UNIVERSITY OF MISKOLC
DEPARTMENT OF MECHANICAL ENGINEERING

Combined Use of Reinforcement Learning and Simulated Annealing: Algorithms and Applications

PÉTER STEFÁN

PH.D. THESIS

Supervisors:
LÁSZLÓ DUDÁS, PH.D.
LÁSZLÓ MONOSTORI, D.SC.

JÓZSEF HATVANY PH.D. PROGRAM IN INFORMATION SCIENCE

Program director:
TIBOR TÓTH, D.SC.

Miskolc-Budapest, 2003

Table of contents

List of Acronyms

ACS	Ant Colony System
AI	Artificial Intelligence
AS	Autonomous System
BA	Boltzmann Annealing
BGP	Border Gateway Protocol
BOE	Bellman Optimality Equations
CIM	Computer Integrated Manufacturing
CPU	Central Processing Unit
DM	Distributed Manufacturing
DNC	Distributed Numerical Control
DP	Dynamic Programming
EGRP	Exterior Gateway Routing Protocol
FMC	Flexible Manufacturing Cell
FMS	Flexible Manufacturing System
FWA	Floyd-Warshall Algorithm
GA	Genetic Algorithm
ICMP	Internet Control Message Protocol
IGRP	Interior Gateway Routing Protocol
IMS	Intelligent Manufacturing System
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
MDP	Markov Decision Process
ML	Machine Learning
MS	Manufacturing System
NIC	Network Interconnection Card
NP	Non-Polynomial (time algorithm)
OS	Operating System
OSPF	Open Shortest-Path First
RIP	Routing Information Protocol
RIS	Real Information System
RL	Reinforcement Learning
RP	Routing Protocol
RPS	Real Production System
RRR	Restricted Rank-based Randomized (policy)
SA	Simulated Annealing

SMQ	“Send me Q”
TCP	Transmission Control Protocol
TD	Temporal Difference (learning)
TH	Time Horizon
TSP	Traveling Salesman’s Problem
TTL	Time to Live (field)
TU	Time Units
UDP	User Datagram Protocol
VE	Virtual Enterprise
VIS	Virtual Information System
VMS	Virtual Manufacturing System
VPS	Virtual Production System

List of Symbols

$[a,b],[c,d]$	real intervals
$\nabla_{\mathbf{w}}$	partial derivatives of a quantity according to the weight vector elements
A_i, B_i, C_i	machining times on machines *A*, *B*, and *C* respectively
a_{i_j}	*j*th action in the *i*th state of the agent
a	learning rate for "*mean value type*" action-state values
$\mathbf{b}$	vector of boundary conditions in flow-shop scheduling
b	learning rate for "standard deviation type" action-state values
b	"inverse temperature" (Chapter 3)
$\mathbf{C}$	set of constraints (Chapter 5)
C	number of precedence constraints
c, c_i	constant parameters (Chapter 3)
c_i	precedence constraints (Chapter 5)
$\mathbf{c}$	vector of parameters (Chapter 3)
c_{ij}	cost of delivery between nodes *i* and *j* (Chapter 4)
g	discount factor
$\mathbf{D}$	compound matrix
d_i	distance label on node *i*
d_{ij}	distance label on node *i* (matrix representation)
$\dim(\mathbf{a})$	dimension of vector $\mathbf{a}$
d	temporal difference factor
$E_p\{x\}$	expected value of x following policy p
E_i	"potential energy"
$e_t(s)$	eligibility value of state s
e	the natural number $\approx 2.71...$
$\mathbf{e}_i$	base vector in dimension *i* (Chapter 5)
e	small positive error
$f(x)$	transfer function
$f'(x), f''(x)$	reward function
$g(x)$	reward function
$\mathbf{g}$	auxiliary machining-delay vector (Chapter 5)
$h'(x), h''(x)$	reward function

I_j	Palmer index for job j
h_{ij}	visibility of a trail from node i to node j
j_j	the jth job under processing
k	the number of actions sharing the same, maximal action-state value
k	minimum of the difference between each action-state value pair
$L(A)$	link state description of node A
l	decay rate for eligibility traces
$\mathbf{M}$	matrix of machining times
$m(i)$	number of actions in the ith state (Chapter 2)
n	number of states
n	small positive redundancy variable
o_1, o_2	abstract machining times
$P_{ss'}^a$	probability of getting into state s' from state s via action a
p_i	probability of selecting action a_i
$\hat{p}$	scaled probability value
$p(x)$	continuous probability distribution function
$\mathbf{p}, \mathbf{r}$	permutation vectors (Chapter 5)
$pred_i$	predecessor node
$pred_{ij}$	predecessor node (matrix representation)
$p(s_i, a_{i_j})$	policy of the agent
p^*	optimal policy
p'	non-optimal policy
Q, Q_i	action-state values
$\mathbf{Q}$	matrix of action-state values
Q^*	optimal action-state value
$\hat{Q}$	scaled action-state value
$Q_{\max}$	the largest action-state value
$Q_{\min}$	the smallest action-state value
$Q(x)$	continuous function of the action-state value
$\mathbf{q}$	vector of action-state values
R_T	total reinforcement on time horizon T
$R_{ss'}^a$	reward of getting into state s' from state s via action a
r_t	reinforcement in time step t
s_i	the ith state of an agent
x	integration auxiliary variable

s	standard deviation
T	time horizon (Chapter 2)
T	temperature (Chapter 3)
T	total machining time (Chapter 5)
$\hat{T}$	scaled temperature
$T_{\max}, T_i^{\max}$	the largest temperature
$T_{\min}, T_i^{\min}$	the smallest temperature
$T(t)$	time-temperature function (Chapter 3)
$T(A)$	routing table of node A (Chapter 4)
t	time step
t_{ij}	machining time of job j on machine i (Chapter 5)
t_{start}	starting time of the annealing process
t_{end}	ending time of the annealing process
$\mathbf{t}$	vector of temperature values
$\mathbf{t}^{\max}$	vector of maximal temperature values
$\mathbf{t}^{\min}$	vector of minimal temperature values
t_{ij}	strength of a trail from node i to node j
Θ	fix-point operator applied to action-state values
V, V_i	state values
V^*	optimal state values
$\Delta\mathbf{w}$	vector of weight components' changes
X_i, Y_i	off-machining times on machines B and C

Acknowledgements

The dissertation summarizes the results of my four years research in the field of machine learning. During this time I learnt many-many new concepts which governed my work, and formed the way how I look to the world and how I approach to problems and to the solutions. Here I would like to express my gratitude to many-many people.

First of all I would like to render my acknowledgements to my supervisors *László Dudás* and *László Monostori* who introduced the field of artificial intelligence and machine learning to me, and who gave me continuous scientific support during the whole of my research. They taught me how to be able to see the same thing from the engineer's as well as from the scientist's point of view. Many thanks to both of them!

I'm also grateful to *András Márkus*, *József Váncza* for their inspiration, and for the interesting discussions conducted not only in the field of AI, but other interesting and important topics as well. Special thanks to *Anikó Ekárt* who helped me to formulate my mathematical results.

I'm indebted to my former and present colleagues *Elizabeta Zudor*, *Botond Kádár* and *Zsolt János Viharos*, *Gábor Ivánszky* who could always give me new ideas whenever I got into a deadlock. I would also like to thank *Tibor Tóth*, *Ferenc Erdélyi* and *Dénes Vadász* for revealing me the broadness of the information science. I'm also indebted to the community of the *Department of Information Science at the University of Miskolc*.

Last but not least, I'm especially grateful to my family, *Zsuzsa* and my parents, *Mária Stefán* and *György Stefán* for their patience in the "hard days" and for the moral support.

Chapter 1

The Scope of the Research

In this dissertation we focus on the question: How different methods, especially combined intelligent methods can be used in solving practical problems, to which exact mathematical solutions can awkwardly be tailored, or do not exist at all. The term intelligence has many interpretations in the literature. Most of the definitions treat intelligence as some model of human being's information processing capability, but from different aspect. Throughout the dissertation we use the term "intelligent" to describe all artificial or natural creatures that are able to make decisions based on their own knowledge. The decision process from this respect is not a pure mechanical task, but a consideration of different choices.
In Figure 1.1 three areas within the artificial intelligence science to which this dissertation joins are shown.

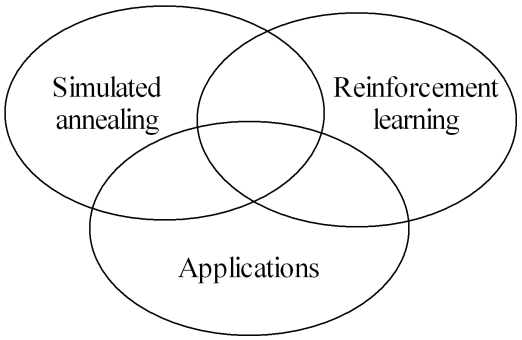


Figure 1.1
The scope of the dissertation

Reinforcement learning algorithms model the decision maker as an automata-like goal-driven agent with the aim of reaching some goal states in the problem representation state space. For example an automated robot vehicle in a room can have the goal to reach a certain position; a resource broker agent can have the goal to reach a certain resource configuration, etc. The agent accomplishes different choices among its numerous possibilities, but each choice can make different sense in the environment. Each decision has some immediate or future effects which are expressed in the form of numeric honor or dishonor, say, of a reward value. The agent utilizes the feedback in order to try to recognize which actions are appreciated by the environment and which

actions are not. The agent then tries to govern its decision sequence into the direction that maximizes the "environment's satisfaction".

The concept of simulated annealing is based on the analogy of how liquids freeze or recrystalize. In an annealing process an initially high temperature and disordered melt is slowly cooled down and reaches thermal equilibrium. As the cooling proceeds, the system becomes more ordered and approaches a so-called "frozen" state. The process can be thought as an adiabatic approach to the lowest energy state. Simulated annealing is analogous to the real annealing process: the state of the thermal system corresponds to the current solution of the model analogy; the thermal equilibrium corresponds to the global optimum; the annealing schedule corresponds to the objective function. The greatest advantage of the method is that it avoids running into local minima.

Chapter 2 gives an overview on the reinforcement learning concept, and surveys several reinforcement learning techniques ranging from dynamic programming to temporal difference learning. All methods have something in common: all algorithms converge to a so-called optimal decision sequence, or optimal policy which guarantees the maximization of the total reward given by the environment on the long run. The chapter concludes with the question: It is good, that all reinforcement learning methods are convergent, but what about the speed of the convergence? Is there any decision policy that can be used to boost the convergence? This is the point where combined simulated annealing and reinforcement learning methods come to the focus.

Chapter 3 reveals a new simulated annealing technique that is tailored to the goal driven agents. The method is called Boltzmann annealing schedule, and utilizes the control properties of the Boltzmann distribution computed over the action selection preference values. As in the case of general simulated annealing algorithms the annealing temperature value is characteristic to the action selection probability distribution, in the sense that it influences the way the agent makes its decision. We show in a theorem that there are two extremes of the distribution that are useful in practice: 1. when the agent treats all choices equally probable and let itself try actions that have never been selected before, in the hope of larger reward afterwards, or 2. when the agent always selects an action that it thinks to be the best. The former is referred to as exploration, while the latter is called exploitation.

Though the two extremes are reached only in the limit, it is still an interesting question if there are finite and nonzero temperature values for which the two extremes can be approached with a sufficiently small error. We show that if the preference values are bounded, such finite temperature bounds do exist, defining a temperature parameter domain in which any decision control has effect. Outside this domain there is no improvement, since exploration or exploitation is already performed accurately enough.

Temperature bounds have several benefits. Firstly they can be computed on-line. Secondly they can be used to define an exploration-exploitation balancing annealing schedule if the length of the intended exploration time is given. Thus the agent approaches to the original decision problem in an inverted sense: "*I have certain*

amount of time to look for solutions, and after some elapsed time, I would like to have a feasible solution. It does not matter if it is not the optimal one (it is good if it is the optimal solution), but it should be one that is practically usable. I would like to avoid the really bad cases." This is referred to quasi-optimality, when not the best decision sequence is sought, but the one which is good enough in practice.

We describe an annealing model which guarantees sufficient exploration at the beginning of the decision making process, and the range of decision gradually shrinks around the best decision sequence found as the agent's experience increases. A general annealing framework is shown first then special annealing functions are derived from the general model, which are simplistic enough for practical purposes. At the end of Chapter 3, computation validation of the temperature bounds as well as Boltzmann distribution based annealing schedules are shown.

Chapter 4 and Chapter 5 detail two application examples. Chapter 4 concentrates on Internet Protocol packet routing problems, and introduces a Boltzmann annealing based Q-routing algorithm. It is shown that our new routing method prevents the router agents from the classical problems of dynamic routing protocols, and eliminates two problems of the basic Q-routing algorithm: loop detection and path recovery. Loop detection in Q-routing refers to detecting and punishing the looping parts of a data delivery path, while retaining and rewarding those parts that do not contribute to the loop. Path recovery is the method of diverting the traffic back to the regular (and optimal) data delivery path from a backup route when the failure of the regular path is over. Ordinary Q-routing algorithms are not capable of doing this. A distributed simulator is also developed in Java which emulates the setup and the operation of a real network.

Chapter 5 illustrates Boltzmann annealing based flow-shop scheduling method within the framework of virtual manufacturing concept. Given a real manufacturing system, a virtual service is worked out which maps the real system to computational objects and utilizes the property that the speed of evaluation of real processes in the virtual model can be thousand times quicker than executing it in the real environment. An on-line virtual scheduler service is proposed which works in the virtual manufacturing space and which uses reinforcement learning principles in implementing domain-specific heuristics to solve the flow-shop scheduling problem. However there is a large difference between routing and scheduling: The latter problem is non-Markovian. We illustrate that the new on-line scheduling concept provides better results than heuristic approaches and on-line re-scheduling capability in spite of the non-Markovian environment.

Chapter 6 summarizes the main contributions to the interdisciplinary field of reinforcement learning, simulated annealing, internet routing and on-line flow-shop scheduling presented in this document.

Chapter 2

Fundamentals of Reinforcement Learning

In this chapter a general "learning from zero-prior-experience" method, called reinforcement learning (RL) will be surveyed. RL has its roots in automata theory as well as in dynamic programming where large computational problems are split into small sub-problems by setting decision points and using evaluation functions to build globally optimal solutions on the basis of sub-optimal solutions.

Reinforcement learning adds iterative environment sampling capability to the original dynamic programming model, thus making it capable of adapting to changes in the external environment.

2.1 Machine learning[1]

The definition of machine learning used to be ambitious, aimed at giving an artificial replacement of human beings' information processing and knowledge synthesizing method. Recently the definition has focused around the goal to produce algorithms that are limited models of human beings' learning capability but are practically usable in many real-life applications. According to *Kohavi* and *Provost "machine learning is the field of scientific study that concentrates on induction algorithms and on other algorithms that can be said to learn"* [R18].

The common point of all artificial learning systems is that all use some internal data structure consisting of different parameters that may affect the behavior of the whole system. The artificial learner modifies its internal structures and/or parameters in order to give improved performance, i.e., improved expected system output for the same input as time goes on [R26]. The natural learning procedure can also be studied from this perspective, although it is more complex than its artificial counterpart.

Kaelbling classifies machine learning algorithms in [R16] using different criteria. One criterion that examines the role of the system's environment, distinguishes three groups of algorithms: supervised learning, unsupervised learning and reinforcement learning.

Supervised learning algorithms have the strongest environment dependency since these algorithms use an external (and often stationary) reference, called teacher,

[1] The terms "artificial learning" and "machine learning" are used as the synonyms of each other throughout the document.

that can tell the expected input-output mapping all the time, and can directly instruct the learning system which parameter and how to modify. This type of learning is also regarded as instructive learning, since the environment directly intervenes into the internal structure of the learning system [R16]. Commonly used neural network learning algorithms, such as back-propagation, or classification algorithms fall into the category of supervised learning.

Unlike supervised learning methods, unsupervised learning methods do not require any external resource to tell the right output all the time, but have some internal metric that is applied to build internal multi-dimensional mappings of the parameter space [R19]. The learning system then either generalizes or specializes on different sections of these maps in order to introduce new classification rules, or remove old rules as learning advances. Algorithms of self-organizing maps and *Kohonen*-networks are typical examples for unsupervised learning.

The third group of learning algorithms is called reinforcement learning (RL) algorithms that have both supervised and unsupervised features [R16]. RL algorithms use the autonomous learning system model, familiar from unsupervised learning, which means that the system cannot be instructed by any external reference. However, it keeps interactive relationship with the environment and tries to get some information in order to build up its knowledge base. The feedback is either utilized to update the knowledge or simply ignored depending on the learner's own decision. In fact, the responsibility of gaining feedback is also on the learner, so this type of learning is also called explorative learning [R16]. RL methods are capable of learning from zero prior knowledge provided that the number of learning examples is sufficiently large.

2.2 The reinforcement learning problem

Sutton and *Barto* in [R46] defines the reinforcement learning problem as the problem of a goal driven agent which is in contact with its external environment and which performs interactive actions, gets feedback and gathers experience in order to reach a specified goal state. RL algorithms are not considered to be a separate group of methods but all kinds of algorithms that give possible solutions to the defined problem.

The agent can be described by discrete states that are the only parameters that uniquely define the agent's behavior. States are either simple data elements or compound data structures. State information is furthermore discrete in time, which means that in each time step the agent can be described by a single state. State representation can either be the result of internal implicit rules, the behavior of the agent, or the result of the mapping of external perceptions [R46].

When the goal state has not been reached yet, each state is considered as a decision point, where something must be done in order to get into a new state. Thus, in each time step, the agent selects an action out of a set of actions, which influences the

environment and makes the agent get into a new state[2]. The mechanism is shown in Figure 2.1.

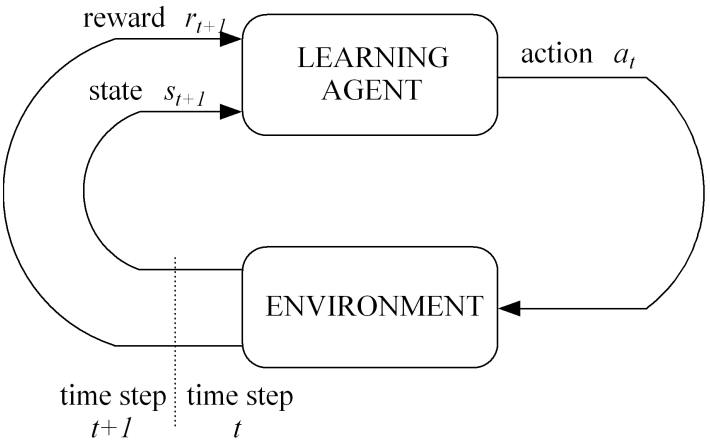


Figure 2.1
Schematic representation of the reinforcement learning model

The representation of actions can be quite different. In the simplest case actions are stored in look-up tables, but they can be the result of a complex search algorithm or more complex functions as well [R16]. The agent's *"doing nothing"* between time steps, is also treated as an "*identity*" action selection.

State changes may be stochastic, which means that selecting the same action from the same state yields different resulting state. It is expedient to describe the system's dynamics using state-action-state probability distribution [R16][R46].

Apart from state change, a reinforcement signal that indicates the environment's evaluation is fed back to the agent in every time step. A large positive scalar, for example, may show that the environment appreciated the selected action honoring it with a large reward, and a large negative number may be provided if the action selected is improper in that situation and should be avoided in the future. The agent evaluates the scalar reward/punishment signal to adjust its knowledge base, which makes possible to select some better action next time. The goal of the agent is thus, to maximize the sum of rewards gained in a learning period, called an episode, on the long run. Rewards are always considered to be external to the agent [R46].

The environment is not supposed to be stationary, but is supposed to be consistent in rewards, since confusing rewards/punishments in a short period may spoil the agent's state-action assignments[3]. However, rewards are permitted to change on the long-run. It is also an interesting question where the boundary between the agent and its environment is. *Sutton* and *Barto* gives the answer that *"everything that falls outside of*

[2] In fact state change may occur not only due to the influence of the environment, but to the agent's internal mechanisms as well.
[3] If the agent has some inertia, the fluctuating variances can be handled.

the agent's control is treated as part of the environment". This means that natural physical boundaries (e.g. learning robot vehicle in a room) cannot always coincide with the agent-environment boundaries.

Throughout learning, the agent's task is to find the optimal decision sequence, which guarantees the maximal sum of rewards in the long run. The way how the agent makes a decision is called policy [R46] which is defined as a state-action mapping. Let $S = \{s_1, s_2, ..., s_n\}$ define the set of states (where n indicates the number of states) and $A(s_i) = \{a_{i_1}, a_{i_2}, ..., a_{i_{m(i)}}\}$ the set of actions available in state s_i (where $m(i)$ denotes the number of available actions). Equation 2.1 defines the stochastic policy over the set of actions, i.e., assigns a probability value to each state-action pair.

$$p(s_i, a_{i_j}) = p_{ij}, \; i = 1,2,..., n, \; j = 1,2,..., m \tag{2.1}$$

Policies represent part of the agent's knowledge, since it is crucial information which action may lead to better results. In order to let the agent discriminate between different states and different actions, preference values (or simply values) are used. There are two kinds of values: state values, and action-state values. State values show how valuable for the agent it is to be in a certain state. The precise definition of a state value is given by *Sutton* and *Barto* in [R16] as "*the sum of expected discounted rewards from the current state to the goal state*". Although it is important to differentiate states since their utility can be quite different, state values do not take forthcoming state changes into account. It is the point where action-state values come into the focus: by definition, action-state values approximate "*the sum of expected discounted rewards from the current state to the goal state selecting a particular action*". Note that policies which are probability distributions over the set of actions and action-state values are different, but are in strong relationship with each other. Traditionally, state values are denoted by V, action-state values are denoted by Q.

2.3 Time-horizon models, Markov property, Bellman optimality equation

Throughout an episode, the agent can receive lots of rewards that, due to the time locality of the decision, should be taken into account differently. The way, how an agent treats temporal rewards is regarded as a time-horizon (TH) model. There are three fundamental TH models: finite horizon, infinite horizon and average reward models.

In the case of the finite horizon model the agent takes a finite time window and works to obtain maximal sum of rewards within this time period. This method is useful when the learning process can be divided into episodes and each episode ends up in a terminal state in finite time steps.

Suppose that the agent looks T steps ahead, let R_T denote the sum of rewards on the given time-horizon from t to $T+t$:

$$R_T = r_t + r_{t+1} + r_{t+2} + ... + r_{t+T} = \sum_{i=0}^{T} r_{t+i} \qquad (2.2)$$

There are two ways the above equation can be used:

1 at time step $t+1$ the agent looks $T-1$ steps ahead, at $t+2$, looks $T-2$ steps ahead, and looks only one step ahead in $t+T-1$, or

2 the agent always takes the optimal action looking $T-1$ steps ahead.

Finite horizon models can only be applied to episodic processes (processes that can be naturally divided into finite decision sequences), since in the case of continuing processes (i.e. non-episodic processes), *T* goes to infinity, and so does R_T.

The infinite horizon model introduces a mathematical trick to keep R_T in a finite range even if $T \to \infty$. The discount method weights rewards situated closer to the decision point with larger values than rewards situated in the farther future. The discount rate determines the present value of the future reward: reward *r* received in time step *k* is equivalent with reward $g^{k-1}r$ received immediately. The discount formula is as follows:

$$R_T = \sum_{i=0}^{T} g^{i-1} r_{t+i}\ ,\ 0 \le g \le 1. \qquad (2.3)$$

If *T* is finite and $g = 1$ then equation 2.3 is identical to equation 2.2. When $g = 0$ then the agent has one-step look-ahead, or follows greedy policy. If $T \to \infty$ and $g < 1$ then the learning task is continuing and the reward is finite.

Average reward models take the average of rewards as shown by equation 2.4. This method hides differences between the policies that are performing well at the beginning and worse afterwards, and the policies that are performing better in the final phase; only the average reward matters.

$$R_T = \frac{1}{T} \sum_{i=0}^{T} r_{t+i} \qquad (2.4)$$

Let $P_{ss'}^{a} = \Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\}$ denote state transition probabilities, that is the probability of getting into state s' provided that the current state is *s* and the chosen action is *a*. If state transition probabilities are dependent only on *s*, *a*, and s', the decision process has the Markov property and is referred to as Markov Decision Process (MDP). A MDP does not maintain any history, i.e. a sequence of states traversed so far or a set of actions that have been selected. Let $R_{ss'}^{a} = E\{r \mid s_t = s, a_t = a, s_{t+1} = s'\}$ denote the expected value of the reward of state transition from *s* to s' when choosing action *a*. $P_{ss'}^{a}$ and $R_{ss'}^{a}$ are referred to as the environment dynamics of a Markovian process and

these are the only necessary input data that describe the process [R46]. However the environment dynamics are rarely known, but are estimated from trial and error experience.

In the previous section the verbal definition of state and action-state values were given. The formal definitions are as follows for the infinite horizon model:

$$V^{p}(s) = E_{p}\{R_{t} \mid s_{t} = s\} = E_{p}\left\{\sum_{i=0}^{T} g^{i-1} r_{t+i} \mid s_{t} = s\right\}, \tag{2.5}$$

and

$$Q^{p}(s,a) = E_{p}\{R_{t} \mid s_{t} = s, a_{t} = a\} = E\left\{\sum_{i=0}^{T} g^{i-1} r_{t+i} \mid s_{t} = s, a_{t} = a\right\}. \tag{2.6}$$

where p denotes the policy followed by the agent.

As it is indicated in equations 2.5 and 2.6, values are dependent on policies. Different policies may result in different values. Also, *"value functions define partial ordering among policies"* [R46]. It is established by *Bellman* in [R4] that if a decision process is a MDP and the environment dynamics exist (not necessarily known), there exists a unique policy denoted by p^{*} called the optimal policy for which $V^{p^{*}}(s) \geq V^{p}(s)$ for all $s \in S$ where p is an arbitrary policy. It is also established that the existence of the optimal policy determines the optimal state and action-state values as:

$$V^{*}(s) = \max_{p} V^{p}(s) \tag{2.7}$$

and

$$Q^{*}(s,a) = \max_{p} Q^{p}(s,a). \tag{2.8}$$

Bellman also shows that the unique optimal values can be obtained by solving the nonlinear Bellman optimality equations (BOE) shown in equations 2.9 and 2.10 [R4][R46].

$$\begin{aligned} V^{*}(s) = \max_{a \in A(S)} Q^{p^{*}}(s,a) = \max_{a} E_{p^{*}}\{R_{T} \mid s_{t} = s, a_{t} = a\} = \\ \max_{a} E_{p^{*}}\left\{\sum_{i=0}^{\infty} g^{i-1} r_{t+i} \mid s_{t} = s, a_{t} = a\right\} = \\ \max_{a} E_{p^{*}}\left\{r_{t+1} + g\sum_{i=0}^{\infty} g^{i-1} r_{t+i+1} \mid s_{t} = s, a_{t} = a\right\} = \end{aligned} \tag{2.9}$$

$$\max_a \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma E_p \left\{ \sum_{i=0}^{\infty} \gamma^{i-1} r_{t+i+1} \mid s_t = s, a_t = a \right\} \right] = \max_a \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma V^*(s') \right] \tag{2.9}$$

$$Q^*(s,a) = E\left\{ r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a \right\} = \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma \max_{a'} Q^*(s', a') \right] \tag{2.10}$$

Both equations establish a relationship between the value of the current state and the values of successor states. *Szepesvári* and *Lorincz* shows that the unique solution of the BOE can be found by using fix-point theorem [R49][R50].

The goal of any reinforcement learning algorithm is to find or approximate the optimal values and the optimal policy by either solving equations 2.9 and 2.10 if the environment dynamics are known or by estimating them using on-line estimation.

