas exploitation means behaving in the *optimal* way **given the current model**. We want the agent to explore to find changes in the environment, but not so much that performance is greatly degraded. As in the earlier exploration/exploitation conflict, there probably is no solution that is both perfect and practical, but simple heuristics are often effective.

For example, the **Dyna-Q+** is an improved version of Dyna-Q with heuristic search. This agent keeps track for each stateaction pair of how many time steps have elapsed since the pair was last tried in a real interaction with the environment, which is denoted as $\boldsymbol{\tau(s,a)}$. The more time that has elapsed, the greater (we might presume) the chance that the dynamics of this pair has changed and that the model of it is incorrect. During the planning phase, the model uses a special "bonus reward" $= \underline{r + \sqrt{\tau(s,a)}}$ is given on simulated experiences involving the long-untried actions. This



Do we already have access to a **model**?
Yes
No
Is this a **control** problem?
Will we learn a **model**?
Yes
No
Yes
No
Q-Planning
Dyna-Q+
Dyna-Q
Will we learn on each **timestep**?
Value Iteration
Policy Iteration
Iterative Policy Evaluation
Yes
No
Is this a **control** problem?
Is this a **control** problem?
Yes
No
No
Yes
Q-Learning
Expected SARSA
SARSA
TD
Off-Policy MC
MC Prediction
e-soft MC
Exploring Starts MC

Figure 8: Decision-tree to find suitable tabluar methods for reinforcement learning

encourages the agent to keep testing all accessible state transitions and even to find long sequences of actions in order to carry out such tests.

The Dyna-Q+ agent was changed in two other ways as well. First, actions that <u>had never been tried before</u> from a state were allowed to be considered in the planning step of the Tabular Dyna-Q algorithm in the box above. Second, the initial model for such actions was that they would <u>lead back to the same state</u> with a <u>reward of zero</u>.

3.2 Prioritized Sweeping

Sometimes, planning can be much more efficient if simulated transitions and updates are focused on particular stateaction pairs. This is called **Prioritized Sweeping**. The priority is measured according to the size of change for the value estimates when the model updates.

Suppose that the agent discovers a change in the environment and changes its estimated value of one state, either up or down. Typically, this will imply that the values of many other states should also be changed, but the only useful one-step updates are those of actions that lead directly into the one state whose value has been changed. If the values of these actions are updated, then the values of the **predecessor** states may change in turn. If so, then actions leading into them need to be updated, and then their predecessor states may have changed. This is called **backward focusing** of planning computation.

4 Summary of all RL methods

Much of this book has been about different kinds of value-function updates, and we have considered a great many varieties. Focusing for the moment on one-step updates, they vary primarily along three binary dimensions.

- update **state-value**s vs. **action-value**s: i.e. $v(s)$ or $q(s,a)$
- estimate the value for the **optimal policy** vs. for an **arbitrary given policy**: i.e. $v_*(s)$ vs. $v_\pi(s)$ or $q_*(s,a)$ vs. $q_\pi(s,a)$
- **expected updates** vs. **sample updates**: the former considers all possible events that might happen; the later considers a single sample of what might happen

Figure 9 compare their backup diagrams. We can see how DP, MC, TD(0), Sarsa, Q-learning fall into different categories. Dyna-Q uses the Q-learning but it can be replaced by other algorithms in different category since the Dyna architecture is generic and can be applied to other RL algorithm.

Expected updates yield a better estimate because they are uncorrupted by **sampling error**, but they also require more computation, and computation is often the limiting resource in planning. To properly assess the relative merits of expected and sample updates for planning we must control for their different **computational requirements**. This is also determined by the ***branching factor***: the number of possible next states with non-zero transition probability. The higher the branching factor, the more computational resources expected updates required compared to sample updates. In a large problem with many state-action pairs, we are often in the latter situation where the sample update is more feasible.

The following decision-tree diagram Figure 8 shows us how to decide which ***Tabular*** *RL method* is suitable for our problem. In Tabular RL, the value functions are stored in a table with each state/state-action pair pointing to a real-time value. This is different from Functional Approximation RL methods, which assume that the value function is of some parametric functions.

According to our knowlege on environment, the RL methods learned before can be categorized as model-based learning and model-free learning

- **model-based learning**: as discussed above including the DP method. The followings are incremental updates in policy evalution stage from **Dynamic Programming** method via Bellman equation or Bellman optimality equation. The *state-value function* (prediction) **expected updates** are formulated as below:

$$\begin{aligned}
v_\pi(s) &\leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)\left[r+\gamma v_\pi(s')\right] \\
&\leftarrow \mathbb{E}_\pi\left[R_{t+1}+\gamma\, v_\pi(S_{t+1})|S_t=s\right] \\
v_*(s) &\leftarrow \max_{a\in\mathcal{A}(s)} \sum_{s'} \sum_r p(s',r|s,a)\left[r+\gamma v_*(s')\right] \\
&\leftarrow \max_{a\in\mathcal{A}(s)} \mathbb{E}\left[R_{t+1}+\gamma v_*(S_{t+1})|S_t=s, A_t=a\right].
\end{aligned}$$