2.4 Solutions to the RL problem

In this section several methods that solve the RL problem will be shown. Solutions can be of two types: the ones that require exact knowledge of the environment dynamics ($P_{ss'}^a, R_{ss'}^a, \forall s \in S', \forall a \in A(s_i)$), and build the exact model determining the optimal policy, and the others that estimate the dynamics and also the value functions trough trial and error probes. The first type of methods is also referred to as dynamic programming methods, while the second type as temporal difference algorithms.

2.4.1 Dynamic programming

Dynamic programming (DP) refers to a collection of algorithms that are used to compute values, as well as optimal policies if the perfect model of the environment is given. DP algorithms are fairly important since they give theoretical background to all other RL algorithms. As a general law, all DP methods aim at breaking Bellman optimality equation (formulas 2.9 and 2.10) into successive iterative steps; the iterated value approaches to the unique solution of the Bellman optimality equation.

The very first step to determine the optimal policy is to compute values to an arbitrary policy. This method is known as policy evaluation. The key idea of policy evaluation is to let the policy be fixed. Then equation 2.9 is re-written as:

$$V_{k+1}(s) = \sum_a p(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + gV_k(s') \right]. \qquad (2.11)$$

Note that the policy is not the optimal one, thus the maximization operator cannot be used. Instead, summation over the state values as well as the known rewards weighed by the action selection probabilities of each available successor state choosing a particular action is applied. The summation is done for all possible actions. Also note that the value function is not the optimal one either. The policy evaluation algorithm is shown in Figure 2.2.

```
input: π policy to be evaluated, θ small error value
input: n number of states, m number of actions
output: V(s)
initialize V(s) for all s∈S
function policy_evaluation(π,θ)
begin
  while Δ>θ do
    Δ:=0;
    for i:=1 to n do
      v:=V(s[i]);
      for j:=1 to n do
        for k=1 to m do
          V(s[i]):=V(s[i])+π(s[i],a[j])P(s[i],a[j],s[k])
                                  (r(s[i],a[j],s[k])+γV(s[k]));
        end
      end
    end
    Δ:=max(Δ,|v-V(s[i])|);
  end
end
```

Figure 2.2
The policy evaluation algorithm

In the algorithm, environment dynamics ($P^a_{ss'}, R^a_{ss'}$) are denoted slightly differently and all sets are treated as arrays[4]. If the number of iterations goes to infinity, iterative values of V converge to V^p. This procedure is also referred to as full-backup method since it takes all possible successor states in a given state into account rather then a single sample state. The computational complexity of iterative policy evaluation algorithm is of order $O(n^2mz)$ where n is the number of states, m is the number of actions, and z is the number of iteration steps.

The purpose of policy evaluation is to determine better policies. Suppose that the value V^p of policy p is determined by policy evaluation. Suppose further that in the

[4] There is no sequence relationship between elements of sets, but there is between the elements of arrays, at least from the storage point of view. From theoretical aspect this difference does not make sense, and arrays are used as computational representation of any sets.

current state s there is an action a' which is different than the one that would be chosen by policy p. Note that action a' may yield better reward, than action a. Select action a', for which $p(s) \neq a' = p'(s)$, where policy $p'(s)$ sets different action selection rules than $p(s)$ at the first step, but they are identical for the remaining steps. It comes from equation 2.10 that for the modified policy p'

$$Q_{k+1}^{p'}(s, p'(s)) = \sum_{s'} P_{ss'}^{p'(s)} \left[R_{ss'}^{p'(s)} + gV^{p}(s') \right]. \tag{2.12}$$

It is proven in [R46] that if $Q^{p'}(s, p'(s)) \geq V^{p}(s)$, then $V^{p'}(s) \geq V^{p}(s)$ is also true. The key question is how to modify the original policy to give improved values. It is also shown in [R46] that defining a one-step greedy policy, that takes the best action immediately, meets the conditions of equation 2.12 thus:

$$p'(s) = \arg\max_{a} \sum_{s'} P_{ss'}^{p'(s)} \left[R_{ss'}^{p'(s)} + gV^{p}(s') \right]. \tag{2.13}$$

The argmax operator means selecting the action that maximizes the sum. The method of refining the policy from the values of another policy is called policy improvement.

Given policy evaluation and policy improvement, it is reasonable to join these algorithms together: values can be computed by policies, and policies can be improved by new values. It is shown by *Sutton* and *Barto* in [R46] that $p^{0} \xrightarrow{E} V^{p^{0}} \xrightarrow{I} p^{1} \xrightarrow{E} V^{p^{1}} \xrightarrow{I} p^{2} \xrightarrow{E} \ldots \xrightarrow{I} p^{*} \xrightarrow{E} V^{*}$, where $\xrightarrow{E}$ denotes policy evaluation and $\xrightarrow{I}$ denotes policy improvement. The whole process that outputs the optimal policy as well as the optimal values is called policy iteration.

In most of the practical cases policy iteration converges to the optimal value too slowly since the set of states has to be traversed many times and policy iteration consumes considerable amount of time. An algorithm that combines policy improvement with one-step policy evaluation is called value iteration. Value iteration uses the best action's value (see equation 2.14) instead of doing a full backup on all possible successor states. The simplification speeds up the convergence:

$$V_{k+1}(s) = \max_{a} \sum_{s'} P_{ss'}^{a} \left[R_{ss'}^{a} + gV_{k}(s') \right]. \tag{2.14}$$

Similarly for action-state values:

$$Q_{k+1}(s, a) = \max_{a} \sum_{s'} P_{ss'}^{a} \left[R_{ss'}^{a} + gQ_{k}(s', a') \right]. \tag{2.15}$$

The value iteration algorithm is shown in Figure 2.3.

DP methods can be applied to RL problems when the number of states is large; for smaller states-spaces linear programming as well as direct search algorithms can also be used to solve equations derived from the Bellman optimality equations.

```
input: θ small error value
input: n number of states, m(i) number of actions in state i
output: V(s) the optimal policy
initialize V(s) for all s∈S
function value_iteration(θ)
begin
  while Δ>θ do
    Δ:=0;
    for i:=1 to n do
      v:=V(s[i]);
      for j:=1 to m(i) do
        Vtemp:=0;
        for k=1 to m(i) do
          Vtemp:=Vtemp+P(s[i],a[j],s[k])(r(s[i],a[j],s[k])+γV(s[k]);
        end
        if Vtemp>=Vmax then
          V[s[i]):=Vtemp;
        end
      end
    end
    Δ:=max(Δ,|v-V(s[i])|);
  end
end
```

Figure 2.3
The value iteration algorithm

2.4.2 Temporal-difference learning

Dynamic programming methods take the assumption that environment dynamics ($P^a_{ss'}, R^a_{ss'}$) are known, which is rarely the case for most practical applications. The environment dynamics can be estimated from experience if the agent can make a large number of iteration samples [R46]. Estimated values as well as $P^a_{ss'}$ and $R^a_{ss'}$ are backed up at each time step with respect to the previous samples, which explains the general name of these type of algorithms as temporal difference (TD) learning methods [R16].

Kaelbling classifies TD methods into model-based and model-free types [R16]. Model-based TD algorithms first build up the environment, then compute state and action-state values by one of the DP methods. This type of learning often breaks into two distinguishable temporal phases: environment estimation and computing optimal policy. On the other hand, model-free methods estimate optimal values and optimal policy without the explicit knowledge of the environment.

The Dyna algorithm of *Sutton* [R47] is one of the most well-known model-based algorithms. Dyna operates in a loop of interaction steps. The "experience" samples in each time step are described by a quintuple of $\langle s, a, r, s', a' \rangle$, the elements denoting state, action, reward, next-state, and action after the next state, respectively. Due to the nature of the experience, the agent knows its next state in each time step, so there is no need to make a full backup on each possible successor state. Using the experience quintuple $E\{P_{ss'}^a\}$ and $E\{R_{ss'}^a\}$ (i.e. the estimated values of the environment dynamics) are updated by simple averaging technique. Equation 2.15 can be rewritten as

$$Q_{k+1}(s,a) = E\{R_{ss'}^a\} + g \sum_{s'} E\{P_{ss'}^a\} \max_{a'} Q_k(s',a') . \qquad (2.16)$$

During the interaction cycles, *n* updates are performed using random exploration strategy, i.e. the agent should visit all possible states and select all possible actions in order to get real convergence.

Although Dyna converges to the optimal action-state values, it is computationally inefficient because of the random exploration. An extension of Dyna called Queue-Dyna introduced by *Moore at al.* in [R28] focuses on only the "*interesting parts*" of the state space: the algorithm maintains a list of predecessor states for each state. In addition, states in the list have priority values. The agent selects actions having nonzero state transition probability and follows not random exploration, but exploration that is based on the accumulated priority values. Priorities are initialized to zero at the beginning and then updated to be proportional to the value improvement since the last improvement of that state.

Queue-Dyna improves Dyna significantly, in terms of the number of computational steps needed to learn the optimal policy.

The most general model-free temporal difference learning method is called TD(λ), or SARSA. A two-layered hierarchical extension is called adaptive heuristic critic algorithm, i.e. an adaptive version of policy iteration. TD(0), later generalized to TD(λ), was introduced by *Sutton* [R48], while SARSA was developed by *Rummery* and *Niranjan* [R41].

Let $\langle s, a, r, s', a' \rangle$ denote the experience in an iteration step (the name SARSA refers to the quintuple). The crucial thing for the learner is whether the new state has made the values better or worse. This quantity is expressed as a temporal difference:

$$d = r + gV(s') - V(s) . \qquad (2.17)$$

If $d > 0$, then action *a* increases the value $V(s)$, otherwise it decreases it. Equation 2.17 can then be used for writing update rule[5] to the state values as well as to the action-state values [R46], such as

$$V_{k+1}(s) = V_k(s) + \beta d\,, \tag{2.18}$$

or

$$Q_{k+1}(s,a) = Q_k(s,a) + \alpha d\,. \tag{2.19}$$

where α and β are the controllable step-size parameters called learning rates and *k* denotes the time step [R44]. Equations 2.18 and 2.19 are equivalent to the backup rule of value iteration (equations 2.14 and 2.15) with the only difference that the sample is drawn from on-line sampling of the real world rather than by preliminary sampling and simulation of a known model [R16]. Any algorithm that uses update rules 2.18 and 2.19 is referred to as TD(0) algorithm and it is guaranteed to converge to the optimal value function, which was established by *Sutton* [R48].

TD(0) algorithms are special cases of a more general temporal difference algorithm class called TD(*n*). While TD(0) algorithms take only the immediate reward and the value of the next state into account, TD(*n*) algorithms use *n*-step look-ahead to determine the value of the current state. Thus all TD(*n*) algorithms define *d* as

$$d = \sum_{i=0}^{n} \gamma^{i} r_{t+i} + \gamma^{n} V(s_{t+n}) - V(s)\,. \tag{2.20}$$

A more general view of temporal difference methods is provided by TD(λ) algorithms that use weighed sums of *k*-step look-ahead rewards, where *k* takes values from 1 to *n*. This means that the rewards expected in the near future or rewards obtained in the near past weigh more significantly than those farther in the future or past, the update rule uses the following weighted average:

$$d = (1-\lambda)\sum_{j=1}^{\infty} \lambda^{j-1}\left[\sum_{i=0}^{n} \gamma^{i-1} r_{t+i} + \gamma^{n} V(s_{t+n}) - V(s)\right]. \tag{2.21}$$

TD(λ) algorithms can be considered as forward view or backward view algorithms. Equation 2.21 combined with 2.18 or 2.19 defines forward view TD(λ) which is also called: the theoretical approach. A much more practical model can be gained by using the backward view: through a memory variable referred to as the eligibility trace. Eligibility traces are concerned with all states visited so far:

[5] The update rule specifies how the agent uses the experience.

$$e_t(s) = \begin{cases} g l e_{t-1}(s), s \neq s_t \\ g l e_{t-1}(s) + 1, s = s_t \end{cases} \tag{2.22}$$

The eligibility of a state is the degree of how frequently it has been visited in the recent past. Whenever a state is reached, the trace concerning that state is amplified, and similarly the trace smoothly becomes less influential if the corresponding state is not visited. The temporal difference of the backward view TD(λ) is defined as:

$$d = \left[r + gV(s') - V(s)\right] e_t(s). \tag{2.23}$$

Update of eligibility traces can either be online, when backups are performed in each time step, or off-line, when a summary backup is carried out at the end of the episode[6]. Any backward view TD(λ) method is easy to implement; in Figure 2.4 the on-line version is shown.

```
input: n number of states, T number of steps
output: V(s) the optimal value function
initialize V(s) for all s∈S
function TDλ(n,T)
begin
  for j:=1 to T do
    choose action a in state s;
    take s' and r;
    δ:=r+γV(s')-V(s);
    e(s):=e(s)+1;
    for i:=1 to n do
      V(s):=V(s)+αδe(s);
      E(s):=γλe(s);
    end
    s:=s';
  end
end
```

Figure 2.4
The on-line, backward view TD(λ) algorithm

It is proved in [R46] that backward view and forward view TD(λ) algorithms implement the same weight-layout, and thus are equivalents of each other.

The actor-critic methods separate policy and value structures, thus introducing an implicit hierarchy in the system. The policy is bound to the actor of the process, and the value to the critic. The actor follows policy p and whatever this policy is, the critic evaluates it and makes changes. The concept is shown in Figure 2.5. Control signal c has similar functionality to the reward signal, but inside the learning agent.

[6] In case of episodic processes.

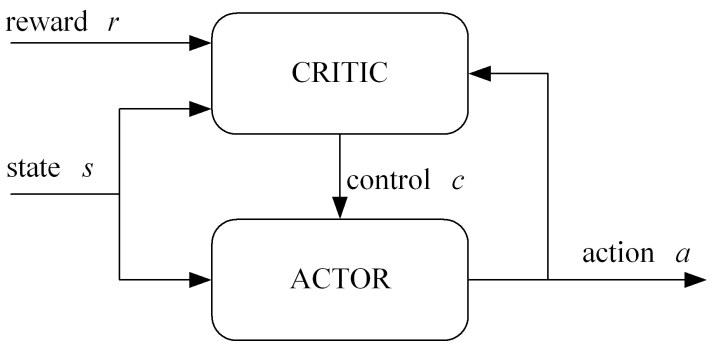


Figure 2.5
The actor-critic agent architecture

2.5 Q-learning

Although Q-learning implements a kind of TD learning, due to its practical relevance it is discussed in a separate section. The algorithm of Q-learning was introduced by *Watkins* and *Dayan* in [R60], where the complete proof of the convergence to optimal action-state values as well as optimal policy is given. The key idea of the proof is to replace the agent's *"real process"* to an equivalent abstract MDP, a *"biased coin flipping"* task.

It is common in TD-learning and in Q-learning that a sample experience sequence is given, i.e. the agent senses current state, selected action, given reward, next state and next action quintuples, and whatever would follow next, the agent considers optimal action choice then. Experience quintuples are denoted in the usual way by $\langle s, a, r, s', a' \rangle$ and are supposed to be random samples of a given probability distribution.

Having selected an action that provides the best action-state value, the optimal state value turns out to be $V^*(s) = \max_a Q^*(s, a)$.

Writing a backup rule to Q-learning can be done similarly to writing a backup rule to the general model (equation 2.15):

$$Q^*(s,a) = R^a_{ss'} + g \sum_{s'} P^a_{ss'} \max_{a'} Q^*(s', a') \qquad (2.24)$$

Note that there is no stochastic action selection, since both the next action and the next state are known. Also note that the policy is implicit, i.e. the best action is as follows: $p^*(s) = \arg\max_a Q^*(s, a)$.

If the update rule is given as

$$Q_{k+1}(s,a) = (1-\alpha_{k+1})Q_k(s,a) + \alpha_{k+1}\left[r_{k+1} + \gamma V_k(s')\right], \qquad (2.25)$$

then it is true that the sequence of Q-values approach Q^*, provided that α_k is a decaying sequence of the step-size parameter α, and the number of trials sufficiently cover all state-action pairs as the number of trials goes to infinity. The α parameter is also referred to as learning rate, because it controls at what rate the learner allows to modify the values.

Theoretical guarantee for convergence can only be given if the Q values are stored in look-up tables (see section 2.6). A possible Q-learning algorithm is shown in Figure 2.6.

```
input: n number of states, T number of experience tuples
input: γ discount parameter, α learning rate
output: Q(s,a) the optimal action-state values
initialize Q(s,a) for all states and actions
function Q_learning(n,T)
begin
  for i:=1 to T do
    choose action a in state s that maximizes Q(s,a);
    take s',r;
    Q(s,a)=(1-α)Q(s,a)+ α(r+γmaxQ(s',a'));
    s:=s';
  end
end
```

Figure 2.6
The Q-learning algorithm

Q-learning is said to be model-free since there are no explicitly expressed environment dynamics in the model. It is also true that Q^* values are approached sufficiently even by following greedy policy (i.e. selecting the best action in all states). No matter what policy the agent follows, Q-learning will surely converge. This property is called policy independency or exploration insensitivity and discussed in detail in [R16]. However, the speed of convergence, which has serious feasibility impact to real-life applications, strongly depends on the exploration vs. exploitation issue (see in section 2.7). In [R49] there is theoretically grounded estimation on the speed of convergence depending on the learning time provided that the discount factor is bounded and the state-action pairs are sampled from a fixed probability distribution.

The convergence of Q-learning can be boosted up by rearrangement and simplification of the update rule as it is shown in [R60]. The resulting algorithm is based on the observation that the step-size parameter can be dependent on the number of times the current quintuple occurred in the past. If for example a quintuple has occurred many times, the corresponding step-size parameter may be increased. However, there is

a price for the speedup: the agent must store the quintuple-history centrally, which may exclude any distributed applications.

A generalized convergence model to Q-learning can be found in [R50] where a generalized MDP process is used. All operations (summation, maximization, etc.) are expressed as generalized operators on update rules and values. If the operators are non-expansions, learning rates are decaying and the discount rate is definitely smaller than 1, the optimal Q-values are the unique solution (fix-point) of the following equation:

$$[\Theta Q](s,a) = Q(s,a), \tag{2.26}$$

where operator Θ is the dynamic programming operator, defined in equation 2.27.

$$[\Theta Q](s,a) = R^{a}_{ss'} + g \sum_{s'} P^{a}_{ss'} \max_{a'} Q(s',a') \tag{2.27}$$

It is also presented in [R50] that operator Θ can be approximated by successive values, without hurting the convergence.

2.6 Maintenance of state information

All the algorithms mentioned so far assume that values are stored in a look-up table. However look-up tables may raise memory requirements to the limit of infeasibility especially for large or continuous state-spaces. There are several techniques that allow storing values in a more compact form than lookup-tables. These techniques, which allow compact storage of learned information and transfer knowledge between similar states and actions, are called generalization methods [R16].

If both the number of states and the number of actions are relatively large, storing values in a neural network is a reasonable choice. Consider a multi-layer back-propagation neural network for which the input is the state and action information (in any representation), and the output is the action-state value. In each update step training takes place until the expected output of the network approaches its calculated value at sufficiently small error level. Neural network learning rules and RL update rules can be combined in a single equation such as:

$$? \mathbf{w} = a\left[r + g \max_{a'} Q'(s',a',\mathbf{w}) - Q'(s,a,\mathbf{w}) \right] \nabla_{\mathbf{w}} Q'(s,a,\mathbf{w}), \tag{2.28}$$

where $\mathbf{w}$ denotes the weight vector (parameter vector) of the neural network, Q' the Q-value estimated by the network, and $\nabla_{\mathbf{w}}$ denotes the partial derivatives of the Q-value with respect to each individual weight parameter [R10]. This solution saves memory

and space at the price of extra processor cycles needed to train the network in each update step.

Another possible solution is to use decision trees instead of function approximation for basically doing the same job: mapping the description of each state-action combination to values. Decision trees can be applied in discrete Boolean environments [R8]. At the beginning of the learning process the agent learns Q-values supposing that the whole state space consists of a single state. In parallel there is another process that examines if there is any attribute in the state description that may influence Q-values. If such attribute is found, it is used to split the state space into two parts, into two states.

If the number of actions is relatively large, but the number of states remains reasonably small, the method of *Gullapalli* can be used: actions are represented as probability variables sampled from a normal probability distribution with specified mean and standard deviation [R12]. Actions are selected randomly with respect to the defined probability distribution. Those actions that perform better than others make the mean parameter to shift toward the selected action, while narrowing the standard deviation. On the other hand, if all actions perform poorly the standard deviation is adjusted to be large allowing a large scale of actions to be selected.

2.7 Policies, exploration vs. exploitation

Policy, as defined in section 2.1, is a mapping from states to actions in case of deterministic policies, and state-action pairs to probabilities in case of stochastic policies. The policy of the agent is not the same all the time; it is allowed to change as time goes on, as it should converge to the optimal policy. Policy is the unique factor that determines the behavior of the agent (i.e. *"the way how it selects actions"*). The policy is said to be greedy if the agent chooses the best action, i.e. the action with the largest value with respect to its actual knowledge.

In each time-step the agent has a dilemma: either to *explore* or to *exploit*. There are actions that are well-discovered by the agent, and it can tell at a great level of confidence whether selecting those particular actions is expected to be honored or dishonored by the environment. When the agent selects sure actions, or follows greedy policy, its behavior is regarded as exploitation. On the other hand, there may be actions that are not well discovered, or not discovered at all. These actions, however, may be rewarded worse than well-known actions, but may lead to promising large rewards few steps later. So, if the agent has more time to discover, it can select a non-greedy, discovery action, i.e. it explores the environment. Since the agent knows nothing about the environment at the beginning, it can start up with exploration only, which may gradually turn into exploitation as learning goes on. The crucial question that can be addressed is: How can the two extremes, i.e. exploration and exploitation be balanced? Solutions to this problem can fall into two fundamental categories: decaying exploration

and persistent exploration [R44]. In the case of decaying exploration the explorative behavior gradually turns to exploration, and having exceeded a time threshold, the agent never goes back to exploration without external intervention. The advantage of decaying exploration is that the selected actions converge to the optimal actions, at the price of being insensitive to changes in the environment, when the exploration cycle is over. On the other hand, persistent exploration learning policies keep their exploration capability forever, but at the price of slow convergence[7] [R44].

Greedy policy is a singular persistent exploration learning policy. The problem with pure greedy exploration is that the agent can easily run into local extreme, and it cannot exploit all possibilities provided by the environment. However, if the agent has firm knowledge on not only the immediate rewards, but future rewards as well, this policy should be used.

Greedy policy can be naturally extended when exploration-exploitation has a certain time or probability window, i.e. the agent makes exploration at certain time window, and makes exploitation throughout the remaining time. This policy is called e-greedy, and it can be illustrated as follows: at each time step a e-biased coin is flipped, and if the output turns out to be heads the agent makes exploration or else it makes exploitation. e-greedy policies are typical examples of persistent learning.

It is possible to implement decaying exploration using Boltzmann formula. In this case a control parameter called temperature is used to determine whether exploitation or exploration step follows. Boltzmann distribution based exploration will be discussed in detail in Chapter 3.

There is a general policy class called restricted rank-based randomized (RRR) policies that involve all previously mentioned learning policies. RRR uses rank vector to indicate the relevance of each available action. Though ranks are assigned on the basis of action-state values, the decision itself is done on the basis of ranks, not on the basis of values.

Many of these techniques focus on the convergence in a certain regime. It is appropriate when the environment is not allowed to change. In most of the practical applications the environment is not stationary, thus repeated exploration cycles should be executed.

2.8 Discussion

In this chapter the reinforcement learning concept and the two major classes of reinforcement learning algorithms have been surveyed. Dynamic programming algorithms give theoretical grounds to RL methods, but suffer from the lack of information about environment dynamics. A much more practical approach is to use temporal difference learning that builds up the estimate of the environment by taking

[7] The system will not converge in the classical terms of convergence.

experience samples. Convergence of temporal difference learning algorithms is theoretically grounded, but the speed of convergence is still a key issue. Exploration-exploitation balancing methods are used for speeding up convergence, as well as giving exploration schedules.

Chapter 3

Annealing Schedules Using the Boltzmann Distribution

In this chapter an exploration and exploitation balancing method using the Boltzmann distribution will be introduced. First we examine the convergence behavior and other relevant properties of the Boltzmann formula then we derive temperature bounds which mark a feasible annealing domain.

In each learning period the agent determines what proportion of the episode is spent on exploration and how much time it intends to exploit its actual knowledge to maximize the sum of expected rewards. Given the temperature bounds, and the length of exploration time we propose an annealing schedule technique based on function interpolation, which gradually turns the agent's decision making behavior from exploration to exploitation.

The annealing model is computationally validated on a general test-bed called "*n-armed bandit problem*" that is also used in automata theory. The results of the chapter are based on our paper [P3].

3.1 Introduction

In Chapter 2 an overview of general reinforcement learning solutions has been given. We discussed the model of the goal-driven, reward-maximizing learning agent, and possible solutions to the defined RL problem. We concluded that the agent can make its decision in two different ways: either it makes exploration, i.e. selects non-profitable actions in order to find better states afterwards, or it makes exploitation, i.e. it uses its actual knowledge to select the best action to get large immediate reward. Two important quantities were also introduced: state values and action-state values which indicate the utility of each state and each state-action pair, respectively. These quantities express action choice preferences. The key questions are, how the agent should turn its behavior from exploration to exploitation, or when it should turn back into exploration again.

Balancing between exploration and exploitation is, however, a difficult task. *Sandholm* and *Crites* suggests problem dependent heuristic solutions to exploration exploitation balancing [R32][R42]. *Sutton et al.* in [R46] also discuss that the heuristic solutions "*make strong assumptions about stationary and prior knowledge that are either violated or impossible to verify in applications*".

In this chapter we propose an exploration-exploitation balancing method based on simulated annealing, which is both theoretically grounded and problem-independent.

The decision making problem can be defined as follows: Suppose that the agent has n action choices denoted by $a_1, a_2, ..., a_n$ at time step t. For each action a finite preference value[8] is assigned (or accumulated), which represents the "*utility*" of the particular choice. Preferences corresponding to actions are denoted by $Q_1, Q_2, ..., Q_n$ which are integers and $Q_1 \le Q_2 \le ... \le Q_n$. Suppose that the agent defines a probability distribution over the preference values and selects an action randomly. The control parameters that influence how the actions are selected are the parameters of the probability distribution. A special way of assigning probabilities to values is the Boltzmann distribution proposed by *Sutton* in [R46], defined as

$$p_i = \frac{e^{\frac{Q_i}{T}}}{\sum_{j=1}^{n} e^{\frac{Q_j}{T}}},\ i = 1,2,...,n. \qquad (3.1)$$

The Boltzmann-distribution has a single non-negative real-valued control parameter T called the temperature that governs the action selection policy. (Note the analogy with statistical mechanics where the exponential is $-\frac{E_i}{kT}$ where E_i is the potential energy, and k is Boltzmann's constant.)

There are several appealing properties of the Boltzmann formula (equation 3.1). Two of the most important properties are the invariance property and the scaling property. Both are introduced by *Mahnig et al.* in [R22] in relation to Boltzmann-selection scheduled genetic algorithms.

Property 3.1: *If $\hat{Q}_i = Q_i + c$, where $i = 1,2,...,n$, c is an arbitrary constant value and $\hat{p}$ denotes Boltzmann probability distribution over $\hat{Q}$, then $\hat{p}_i = p_i$, for $i = 1,2,...,n$. In equations:* $\hat{p}_i = \frac{e^{\frac{\hat{Q}_i}{T}}}{\sum_{j=1}^{n} e^{\frac{\hat{Q}_j}{T}}} = \frac{e^{\frac{Q_i + c}{T}}}{\sum_{j=1}^{n} e^{\frac{Q_j + c}{T}}} = \frac{e^{\frac{Q_i}{T}} e^{\frac{c}{T}}}{\sum_{j=1}^{n} e^{\frac{Q_j}{T}} e^{\frac{c}{T}}} = p_i.$

Property 3.2: *Let $\hat{Q}_i = cQ_i$ denote "scaled" preference values, where $i = 1,2,...,n$, and c is an arbitrary constant value. Let $\hat{p}$ denote Boltzmann probability distribution over $\hat{Q}$ using $\hat{T} = cT$ as temperature value, then $\hat{p}_i = p_i$, $i = 1,2,...,n$.*

[8] Throughout the chapter terms "preferences", "action preferences" and "action state values" are used interchangeably covering the same meaning.

Property 3.2 means, that if (for some reason) preference values are re-scaled, in order to have the same distribution, temperature values should also be modified. The property is of great importance in practical applications where preferences may grow without converging to a specified value due to the changes in the environment and are needed to be re-scaled periodically.

3.2 The convergence property of the Boltzmann formula

Theorem 3.1 summarizes the behavior of the Boltzmann distribution with respect to the convergence of the temperature values to zero or to infinity.

Theorem 3.1: *If temperature T approaches infinity, the action selection probability of all the actions approaches the uniform distribution; if T goes to zero the probability of selecting the strictly highest Q-valued action goes to 1, while the selection probability of others' goes to 0. If there are k maximal equally preferred actions, the probability of making selections from among these actions goes to* $\frac{1}{k}$ *as T goes to zero.*

Note that T may never reach 0. Also note that if T goes to 0, the action selection becomes more deterministic. Proof to theorem 3.1 can be found in Appendix A.

3.3 The accuracy of the approach

It is proven that the Boltzmann distribution converges to uniform distribution as T goes to infinity and to the greedy distribution[9] as T goes to 0. It is also an interesting question to what degree the formula approaches the extremities when parameter T changes, if the Q-values are kept constant. The question can be reformulated: Can a maximal temperature be found so that the probabilities p_i approach the uniform distribution with any small error, say e, and also, can a minimal temperature be determined when approaching the greedy distribution is also guaranteed with a sufficiently small error. The answer to both questions is summarized in the following theorem:

Theorem 3.2: *Consider* $e > 0$*, as a small positive number, and an upper and a lower bound on Q-values,* Q_{max} *and* Q_{min}*. The following inequalities are held under these circumstances:*

[9] Greedy distribution is used as a synonym for a distribution in which the probability of selecting a single action is 1 and those of the others are 0.

(a) $\left| p_i - \frac{1}{n} \right| < e$ *if* $T > \frac{Q_{\max} - Q_{\min}}{\ln \min \left\{ \frac{1}{1-ne}, 1+ne \right\}}$, *for* $i = 1,2,...,n$ *and*

(b) $|1 - p_i| < e$ *if* $T < \frac{k}{\ln\left[\left(\frac{1}{e} - 1\right)(n-1)\right]}$, *where* $Q_i > Q_j$, $j \neq i$, $k = \min_{\substack{j=1,..,n \\ j \neq i \\ Q_i \neq Q_j}} \left| Q_i - Q_j \right|$.

Note that k is the minimal difference between the largest and second largest Q-value, and due to the integer nature of the preferences, the difference is minimally 1. Also note that the maximal Q-value is unique.

The minimal T for which the inequality (a) is satisfied will be referred to as T_{max} (or exploration temperature) and the maximal T for which inequality (b) is satisfied will be referred to as T_{min} (or exploitation temperature) throughout the rest of the chapter. The proof of the theorem can be found in Appendix A.

Property 3.3: *In Theorem 3.2 (a) the denominator's independent variables are restricted and cannot take any real value. If* $ne > 1$, *then the argument of the logarithm function is negative, so temperature values exist only on the set of complex numbers.*

However, if

$$e < \frac{1}{n} \tag{3.3}$$

is true, then the denominator of the inequality in Theorem 3.2 (a) becomes

$$\ln \min \left\{ \frac{1}{1-ne}, 1+ne \right\} = \ln(1+ne), \tag{3.4}$$

since $\frac{1}{1-ne} > 1+ne$, if $0 \leq ne < 1$. $0 \leq ne$ is trivial, since neither the number of actions, nor an interval radius can be negative, $ne < 1$ is equivalent to 3.3.

A key question is: given the conditions above, is there any T value satisfying the inequality of Theorem 3.2 (a), as well as minimizing $T(n,e)$. Constraint 3.3 can be rewritten as $ne + n = 1$, where n is a small positive redundancy variable, thus

$$\min_{ne} T(n,e) = \frac{Q_{\max} - Q_{\min}}{\lim_{n \to 0} \ln(2-n)} = \frac{Q_{\max} - Q_{\min}}{\ln \lim_{n \to 0} (2-n)} = \frac{Q_{\max} - Q_{\min}}{\ln(2 - \lim_{n \to 0} n)} = \frac{Q_{\max} - Q_{\min}}{\ln 2}. \tag{3.5}$$

Similarly, the extreme value of the temperature expressed in Theorem 3.2 (b) can also be determined, thus the inequality of Theorem 3.2 (b) can be rewritten as:

$$T < \frac{k}{2\ln(n-1)}. \tag{3.6}$$

Note that using this simplification implies $n > 2$.

Property 3.4: *The computational cost of determining temperature bounds is* $O(n)$ *where n is the number of alternatives in the unsorted preference-values case and* $O(1)$ *in the sorted case.*

3.4 The continuous case

Suppose that preferences and probabilities are not discrete values but functions of some other quantity. Thus, $p:[a,b] \to [0,1]$ and $Q:[a,b] \to [b,c]$, i.e. map real intervals to real intervals. Both functions are bounded, continuous[10], and $[a,b],[c,d] \subseteq \mathbf{R}$. In this case the formula

$$p(x) = \frac{e^{\frac{Q(x)}{T}}}{\int_a^b e^{\frac{Q(x)}{T}} dx} \tag{3.7}$$

also convergent (where $x \in [a,b]$), as T goes to zero or infinity, and the temperature bounds are as follows:

$$T_{\max} = \frac{\max_{x\in[a,b]} Q(x) - \min_{x\in[a,b]} Q(x)}{\ln 2} \tag{3.8}$$

$$T_{\min} = 0 \tag{3.9}$$

Notice that $k = \min_{x\neq y} |Q(x) - Q(y)|$, and in the case of continuous functions $\lim_{x\to y} \min_{x\neq y} |Q(x) - Q(y)| = 0$, where $x, y \in [a,b]$, thus $k = 0$.

[10] In the case of using Lebesque-integral instead of Riemann-integral, this restriction is even unnecessary, which also yields that *?* is nonzero.

3.5 Illustration of the temperature bounds theorem

Preference values of a 7-way example decision problem can be represented by a vector of preference values as follows

$$\mathbf{q} = \{10,101,128,140,50,25,100\}.$$

Temperature bounds for this problem are calculated by using equations (3.5) and (3.6), as $T_{\max} = 201$ and $T_{\min} = 5$. Figure 3.1 shows the probability distribution of this decision problem at temperature levels $T_{\max}$ and $T_{\min}$. It is easy to observe that the probability distribution is almost deterministic around $T_{\min}$ and follows uniform distribution around $T_{\max}$. Note that the example uses the simplified equations which do not require any error level.

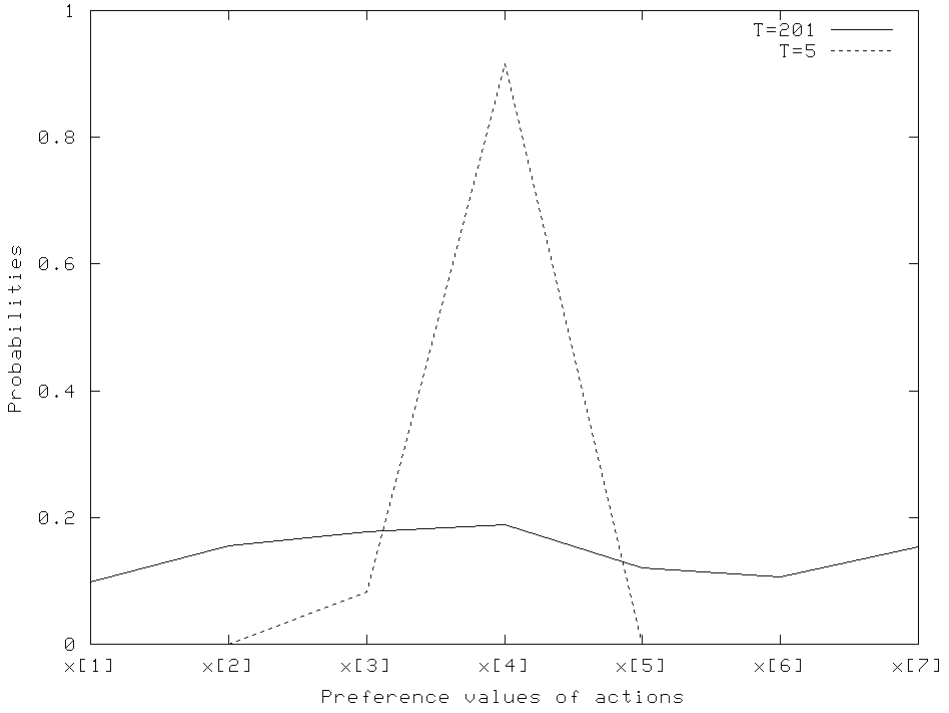


Figure 3.1
Probability distribution of a 7-way decision problem

3.6 Annealing schedules

While the theorems in sections 3.3 and 3.4 establish temperature bounds, this section deals with the question how interpolation methods can be used for determining an

appropriate annealing schedule between these bounds on a given time horizon. A method based on differential equations will be introduced that transforms a parameterized temperature-time function to a recursive annealing schedule function which can be successively applied by the learning agent. Formula 3.10 shows the general iterative function

$$T(t_{k+1}) = f[T(t_k)], \tag{3.10}$$

where $T(t)$ denotes the time dependency of the temperature. Iterative application of 3.10 is applied to descent temperature from $T_{\max}$ to $T_{\min}$.

First we show the general method, then, four special annealing functions will be derived: linear, exponential, quadratic and inversely quadratic annealing.

3.6.1 General annealing method

Given a parameterized function of temperature in the following form

$$T(t) = \sum_{i=0}^{n-1} c_i f_i(t), \tag{3.11}$$

where constants c_i are the parameters being sought, and $f_i(t)$ functions are arbitrary continuous functions.

The derivative of equation 3.11 is expressed as

$$\frac{dT(t)}{dt} = \sum_{i=0}^{n-1} c_i \frac{df_i(t)}{dt}. \tag{3.12}$$

In order to determine all parameters, preconditions must be set, such as

$$T(t = t_j) = T_j,\ j = 0,1,...,n-1 \tag{3.13}$$

where T_j is an arbitrary temperature value. Note that all T_j values are recommended to fall between $T_{\max}$ and $T_{\min}$. However, this is not vital, but as a direct consequence of Theorem 3.2 there is no reason to set any temperature value outside the bounds.

For simplicity, vector abbreviations are used. Let vector $\mathbf{t} = \{T_0, T_1, ..., T_{n-1}\}$ denote all the temperature values set by boundary conditions in 3.13. Let $\mathbf{c} = \{c_0, c_1, ..., c_{n-1}\}$ denote the parameter vector, and let $\mathbf{F} = \{f_i(t_j)\}$ be the matrix of function values at $t_0, t_1, ..., t_{n-1}$.

Parameters can be determined by solving the following linear equation system:

$$\mathbf{t} = \mathbf{Fc} \tag{3.14}$$

thus if **F** can be inverted and $\mathbf{F}^{-1}$ denotes the inverse

$$\mathbf{c} = \mathbf{F}^{-1}\mathbf{t}\,. \tag{3.15}$$

Equation 3.12 is to be used to get an iterative annealing function:

$$\begin{aligned} T(t=0) &= T_{\max}\,, \\ T(t_{k+1}) &= T(t_k) + \sum_{i=0}^{n-1} c_i \frac{df(t_k)}{dt}\,. \end{aligned} \tag{3.16}$$

Equation system 3.16 gives the general annealing schedule model. Simple methods such as linear or quadratic schedules are treated as special cases of the formula above.

3.6.2 Linear annealing

Equation 3.16 determines the general form of recursive annealing schedule. In practical applications polynomial functions are used. The simplest polynomial is the linear function interpolated between the points determined by t_{start}, t_{end}, $T_{\max}$ and $T_{\min}$.
The linear annealing function is given in the following form:

$$T(t) = c_1 t + c_0\,, \tag{3.17}$$

where t_{start} denote the beginning of the annealing process, t_{end} denotes the end, c_0 and c_1 are constants. The derivative equation with respect to variable *t* is as follows:

$$\frac{dT}{dt} = c_1\,. \tag{3.18}$$

The boundary conditions in this case are defined as follows:

$$T(t = t_{start}) = T_{\max}\,, \tag{3.19}$$

$$T(t = t_{end}) = T_{\min}\,. \tag{3.20}$$

Substituting 3.19 and 3.20 into 3.17 yields $c_1 = \dfrac{T_{\min} - T_{\max}}{t_{end} - t_{start}}$. Equation 3.18 can also be rewritten as $dT = c_1 dt$, that is the differential form of $\Delta T = c_1 \Delta t$. The later difference equation is used for writing the recursive form as follows:

$$T(t=0) = T_{\max}, \tag{3.21}$$

$$T(t_{k+1}) = T(t_k) + \frac{T_{\min} - T_{\max}}{t_{end} - t_{start}}. \tag{3.22}$$

3.6.3 Quadratic, inversely quadratic and exponential schedules

Following similar reasoning as in the case of linear annealing, higher order polynomials, such as

$$T(t) = c_n t^n + c_0 \tag{3.23}$$

take the following recursive form:

$$T(t=0) = T_{\max}, \tag{3.24}$$

$$T(t_{k+1}) = T(t_k) + n\frac{T_{\min} - T_{\max}}{t_{end}^n - t_{start}^n} t^{n-1}. \tag{3.25}$$

Figure 3.2 shows different annealing schedules where $T(t) = c_4 t^4 + c_0$, $T(t) = c_2 t^2 + c_0$, $T(t) = c_1 t + c_0$, and $T(t) = c_{-1}\sqrt{t} + c_0$ when $T_{\max} = 1000$, $T_{\min} = 100$, $t_{start} = 0$, $t_{end} = 150$.

3.7 Variable preferences

The annealing models we have shown so far assume that all preference values are constant. In real-life applications, however, this is not true, since preference values are allowed to change in time, as a result of learning. Varying preferences may also yield varying temperature bounds if the maximal, the second largest or the minimal values change. Since both $T_{\max}$ and $T_{\min}$ are bounds, it is possible that the Boltzmann-distribution defined by equation 3.1 may approach the greedy distribution at larger temperatures than $T_{\min}$ and uniform distribution at temperatures smaller than $T_{\max}$ with error level of e. If the temperature bounds are allowed to change in a hysteresis-like manner, the system is much more fault-tolerant, in terms of following the theoretical bounds much more accurately.

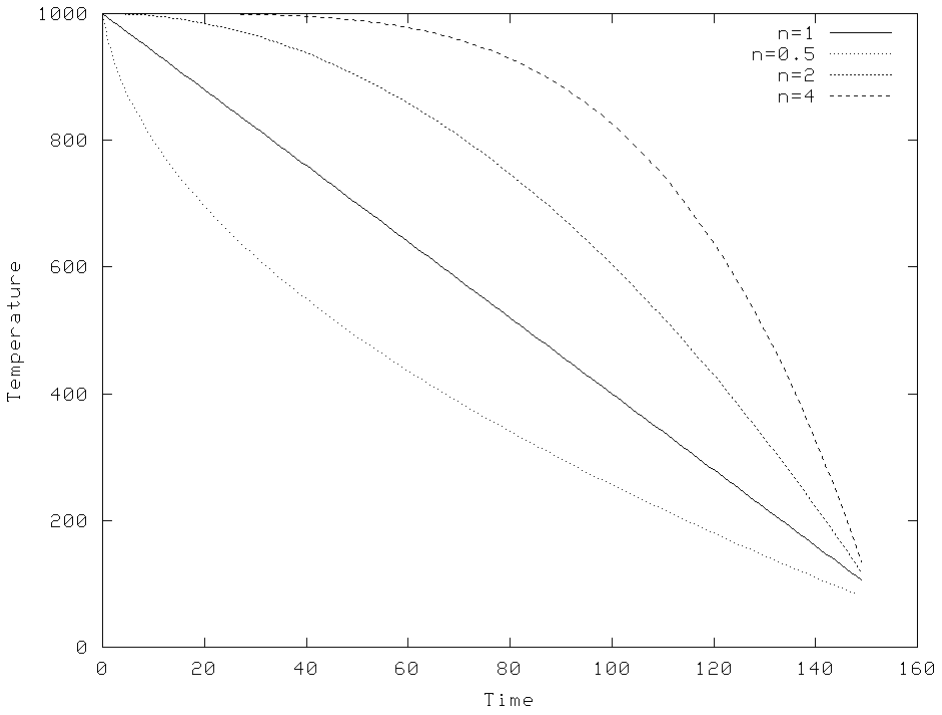


Figure 3.2
Polynomial annealing schedules (equation 3.25) satisfying boundary conditions for different values of n

In many cases temperature bounds may change significantly especially when extra large rewards or penalties occur. The fundamental rule of thumb in this case is that bounds must be changed in a discrete-valued manner, i.e. two versions of bounds are maintained: the original version and an actually computed version. Let the actual value of the upper bound be denoted by $T'_{\max}$ and the original value by $T_{\max}$, and similarly for the lower bound: $T'_{\min}$ and $T_{\min}$. The annealing schedule is always defined for the original temperature values, $T_{\max}$ and $T_{\min}$.

If the actually computed maximal temperature value $T'_{\max}$ grows beyond a certain limit, say, $cT_{\max}$ where c is a constant, the new value of $T_{\max}$ is set to $T'_{\max}$ and an annealing sub-period takes place. (The constant is referred to as re-scale factor.) The similar is true for $T_{\min}$. An annealing sub-period means a new annealing process from the points determined by the new, varied temperature values and the current time, as a start point of the sub-period, and the original end-point. The re-annealing principle can only fail if one of the preference values grows significantly beyond that of the others at a larger pace than annealing could decrease the temperature. The rule of thumb for this case, which is in harmony with the way how the re-scaling automatically works anyway, is that if the preference of one action grows to a large value, the agent is convinced of selecting this action later in the exploitation phase, since it is significantly

better than the others. In this situation the decision-making agent would not need annealing at all.

In Figure 3.3 a linear annealing re-schedule is shown. T_{max} is computed as 10 at the beginning, temperature re-scale factor on the maximum bound is set to 1.5, T_{min} is computed as 0.3, exploration time is set to 2/3 of the whole episode time. It can be seen that the temperature decreases linearly from time step 0 until time step 50. At this point preferences are modified significantly, which influences temperature bounds that are re-scaled, i.e. increased over 1.5 times of the old value. Maximal temperature is set to 15 at this point and a new and quicker scheduling sub-period continues as the temperature reaches T_{min} at time step 66.

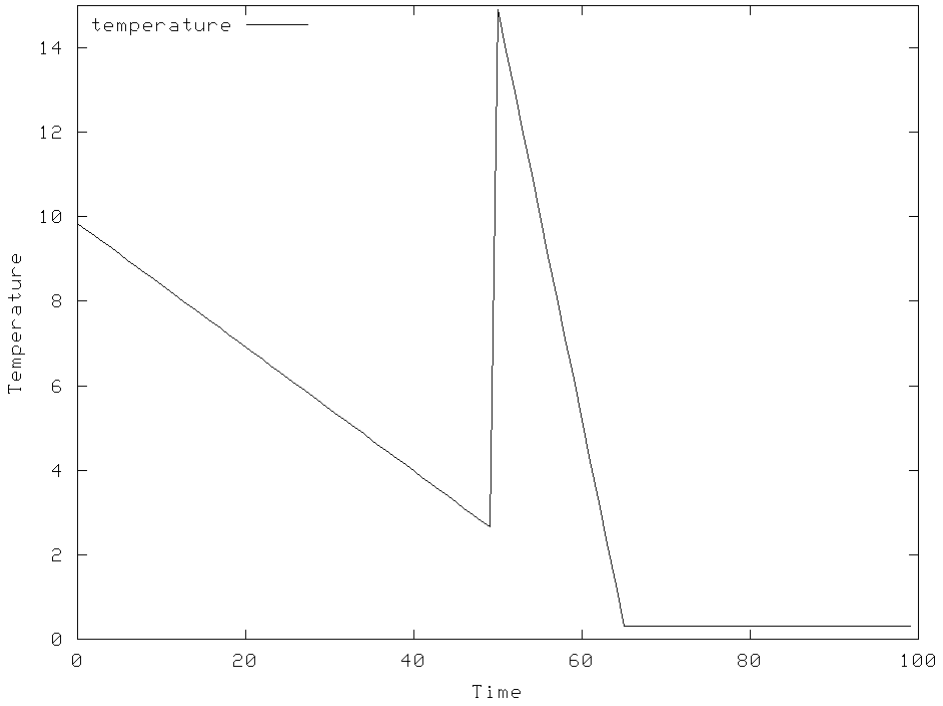


Figure 3.3
Linear re-annealing process

3.8 Computational validation of the annealing model

Although the annealing schedule model has theoretical roots, it is necessary to validate it. There is a famous test-bed in automata theory known as the "*n-armed bandit problem*" [R46].

Consider the following learning task: An agent faces to a decision problem in which there are *n* choices. Selecting one of the options results in a reward taken from a stationary probability distribution. The value of each choice is consistent, i.e. repetitive samples are taken from the same probability distribution, thus each decision result is

treated as a probability variable which is unknown to the agent. What the agent can do is to make estimates on these variables through trial and error.

The bandit problem has practical interpretations as well: the *n*-armed bandit is the *n*-dimensional extension of the one-armed bandit slot machine. The agent selects an arm, makes a pull, then either he wins or he looses, in numerical terms, it receives reward of 1 or 0 (i.e. he makes a play). The goal is to find a *"winner strategy"* in which the agent maximizes the amount of rewards obtained through series of plays, episodes.

The agent can basically do two things: if it has lots of pull opportunities, it is better to explore which options (arms) are better, and which arms are worse than the other ones. When he has not too many pulls remained, it is better to select the action that he thinks to be the best, it is better to exploit the knowledge.

As a validation, a program written in C has been used for studying different action selection strategies and annealing schedules. Parameter setting of the test-bed is shown in Table 3.1.

Parameter	Value
Number of actions	10
Number of plays in an episode	100
Number of episodes	10000
Annealing method	linear, polynomial annealing, variable temperature bounds

Table 3.1
Properties of the *n*-armed bandit test-bed

The number of episodes is set to a large value in order to average individual differences and concentrate on the average behavior. All the results presented in the rest of the section are averages on successive episodes. Exploration/exploitation balancing works well if the number of plays is relatively large to the number of choices, in this case the pull number 100 is sufficiently larger than the number of choices 10. (Note that plays in this case function as time steps.)

In Figure 3.4, comparison of different annealing schedules is shown. The continuous line denotes e-greedy action selection schedule when e is set to 0.1; the x-axis denotes the number of plays and the y-axis denotes the development of reward obtained during the play. The line entitled by "boltzmann-t1-lin-fix" indicates Boltzmann distribution based action selection using fixed temperature bounds, linear annealing schedule and the whole episode was dedicated to exploration (i.e. the end of annealing schedule is coincident of the end of the episode). As another strategy "t1/10" indicates that only 10 percent of the episode is spent on exploration, and the rest of the time was spent on greedy action selection. Similar comparisons of different strategies can also be found in [R46].

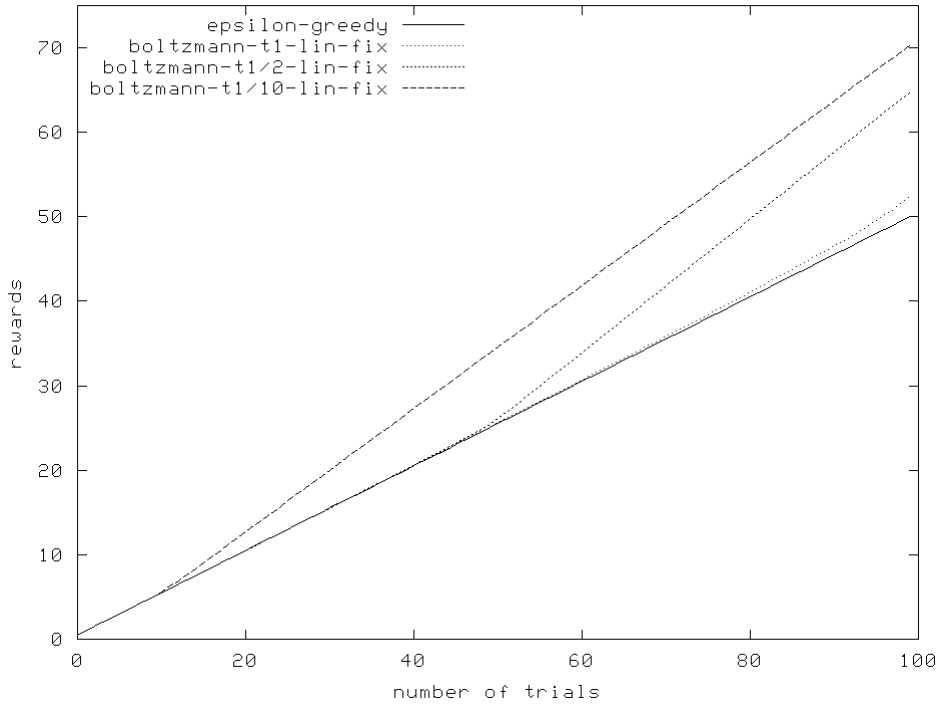


Figure 3.4
Comparison of different annealing strategies, 0.1-greedy vs. Boltzmann action selection with different annealing schedule length

It is easy to see that the reward development as well as the final sum of rewards is dependent mostly on the percent of time spent on exploration. 0.1-greedy strategy shows slightly better performance than random behavior on average since the exploration was continuous (i.e. with percent of 0.1 the agent explore) and the gain was distributed linearly along the development of the reward. Increasing e to 1 turns the system to totally random; decreasing e to 0.05 gives better performance again. Note that pure greedy method is missing from the comparison since it is not considered be a "*self-initializing*" method; the agent has zero initial estimates on the expected reward[11] so "*greediness*" is desirable only at a later stage.

Boltzmann distribution action-selection, that spends the whole time on exploration, gives better results than 0.1-greedy method, but only from the middle of the episode. The reason why is that it explores too much of its time, and the agent starts to use its knowledge only at the end of the period.

Boltzmann selection having half-time exploration slightly diverges from the 0.1-greedy reference approximately from the middle of the episode. At this point the behavior of the agent gradually turns from exploration to exploitation, and while it gives

[11] If greedy algorithm is used with zero initial values, the average behaviour turns to be random, since further selections are dependent on the first one.

similar results to random behavior in the first half, the agent uses its knowledge to perform better and give better final reward in the second half.

The curve entitled by "boltzmann-t1/10-lin-fix" shows that asymptotically the same amount of reward development can be gained, but only at an earlier stage when exploration finishes at 12-15 percent of the whole time. This also results in a better final reward, since exploration stops at an earlier stage and the agent chooses the best action more times. There is an important remark here: Within a certain time-interval the resulting reward is not significantly influenced by the length of the exploration period (Figure 3.5), but there exists an exploration time at which the resulting reward is maximal and this point is around 10 percent of the episode time in this example, that is the proportion of episode plays to the number of choices.