Same for *action-value function* (control), the *expected updates* are as below:

$$\begin{aligned}
q_\pi(s,a) &\leftarrow \sum_{s',r} p(s',r|s,a)\left[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a')\right] \\
&\leftarrow \mathbb{E}\left[R_{t+1} + \gamma q_\pi(S_{t+1},A_{t+1}) | S_t = s, A_t = a\right] \\
q_*(s,a) &\leftarrow \sum_{s'}\sum_{r} p(s',r|s,a)\left[r + \gamma \max_{a' \in \mathcal{A}(s)} q_*(s',a')\right] \\
&\leftarrow \mathbb{E}\left[R_{t+1} + \gamma \max_{a' \in \mathcal{A}(s)} q_*(S_{t+1},a') \middle| S_t = s, A_t = a\right].
\end{aligned}$$

- **model-free learning**, i.e. sample-based reinforcement learning. We have learned the **Monte Carlo methods** and the Temporal Difference learning methods. The incremental **sample updates** for Monte Carlo methods in policy evalution stage are formulated as below:

  $$\begin{aligned}
  V(S_t) &\leftarrow V(S_t) + \alpha_t \left(G_t - V(S_t)\right) \\
  Q(S_t,A_t) &\leftarrow Q(S_t,A_t) + \alpha_t \left[G_t - Q(S_t,A_t)\right],
  \end{aligned}$$

  where G_t is the sample return accumulated starting at $t+1$ until the end of episode.

  On the other hands, the incremental **sample updates** update for **Temporal Difference learning** methods are formulated as follows: For state-value function updates (prediction)

  $$\begin{aligned}
  & V(S_t) \leftarrow V(S_t) + \alpha_t \left[\hat{G}_t - V(S_t)\right]. \\
  \textbf{TD(0)} \quad & \hat{G}_t = R_{t+1} + \gamma V(S_{t+1}).
  \end{aligned}$$

  For action-value function (control), the *sample updates* are

  $$\begin{aligned}
  & Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha_t \left[\hat{G}_t - Q(S_t,A_t)\right]. \\
  \textbf{Sarsa} \quad & \hat{G}_t = R_{t+1} + \gamma Q(S_{t+1},A_{t+1}) \\
  \textbf{Q-Learning} \quad & \hat{G}_t = R_{t+1} + \gamma \max_{a' \in \mathcal{A}(S_{t+1})} Q(S_{t+1},a') \\
  \textbf{Expected Sarsa} \quad & \hat{G}_t = R_{t+1} + \gamma \sum_{a'} \pi(a'|S_{t+1}) Q(S_{t+1},a').
  \end{aligned}$$

  Note that instead of using the sample returns, TD methods use boostrapping to approximate the return based on reward and old estimate of successor states.

Figure 9 shows the back-up update diagrams for all of these methods.

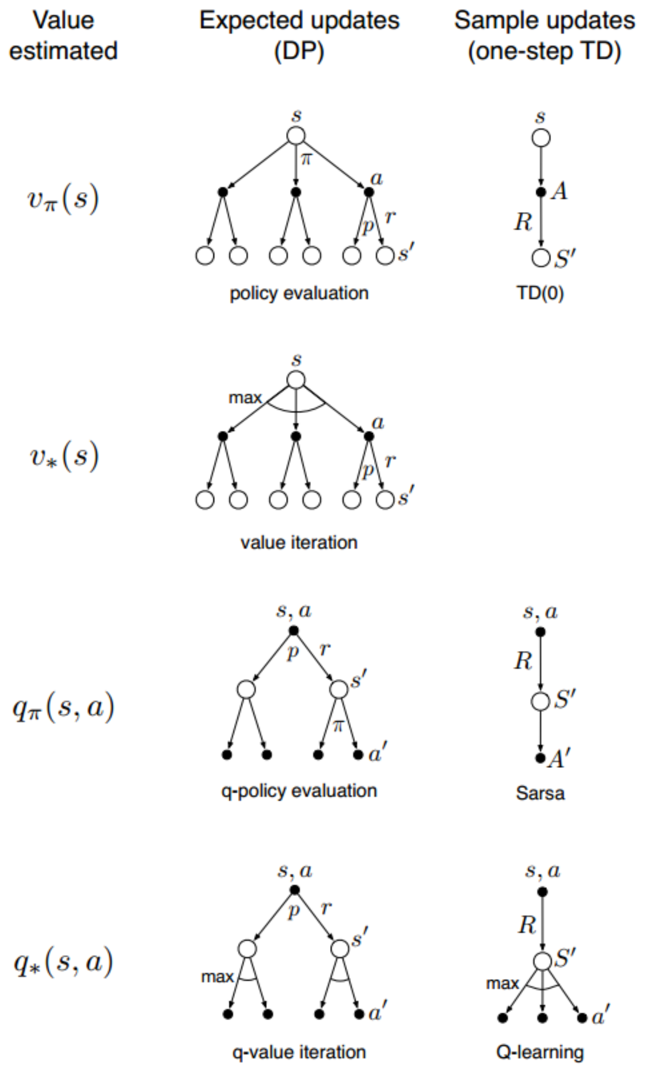


Figure 9: A comparison of all RL methods discussed by far.

References

Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques.* MIT press, 2009.

Jun S Liu and Jun S Liu. *Monte Carlo strategies in scientific computing*, volume 10. Springer, 2001.