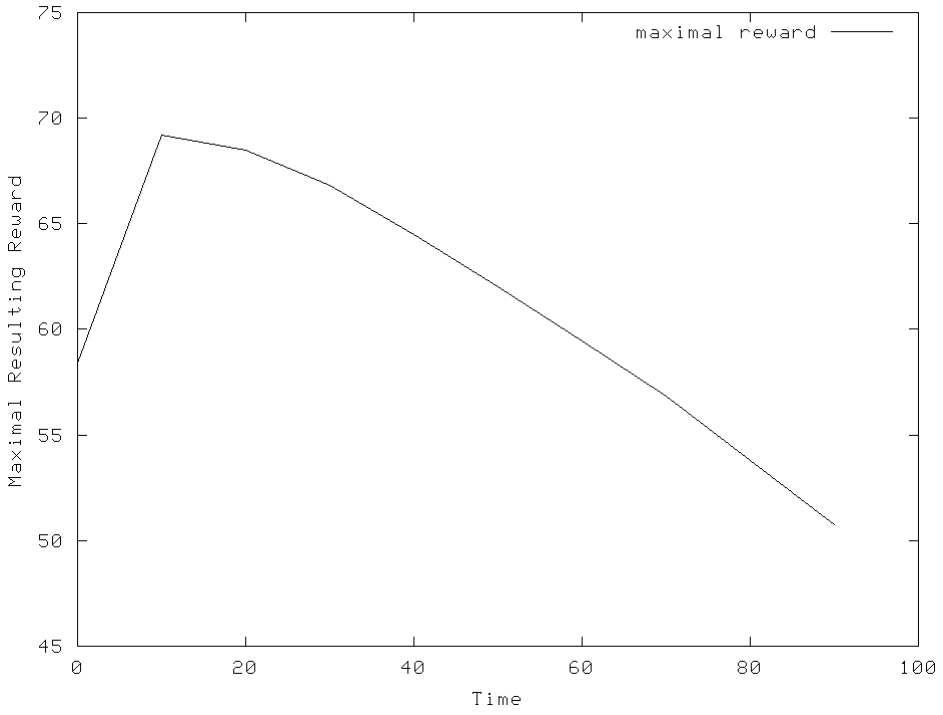


Figure 3.5
Maximal reward vs. exploration time diagram. The maximal reward can be obtained when exploration time is around 10 percent of the whole episode time

In Figure 3.6 the same annealing schedules are shown as in Figure 3.4, but using variable temperature bounds. The temperature re-scale factor is set to 1.5 for $T_{\max}$ and 0.75 for $T_{\min}$, i.e. T is re-adjusted when it grows above $1.5T_{\max}$ or drops below $0.75T_{\min}$. It can be seen that the resulting rewards are better for longer exploration periods as well, than those of fixed temperature bounds. In the case of full-time annealing (exploration during the whole episode) with fixed temperature bounds, the test produces final reward of 52, while using the same schedule, but variable temperature bounds it produces

around 56. And similarly halftime-exploration annealing schedule (denoted by "t1/2") that uses fixed bounds produces sum of reward of 65, while that uses variable bounds produces results above 67. A reasonable explanation for the phenomenon is that the re-annealing "*shakes-up*" the system.

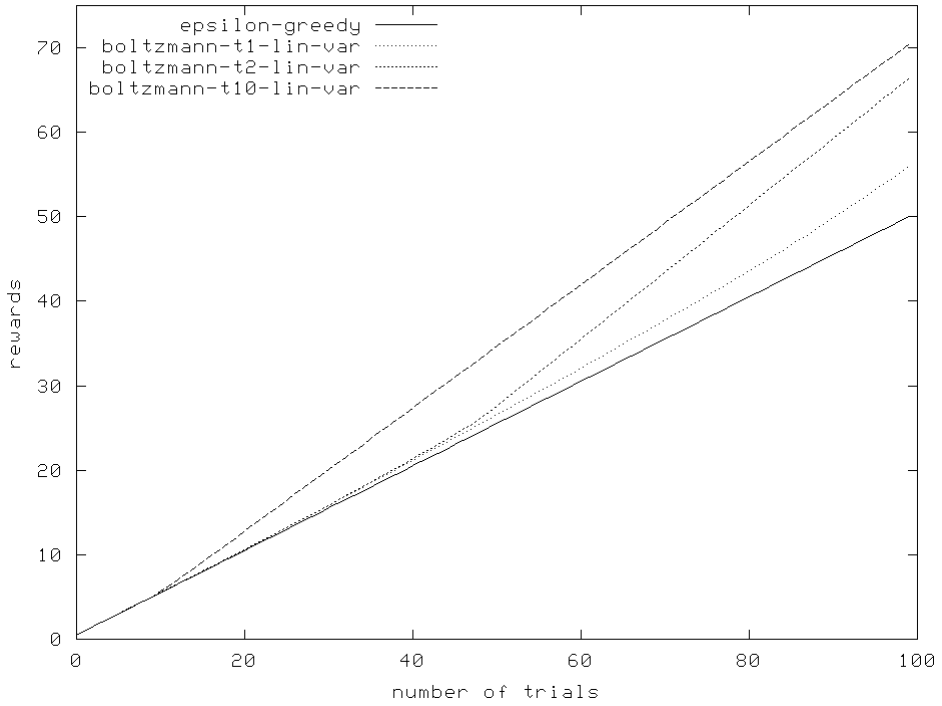


Figure 3.6
Annealing schedules using variable temperature bounds

3.9 The use of Boltzmann distribution in other fields

Using Boltzmann distribution in machine learning algorithms has been popular in the late 1990s. Apart from reinforcement learning, there are two other fields where Boltzmann distribution is used: genetic algorithms (GA) and ant colony systems (ACS).

In [R22] a novel genetic algorithm is proposed which operates on closed population and aims at finding the global extremes of the fitness function of population elements. A probability value is assigned to each population member, which forms Boltzmann distribution over the whole population set. An annealing schedule is used for replacing the existing probability distribution to a new one, which is also a Boltzmann distribution. The new distribution increases the average fitness, thus giving better and better population.

Decision problems also appear in ant colony systems, where an agent (in this case it is called ant) faces to either follow a known trail reinforced by other ants, or to

choose a different path in every time step. Each available path, from node i to node j, a probability value is assigned to a trail as follows

$$p_{ij} = \frac{t_{ij}^{a} h_{ij}^{b}}{\sum_{j \in path\_is\_allowed} t_{ij}^{a} h_{ij}^{b}}, \tag{3.26}$$

where t_{ij} is the strength of the trail on the path, h_{ij} is the visibility factor and a, ß are control parameters. As an alternative to equation 3.26 a two-dimensional Boltzmann distribution can also be used:

$$p_{ij} = \frac{e^{a t_{ij}} e^{b h_{ij}}}{\sum_{j \in all\_paths} e^{a t_{ij}} e^{b h_{ij}}}. \tag{3.27}$$

In equations 3.26 and 3.27 a and ß are referred to as "*inverse temperature*" parameters.

3.10 Discussion

In this chapter a simulated annealing concept tailored to reinforcement learning was shown. The agent's decisions are performed as a result of a random action selection based on a Boltzmann distribution. The nature of the decision is determined by a single control parameter called the temperature. It was shown that there are two characteristic distribution types exist with respect to the value of the control parameter, greedy and uniform distributions. We have shown that both can be approached with a sufficiently small error using finite temperature values; theoretically grounded estimations of these bounds were given.

The agent's decision making policy can be modified by varying the value of the control parameter in the determined temperature range marked by temperature bounds, either manually or in a dynamic way, by applying annealing schedules determined by temperature bounds and an annealing/exploration time period. The general annealing framework and the derivation of simple schedules have also been shown. The whole model was validated on the popular "*n-armed bandit problem*" and was found to perform better than any other schedules.

Chapter 4

Internet Protocol Packet Routing Algorithms

This chapter gives an overview on dynamic IP routing protocols such as border gateway protocol, routing information protocol, or open shortest-path first protocol. Some features, advantages as well as shortcomings will be outlined, and then a different routing concept, which is called agent-based routing, or Q-routing, will be introduced. We show a combined Q-learning and simulated annealing algorithm which uses temperature bounds to Internet Protocol (IP) packet routing to remedy two problems of the original Q-routing: patch recovery and loop detection.

There are three papers on which the results of this chapter are based on: [P4], [P5] and [P8].

4.1 Introduction

Internet Protocol (IP) Packet Routing, or just simply routing, which is a crucial part of Internet data transfer processing aims at providing transparent packet delivering service to applications. The term routing can be defined following *Tannenbaum* [51] as the task of delivering data packets from one computer host to another one through one or more intermediate nodes in finite time steps. If there are no intermediate nodes the data delivery task is simply called switching or bridging. The host that initiates delivery is called source node, the host that receives data is called sink or destination node. Network nodes are identified by unique addresses, in case of IP version 4 network, by 32-bit IP-numbers.

Routing is a compound task and it consists of three basic building blocks: shortest path determination, routing table maintenance, and IP packet forwarding at network level. Figure 4.1 shows the place of the routing process in the operating system (OS) architecture.

IP routing is based on IP packet forwarding, which refers to the process of placing data packets from one network interface connection (NIC) to another one, is basically carried out by the communication protocol stack of the OS kernel. In the figure arrows indicate this process. IP-forwarding is carried out with respect to well-defined routing rules which are summarized in kernel-level data structures, so-called routing tables. Routing tables which hold information about the known network topology can either be manipulated by hand or by user level applications (indicated by BGP, OSPF or RIP blocks in Figure 4.1). Entries in the table implement simple rules:

they assign an outgoing interface to each possible destination entities. Basically there are two kinds of entries: network entry and host entry. A network entry is identified by the IP address-range in network-address/subnet-mask form, a host entry is a unique IP-number without any subnet-masks. Figure 4.2 shows typical routing table entries in UNIX operating systems. (A more detailed routing table example of a Cisco router can be found in Appendix B.) The example routing table joins two networks, 192.111.1.0/24 and 192.111.2.0/24, together. The first two entries tell the kernel to send packets that have destination address of 192.111.1.x to the Ethernet device named eth0 and send packets belonging to 192.111.2.x to device eth1, to the other side of the router. There is also a "default" line that gives default direction to those packets that belong to neither of the previously specified networks.

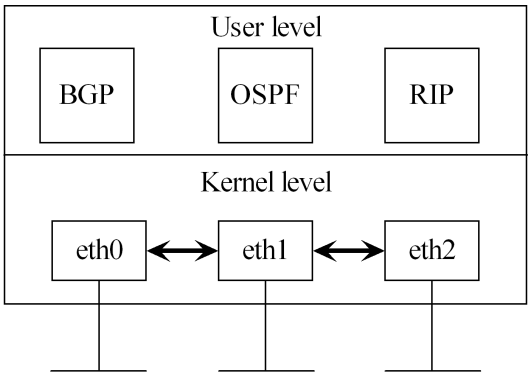


Figure 4.1
Routing processes in the operating system

Destination	Netmask	Gateway	Flag	Interface
192.111.1.0	255.255.255.0	192.111.1.254	G	eth0
192.111.2.0	255.255.255.0	192.111.2.254	G	eth1
default		192.111.3.1	G	eth2

Figure 4.2
Routing table entries in UNIX operating system

If the default line is missing or the packet does not match any of the rules, it is simply dropped. IP-forwarding is just a mechanical task that does not require any knowledge on the real network topology; it is just a sequence of simple matching and switching operations.

Those routing algorithms, for which routing table entries always remain the same, are referred to as static routing or non-adaptive routing algorithms. Static routing

can be applied whenever the network topology remains stationary or changes may occur only under full administrative control.

On the other hand, there is another class of algorithms named dynamic routing algorithms, which let the routers exchange link or address information with each other on the application level. These algorithms are allowed to make modifications in the routing table with respect to the routing information received, thus allowing adapting to changes in network topology. Information exchange is performed through specified routing protocols that describe exactly what information the neighboring routers exchange. The exchange communication is usually performed over the network layer by user level background processes. The routing table is a shared resource in the sense that multiple routing algorithms may operate on the same table at the same time: most of the routing daemons tag rules they last inserted or modified. Specific network appliances allow the usage of multiple routing tables in order to create isolated private networks.

Dynamic routing algorithms store routing information in a separate, user-level database on which they carry out computations. No intermediate rules are permitted to enter the routing table; only secure rules are registered.

4.2 Principles of dynamic routing algorithms

4.2.1 Classification and metrics

Tannenbaum in [R51] classifies dynamic routing algorithms by using various criteria.

Single-path algorithms determine a single path to the destination node, while multiple-path algorithms use multiple routes to the same destination, thus letting multiplexing of IP-packets.

Flat algorithms use a single level of hierarchy, each router knows about each other. Hierarchical routers define special network areas, or autonomous systems (AS) that belong to the same administrative domain. Routers, which are responsible for exchanging packets between autonomous systems, are referred to as backbone routers or edge routers. Edge routers may build up a backbone topology for backbone information in separate routing data structures as if the backbone topology was a separate network. Outgoing packets coming from non-backbone routers are directed to the backbone router where it leaves the AS. Intra-domain routers are allowed to recognize routers within the same AS only.

Routing algorithms may use many distance metrics when calculating the best route, which are as follows: path length, reliability, routing delay, bandwidth, and load [R40].

- One of the simplest metric is the cost value assigned by network administrators to each link. The sum of costs along a route traversed is used as path length. A

special case is when unit costs are assigned to each traverse through a router: hop counts indicate how many network devices the packet has passed through.

- Reliability can also appear as a metric. Some network links may be down more often than others. Similarly, after a failure some links are repaired faster than others. The reliability of each network device can be measured numerically by an error ratio.
- Routing delay refers to the total length of time needed to transport a package from the source to the destination. Delay actually is a compound metric since it may be influenced by many other factors, such as bandwidth, congestion rate, or physical distance. It is a common and useful metric in practice.
- Bandwidth refers to the maximal available traffic capacity of a network link. Larger bandwidth does not necessarily mean larger transport speed. If, for example links with large capacity are busy it is reasonable to choose a link that has lower bandwidth, but is less busy.
- Load metric measures how busy the neighbor router is in terms of CPU-load, memory-load or packet-load.

4.2.2 Design goals

There are several design goals that a routing algorithm must fulfill. During years of operation the network topology may change, links can go up and down hardware or software errors may occur, but the whole network must operate robustly, without any single point of failure[12] [R51]. The properties are as follows: simplicity, robustness, convergence, flexibility and optimality.

Optimality in the definition of *Pierre et al.* in [R34] refers to the capability of finding the best route from the source node to the destination with respect to a specified metric. Optimality may also be defined as reaching the maximal throughput [R40].

Simplicity matters in two ways: firstly the algorithm must utilize as little software resources as possible to be efficient. Routing decisions are made for every packets or group of packets[13], so there is no place for any software or hardware overhead. Secondly the larger the algorithm, the greater the probability of having bugs inside.

Robustness refers to the capability of the algorithm to behave in the expected way even under unexpected or unforeseen circumstances, such as hardware failures, large load conditions. Routers have central role in a network, so if they fail, the whole AS may break down.

[12] The term single point of failure is used to refer a single critical point in the system: if the single point damages, the whole system breaks down.
[13] When routing or switching decision is made for a sequence of packets with common source and destination it is called flow-based switching.

Convergence refers to the agreement among different routers on the best routes. The speed of convergence is also a crucial issue. If, for example, a link goes down, all other routers must be informed as soon as possible, to keep their routing tables in consistent state. Corrupt routing tables may cause loops, or improper delivery.

Routing algorithms should also be flexible that means to quickly adapt changes in the environment. In case of link failure the algorithm must quickly divert the traffic to the second best route.

4.2.3 Distance-vector algorithms

Distance-vector algorithms were one of the first algorithms used in the early ARPANET and Novell IPX systems. They are also called Bellman-Ford or Ford-Fulkerson algorithms for the honor of its developers *Bellman*, *Ford* and *Fulkerson*.

Distance-vector algorithms are based on the following principle: each router maintains a table, or more precisely a vector indexed with each possible destination routers and stores known distance values as well as neighbor identifiers that provide route to them. Each router knows only which neighbor will take the packet closer to the destination. The distance from the immediate neighbors is measured directly by special "hello" packets. After receiving routing table information update from the neighbors, all entries are checked, and if a neighbor provides better route to the destination, than the one which is already known, the routing table entry is updated. Operation of the algorithm is illustrated in Figure 4.3.

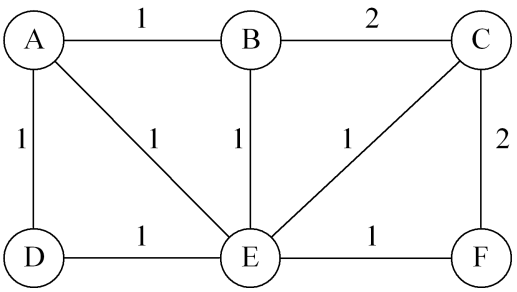


Figure 4.3
Network example of distance-vector routing. Numbers on edges indicate distance, in terms of some metric weights, e.g. packet delays

In the example, routing information is exchanged in $\langle destination, distance, neighbor \rangle$ triples, which carry the distance information to the destination via a certain neighbor. For example the full routing table of node A at the beginning can be written as

$$T(A) = \{\langle \boldsymbol{?,0,?} \rangle, \langle \boldsymbol{B,1,B} \rangle, \langle C,-,\infty \rangle, \langle \boldsymbol{D,1,D} \rangle, \langle \boldsymbol{E,1,E} \rangle, \langle F,-,\infty \rangle\}$$

where A knows the distance to its direct neighbors, and knows no route to nodes C or F. It is a trivial assumption that it knows a route to itself. (The router and all immediate neighbors are highlighted by bold letters.) Similarly tables for nodes B and E are as follows:

$$T(B) = \{\langle \mathbf{?,1,?} \rangle, \langle \mathbf{B,0,B} \rangle, \langle \mathbf{C,2,C} \rangle, \langle D,\text{-},\infty \rangle, \langle \mathbf{E,1,E} \rangle, \langle F,\text{-},\infty \rangle\},$$
$$T(E) = \{\langle \mathbf{?,1,?} \rangle, \langle \mathbf{B,1,B} \rangle, \langle \mathbf{C,1,C} \rangle, \langle \mathbf{D,1,D} \rangle, \langle \mathbf{E,0,E} \rangle, \langle \mathbf{F,1,F} \rangle\}.$$

Suppose that router B sends table update to router A. Router A adds the measured distance value to B (which is 1) to all distances sent by router B, and compares the new values to the old ones, if they provide smaller distances. There is no improvement for entries A and B, but B knows a route to C with distance value of 2. So, router A registers router B to be the next hop toward router C and the estimated distance is 3. Note that B still does not know anything about router F, so the corresponding entry remains untouched. Also note that router A still has not sent its routing table to router B so B's entries also remain untouched. Router A's resulting table is thus:

$$T(A) = \{\langle \mathbf{?,0,?} \rangle, \langle \mathbf{B,1,B} \rangle, \langle C,3,B \rangle, \langle \mathbf{D,1,D} \rangle, \langle \mathbf{E,1,E} \rangle, \langle F,\text{-},\infty \rangle\}.$$

Suppose that router E also sends its routing table update to router A. Router A checks the table received in the same way as before and finds that a better route exist to router C via node E rather than via node B. It also finds a route to node F, thus entries for nodes C and E in the routing table of node A are updated resulting:

$$T(A) = \{\langle \mathbf{?,0,?} \rangle, \langle \mathbf{B,1,B} \rangle, \langle C,2,E \rangle, \langle \mathbf{D,1,D} \rangle, \langle \mathbf{E,1,E} \rangle, \langle F,2,E \rangle\}.$$

Although the distance-vector algorithms function well in theory, they have a serious practical drawback: they converge to the correct routes, but they do it slowly. Good news, i.e. a link is up, spread over the network at the speed of 1 hop/exchange, but bad news, such as some routers or links have been damaged spread over much slowly. *Tannenbaum* refers to this problem as counting-to-infinity [R51]. Since all routers must rely on the information received from its neighbors, and this is the only source of information from indirect nodes, it is possible when a router goes down, one of its neighbors may receive information exchange packets from those routers that are not directly connected to the damaged routers (say, from an indirect router) establishing routes to the damaged router via its direct neighbor. The neighbor router does not take into account that the route, which may be proposed by the indirect router, is actually a route that traverses via itself, and adjusts its routing table to have a longer route to the damaged router via the indirect router. This yields loops in the delivery as well as causes improper transmit rules. Several solutions exist to solve counting-to-infinity

problem. The most popular one, though it also fails in some situations, is the split horizon model. The essence of split horizon is to send infinitive distance from router A on those outgoing lines which are used for reaching router A thus preventing the farther router from giving misleading information. The indirect neighbor, in this case, naturally says infinitive path length to the direct neighbor about the damaged router, so the router can be recognized as unreachable.

4.2.4 Link-state algorithms

Link-state algorithms are conceptually different from distance-vector algorithms. They were used in late ARPANET system and are widespread in contemporary commercial software products as well.

Link-state algorithms also operate by topology information exchange over the network protocol, but not the whole topology is exchanged, just portions of it. Each router reports state information about its links, neighbors, and delays. Link information is then forwarded throughout the network without any modification by intermediate nodes, and finally all nodes receive information on all other nodes and their links. Then, the topology on all routers can be built up using the information portions, and shortest path calculations can be executed to determine the best routes [R51].

The first task a router should do is to discover all of its neighbors. This is done in a similar way as in the case of distance-vector based algorithms, using special "hello" packets. Further, each router need to have a unique identifier (which is not necessarily the IP address, but a protocol dependent number commonly provided by the manufacturer) that helps to identify all routers on all possible nodes of the network. Then the line costs are measured using Internet Control Message Protocol (ICMP) "echo" packets. Measurement can either be load dependent, or independent, which is a configuration issue. For the sake of precise results, it is common to repeat measurements and take the average on all measurements.

When a router knows the distances to all neighbors it builds link-state packages that basically consist of the identifier of the sender, and all neighbors, and also the distance among each node pair. For example, in the network shown in Figure 4.3, router A may send a packet like $L(A) = \{\langle B,1\rangle, \langle D,1\rangle, \langle E,1\rangle\}$ or router F $L(F) = \{\langle C,1\rangle, \langle E,1\rangle\}$, which are relayed by other routers, so all routers receive them.
Link-state packages can be sent not only periodically but whenever some event occurs, thus preventing the network from unnecessary communication burden.

The most crucial part of any link-state routing algorithm is how the packets are distributed. Routers that are closer to the sender may receive link-state packets earlier than those are being farther from it, which may also lead to loops, inconsistent routing databases, or unreachable network portions. Basically flooding is used for distributing link-state information with some extensions: a counter that functions as a timestamp on each packet is maintained by each router. If multiple packets are found which have

identical timestamps, one of them is dropped. If several packets are received from the same node, the older packet is dropped. There is an auxiliary life field in each packet, which is decremented in every minute. When the life of a packet reaches a threshold value, it is simply dropped.

Whenever a link-state packet is received, it is not immediately relayed, but it is stored for some time in a queue to let some further checks be carried out. When no packets are received, packets still in the waiting queue are sent over.

When enough link-state information is gathered, each router can build the model of the whole network topology. In order to determine the source spanning tree of the network, i.e. a tree graph structure which contains the shortest paths to other nodes, shortest-path computations are carried out. Such an algorithm is Dijkstra's algorithm (see section 4.4). The tables are updated with respect to the results of the shortest path computations.

4.3 Dynamic routing protocols

Dynamic routing protocols refer to the implementation of routing principles. Routing protocols should not be mixed up with routed protocols. While the former refers to data structures and algorithms responsible to computations finding the best way to a specified destination, the latter refers to protocols routed over the network. Typical examples to routed protocols are IP, IPX, Novell NetWare or DECnet. The most popular routing protocols are discussed in the following.

4.3.1 Routing Information Protocol

Routing Information Protocol (RIP) was the first routing protocol used in networking. The protocol specification was proposed by *Hedrick*. RIP is basically a distance-vector algorithm that uses hop count as a metric. In order to avoid routing loops RIP sets a limit on hop counts, which is a defined number, 15, so any routing entry that informs about destination farther than 15 hops are considered invalid or unreachable. 15 is also the maximum network diameter[14] that can be scanned by a RIP router. The real advantage of the limitation is that the protocol is very robust.

Routing updates are scheduled at regular intervals with a small amount of added perturbation in order to avoid network congestion. Only the best routes are considered to be valid, all "second best" entries are dropped. Each entry has also got an own timer which makes old entries to be invalid, when a certain time threshold is exceeded.

[14] The term diameter is used to refer to the maximum distance that can be achieved on the network in the networking literature. Since it is measured from the current host, the term radius would be more expressive.

RIP uses IP addresses for identification. RIP2, which is an update version of the protocol, allows storing not only host entries, but network entries in the routing table as well. There is also a special tag that enables to distinguish rules learned by RIP and rules learned by other protocols. The full specification of RIP can be found in [R14].

4.3.2 Interior Gateway Routing Protocol

Interior Gateway Routing Protocol (IGRP) is also based on distance-vector routing updates, such as RIP but differs from it in many ways. IGRP stores data in separate structures such as: neighbor table, topology table, feasible successors table and routing table.

Each neighbor has an entry in the neighbor table containing identification and distance information. For distance metric in IGRP a 5-dimensional vector is used.

Topology table contains all possible destinations advertised by neighboring routers. It is also an important difference from RIP that for each destination node a list of neighbors is stored which advertise routes to that destination, thus less appealing routes are also stored. Those advertised routes that have smaller cost values than the one currently used to transport packets are considered to form the set of feasible successors for that target. Whenever a route becomes unusable the set of feasible successors is sought for a new route. If the set is empty, the route to the destination turns into active state, which means that route recalculation is carried out by a special DUAL finite state machine. DUAL guarantees to compute loop-free routes and forms a new set of successor states, as well as updates the routing table.

Though update-packets are sent regularly, routers may initiate an update request to their neighbors. This is typically done when a route becomes active.
Like advanced RIP protocols, IGRP also use tagging to distinguish rules that have been added by the IGRP protocol (internal routes) and rules that have been inserted by another algorithm. More details on IGRP can be found in [R38].

4.3.3 Open Shortest Path First

Open Shortest Path First (OSPF) is a link-state algorithm based routing protocol. As its name prompts, OSPF is an open standard, thus all specifications are available in [R29]. OSPF is designed to work in different levels of hierarchy. Different areas can be defined within an autonomous system that can be connected by an area backbone which is also an OSPF area. Routing information concerning to different levels of the hierarchy are separated from one another in different data structures. The backbone network topology is invisible at lower levels of the hierarchy.

All neighboring routers that have synchronized link-state databases are adjacent routers. Each router periodically sends link-state advertisement on adjacency

relationships to inform other routers on certain part of the topology. As it is usual in link-state algorithms, from all received and feasible advertisement a topology database is built up on each router which is used by a shortest-path algorithm to compute the spanning tree with itself as a root node and determine the best next-hop to each possible destination.

4.3.4 Border Gateway Protocol

Border Gateway Protocol (BGP) is an exterior protocol that is used for routing between autonomous systems. There is a slight implementation difference between interior protocols (such as all previous ones) and exterior protocols. The latter should be capable to advertise lots of network entries, and should be capable to synchronize and keep routing database in consistent state even if the autonomous system has more than one exterior gateway.

BGP runs over a reliable transportation protocol, typically over TCP, which eliminates the need to implement explicit update fragmentation, retransmission, fragmentation and sequencing. BGP may be implemented on computers that do not route at all; any host using BGP is permitted to relay BGP information.

Each BGP router holds an active connection with its peers on which it periodically sends keep-alive messages. If the connection is closed for some reason, all the routes toward the closed connection are also invalidated. No deletion takes place, since it is needed to send information on invalid routes over the network, to let other routers know, which particular links are down. At the beginning of the BGP connection full routing information is exchanged; later only change updates are reported. BGP can aggregate rules in order to allow compact routing tables.

BGP can work in hierarchy as well. The fundamental BGP specification is written in [R39].

4.4 Shortest-path computation

Shortest-path computations are executed in any of the link-state algorithms whenever enough information is gathered about the network topology. Following the notation of *Ahuja at al.* in [R1], network topologies are represented by directed graphs having the following assumptions:

- All metrics are represented by positive integers.
- The graph is connected; that is every node has a directed path to every other node.
- No negative cycles are present[15].

[15] This assumption comes from the nature of the application, as well as the definition of the metric.

- Bidirectional links are treated as double edges in the opposite direction.

Algorithms that determine shortest-path among network nodes are referred to as shortest-path algorithms and can be classified into label setting and label correcting methods. Both use labeling technique to indicate the minimum distance from the source node, as well as the predecessor node, through which the shortest distance is achieved; both algorithms are proved to be convergent to the optimal solution. Distance labels are upper bounds on minimum distances, and are updated in interactive steps. Label-setting algorithms divide the set of nodes into two subsets: temporarily labeled nodes and permanently labeled nodes. As the computation advances, the number of permanently labeled nodes grows until all nodes are labeled as permanent. At this stage the algorithm terminates. Label-correcting algorithms sign all nodes as temporary, and it is possible to modify all the labels at any stage. Label-setting algorithms are simpler than label-correcting algorithms and have smaller computation cost. However, label-correcting algorithms can be applied to a broad range of problems, such as negative cycle-detection, and offer much more flexibility [R1].

4.4.1 Dijkstra's algorithm

One of the simplest and most popular methods of label-setting algorithms is Dijkstra's algorithm which is used for finding the shortest path between a particular source node, and all other nodes, thus forming a spanning tree which has the root in the source. The algorithm initializes labels on the source node to zero distance and void predecessor and any other nodes to infinitive distance and unknown predecessor. A neighbor node is then selected with the lowest label value as an initial step. Labels are denoted by $\langle predecessor\_node, distance\_from\_source \rangle$ tuples. The selected node is examined by an optimality criterion, whether $d_j > d_i + c_{ij}$ is true, where d_i is the distance from the source to node *i* (the source-source distance is labeled as 0), d_j is the distance from the source to node *j*, and c_{ij} denotes the distance between nodes *i* and *j*. If the label on node *j* is larger than the new path which is found via node *i*, the distance on *j* is updated to $d_j = d_i + c_{ij}$ and the predecessor of node *j* is marked to *i*. If the condition is false, then no action is performed. Figure 4.4 gives a formal algorithmic description where $pred_i$ denotes the predecessor node of *i* in the shortest path. An illustrative example is shown in Figure 4.5.

Property 4.1: *The computational cost of Dijkstra's algorithm is proportional to* $O(n^2)$

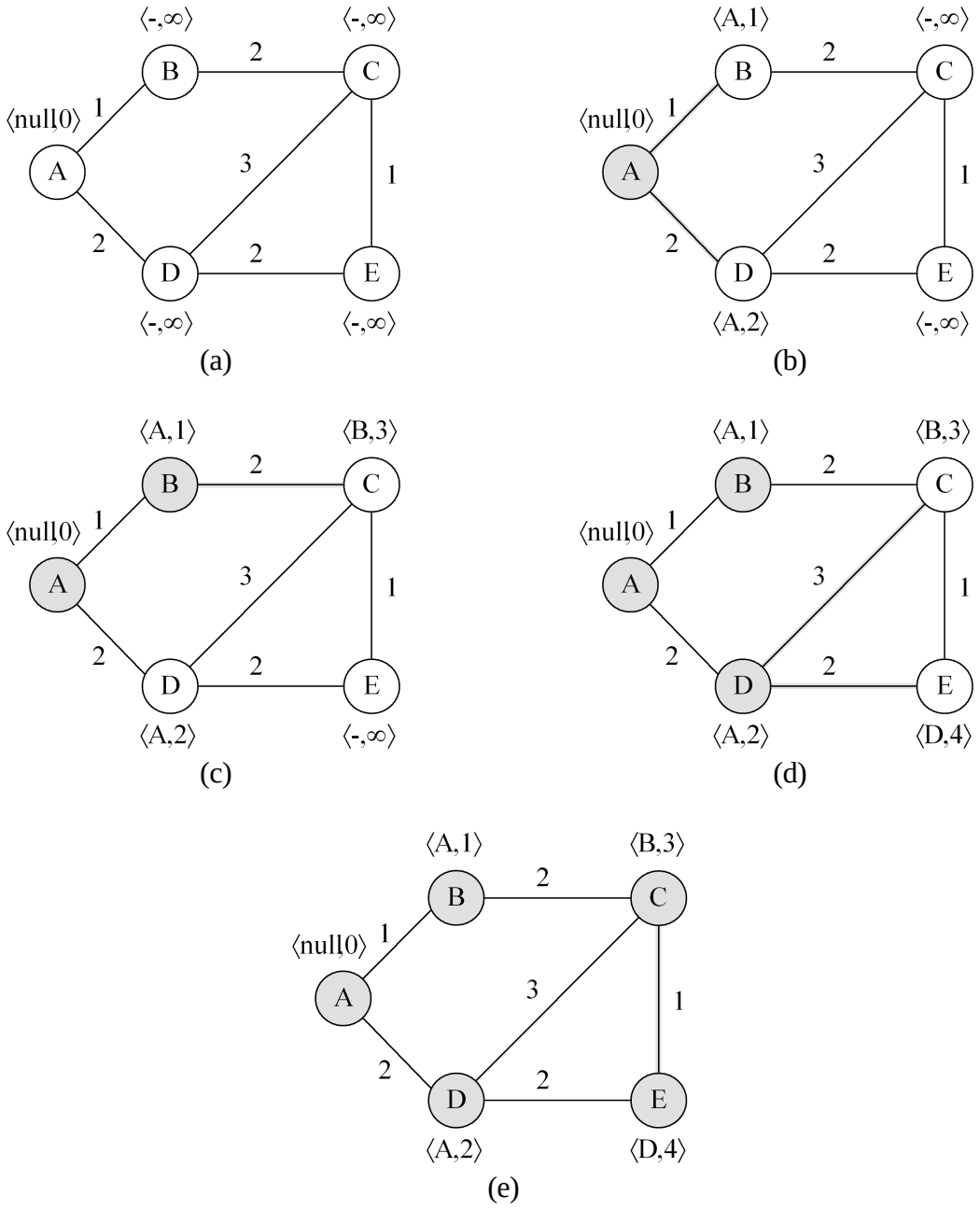


Figure 4.5

Example to how Diskstra's algorithm works. Nodes belong to set S are denoted by grey. The two final steps are abbreviated in a single pane, pane (e)

```
input: N set of nodes, E set of edges, s source node
input: A(i) the nodes available from node i, c[i,j] distances
output: labels on all nodes, pred[i] vector
initialize set S:=ø; S':=N; d(s):=0; d(i)=8; for all other nodes;
initialize pred(s):=null;
function Dijkstra(N,E,c)
begin
  while |S|<n do do
    let node i∈S' be the node with the smallest label value
    S:=S+{i}; S'=S'-{i};
    for each (i,j) ∈A(i) do
      if d(j)>d(i)+c(i,j) then d(j)=d(i)+c(i,j);
                               pred(j):=i;
      end
    end
  end
end
```

Figure 4.4
Formal description of Dijkstra's algorithm

4.4.2 Floyd-Warshall algorithm

Floyd-Warshall Algorithm (FWA) belongs to the class of label-correcting algorithms. It solves the all-pairs shortest-path problem, i.e. it determines shortest path from every node to every other node. As general label-correcting algorithms FWA scans through the set of edges to find nodes for which the optimality equation discussed at Dijkstra's algorithm is hurt. The basic idea of the algorithm is the assumption that all the distance labels between two arbitrary nodes are either updated in every time step or left untouched depending on the existence of any intermediate node, that makes a shortcut. Let d_{ij}^{k} denote the distance between nodes *i* and *j* provided that *k*-1 intermittent nodes are involved in the path. FWA initializes d_{ij}^{1} with distances c_{ij}, if link exist between *i* and *j*, or 8 if it does not.

The value of d_{ij}^{k+1} is computed using the following recursive rule

$$d_{ij}^{k+1} = \min\left\{d_{ij}^{k}, d_{ik}^{k} + d_{kj}^{k}\right\} \tag{4.1}$$

where *k* goes from 0 to *n*+1. The rule is valid since the shortest path that uses nodes 1, 2, …, *k* as internal nodes either miss to go through node k ($d_{ij}^{k+1} = d_{ij}^{k}$), or does pass through node *k* ($d_{ij}^{k+1} = d_{ik}^{k} + d_{kj}^{k}$) [R1].

Figure 4.6 shows Floyd-Warshall algorithm. In the case of all-pairs shortest-path problem precedence relations among nodes in any path is stored in the $pred_{ij}$ matrix.

From this matrix the shortest-path between all sources and destinations can be determined by pointing on the destination node, and traverse in backward order. It is shown in [R1] that:

Property 4.2: *The FWA computes the shortest distance between all nodes at the cost of* $O(n^3)$.

```
input: N set of nodes, E set of edges
input: c(i,j) the known one-step distances between nodes
output: labels on all nodes, pred(i,j) matrix
initialize for all i and j d(i,j):=8; pred(i,j):=null;
initialize for all nodes i d(i,i):=0;
initialize for all edges d[i,j]:=c(i,j);
function Floyd_Warshall(N,E,c)
begin
  for k:=1 to n do
    for each edge (i,j)∈E do
      if d(i,j)>d(i,k)+d(k,j) then
        d(i,j):=d(i,k)+d(k,j);
        pred(i,j):=d(i,k)+d(k,j);
      end
    end
  end
end
```

Figure 4.6
Floyd-Warshall shortest-path algorithm

4.5 Reinforcement learning based routing

Although current dynamic routing algorithms are wide-spread, and robust enough, they have some shortcomings to deal with as it is pointed out by *Littmann* and *Boyan* in [R6].

Both distance-vector and link-state algorithms implement a naive approach: the routers exchange reliable information over the network layer with one an other, thus, all information must be confident and consistent. Especially link-state algorithms are sensitive to inconsistencies, since all routers maintain the model of the whole topology, and if, for some reason, different routers process the same link-state information differently, it may lead to unavailable network portions or loops. The problem would disappear if there was a global observer which could keep the topology maps in consistent state and tell the packet the optimal path all the time. However, in networking technology there is no global observer, and decisions should be made on the basis of local information. Global information (or parts of global information) is produced by periodical or if-needed information exchange at the price of a definite communication cost. Consider a distance-vector algorithm: for small networks the routing tables are

relatively small, so periodical exchange is feasible at sufficiently low communication cost, but in the case of large networks exchanging full routing tables may consume considerable amount of time.

Global information can be obtained in smaller chunks as well, forming the concept of a new routing model called preference-based or agent-based routing. The basic idea of preference-based routing is similar to distance-vector algorithms: all routers use the estimation of their neighbors on how far the destination is plus the direct distance to the router. The difference is, that instead of transportation-layer information exchange, the estimated delivery times are determined by trial and error probes, thus each router has a realistic, on-line estimate on the delivery times, which also gives the possibility of adapting to topology changes. Each node maintains a preference table, which is called Q-table, assigning estimated delivery times to each destination-next hop pair. The estimates are updated each time the router sends a packet to one of its neighbors. As the node routes packets, preference values gradually reflect more and more accurate model of the global network topology.

Preference-based routing is formed on the basis of reinforcement learning. Each router is treated as an intelligent agent that makes routing decisions on the basis of estimated delivery times and gathers reinforcement signal proportional to the quality of routing[16] from neighbors or from possible destinations each time a packet is sent.

To formulate the routing problem as a reinforcement learning problem, first the states, actions, and reinforcement functions are to be defined. In the case of routing states are identified simply as destination nodes [R6]. Note that the real state identifier is a triplet: $\langle source, destination, packet\_identifier \rangle$.

Property 4.3: *The state of a router agent can be uniquely identified by the source, the destination nodes and the packet identifier.*

In each delivery step the router may decide which neighbor to send the packet to. It is reasonable to define selected next-hop neighbors as actions. When the packet arrives at the neighbor, it immediately sends two pieces of information back to the sender: the neighbor's estimation on the delivery of the packet to the destination and implicitly a reward signal. When the packet is sent, a timer is started, which stops when the acknowledgement is received from the neighbor. Various functions of the time difference are then used as a reinforcement feedback. The estimates are updated by the following rule:

$$Q_{k+1}^{s}(d,a) = (1-\alpha)Q_k^s + \alpha\left[f(t) + \gamma \min_{a'} Q_k^{s'}(d,a') \right]. \tag{4.2}$$

[16] The quality of routing may refer to constraints on different additive or non-additive metrics as well as to the confidentiality of data delivery.

Here $Q_k^s(d,a)$ denotes the actual node's estimate on the delivery time to destination d in time step k, when selecting neighbor a. Learning-rate is denoted by α, discount rate is by γ, and the delivery time estimation of neighbor a by $Q_k^{s'}(d,a')$ where a' the next hop after the next hop. Note that instead of maximization, minimization is used due to the interpretation of the Q-values: estimated delivery times; the shorter they are, the better the delivery is.

Although both denote the same thing and for the environment dynamics, $P_{ss'}^a = P_s^a$, there is a semantic difference between a and s': while the former means "*send this packet out on a certain interface*", the latter means *"the packet has arrived at the neighbor, and it has sent an acknowledgement"*. From the routing point of view there is no reason to distinguish between the two terms, since local network interface problems are as wrong as remote machine breakdown with respect to the packet flow.
Note that the preference values are distributed among all routers. Also note that if all tables from all nodes would be put together the resulting table, which would specify states as $\langle source, destination \rangle$ tuples, will be a large preference table of the whole network, and slightly modified update rule of equation 4.2 could be used to operate on it. Following the work of *Watkins* and *Dayan* [R59] it is easy to see that such Q-values converge to the optimal estimated data delivery times under certain conditions. In the routing case, this large Q-table is stored in smaller chunks on the different nodes, and joining them is carried out by network communication. Since, the distributed nature has no effect to the convergence, the following property is true:

Property 4.4: *The distributed Q-learning algorithm that uses equation 4.2 as update rule retains convergence to the optimal Q-values.*

Q-routing algorithms can be boosted up by dual Q-routing, when not only the next-hop node sends estimation about the delivery time to the destination node, but the source itself sends an update to the next-hop to adjust delivery times from the source, as it is a potential destination to other packets. The next-hop node forwards this information along the path. The method is also called backward exploration and details can be found in [R21].

There are many appealing properties of distributed preference-based routing, but several questions are still open: Which neighbor to choose at the initial steps? How cycles are detected and avoided? How good news and bad news traverse along the network? The basic model does not deal with these questions, so an extended version is proposed in the next section.

4.6 Proposed extensions

In this section we give some proposal to the original Q-routing algorithm worked out by *Littmann* and *Boyan*, and we give the framework how Q-routing algorithm can be integrated with Boltzmann-annealing to remedy most of its original shortcomings. The new routing algorithm is referred to as extended Q-routing.

If a node is considered as an intelligent agent and it enters the system, it has no knowledge about the network topology. In this scenario, the expected behavior of the agent is to make some explorative steps around the neighbors. This can be accomplished by selective flooding, where "*selective*" refers to be selective in time. When the agent has gathered information enough, or has no more time to explore, it is expected to start using one of its discovered routes to transport packets to the desired destination. The proposed method how the agent focuses on the best route is using Boltzmann exploration, which either gives equal chance to all neighbors or it is discriminative i.e. use only the best neighbors to a given destination depending on a single control parameter named temperature. For each destination an individual temperature value is assigned that controls the "sureness" of the data delivery toward a particular node. Separate temperatures allow distinguishing between destinations, so there may be destinations to which firm routes are used, and destinations toward the delivery is under exploration.

Router table initialization can be boosted up by initial communication with the neighbors. Whenever a router is new to the network, it first sends "hello" messages to the other routers which in turn send the list of possible destination nodes back. The new router is then registers its neighbors, initializes its tables to zero and sets temperature vector to some initial, non-zero value. The router agent then makes exploration during a given time period using dynamically adjusted temperature bounds following a predefined simulated annealing schedule. When the exploration is over, the temperature values corresponding to each destination go below the minimal value, and the router is ready to use the best route toward that destination. When this point has been reached, the routing procedure is identical to the ordinary Q-routing algorithm.

Whenever a router or a link breaks down, all neighbors begin to use the second best route bypassing the damaged part of the network. Alternative paths may be simple bypasses, but can be totally different routes depending on the network structure, or the speed of the links. However, there is no guarantee to appropriate path recovery, i.e. when the router or link comes up again after failure the traffic should be adjusted back to the original, and presumably, better route. The reason why this may occur is that the Q-values are updated only along a used (or partially used) path, but if the recovered path is just bypassed by another route, no packets are transmitted along it, thus no Q-value update happens. This problem is pointed out by *Boyan* and *Littmann* [R6] as well as by *Choi et al.* [R9], and is also referred to as the *"path-hysteresis"* problem. The solution in the Boltzmann-annealing model is straightforward: Temperature values of certain routers are to be re-adjusted to their theoretical maximal value, and a re-

annealing process should follow, but only for a single temperature value. This process can be initiated by a special "raise temperature" packet sourced by the neighbors of the damaged router (or the previously damaged, now recovered router itself), or endpoints of a damaged link, when they detect that the failure is over. The "temperature up" signal is flooded throughout the network, as long as its time-to-live (TTL) value is exceeded, or a router finds that the sender is neither a neighbor nor a destination entry. The temperature raising signal for a given destination is always dropped when any annealing procedure is being done for that destination.

Temperature can be raised up as a consequence of the routing agent's own decision as well. When the agent experiences, that for a particular destination the data delivery has significantly increased, it may raise its own temperature up, and may carry out an exploration cycle. This requires the storage of not only the expected values of delivery times, but their standard deviation from the mean value as well.

Cycle detection along the path, especially in the exploration phase, is also a key issue. Ordinary Q-routing does not involve any explicit cycle detection. As it is shown in [R6], the cycles are punished in the regular way, and for small networks the learning system will surely find a cycle-free path superior to any other paths involving a cycle. However, in large networks cycle-free path learning cannot be guaranteed. An efficient cycle detection algorithm is, thus, needed: each router can detect a cycle if the package, that has already traversed it, has some clues there. It is satisfactory if the header of each packet is stored in the memory as long as an acknowledgement packet arrives or the memory is short of free space.

Acknowledgements are also important parts of the successful data delivery and the learning process. There are basically two kinds of acknowledgements which actually aid cycle detection: immediate reward from the neighbor and remote feedback from the destination (via the same neighbor). The first kind of acknowledgements used in the original Q-routing model is applied to give an immediate estimate on the expected delivery time. On the other hand remote reward gives a real estimate on the total packet delivery time. If both types of rewards coexist in the system, equation 4.2 varies as:

$$Q_{k+1}^{s}(d,a) = (1-2a)Q_k^s + a\left[f'(t) + f''(t) + g(l) + g \min_{a'} Q_k^{s'}(d,a') \right]. \qquad (4.3)$$

The immediate rewards are denoted by $f'(t)$, the indirect reward is denoted by $f''(t)$ which are both the functions of measured times, i.e. the time between sending the packet from the source and receiving it at the destination, as well as the time sending the packet from the source and receiving it by the immediate neighbor. Function $g(l)$ is the punishment function when a packet is in a loop. In the simplest case it is inversely proportional to the length of the loop.

Action-state values can be normal distribution probability variables as well, as *Gullapalli* shows in [R12]. In this case each action-state value is represented by a mean,

and a standard deviation. Equation of the mean value is symmetrical to equation 4.3, while that of the standard deviation for all Q-table entries can be written as follows:

$$s^{s}_{k+1}(d,a) = (1-b)s^{s}_{k} + b\left[h'(t) + h''(t)\right]. \tag{4.4}$$

4.7 Implementation, experimental results

4.7.1 General principles

Based on the extension proposal principles in section 4.6, we developed a distributed extended Q-routing network topology simulator in Java. The simulator imitates IP-layer transportation through UDP packets. The reason for this design principle is that in extreme case the whole simulator may run on a single computer. Each router is implemented as an individual UNIX process that has multiple Java threads. Each router connects to its neighbors via UDP socket connection. Since the topology is simulated, there is a dispatcher agent which maintains the network topology, and tells each router the appropriate neighbor-mapping. This agent has no function in a real environment, since delivery times as well as topology are real; there is no reason to simulate them. Figure 4.7 shows the sketch of the routing simulator. The dispatcher agent forms a central access point to the simulated topology, and enables to gather statistics and provides some visualization tools as well.

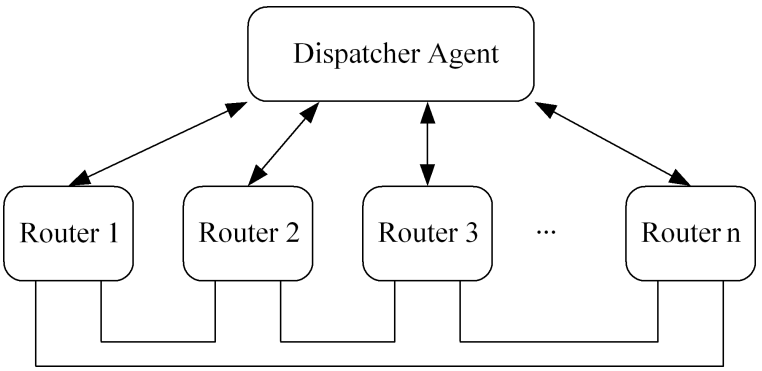


Figure 4.7
The scheme of reinforcement learning routing simulator

Delivered packets are simulated by objects. There are five basic packet types: hello, data, acknowledgement, raise temperature, and “send me Q”.

Hello packets are similar to those used in the classical dynamic routing protocols, with the only difference that they are forwarded throughout the network.

When an agent receives a hello packet, it registers the sender into its Q-table and initializes the corresponding preference values to 0. If it finds that an entry for that host already exists, the packet is simply ignored.

Data packets are also referred to as routed packets. Whenever a data packet arrives at the agent and the agent is not the destination, a decision is made where to forward the packet with respect to the following algorithm: The node where the packet has come from is excluded from the selection possibilities; then a probability distribution is calculated with respect to the preferences and the actual temperature value; a random action is generated out of the probability distribution. Note that when the temperature is sufficiently small, the only action that can be selected with almost probability of 1 is the best Q-valued action. Also note that the preferences are in reverse order, i.e. the smallest the Q-value is, the better the action.

Relevant header information of all packets is stored in an associative array for future processing. If the packet entry already exists in the array, the packet probably has already been passed through the agent, thus a loop emerges in the delivery path. The loop detection method is detailed in subsection 4.7.3.

As it was stated before, remote acknowledgement packets are sent whenever a data packet arrives at its destination node. An acknowledgement packet traverses back to the source on the same path as the data packet arrived. It holds two pieces of information: the Q-value of the next hop[17] is inserted into the packet as in the case of ordinary Q-routing; the reward is computed from the difference between the time stamp of the data packet header stored and the time stamp of the acknowledgement. If an acknowledgement is received without corresponding stored data header, the acknowledgement is dropped, and warning message is send to the dispatcher agent. There is also an immediate acknowledgement and feedback which arrives whenever the packet reaches the next hop router. When all reward-like information is gathered, Q-values are backed up, and the acknowledgement is forwarded backwards to the source. If the source node is reached the acknowledgement is dropped after backup.

Raise temperature packets are used to explicitly control the temperature value corresponding to a given destination. Whenever an agent receives this kind of packet, it calculates temperature bounds and sets the actual temperature to its maximal value corresponding to the destination from which the packet has come.

Before a stored data packet header is dropped due to timer expiration, the agent sends “send me Q” packet to the neighbor which the data packet is forwarded to. The neighbor then replies with a “send me Q acknowledgement” holding its estimated packet delivery time to the destination of the dropping packet. This results in immediate reinforcement.

The exact specification of the used packet formats for Internet Protocol version 4 (IPv4) can be found in Appendix C.

[17] Next hop from the point of view of the data packet.

4.7.2 Time to live fields

There are two kinds of time to live (TTL) fields: packet-wise, and queue-wise, which are independent from each other. Packet-wise TTL field is a counter which counts through how many routers the packet has already traversed so far. If the value exceeds some threshold, the packet is dropped in order to avoid network congestion. Queue-wise TTL field corresponds to packet header storage on routers. Each queued header has got a TTL field which is proportional to the amount of time the header has spent on staying in the queue. Since there is no reason to store header information forever, having exceeded some time value, the header is dropped. In case of an acknowledgement that regards to an entry that has already been dropped arrival no learning takes place. Queue-wise TTL is also proportional to the amount of memory available to store header information. In the physical implementation, a special cleanup process is used to scan for free memory, and if the queued header information grows beyond a threshold value, old entries, i.e. entries that have old queue-wise TTL values are expunged from the queue. Under normal load conditions queue-wise TTL values decrease slower than packet-wise TTL values, so each acknowledgement will probably find a corresponding entry in the stored header values. Under heavy load conditions queue-wise TTL values may decrease quicker, and acknowledgement will not find valid entries in the header queue.

4.7.3 Loop detection

Although any preference routing algorithm could survive without loop-detection, the efficiency can be seriously improved by using it. Since negative path length are excluded from the system, there is no reason to enter any packet into a delivery loop, since if there exists two paths between two arbitrary nodes, one with and the other without a loop, the loop-free path will always be shorter in terms of any non-negative metric, than the looped one and it receives better reinforcement. In the extended Q-learning model, whenever a loop is detected in the delivery path, it is wholly unfolded, and only the loop-section is punished in order to let other useful pieces of the path to be selected it in the future.

Two-hop bops have already been excluded, since the previous hop is always removed from the possible next-hop selection palette. Larger loops are, however, more difficult to detect. The basic principle is to use a special data field, called “looped bit” to indicate whether the packet is outside, or inside a loop. The looped bit is set by an agent that first detects, that the packet has already been there without acknowledgement. Suppose a simple loop topology, such as in Figure 4.8. Suppose further, that a packet traverses nodes in the following sequence: EABCDA. In this case node A is the first node that sets the looped bit, since it is the first node which receives the packet twice. Every other node can detect that if the looped bit of the received packet has set or not. If

it is set and the original sender of the packet, i.e. the one maintained in the header queue, and the currently chosen next hop is the same, then the packet is backtracking in a loop, and is forwarded back to the original sender with looped bit set, along with a large artificial delivery time value which serves as punishment. If the chosen next hop is different, than the packet exits the loop, and the looped bit is cleaned. When the direction of the packet is reversed at the first time, node A also indicates the fact of the loop by setting the looped bit in the packet header, and when the backtracking procedure reaches node A again, it cleans the looped bit and forces the packet to be sent to any directions other than node B (i.e. the packet cannot be forwarded to any of its senders, neither to node E which originally sent the packet, nor to node B who sent the backtracked packet with looped bit cleaned). Whenever a packet is received with looped bit set, the punishment update is applied to the original next hop, since the actual agent knows that the next hop contributed to the loop. During the backtracking process the header information is removed from the header queue to keep the system clean.

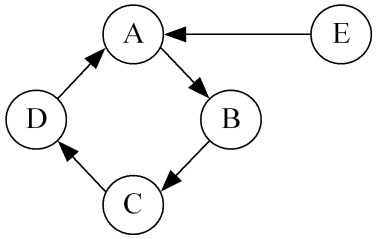


Figure 4.8
Simple loop in a delivery path

4.7.4 Experimental results

The routing algorithm was tested on different network topologies, with node number up to 100. The topology information was loaded artificially through the dispatcher agent. Results were monitored via the logs of the dispatcher. All networks were trained to the all-pairs shortest path problem, i.e. all nodes were sender and receiver at the same time, and the shortest path is sought among all nodes. Test parameters of the algorithm are summarized in Table 4.1.

Studying the output of the test runs the random behavior can be stated by packet losses, dropped header queue information and selection of non-optimal paths. This is the exploration stage when the agents discover the topology. At later stage, when the agents have reasonable estimate about the network topology, there are less packet losses, but non-optimal routes are also selected. As the temperature value decreases, the agents more and more often select the optimal path, and when the minimal temperature value is reached, only the best path is chosen. A test topology (the backbone of the Hungarian Academic Network) is shown in Figure 4.9.

Parameter	Value
Learning rate	0.1
Discount rate	0.9
Annealing schedule	linear annealing with variable temperature bounds
Exploration interval	proportional to nodes × links

Table 4.1
Most important parameter settings of Q-routing algorithm with Boltzmann exploration

The shortest path results were validated by using Dijkstra's algorithm. In most of the cases the modified Q-routing algorithm found the theoretical optimum even under large load conditions.

Figure 4.10 illustrates the delivery time between two towns (which are indicated by asterisks in Figure 4.9) as the function of time steps. It is easy to see, that at the beginning and throughout the exploration cycle, the path lengths are of variable size, the fluctuation is large. As the exploration cycle is over, only the best path is used. Since there may be stochastic delays in the delivery chain, the delivery time is allowed to show small variance in the exploitation stage as well.

Link and router breakdown tests were performed manually. The algorithm was capable to detect changes and diverted the traffic onto an alternative path (see the curves in the middle of Figures 4.11b and 4.12b). At the beginning, the usual exploration shows that the length of delivery heavily fluctuates. Then a stable, bypass route is chosen. When the link comes up again, all nodes perform similar annealing cycles for the damaged router, or routers on both side of a damaged link as in Figure 4.10, and finds the best path to be the original one, thus fulfilling a complete path recovery. Figure 4.12 illustrates the same behavior, but for two pairs of nodes.

The scheme of the router agent algorithm can be seen in Figure 4.13.

4.8 Implementation proposal

4.8.1 IP-layer implementation

Though the simulator is implemented using UDP, all the applied packet format extensions can be implemented at network layer level. The specification of IP-header allows inserting extra options into an ordinary IP-header up to 60 bytes, and since all options required by Boltzmann exploration based Q-routing are within this limitation, it is reasonable to specify IP-packet format extensions as IP options.

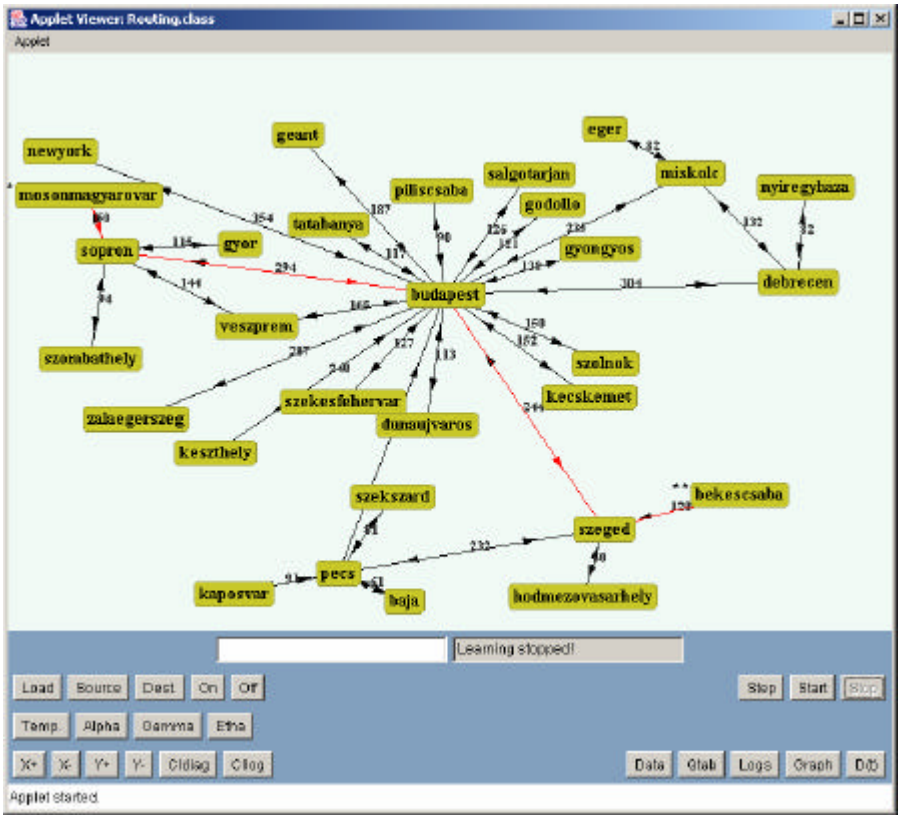


Figure 4.9

The backbone topology of the Hungarian Academic Network. Nodes of the graph indicate Hungarian regional centers; edges denote “delay distances” among them

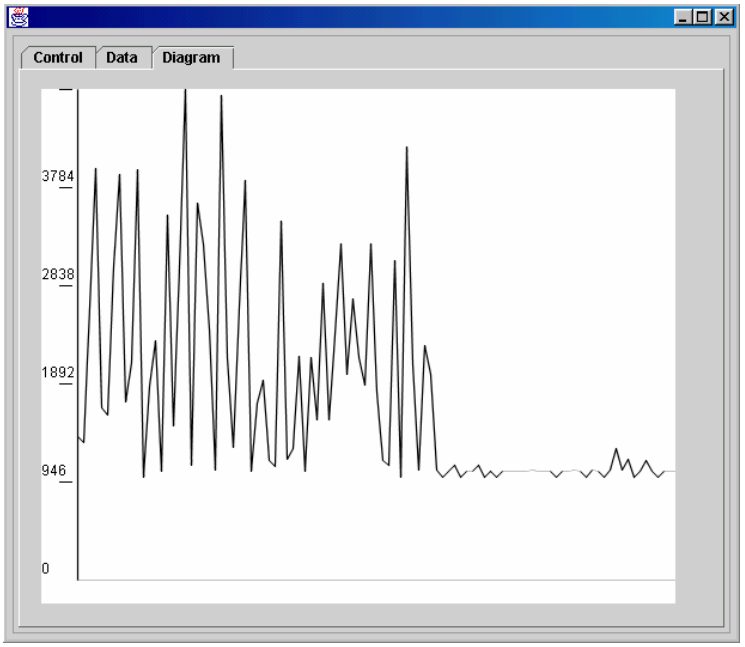
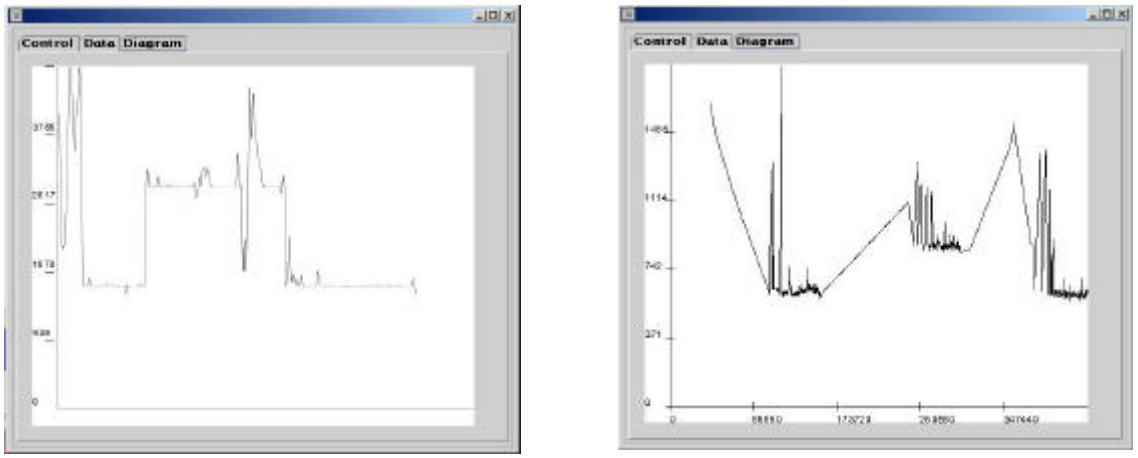


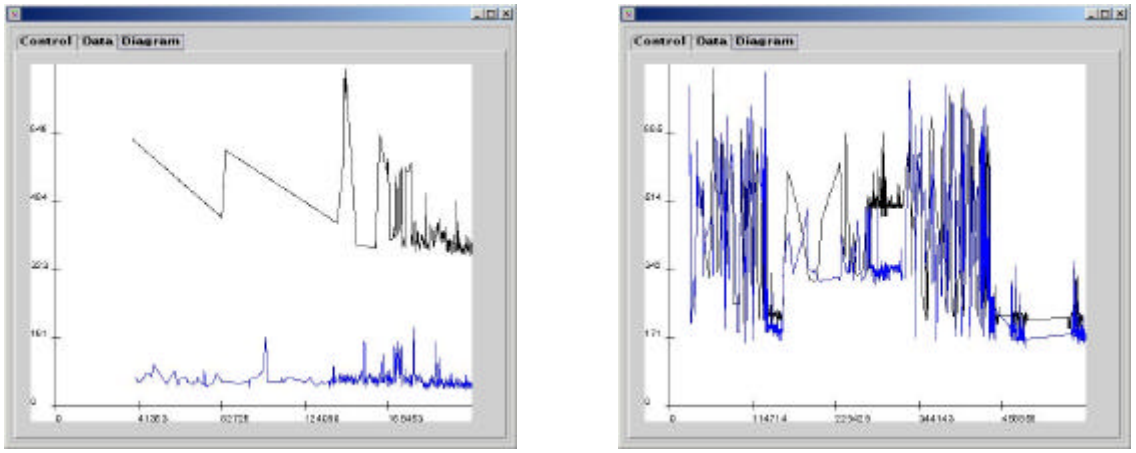
Figure 4.10

Delivery times between two towns indicated by red line in Figure 4.9 vs. received acknowledgement time steps diagram



(a) (b)

Figure 4.11
Re-annealing schedules between a source-destination pair



(a) (b)

Figure 4.12
Re-annealing schedules between two source-destination pairs

```
function routing_agent()
begin
  initialize internal data structures;
  thread cleanup
    if free memory falls below a threshold value then
      clean up headers with old TTL value;
    end
  end
  thread timer
    start/stop timers on event;
  end
  thread annealing
    if Tmax, Tmin and Tactual are set then
      start annealing using any of the annealing functions;
    end
  end
  thread routing_loop
    if packet arrives then
      if destination reached then
        send acknowledgement back to the source;
        process packet;
      else
        case packet type in
          data: select outgoing interface using Boltzmann-eq.;
                switch the packet;
                send immediate Q-estimate to the previous
                  node;
          acknowledgement: calculate parameters;
                           use backup rules to update Q-
                             values;
                           forward the acknowledgement;
                           if source node is reached then
                             sink acknowledgement;
                           end
          looped: unfold loop;
          hello: register the router;
                 forward hello packet;
          raisetemp: raise the temperature corresponding to
                       the node that originated the packet;
                     forward raisetemp if needed;
          smq: ask the neighbor for Q-values corresponding
                 to a specified packet header;
          sqm-ack: update Q-values with values received;
        end
      end
    end
  end
end
```

Figure 4.13
The schematic algorithm of Boltzmann exploration based extended Q-routing

Any IP option can be specified by the following scenario:

- Each option has a unique code which indicates the control of that option. There are specific codes reserved for specific purposes, but there are free slots for individual use. The option code is the first field.
- Each option code is followed by a length field which indicates the total length of that option as it comes from *Portel's* IP specification [R35].
- The last field is the value of the option which can be of different length depending on the previous field.

Appendix C also details the IP version 4 packet formats for the extended model. All the extensions are designed to be compatible with all the standard IP specifications [R35].

The vast majority of protocol proposals are put into the data packet due to the loop detection algorithm. Each packet type, such as data, hello, acknowledgement, raise temperature, send me Q, and send me Q acknowledgement has a common field called control code which involves all control information related to Boltzmann exploration based Q-learning. In most of the packet types there is no need for any extra option other than the control code.

Reverse path packets such as any type of acknowledgement packet or raise temperature packets are based on Internet Control Message Protocol (ICMP) specified in [R36]. ICMP packets consist of header information only without any data part, and are used to transfer control information between any pairs of nodes. Note that in the case of large amount of data, the packets are fragmented, there is only a single reverse path packet corresponding to the first packet in a fragmented data set needed, which largely reduces communication overhead. IP layer acknowledgement may be combined with transportation layer acknowledgement in case of reliable transportation protocol such as TCP, thus, further reducing the number of extra packets.

Q-values in all types of acknowledgement packets are stored as 4-byte floating point numbers.

4.8.2 UDP implementation

The proposed model can also be implemented by using higher level protocols, in the same way as ordinary dynamic routing protocols are implemented. In this case there are two separate routing tables, the ordinary routing table, and a table which is used for routing the exploration packets. In the ordinary table, stable rules, which have been the result of an exploration cycle, are inserted only. The other table may contain temporary rules, and all routing protocol packets are transferred via this table. IP-route version 2 and Linux Netfilter modules are capable to maintain different routing tables as well as pre-routing chains to let different packets be routed by the different tables [R62]. All the proposed option-level IP-header extensions (see Appendix C) can be used to build up the Boltzmann exploration based routing protocol.

4.9 Discussion

In the chapter we studied the most important functionalities of packet routing. Different dynamic routing protocols, such as RIP, OSPF or IGRP have been surveyed. All these protocols use transportation level reliable information exchange to share routing table data with one another.

There is a new concept, named Q-routing, which uses network layer information exchange through trial and error probes and learns routing information from experience. This concept is proved to give better flexibility and robustness, than general shortest path algorithms, but suffers from path recovery and loop detection problems.

We proposed an extended Q-routing model based on Boltzmann-exploration to overcome both problems. Simulated test results have shown that the reward-punishment procedure can be successfully applied to detect and to punish looping parts of any path while retaining the non-looping parts be competitive. Environment changes are detected either by protocol included warning packets, or by the routing agent's self recognition.

Chapter 5

Flow-shop Scheduling in Virtual Manufacturing Environment

In the first part of the chapter a brief survey on manufacturing disciplines will be given, with special emphasis on the virtual manufacturing concept. Then job scheduling issues are examined where special attention is paid to the flow-shop scheduling problem.

In the second half of the chapter we show the design and the implementation of combined reinforcement learning and Boltzmann-annealing based scheduling algorithm which uses the framework of the virtual manufacturing concept and provides dynamic scheduling capabilities as well. The algorithm aims at solving the simple *m*-machine, *n*-job flow-shop scheduling task on-line. The results shown in this chapter are based on our papers [P1] and [P2].

5.1 Introduction

The term manufacturing refers to the process of making ready products from different input resources, such as processing equipment, material, energy and information in finite time steps. In a broader sense the term manufacturing is defined as "*all the activities and processes from order receiving to delivering customer goods*" [R17]. Manufacturing processes are accomplished by manufacturing systems (MS) which, following the definition of *Tóth* in [R52], are the structured set of humans, machinery and equipment bounded to a material and information flow. MS is "*a complex technological object composed of machining, material handling, tooling and controlling sub-systems*".

Intelligent manufacturing systems (IMS) introduced by *Hatvany* and *Nemes* in [R13] aim at integrating the fields of artificial intelligence (AI) and manufacturing systems in the sense that the resulting intelligent manufacturing systems are expected to solve tasks, or sub-tasks in unexpected or unforeseen environmental conditions with certain limitations. These conditions may mean changing market demands, late deliveries of suppliers, failed operations, machine break-downs, etc. The key benefit of IMS is that the internal structure of the manufacturing process is capable to exploit and feed back experience on a particular manufacturing process in order to let the system improve the execution of the same task in the future.

A step toward contemporary manufacturing research is the recognition that computer systems and information technology (IT) can provide firm infrastructure as well as excellent computational capabilities to create simulation on manufacturing process models, which make the system more predictive. The integration of IMS and IT brought the disciplinary field of virtual enterprises (VE) and virtual manufacturing systems (VMS). *Tóth* in [R52] gives a more comprehensive view of the terms as well as the history of manufacturing from direct numerical control (DNC) systems to up-to-date computer integrated manufacturing (CIM) approaches.

5.2 The concept of virtual manufacturing, distributed models

In a nutshell, all virtual manufacturing concepts are about to create computer-based models of real manufacturing systems, accomplish performance improvement calculations on these models utilizing the fact that a simulation step can be carried out in the fraction of time of a real manufacturing operation, and feed the results of the computation back to the real manufacturing system. Figure 5.1 gives an example on the VM concept on the flexible manufacturing cell's (FMC) level. The manufacturing system can be divided into four parts: real production system (RPS), real information system (RIS), virtual production system (VPS), and virtual information system (VIS) [R17]. The real part (RIS and RPS), on one hand, represents the real system which consists of all the machines, workstations, shop-floor network, control system and monitoring tools that make up the manufacturing environment in the cell. On the other hand, the virtual parts represent the computational model of the above mentioned system elements which take the form of objects[18] on a large-performance computer, or the network of computers. Note that the mapping between the real part and the virtual part is bijective, however, the virtual part may be structured in a totally different way than the real system. As *Monostori* and *Kádár* points out the virtual system may work using distributed manufacturing concepts, such as heterarchical control, while the real system uses hierarchical control structure [P7][R5][R17][R27].

Holonic manufacturing (HM), or as it is most frequently referred to, agent-based manufacturing aims at modeling the manufacturing system as network of individual decision makers, so-called agents. The agents represent either manufacturing resources, such as cells, machines, parts, or manufacturing functions, hardware or software entities; they try to reach a common goal while they also pursuit their own goals. *Márkus* and *Váncza* define heterarchical manufacturing systems as *"transformation of manufacturing organizations"* to *"network-like, reconfigurable federations where production is carried out by more or less autonomous and cooperative production units*". The communication among the agents is accomplished in two fundamental ways: either using blackboard communication, or using message passing protocol.

[18] The term object refers to data structures and algorithms.

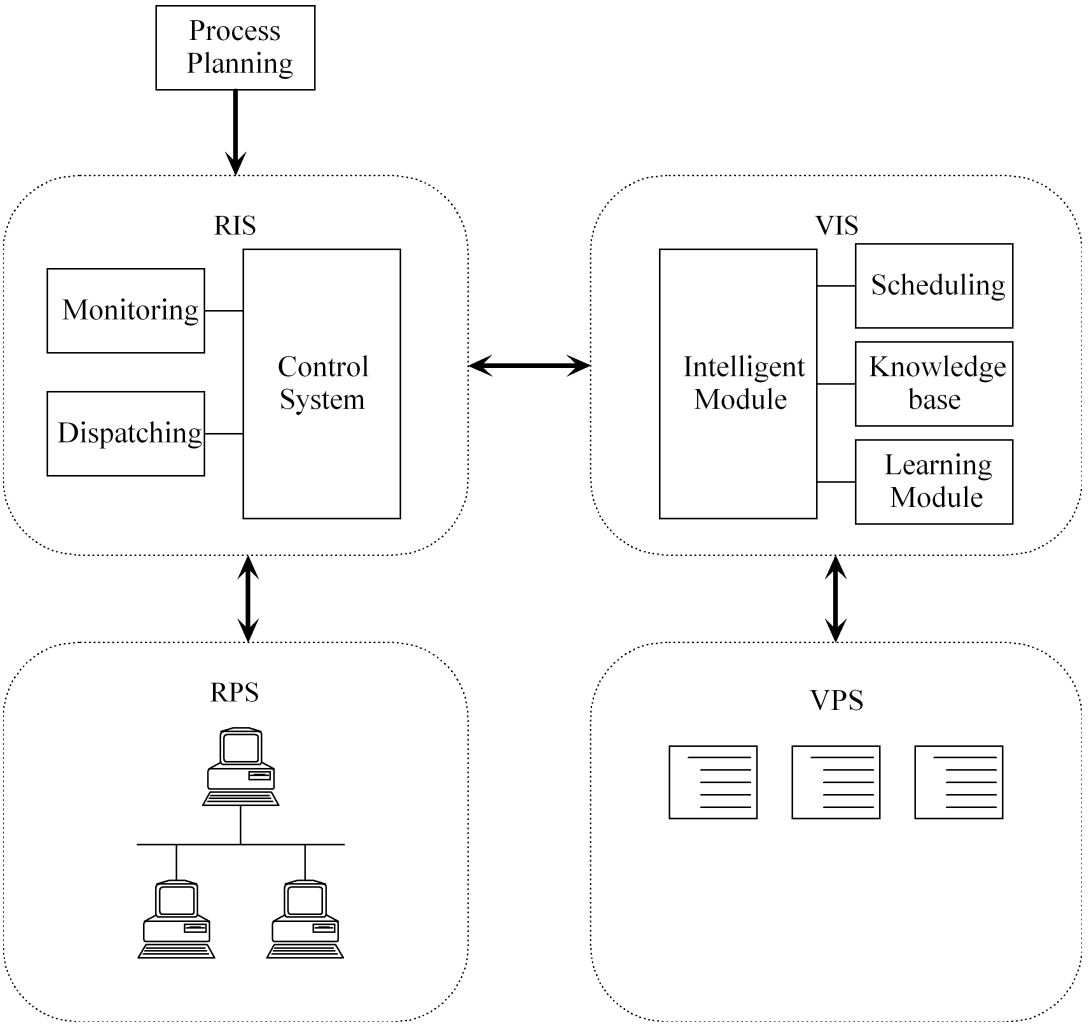


Figure 5.1
The concept of virtual manufacturing on the cell level

Blackboard communication is a kind of shared memory communication where all the agents access a shared storage resource, e.g. a shared memory segment, and they synchronously and consistently read and write that area. Message passing is based on network communication when the agents exchange information packets via packets over the network layer (e.g. the IP layer). Message passing can be used on the enterprise networks, that connect different elements within the enterprise, and shop-floor networks that interconnect manufacturing elements, such as machines, workstations.

5.3 Job scheduling

In all manufacturing processes, the schedule of different jobs on different resources is of fundamental importance. Following *Baker* [R2] scheduling is the process of planning and applying optimal job allocation or assignment to the different resources.

Typical scheduling tasks can be classified by using $\langle a \mid b \mid g \rangle$ triples where a denotes the machining environment, such as the number or type of machines, ß denotes the job characteristics, like different relations among the jobs, and ? denotes the optimality criteria. *Vízvári* gives a thorough survey on the typical scheduling tasks and their classifications in the domain of manufacturing in [R59].

Finding optimal job schedules is, however, difficult. Mathematically grounded solutions exist to a limited set of small scheduling tasks only. In most of the cases direct enumeration does not help either, since the scheduling search space is extremely large making the evaluation of all states practically infeasible. In order to overcome this difficulty, either heuristics or some directed search methods driven by artificial intelligence or learning algorithms are applied. In real manufacturing environments the demands are even greater, since the schedulers are expected to sense any changes in the environment and to provide feasible and close-to-optimal re-schedules of the given task. "*In the case of on-line dynamic scheduling there is a time constraint for the scheduler for finding the best possible result*" [R27].

In the rest of the chapter a specific scheduling class, called flow-shop scheduling is addressed. First the flow-shop task is defined, then a survey on the classical solutions are given. The classical heuristic solutions are non-improving algorithms, which means that in unchangeable environment they always provide the same, quasi-optimal solution. On the other hand, improving algorithms are capable to find better schedules if they have enough time to evaluate the different possibilities.

We provide a scheduling framework which is capable of dynamic behavior and which utilizes computational time frames determined by real manufacturing events, such as job starting and finishing to make directed-search in the scheduling state space. The search process starts from a firm, but non-optimal solution provided by one of the classical algorithms and ends up in a new, improved solution if the global optimum is not reached. As the time goes on, exploration of new schedules gradually turns into focusing on the best solution, or solutions that have been found. This is the point where Chapter 3's simulated annealing combined with reinforcement learning algorithm can contribute to the general scheduling framework.

5.4 Flow-shop scheduling

5.4.1 The definition of flow-shop scheduling

Flow-shop scheduling is also referred to as pipeline scheduling, and it can be defined as follows: given a time horizon, m processing machines, n jobs to be executed, a job

execution sequence[19] is sought which yields the minimization/maximization a particular objective function. Let the processing machines be denoted by $m_1, m_2, ..., m_m$ and the jobs by $j_1, j_2, ..., j_n$. All jobs are processed on all machines, no preemption is allowed and each job as well as each machine is unique. Processing each job on the machines consumes certain amount of time, which times are structured in a processing time matrix $\mathbf{M} = \{t_{ij}\}$ where t_{ij} is the processing time for job j on machine i.

The optimality criterion of the scheduling problem can be various ranging from exact cumulative properties to stochastic ones. The most popular criteria are maximization of throughput, minimization of lateness, minimization of processing costs. One possible cost is the cost of machines that is spent on waiting for jobs to be processed. This is often regarded as the sum of off-machining times. For the easy comparison, throughout this chapter, the off-machining metric is used.

Using the $\langle a \mid b \mid g \rangle$ triplet notation, the flow-shop scheduling is abbreviated as $\langle F_m \mid\mid O_{\min} \rangle$ when each job is allowed to follow each other job, and $\langle F_m \mid prec \mid O_{\min} \rangle$ when precedence relations are defined among different jobs. F_m indicates that the number of machines can be arbitrary and all jobs are executed on all machines; *prec* indicates precedence relations among jobs; O_{min} denotes minimization of the sum of off-machining times.

The computation complexity of the general problem, apart from special cases, is non-polynomial [R59].

5.4.2 Johnson's algorithm

The first algorithm that solved the 2-machine flow-shop scheduling problem was published by *Johnson* in 1957. The philosophy behind the Johnson's algorithm is *"not to let the second machine wait for processing"*. Thus all jobs that have smaller execution time on the first machine are processed at the beginning, while those jobs that have shorter execution times on the second machine are left behind. This type of ordering aims at filling the second machine with jobs as soon as possible [R59], thus, minimizing its off-machining time. Note the simplification that the first machine can process jobs one after another, without off-machining times.

Figure 5.2 shows a simple 2-machine flow-shop scheduling task involving 4 jobs. In the figure, A_1, A_2, A_3, A_4 denote the job execution times on machine A, while B_1, B_2, B_3, B_4 those on machine B, and X_1, X_2, X_3, X_4 indicate the off-machining

[19] Since the job sequence is the only control parameter, it determines the schedule as well, thus the terms "sequence" and "schedule" are used interchangeably in the flow-shop context throughout this chapter.

times. It is easy to observe that the total length of processing is determined by machine B as $T = \sum_{j=1}^{4}(B_j + X_j)$, or for n jobs as

$$T = \sum_{j=1}^{n}(B_j + X_j)\,. \tag{5.1}$$

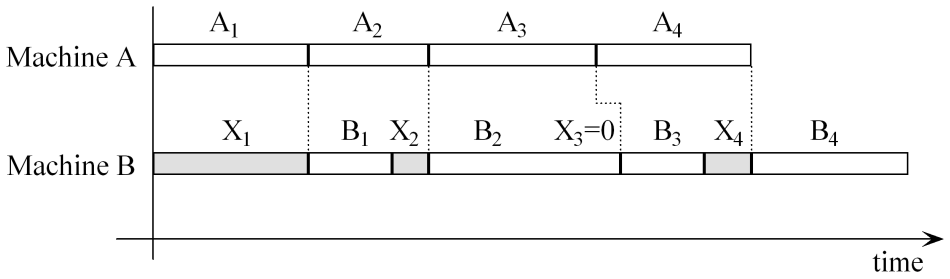


Figure 5.2
2-machine, 4-job flow-shop scheduling task

Note that the objective is to minimize the sum of off-machining times X, where X is defined for n jobs as

$$X = \sum_{j=1}^{n} X_j\,. \tag{5.2}$$

5.4.3 Palmer's method

Johnson's algorithm can be extended up to 3-machines, but cannot be applied for larger problems. There are two fundamental approaches to expand the philosophy behind Johnson's algorithm above 2 machines: one group of algorithms forms two virtual machines out of the m machine, and runs Johnson's algorithm on them; the other group of algorithms uses the sorting principle applied in the 2-machine case, i.e. the shorter machining time at the beginning of the pipeline, the earlier execution, or the shorter machining time at the end, the later the execution.

Palmer's method belongs to the first class of algorithms. To each job a priority index is assigned with respect to the corresponding execution times on the different machines, and then the jobs are sorted in decreasing order of their priority indices [R33]. Equation 5.3 is used for computing priority values for each job j:

$$I_j = -\sum_{i=1}^{m}\left[\frac{m-(2i-1)}{2}\right]t_{ij} \,. \tag{5.3}$$

5.4.4 Dannenbring's algorithm

The "*quick availability method*" worked out by *Dannenbring*, defines abstract machines reformulating the *m*-machine problem into a single 2-machine task where the abstract machining times (o_{1j}, o_{2j}) are as follows:

$$\begin{aligned} o_{1j} &= \sum_{i=1}^{m}(m-i+1)t_{ij} \,, \\ o_{2j} &= \sum_{i=1}^{m} it_{ij} \,. \end{aligned} \tag{5.4}$$

Johnson's algorithm is then applied to o_{1j} and o_{2j} to determine the job execution sequence [R11].

5.4.5 The quality of the solution

The quality of the solution can be expressed in terms of optimality, efficiency, effectiveness and feasibility [R54].

Optimality means that the solution is the best with respect to the given optimality criterion. Since the NP-hard nature of the problem, this is rarely guaranteed, so instead of optimality, quasi-optimality is used. Quasi-optimality is difficult to define; its meaning ranges from *"getting close-to-optimal"* to *"avoiding worst-case"* scenarios.

Efficiency defines the measure how the computational time spent on finding the solution relates to the time scale of the real processes. An inefficient scheduling algorithm solves the task within the same time, as the manufacturing system accomplishes the real manufacturing task.

The purpose of scheduling effectiveness is to measure the distance between the solution and the theoretical optimum. In practice, however finding the global optimum is difficult.

Feasibility is used to express that the solution matches all scheduling constraints.

5.5 The structure of the proposed dynamic scheduler

Classical flow-shop scheduling algorithms can be improved by using the appropriate combination of the concepts mentioned above. In [R27] a service-like architecture, called scheduling agent is proposed, that is a "*system, which in normal static condition*

can ensure global performance if other agents follow its command or its advice". "*In dynamically changing conditions, however, through increased autonomy of agents, a more dynamic behavior can be reached.*" Improving the idea we propose the following structure:

- using the concept of the scheduling agent,
- in a virtual manufacturing environment,
- where the scheduling algorithm is of an improving type,
- and which can be initialized by one of the classical static scheduling algorithms.

One of the results of using virtual manufacturing concept is that scheduling and dispatching are functionally separated. Note that scheduling is the way how a job sequence is created, so the whole scheduling window is examined, while dispatching is just an execution-like local decision. The differences between the two are discussed by *Kádár* in [R17].

5.5.1 General principles

The general idea of using reinforcement learning (RL) combined with simulated annealing in solving scheduling problems is originated by *Zhang* and *Diettrich* [R64]. Their goal was to provide automatic, repair-based, domain-specific heuristics to build optimal job-resource allocations for the job-shop scheduling problem. In their solution simulated annealing probabilities appeared as an acceptance factor assigned to a schedule, and they also used experimental temperature bounds. The novelty of our proposed solution is that simulated annealing (SA) is used for building the job sequence, thus no unnecessary or "*rejected-in-the-future*" schedules are generated, and due to Theorem 3.2 temperature bounds that can be dynamically computed.

In the routing problem, which is detailed in Chapter 4, the problem definition is as simple as searching the shortest path between two determined nodes in a network graph. Consider the graph representation of the flow-shop scheduling task: each job is represented as a node, each edge a feasible job transition having the expected off-machining times on the edges. The graph is dynamic and there is a zero cost path from any node to the starting node whenever no other nodes are remained to traverse. The goal is to find the shortest possible round trip provided that all nodes are visited once and only once. In this respect, scheduling problem appears to be similar to the traveling salesman's problem (TSP).

To view the flow-shop scheduling as a reinforcement learning problem, the problem space, i.e. actions and states, and reinforcement functions should be determined. A significant difficulty is that all RL methods are convergent only if the decision process is Markovian. Neither the flow-shop scheduling, nor the TSP is Markovian, since each decision point excludes one or more opportunities from the

available future decision set, thus influencing future decisions. This is the reason why the expression "*domain-specific heuristics*" is used.

States are crucial parts of RL based algorithms since they uniquely identify the nature the process in any time step. There are two approaches how states are defined: a theoretical one, and a practical one. From theoretical aspect, a state is represented as a job sequence selected so far at any point of the sequencing process. However, in practical cases it is inconvenient, or even infeasible to store job sequences, especially for large number of jobs. It is much more convenient to split the sequence into two parts: the sequence of jobs that have already been selected, and the set of jobs still available. At the decision point only one job, i.e. the one that terminates the already-selected job sub-sequence is examined. This gives the natural definition of states as last jobs in the sub-sequence, and also defines the set of actions as the set of available next-jobs. "*The policy tells what scheduling action to make next in order to maximize some measure of quality of the final schedule*" [R64].

The action-state value of each state-action pair is stored in a $n \times n$ matrix, which is denoted by $\mathbf{Q}$. Row i and column j of the matrix shows the estimated reward of continuing the job sequence from job i to job j. The update rule defined over action-state values is as follows:

$$Q_{ij} = Q_{ij} + \alpha (r + \gamma \max_{j \notin \mathbf{s}} Q_{i+1,j} - Q_{ij}) . \qquad (5.5)$$

In the above equation α and γ are the standard RL parameters: learning rate and discount factor, respectively. Array $\mathbf{s}$ denotes the sequence of jobs selected so far. It is easy to see in the update rule 5.5 that the value-update of any state-action pair is influenced by the reward received as well as the estimated value of the remaining job sequence. In graph representation the term $\max_{j \notin \mathbf{s}} Q_{i+1,j}$ is the largest possible value that a successor job estimates on the reward of the total job sequence. Note that the process is naturally episodic: each episode consists of a full job-sequence creation. Reward is provided at the end of each episode, thus whole job sequence is evaluated and reinforced. This is quite important, since no additional jobs in the partially setup sequence can be considered as a "further step" to the optimal solution. Recall that the whole process is not a Markov Decision Process (MDP)!

5.5.2 Evaluation/reward function

Reinforcement values are some measurements of the "fitness" of the job sequence. For the sake of appropriate comparison a special evaluation function, the off-machining time, is used. Off-machining time is defined as the sum of times all machines have to wait for jobs to process from the beginning of processing the first job on the first machine to the end of processing the last job on the last machine. It is shown in

Appendix D that for a particular job sequence the off-machining time can be computed by the algorithm shown in Figure 5.3.

```
input: M(m,n), p(n), b(m);
output: v;
storage: d(n), D(m,n), g(m);
function n_machine (M,p,b)
begin
  v:=0;
  for i:=1 to m do
    g(i):=b(i)-v;
    s(i):=M(i,p(1));
    d(i):=max(0, -g(i));
    D(i,1):=d(i);
    if g(i)<0 then g(i):=0;
    v:=v+M(i,p(1));
  end
  for j:=1 to n do D(1,j):=0;
  for j:=2 to n do
    for i:=1 to m do
      s(i):=s(i)+M(i,p(j));
      D(i+1,j):=max(0,s(i)+d(i)-s(i+1)-d(i+1));
      d(i+1):=d(i+1)+D(i+1,j);
    end
    s(m):=s(m) +M(i,p(j));
  end
  v:=0;
  for i:=1 to m do v:=v+d(i);
end
```

Figure 5.3
Algorithm to determine off-machining times

Property 5.1: *The computation cost of the evaluation algorithm is proportional to* $O(nm)$.

The algorithm uses the processing times matrix **M** and the job sequence permutation vector **p** as input and returns the sum of off-machining times. Vector **p** can take any permutation of the job indices, when no job precedence relations are set. The control parameter is also vector **p**, since the job sequence is sought that minimizes the off-machining time. In some cases not all machines are free when the first job on the first machine starts; some machines may finish old, already-running jobs. The expected finish times on these machines are considered to decrease the initial off-machining times, thus these times appear as temporal boundary conditions for the evaluation algorithm and are subtracted at the beginning. Vector **b** denotes the boundary condition vector which, in some cases, is purely initialized as a zero-valued array.

The off-machining time values can set up a partial ordering among different job sequences, and can be used as a reward: the smaller the value, the better the solution. In

fact different functions of the reward, such as the square-reciprocal or inverse-signal are more appropriate to reward good sequences and punish wrong sequences.

5.5.3 Job sequence setup

Job sequence is determined by using equation 3.1. A special vector **v** is used for storing the value of the first action. The update rule that modifies initial preferences is similar to that of 5.5:

$$v_i = v_i + a\,(r + g \max_{i,j} Q_{i,j} - v_i)\,. \tag{5.6}$$

The method of setting up a job sequence is as follows: First, a probability distribution is defined over preferences **v** by using equation 3.1. Then a job is selected randomly with respect to the defined probabilities. Let the job be denoted by j_i. Then the chosen job is removed from the available set of jobs, i.e. the *i*th column of matrix **Q** is masked, and the uncovered part of the *i*th row of **Q** is used for defining another probability distribution. (Equation 3.1 is applied again.) A job is chosen again with respect to the new distribution, which, in turn, will also cover a column in matrix **Q** and marks a new row of preference values, etc. As a final result, a job sequence is set up, and evaluated by the algorithm in Figure 5.3.

5.5.4 Update rules

Each evaluation step ends up in updating preference values, thus influencing future job selection probabilities. Another factor which determines probability values is the temperature used in the Boltzmann-formula. Since the whole sequencing procedure consists of *n* decisions, exactly *n*+1 different temperature values should be used. $\mathbf{t}^k$ is the vector form of the temperature in time step *k*, where T_1^k is the temperature corresponding to the first decision (the first job), and the sequence $T_2^k, T_3^k, ..., T_{n+1}^k$ is used for storing temperature values corresponding to the remaining decisions. Note that exactly one row, the one corresponding to the last job, is unused in the temperature vector.

Given a time step interval defined by t_{start} and t_{end}, i.e. the start of the sequencing process and the intended finish time, an annealing schedule can be defined as follows:

$$\mathbf{t}^{k+1} = \mathbf{t}^k + n\frac{\mathbf{t}^{\min} - \mathbf{t}^{\min}}{t_{end}^n - t_{start}^n} t^{n-1} \tag{5.7}$$

where $\mathbf{t}^{\max} = \{T_i^{\max}\}$ and $\mathbf{t}^{\min} = \{T_i^{\min}\}$, $i = 1,2,...,n+1$ are computed as

$$T_i^{\max} = \frac{\max(\mathbf{e}_i\mathbf{D}) - \min(\mathbf{e}_i\mathbf{D})}{\ln 2},$$
$$T_i^{\min} = \frac{\max(\mathbf{e}_i\mathbf{D}) - \max_2(\mathbf{e}_i\mathbf{D})}{2\ln[\dim(\mathbf{e}_i) - 1]}, \tag{5.8}$$

where matrix $\mathbf{D} = \begin{bmatrix} \mathbf{v} \\ \mathbf{Q} \end{bmatrix}$ is a $(n+1)\times n$ compound matrix. Note the following:

- When calculating temperature bounds the simplified, but less exact, equations are used. Original equations of Theorem 3.2 can also be applied to define the temperature bounds.
- Operator max() provides the largest element of array **a**; operator min() gives the smallest element; operator $\max_2()$ is used for giving the second largest value in the array; operator dim() returns the dimension of the vector. Recall from Chapter 3, that $\dim(\mathbf{e}_i) > 2$!
- Vector $\mathbf{e}_i$ is the *i*th row of the $(n+1)\times(n+1)$ identity matrix.

5.5.5 Dynamic scheduler

The proposed dynamic scheduler that operates in the virtual part works as follows: Suppose that all system elements are up and running including the controller (RIS), the machines (RPS) and the virtual services (VIS and VPS). As an initializing event, the RIS gets the scheduling task as well as the process plans involving all necessary manufacturing data, such as expected values of processing times, job and resource descriptions. The process plans may also include a sequence plan to initialize the RIS. So far the system is built up in the same way as an ordinary hierarchically controlled manufacturing system. Whenever the RIS receives a task, it looks for virtual services on the enterprise network which provides on-line schedule advisory. If this service is not available, the job sequence provided by the process plan is executed. If the VIS is found, all manufacturing data are synchronized between the RIS and the VIS, and the VIS starts the simulation of schedules. Figure 5.4 shows a cycle of activities that are executed in each simulation step. The RIS waits for a short period of time to get the initial schedule proposal, and starts executing the first job with respect to its actual known job sequence. Whenever the RIS receives a sequence proposal, it makes feasibility checks on it to make sure, that the sequence does not mismatch the already-executed jobs, and is a feasible (i.e. well-formed) sequence. All infeasible schedule proposals are discarded by the RIS. Job starting and finishing events are reported to the VIS by using special synchronization protocols. The virtual system also maintains the set of already-processed jobs to avoid wasting simulation cycles on producing ill-formed schedules. The RIS periodically (and also before selecting a new job) asks the

VIS for a new schedule proposal, and if there is one, it is synchronized, and the new job is selected with respect to the new schedule.

Exploiting the fact that the simulation is definitely faster that the real process, there is plenty of time to make simulation steps among the job events such as job starting and finishing, thus making the system flexible. As a basic rule, a time window which marks the time domain of the annealing process is defined between the job selection events. In each simulation time step a job sequence is prepared with respect to the values of the **Q** matrix. The sequence approaches to the best sequence found if the temperature values are decreased close to their minimal values, thus a guided search takes place which starts from random search in the job-sequence state space, and ends up in a quasi-optimal schedule, as the annealing progresses. Whenever a "next-job" decision is made, a new time slot is also available which mark a new annealing time domain, and a new simulated schedule.

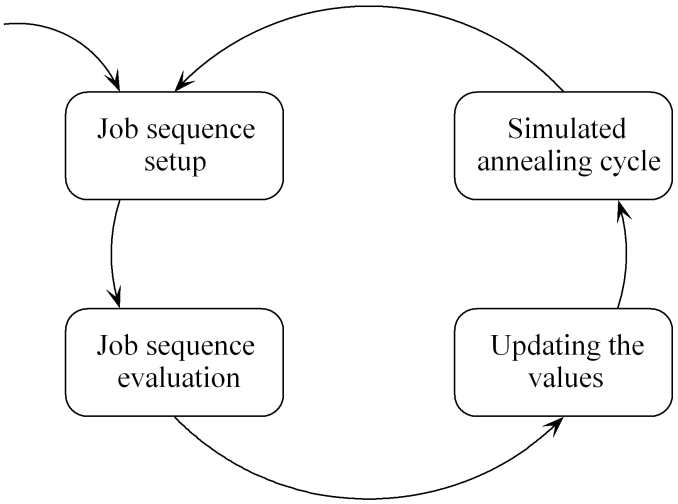


Figure 5.4
A VIS Simulation cycle. An extra arrow indicates at the "job sequence setup" block the entry point of the cycle

5.6 Validation of the model

The proposed model has been extensively tested through randomly generated sample schedules. We statically compared the RL scheduler to other heuristic methods and also studied its dynamic behavior.

5.6.1 Static analysis on different models

The validation code is written in C for efficiency reasons. The features of the sample schedules are summarized in Table 5.1.

Note that the theoretically grounded Johnson's algorithm can be executed on 2-machine tasks only. When "large tasks" are compared, only heuristic and RL-based methods are used. Figure 5.5 shows the comparison of off-machining times computed by Johnson's algorithm, Palmer's method, Dannenbring's method and RL-based simulated annealing method as the function of the number of jobs.

It is trivial to see that neither the heuristic nor the RL-based method can produce better results, than Johnson's algorithm, since it provides the theoretical optimum. In the 2-machine layout, Dannenbring's method produces the worst approach on average, while by using Palmer's indices the difference is significant only when the optimal off-machining times are relatively small. The Boltzmann-annealing based RL scheduler was able to find the theoretical value in most of the cases.

Parameter	Value
Maximum number of jobs	100
Maximum number of machines	40
Minimal job execution time	5 TU[20]
Maximal job execution time	25 TU
Precedence constraints	None

Table 5.1
Properties of sample schedule plans

Parameter	Value
Learning rate	0.2
Discount rate	0.8
Annealing schedule	polynomial annealing (n=5) with variable temperature bounds
Exploration interval	proportional to the square of the number of jobs

Table 5.2
Summary of the RL-scheduler parameters

[20] TU represents "time units". Since the samples are generated randomly, all off-machining time values are dimensioned in an abstract unit called TU.

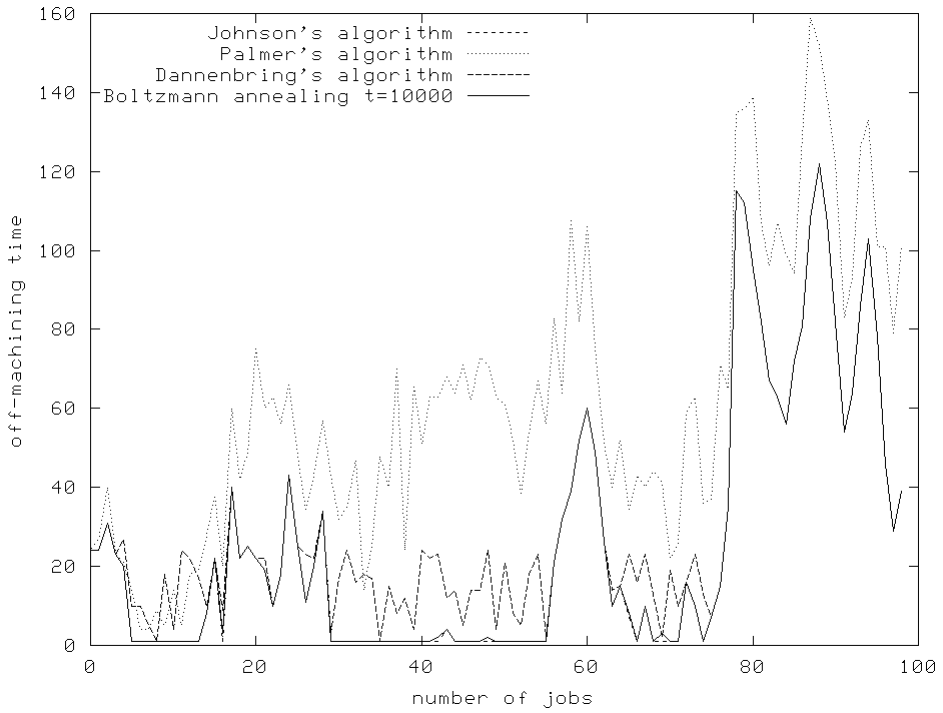


Figure 5.5
Off-machining times vs. the number of jobs diagram for two machines (m=2). Boltzmann-annealing curve runs together with the Johnson's algorithm curve in most of the cases

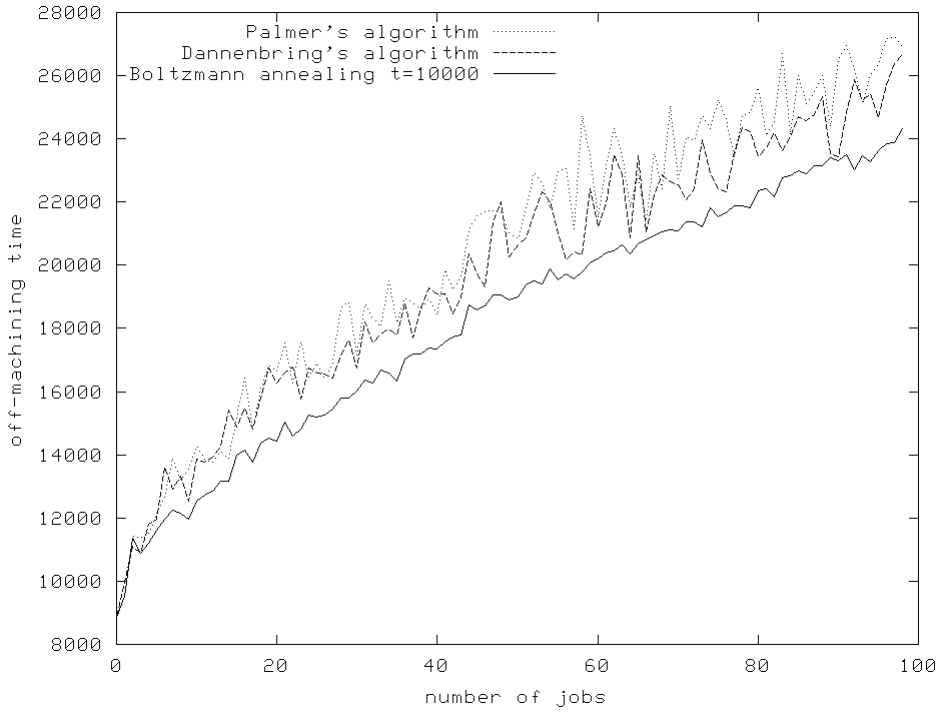


Figure 5.6
Off-machining times vs. number of jobs diagram for large number of machines (*m*=40)

In Figure 5.6 a similar comparison is shown but for large number of machines, 40. It is easy to see that the RL-based method produces definitely smaller off-machining times than classic heuristic models. The key parameters of the RL scheduler are summarized in Table 5.2.

5.6.2 Dynamic behavior

The dynamic test analyzes the scheduler's temporal behavior. Figure 5.6 shows the slice of simulated off-machining times vs. the number of simulation steps diagram of a 20-job 20-machine flow-shop scheduling task.

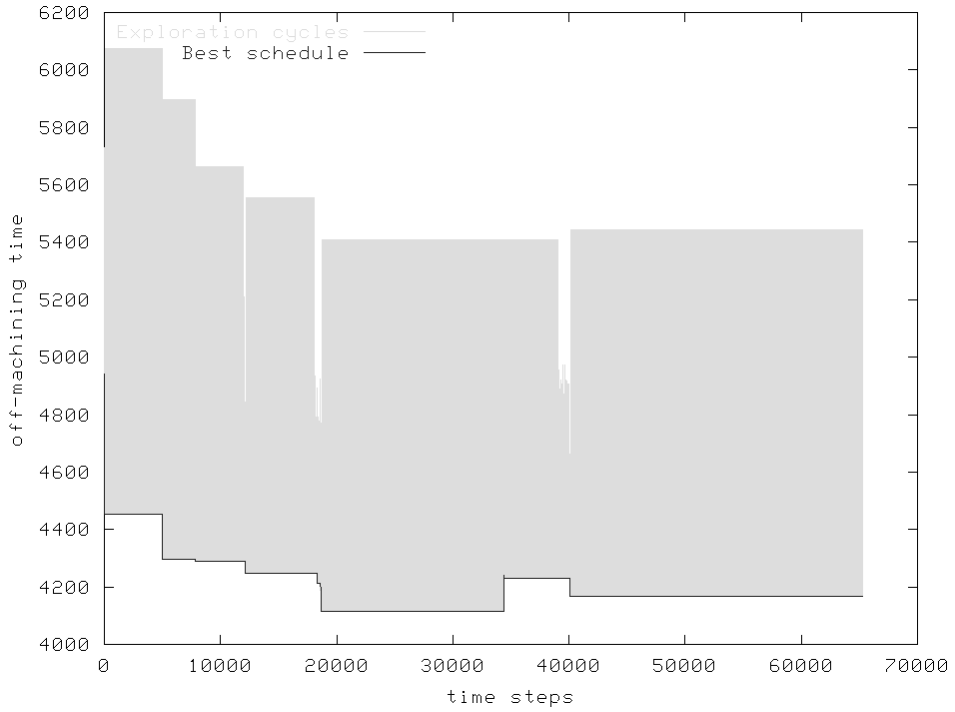


Figure 5.6
Dynamic behavior of the scheduler

For illustration reasons the annealing time was set to 95% of the value suggested in the previous sections, $0.95m_{1j}\frac{speed\_parameter\_of\_VIS}{speed\_parameter\_of\_RIS}$, i.e. the 95% of the estimated time of the selected job's execution. An annealing schedule is thus shorter than the affordable exploration time in a, say, production system. The gray area shows the exploration cycles after selecting a new job, the gaps among the cycles are the exploitation periods. During the simulation the best job sequence is maintained just in the case the job is executed earlier than the simulated annealing, and job sequence

request would arrive before finishing the exploration. The off-machining times corresponding to the best job sequence is indicated by the black line in the figure.

The simulation starts when the VIS receives all the necessary information. The initial exploration time is usually set to some experimental value. It is easy to see that in the given example the off-machining times of the proposed schedules decrease from 5000 to 4100. The latter value is kept as a quasi-optimal solution in the next exploration period. In the third exploration period, around simulation time step 33000, an unexpected event occurs: the machining times vary due to some machine failures in the execution of certain jobs, and the estimated off-machining of the stable job sequence goes up to 4250. Since the job sequence is considered as a quasi-optimum, it would be kept if there was no VIS in the system, thus escalating the failure further in the manufacturing pipeline with respect to time delays. However, the changes are reported to VIS, which is found to be in the exploration period, and which tries to find a new partial job sequence being more appropriate to the modified parameters. As a result of re-scheduling, the estimated off-machining time descends around 4150. It is important that the disturbance occurs before the end of an exploration period, thus the schedule is improved significantly only in the next period. Note that during the simulation the RPS starts and finishes jobs which cannot be executed once again. The already-executed-jobs are considered to be invariables to the VIS simulation.

5.7 Implementation of precedence constrains

So far no job precedence relations have been defined among jobs. In real life applications, however, constraints exist and represent either preferential ordering, or mandatory precedence. E.g. it is not reasonable to start with a finishing operation, and follow a roughing one.

There are two approaches to treat precedence relations: The first approach is often regarded as selection methods, when feasible sequences are selected from arbitrary defined general sequences, like in the case of genetic algorithms' repair methods [R27]. This method has the drawback of spending resources (processor time, memory) on finding solutions which will be surely rejected due to mismatching the constraints.

The other approach allows only those job sequences to be built which surely satisfy all precedence constraints. A precedence constraint can be described as an if-then-rule in the following form

$$c_l : j_{l_1} \rightarrow j_{l_2},\ l = 1,2,...,C,\ l_1, l_2 = 1,2,...,n \qquad (5.9)$$

where C is the number of constraints, c_l is the constraint identifier and job j_{l_1} must be executed prior to job j_{l_2}. Indices l_1 and l_2 are job indices.

The left hand side of the expression is the condition part, while the right hand side is the conclusion part. All constraints are summarized in a constraint set denoted by **C**.

The next key question that emerges: How can restricted job sequences be built based on constraint set **C**? The answer comes from the meaning of the constraint: i.e. execution of the conclusion part is forbidden as long as the condition is not satisfied. It means that when the sequence is created, special attention should be paid to jobs being in the conclusion part of any rule **C**. For all those jobs the action selection probability values are set to 0 regardless to their preferences. This method prevents the job in the conclusion to be selected, as long as the condition job is not completed.

In the computation, the algorithm covers all columns of matrix **Q** corresponding to any constraint conclusion. Whenever a job is selected, the constraint set is sought if the selected job is in the condition part of any constraint. If so, the column corresponding to the consequence of the constraint is uncovered, thus the scheduler allows probability values that correspond to the conclusion part to have nonzero values. The method works for transitive dependencies and constraint disjunctions as well.

Since the set of constraints can be inconsistent, a special check is needed before the first sequence is made to discover if the constraint rules really define partial ordering among jobs. If there is any cycle in the constraint set, there is no ordering, and the constraints define "impossible" task.

5.8 Discussion

In this chapter a novel scheduling concept was shown that implements on-line support for dynamic flow-shop scheduling and integrates the virtual manufacturing concept and reinforcement learning to form a scheduling agent.

Virtual manufacturing implements a kind of service to the real manufacturing system: it simulates different manufacturing layouts in the fraction of time of the real processes and makes schedule proposals whenever they are asked for. When no exact algorithms exist to solve a scheduling problem, improving algorithms such as RL combined with SA can be used as the "learning module" of the agent. The combined algorithm exploits the results of the temperature bounds theorem shown in Chapter 3 and gives reasonable improvement of classical heuristic solutions with respect to off-machining times. Furthermore, the agent can make corrected proposals if some unexpected event, such as machine failure occurs. Static algorithms can be easily used for cost efficiently initializing the scheduling agent.

Precedence constrains are treated naturally by the special "*matrix covering*" technique.

Chapter 6

Summary and Future Work

In the dissertation combined reinforcement learning concepts, problems, proposed solutions, algorithms and application examples have been shown. Our contributions to this broad and continuously evolving interdisciplinary scientific domain can be classified around three topics:

1) We proposed a general simulated annealing model tailored to reinforcement learning which lets the learning agent make smooth exploration and exploitation balancing in a defined control parameter domain.
   a) We used the Boltzmann distribution for assigning probability values to individual decision actions. It was shown that if the control parameter, temperature T converges to infinity, the probability distribution approaches the uniform distribution, and if T converges to 0, the probability distribution approaches to the greedy distribution.
   b) We showed that uniform and greedy distributions can be approached with a sufficiently small error, say e, at finite and nonzero values of T, whenever the preference values are also finite. The smallest T value which guarantees uniform distribution within error level e and the largest T value that guarantees greedy distribution within the error level e were determined.
   c) We worked out a simulated annealing method which defines an annealing schedule between the extremes of T, on a given time horizon.
   d) The model was validated on the *"n-armed bandit"* problem test-bed.

2) The Boltzmann distribution based simulated annealing model was applied in distributed shortest-path computation, and a new combined Q-learning and Boltzmann annealing based routing algorithm was developed which solves the "loop detection" and the "path recovery" problems of the Q-routing algorithm.
   a) We showed that the convergence of Q-learning under distributed action-state value representation is retained.
   b) A loop detection framework was added to the Q-routing model, which aids to reward acyclic parts of a delivery path, and to punish those parts that make up the cycle.
   c) A re-annealing protocol framework was also proposed to support path recovery when router or link comes into normal operation again after recovering damage.

d) A protocol extension was proposed on the *"option-level"* of the Internet Protocol version 4 standard.
e) The whole annealing model was validated on our Java-based distributed network simulator program.

3) An integrated reinforcement learning and simulated annealing based scheduling agent was proposed that uses the concept of virtual manufacturing. The agent is capable to propose improving flow-shop schedules, on-line, and is capable to follow environmental changes, such as machine breakdowns.
   a) A sequencing method based on Boltzmann annealing defined preference values was developed.
   b) An evaluation algorithm (reward function) was derived to determine off-machining times for the flow-shop scheduling.
   c) We compared the model to other heuristic methods were made, both in static and dynamic aspects.
   d) A real system-virtual system communication protocol was proposed, and the scheduling agent and its test framework were implemented.

The three areas mark three possible directions of future development:

- The annealing model can be improved in two ways: It would be interesting to examine the behavior of the Boltzmann distribution over the set of complex numbers and to examine how annealing schedules and application domains are interrelated.
- Implementing the proposed routing protocol within the frameworks of ordinary dynamic routing protocols is also a key issue, which would open the possibility of real-life testing for a broad range of users.
- The proposed scheduler agent is only the first step toward the implementation of intelligent, on-line *"services"* to support real manufacturing systems. It is still interesting questions how the proposed framework can be used for supporting different scheduling (or not necessarily scheduling) problems, or how the agent can be made more intelligent in terms of utilizing feed-back from the real manufacturing systems.

References

[R1] Ahuja, R.; Magnanti, T.L.; Orlin, J.B.: Network flows, Prentice Hall, 1993

[R2] Baker, A.: Case study results with the market-driven contract net protocol planning and control system, Proceedings of the AUTOFACT Vol. 31, 1992, pp. 17-55

[R3] Baker, A.: A survey of factory control algorithms that can be implemented in multi-agent heterarchy: dispatching, scheduling and pull, Journal of Manufacturing Systems Vol. 37, pp. 297-320

[R4] Bellman, R: Dynamic Programming, Princeton University Press, 1957

[R5] Bongaerts, L.; Monostori, L.; McFrlane, D.; Kádár, B.: Hierarchy in distributed shop-floor control, Computers in Industry Vol. 43, 2000, pp. 123-137

[R6] Boyan, J. A.; Littman, M. L.: Packet routing in dynamically changing networks: a reinforcement learning approach, Advances in Neural Information Processing Systems Vol. 6, 1993, pp. 671-678

[R7] Braden, R: Requirements for internet hosts, RFC 1122, http://www.ietf.org/rfc/rfc1122.txt

[R8] Chapman, D.; Kaelbling, L.P.: Input generalization in delayed reinforcement learning: an algorithm and performance comparisons, Proceedings of the International Joint Conference on Artificial Intelligence, Sydney, Australia, 1991, pp. 726-731

[R9] Choi, S. P. M.; Yeung, D.: Predictive Q-routing: a memory-based reinforcement learning approach to adaptive traffic control, Advances in Neural Information Processing Systems Vol. 8, 1996, pp. 231-238

[R10] Crites, P.H.; Barto, A.: Improving elevator performance using reinforcement learning, Advances in Neural Information Processing Systems Vol. 8, MIT Press, 1996

[R11] Dannenbring, D.G.: An evaluation of flow-shop sequencing heuristics, Management Science Vol. 23(11), 1977, pp. 1174-1182

[R12] Gullapalli, V.: A stochastic reinforcement learning algorithm for learning real-valued functions, Neural Networks, 1990, pp. 671-692

[R13] Hatvany, J.; Nemes, L.: Intelligent Manufacturing Systems – a tentative forecast, Proceedings of the VIIth IFAC World Congress Vol. 2, 1978, pp. 895-899

[R14] Hedrick, C.: Routing Information Protocol, RFC 1058, http://www.ietf.org/rfc/rfc1058.txt

[R15] Heragu, S.; Graves, R.; Kim, B.: Intelligent agent-based framework for integrating planning and design in material handling systems, St. Onge Company, 2002

[R16] Kaelbling L.P.; Littman, M.; Moore A.J.: Reinforcement learning: a survey, Journal of Artificial Intelligence Research Vol. 4, 1996, pp. 237-285

[R17] Kádár, B.: Ph.D. thesis, Technical University of Budapest, 2001

[R18] Kohavi, R; Provost, F: Glossary of terms, Special issue on machine learning and knowledge discovery processes, Machine Learning Vol. 30, 1998, pp. 271-274, http://robotics.stanford.edu/~ronnyk/glossary.html

[R19] Kohonen, T: Self-organizing Maps, Springer Series in Information Sciences, Springer, Berlin, 2001

[R20] Kuipers, F. A.; Korkmaz, T.; Krunz, M.; Van Mieghem, P.: Overview of constraint-based path selection algorithms for QoS routing, IEEE Communications Magazine, 2002

[R21] Kumar, S; Miikkulainen, R.: Dual reinforcement Q-routing algorithm: an on-line adaptive routing algorithm, Proceedings of Artificial Neural Networks in Engineering, 1997, pp. 231-233

[R22] Mahnig, T.; Mühlenbein, H.: A New Adaptive Boltzmann Selection Schedule SDS, Congress of Evolutionary Computations, 2001, pp. 183-190

[R23] Malkin, G.: RIP version 2 protocol analysis, RFC 1387, http://www.ietf.org/rfc/rfc1387.txt

[R24] Márkus, A.; Váncza, J.: Product line development with customer interaction, Annals of the CIRP Vol. 46/1, pp. 361-364

[R25] Monostori, L.: Intelligent manufacturing systems, D.Sc. Dissertation, Hungarian Academy of Sciences, 1998

[R26] Monostori, L.; Márkus, A.; Van Brussel, H.; Westkämper, E.: Machine learning approaches to manufacturing, CIRP Annals Vol. 45, No. 2, 1996, pp. 675-712

[R27] Monostori, L.; Hornyák, J.; Kádár, B.: Novel approaches to planning and control, Computers in Industry, Elsevier, 1998, pp. 97-113

[R28] Moore, A.W., Atkeson, C.G.: Prioritized sweeping: Reinforcement Learning with less data and less real time, Machine Learning Vol. 13, 1993, pp. 103-130

[R29] Moy, J.: The OSPF specification, RFC 1131, http://www.ietf.org/rfc/rfc1247.pdf

[R30] Nagle, J.: Congestion control in IP/TCP internetworks, RFC 896, http://www.ietf.org/rfc/rfc896.txt

[R31] Nuijten, W.; Le Pape, C.: Constraint-based job shop scheduling with Ilog scheduler, Journal of Heuristics, Kluwer Academic Press, 1998, pp. 271-286

[R32] Numerical recipes in C: The art of scientific computing, Cambridge University Press, 1998, http://www.nr.com

[R33] Palmer, D.S.: Sequencing jobs through a multi-stage process in the minimum total time-a quick method of obtaining near optimum, Operation Research Quarterly Vol. 16, No. 3, 1965, pp. 101-107

[R34] Pierre, S.; Said, H.; Probst, W.G.: Routing in computer networks using artificial neural networks, Artificial Intelligence in Engineering Vol. 14, 2000, pp. 295-305

[R35] Postel, J.: Internet protocol, RFC 791, http://www.ietf.org/rfc/rfc791.txt

[R36] Postel, J.: Internet control message protocol, RFC 792, http://www.ietf.org/rfc/rfc792.txt

[R37] Rekhter, J.: The NFSNET Backbone SPF based Interior Gateway Protocol, RFC 1074, http://www.ietf.org/rfc/rfc1074.txt

[R38] Description if Interior Gateway Routing Protocol (IGRP), Cisco, http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/igrp.htm

[R39] RFC 1105, http://www.ietf.org/rfc/rfc1105.txt

[R40] Routing algorithms, http://www.cisco.org

[R41] Rummery, J.A.: Problem solving with reinforcement learning. Ph.D. thesis, Cambridge University, 1995

[R42] Sandholm, T.; Crites, R.: On multiagent Q-learning in a semi-competitive domain, 14th International Joint Conference on Artificial Intelligence (IJCAI-95) Workshop on Adaptation and Learning in Multiagent Systems, Montreal, Canada, 1995, pp 71-77

[R43] Scheduling in nutshell, http://www.nnh.com/ev/nut2.html

[R44] Singh, S.; Jaakkola, T.; Littman, M.; Szepesvári, Cs.: Convergence results for single-step on-policy reinforcement learning algorithms, Machine Learning Journal Vol. 38(3), 2000, pp. 287-308

[R45] Socolofsky, T.: A TCP/IP tutorial, RFC 1180, http://www.ietf.org/rfc/rfc1180.txt

[R46] Sutton, R.; Barto, A.: Reinforcement Learning (An Introduction), The MIT Press, London, England, 1998

[R47] Sutton, R.: Dyna, an integrated architecture for learning, planning and reacting, SIGART Bulletin Vol. 2, 1991, pp. 160-163

[R48] Sutton, R.: TD models: Modeling the world at a mixture of time scales, Proceedings of the Twelfth International Conference on Machine Learning, 1995, pp. 531-539

[R49] Szepesvári, Cs.: The asymptotic convergence-rate of Q-learning, Neural Processing Information Systems, 1997, pp. 1064-1070

[R50] Szepesvári, Cs.; Littman, M.: A generalized reinforcement-learning model: convergence and applications, Proceedings of the Thirteenth International Conference on Machine Learning, 1996

[R51] Tanenbaum, A.: Computer networks, Panem Press, 1996

[R52] Tóth, T.: Planning principles, models and methods in computer integrated manufacturing (in Hungarian), University of Miskolc, University Press, 1998

[R53] Tóth, T.: The range of concept and the sphere of authority of computer aided manufacturing, Computer Integrated Manufacturing, Miskolc, 1997, pp. 2-21

[R54] Trentesaux, D.; Pesin, P.; Tahon, C.: Distributed artificial intelligence for FMS scheduling, control and design support, Journal of Intelligent Manufacturing Vol. 11, 2000, pp. 573-589

[R55] Ueda, K.; Márkus, A.; Monostori, L.; Kals, H.J.J.; Arai, T.: Emergent synthesis methodologies for manufacturing, Annals of the CIRP Vol. 50/2, 2001, pp. 535-547

[R56] Ünver, Ö.; Anlagan, Ö.; Kiliç, E.; Cangar, T.: A structured methodology for development of heterarchical control software for manufacturing cells, using Windows DNA, Proceedings of the IASTED International Conference Intelligent Systems and Control, 2000

[R57] Váncza, J., Márkus, A.: An agent model for incentive-based production scheduling, Computers in Industry, 2000, pp. 173-187

[R58] Viharos, Zs. J.: Application capabilities of a general, ANN based cutting model in different phases of manufacturing through automatic determination of its input-output configuration; Journal of Periodica Politechnica - Mechanical Engineering Vol. 43, No. 2, 1999, pp. 189-196

[R59] Vízvári, B.: Introduction to the mathematical models of production planning and scheduling (in Hungarian), ELTE University Press, Budapest, Hungary 1994

[R60] Watkins, C.; Dayan, P.: Q-learning, Machine Learning Vol. 8, 1992, pp. 279-292

[R61] Wiering, M.; Schmidhuber, J.: Fast online Q(λ), Machine Learning Vol. 33(1), 1998, pp. 105-116

[R62] http://www.linuxdocs.org/HOWTOs/Adv-Routing-HOWTO.html

[R63] Yamada, T.; Reeves, C.: Solving the Csum permutation flowshop scheduling problem by genetic local search, ICEC'98, 1998, pp. 230-234

[R64] Zhang, W.; Diettrich, T.: A reinforcement learning approach to job-shop scheduling

List of Publications

In English

[P1] Stefán, P.; Monostori, L.: Shop-floor scheduling based on reinforcement learning algorithm, 3rd CIRP International Seminar on Intelligent Manufacturing, ICME 2002, Ischia, Italy, 2002, pp. 71-74

[P2] Stefán, P.; Monostori, L.; Vaskó, Z: Quasi-optimal solution to the traveling salesman's problem in variable environment, 3rd international conference of Ph.D students, University of Miskolc, 2001, pp 415-422

[P3] Stefán, P.; Monostori, L.: On the relationship between learning capability and the Boltzmann-formula, Engineering of Intelligent Systems, Lecture Notes in AI 2070, IEA/AIE-01, 14th International Conference o n Industrial & Engineering Applications of Artificial Intelligence & Expert Systems, Budapest, Hungary, June 4-7, 2001, Springer Book, pp. 227-236

[P4] Stefán, P.; Monostori, L.; Erdélyi, F.: Reinforcement learning for solving shortest-path and dynamic scheduling problems, 3rd International Workshop on Emergent Synthesis, IWES'01, Bled, Slovenia, 2001, pp. 83-88, ISBN 961-6238-49-3

[P5] Stefán, P.; Monostori, L.: On Internet routing problems in dynamically changing environment, MicroCAD International Meeting on Information Technology and Computer Science, University of Miskolc, Hungary, 2001, pp. 221-216, ISBN 963-661-457-1

[P6] Monostori, L.; Kádár, B.; Viharos, Zs.J.; Stefán, P.: AI and machine learning techniques combined with simulation for designing and controlling manufacturing processes and systems, Preprints of the IFAC Symposium on Manufacturing, Modeling, Management and Supervision, MIM 2000, Patras, Greece, 2000, pp. 167-172

[P7] Monostori, L.; Kádár, B.; Viharos, Zs.J.; Mezgár, I.; Stefán, P.: Combined use of simulation and AI/machine learning techniques in designing manufacturing processes and systems, Proceedings of the 2000 International CIRP Design Seminar on Design with Manufacturing: Intelligent Design Concepts Methods and Algorithms, Haifa, Israel, 2000, pp. 199-204

[P8] Stefán, P.; Monostori, L.; Pupp, Z.: Reinforcement learning methods in information engineering, MicroCAD International Meeting on Information Technology and Computer Science, University of Miskolc, Hungary, 2000

[P9] Pupp, Z.; Stefán, P.: Novel applications of self-organizing maps in information technology problems, MicroCAD International Meeting on Information Technology and Computer Science, University of Miskolc, Hungary, 2000

[P10] Stefán, P.; Monostori, L.; Erdélyi, F.: Using symbolic and sub-symbolic methods in solving problems difficult to analyze, MicroCAD International Meeting on Information Technology and Computer Science, University of Miskolc, Hungary, 1999, pp. 195-200

In Hungarian

[P11] Stefán, P: Megerosíto tanulási módszerek alkalmazása Internet Routing problémák megoldására, Fiatal Muszakiak V. Tudományos Ülésszaka, Bolyai Egyetem, Kolozsvár, Románia, 2000, pp. 1-4

[P12] Stefán, P: Szimbolikus és szub-szimbolikus módszerek az analitikailag nehezen kezelheto problémák megoldásában, Fiatal Muszakiak IV. Tudományos Ülésszaka, Bolyai Egyetem, Kolozsvár, Románia, 1999, pp. 137-140

[P13] Stefán, P.: Megerosíto tanulási módszerek alkalmazása az informatikában, Doktoranduszok fóruma, Miskolci Egyetem, 1999, pp. 65-70

Appendix A

Proofs to Temperature Bounds Theorems

Theorem 3.1: *If temperature T approaches infinity, the action selection probability of all the actions approaches the uniform distribution; if T goes to zero the probability of selecting the strictly highest Q-valued action goes to 1, while the selection probability of others' goes to 0. If there are k maximal equally preferred actions, the probability of making selections from among these actions goes to* $\frac{1}{k}$ *as T goes to zero.*

Proof 3.1: First, transformation of the Boltzmann-formula is required serving as the starting point of all proofs through the rest of the appendix.

$$p_i = \frac{e^{\frac{Q_i}{T}}}{\sum_{j=1}^{n} e^{\frac{Q_j}{T}}} = \frac{1}{\frac{\sum_{j=1}^{n} e^{\frac{Q_j}{T}}}{e^{\frac{Q_i}{T}}}} = \frac{1}{\sum_{j=1}^{n} \frac{e^{\frac{Q_j}{T}}}{e^{\frac{Qi}{T}}}} = \frac{1}{\sum_{j=1}^{n} e^{\frac{Q_j}{T} - \frac{Qi}{T}}} = \frac{1}{\sum_{j=1}^{n} e^{\frac{Q_j - Q_i}{T}}} \qquad \text{(A1)}$$

The parameters' domain is set to

$$Q_i \in [Q_{\min}, Q_{\max}] \subset \mathbf{Z},\ i = 1,2,...,n,$$
$$p_i \in [0,1] \subset \mathbf{R},\ i = 1,2,...,n.$$

Here **Z** denotes the set of integers and **R** denotes the set of real numbers.

Since, $|Q_j - Q_i|$ is finite $\forall i, j$, and $\lim_{T \to \infty} \frac{Q_j - Q_i}{T} = 0$,

$$\lim_{T \to \infty} p_i = \lim_{T \to \infty} \frac{1}{\sum_{j=1}^{n} e^{\frac{Q_j - Q_i}{T}}} = \frac{1}{\lim_{T \to \infty} \sum_{j=1}^{n} e^{\frac{Q_j - Q_i}{T}}} = \frac{1}{\sum_{j=1}^{n} \lim_{T \to \infty} e^{\frac{Q_j - Q_i}{T}}} = \frac{1}{\sum_{j=1}^{n} e^{\lim_{T \to \infty} \frac{Q_j - Q_i}{T}}} = \frac{1}{n}.$$

The case of temperature approaching infinity has, therefore, been proved.
In the second case, when temperature approaches zero, there are two sub-cases: the first one is when there is only one maximal preference value, and the second one is when there are *k* equal and maximal preference values.

Before deriving the limit expressions, equation A1 should be further transformed. Since j runs from 1 to n, it is necessary that $j = i$ be satisfied at least once. Thus,

$$\frac{1}{\sum_{j=1}^{n} e^{\frac{Q_j - Q_i}{T}}} = \frac{1}{1 + \sum_{\substack{i=1 \\ j \neq i}}^{n} e^{\frac{Q_j - Q_i}{T}}}. \qquad \text{(A2)}$$

If $Q_i > Q_j$, for $j = 1,2,...,n,\ j \neq i$, then $\lim_{T \to 0} \frac{Q_j - Q_i}{T} = -\infty$, and $\lim_{T \to 0} e^{\frac{Q_j - Q_i}{T}} = 0$, and hence

$$\lim_{T \to 0} \frac{1}{1 + \sum_{\substack{j=1 \\ j \neq i}}^{n} e^{\frac{Q_j - Q_i}{T}}} = 1.$$

If $\exists j$, for which $Q_j > Q_i$, then $\lim_{T \to 0} \frac{Q_j - Q_i}{T} = \infty$, and $\lim_{T \to 0} e^{\frac{Q_j - Q_i}{T}} = \infty$, and therefore

$$\lim_{T \to 0} \frac{1}{1 + \sum_{\substack{j=1 \\ j \neq i}}^{n} e^{\frac{Q_j - Q_i}{T}}} = 0.$$

If $Q_{i_1} = Q_{i_2} = ... = Q_{i_k} > Q_j$, for $j = 1,2,...,n,\ j \neq i_l,\ l = 1,2,...,k$, then the formula A2 becomes

$$\frac{1}{\sum_{j=1}^{n} e^{\frac{Q_j - Q_i}{T}}} = \frac{1}{k + \sum_{\substack{i=1 \\ j \neq i_l, l=1,2,..,k}}^{n} e^{\frac{Q_j - Q_{i_l}}{T}}}.$$

For the rest of Q_js $\lim_{T \to 0} e^{\frac{Q_j - Q_{i_l}}{T}} = 0$, and therefore

$$\lim_{T \to 0} \frac{1}{k + \sum_{\substack{j=1 \\ j \neq i, l=1,2..,k}}^{n} e^{\frac{Q_j - Q_{i_l}}{T}}} = \frac{1}{k}. \ \blacksquare$$

Theorem 3.2: *Consider,* $e > 0$*, as a small positive number, and an upper and a lower bound on Q-values,* Q_{max} *and* Q_{min}*. The following inequalities are held under these circumstances:*

(a) $\left| p_i - \frac{1}{n} \right| < e$ if $T > \frac{Q_{\max} - Q_{\min}}{\ln \min\left\{ \frac{1}{1-ne}, 1+ne \right\}}$, for $i = 1,2,...,n$ and

(b) $\left|1 - p_i\right| < e$ if $T < \frac{k}{\ln\left[\left(\frac{1}{e} - 1\right)(n-1)\right]}$, where $Q_i > Q_j$, $j \neq i$, $k = \min_{\substack{j=1,..,n \\ j \neq i \\ Q_i \neq Q_j}} \left| Q_i - Q_j \right|$.

Note that k is the minimal difference between the largest and second largest Q-value, and due to the discrete nature of Qs, this difference is minimally 1. Also note that only a single value is allowed to maximize Q.

It is the simpler part to establish (a), since the upper bound of temperature comes from the limitation of Q-values. The goal is as follows:

$$\left| p_i - \frac{1}{n} \right| < e,$$

$$\frac{1}{n} - e < p_i < \frac{1}{n} + e,$$

$$\frac{1}{n} - e < \frac{1}{\sum_{j=1}^{n} e^{\frac{Q_j - Q_i}{T}}} < \frac{1}{n} + e. \qquad \text{(A3)}$$

It comes from the preconditions, that $Q_i, Q_j \in [Q_{\min}, Q_{\max}]$, which can be written as $Q_{\min} \leq Q_i, Q_j \leq Q_{\max}$, for $\forall i, j$. So it is easy to see that,

$$\frac{1}{\sum_{j=1}^{n} e^{\frac{Q_j - Q_{\min}}{T}}} \leq \frac{1}{\sum_{j=1}^{n} e^{\frac{Q_j - Q_i}{T}}}, \qquad \text{(A4)}$$

since $e^{\frac{Q_j - Q_i}{T}} \leq e^{\frac{Q_j - Q_{\min}}{T}}$, and $\frac{Q_j - Q_i}{T} \leq \frac{Q_j - Q_{\min}}{T}$, and $Q_i \geq Q_{\min}$ is always true. It is also true that

$$\frac{1}{\sum_{j=1}^{n} e^{\frac{Q_{\max} - Q_{\min}}{T}}} \leq \frac{1}{\sum_{j=1}^{n} e^{\frac{Q_j - Q_{\min}}{T}}}, \qquad \text{(A5)}$$

since $e^{\frac{Q_j - Q_{\min}}{T}} \le e^{\frac{Q_{\max} - Q_{\min}}{T}}$, and $\frac{Q_j - Q_{\min}}{T} \le \frac{Q_{\max} - Q_{\min}}{T}$, and $Q_j \le Q_{\max}$.

Putting inequalities A4 and A5 together a lower bound on p_i is received. Following similar reasoning, an upper bound can also be determined, and the two bounds for p_i can be written as

$$\frac{1}{ne^{\frac{Q_{\max} - Q_{\min}}{T}}} = \frac{1}{\sum_{j=1}^{n} e^{\frac{Q_{\max} - Q_{\min}}{T}}} \le \frac{1}{\sum_{j=1}^{n} e^{\frac{Q_j - Q_i}{T}}} \le \frac{1}{\sum_{j=1}^{n} e^{\frac{Q_{\min} - Q_{\max}}{T}}} = \frac{e^{\frac{Q_{\max} - Q_{\min}}{T}}}{n}. \qquad \text{(A6)}$$

An e should be found for which the bounds of equation A3 is held to the bounds of equation A6 as well. Hence, the stricter property is formulated as

$$\frac{1}{n} - e < \frac{1}{ne^{\frac{Q_{\max} - Q_{\min}}{T}}}, \qquad \text{(A7)}$$

$$\frac{e^{\frac{Q_{\max} - Q_{\min}}{T}}}{n} < \frac{1}{n} + e. \qquad \text{(A8)}$$

Denoting the term $e^{\frac{Q_{\max} - Q_{\min}}{T}}$ by x, formulas A7and A8 take the form of

$$\frac{1}{n} - e < \frac{1}{nx},$$
$$\frac{x}{n} < \frac{1}{n} + e.$$

Arranging these formulas to x, the lower limit of x is given as

$$x < \min\left\{\frac{1}{1-ne}, 1+ne\right\}.$$

Replacing x with the original exponential phrase,

$$e^{\frac{Q_{\max} - Q_{\min}}{T}} < \min\left\{\frac{1}{1-ne}, 1+ne\right\},$$
$$\frac{Q_{\max} - Q_{\min}}{T} < \ln \min\left\{\frac{1}{1-ne}, 1+ne\right\},$$

$$T > \frac{Q_{\max} - Q_{\min}}{\ln \min\left\{\frac{1}{1-ne}, 1+ne\right\}}. \qquad \text{(A9)}$$

As for the (b) part of Theorem 3.2, the theoretical lower bound of the temperature is naturally 0, but using zero temperature causes division by zero error during numerical computations. So, in practical applications the lower limit of the temperature must be slightly higher than zero, even high enough to avoid floating point errors, but small enough to guarantee greedy probability distribution at the specified error level.

Since the probability p_i can never be larger than 1, instead of equation $|1-p_i| < e$, equation $1-p_i < e$ is considered. It was clear in part (a), that bounds on p_i values must be found, but in this special case only lower bound of p_i is to be examined. Taking the form of equation A2 of the Boltzmann formula, the lower bound can be set as

$$\frac{1}{1+\sum_{\substack{j=1 \\ j\neq i}}^{n} e^{\frac{-k}{T}}} < \frac{1}{1+\sum_{\substack{j=1 \\ j\neq i}}^{n} e^{\frac{Q_j - Q_i}{T}}} \qquad \text{(A10)}$$

for some k. Getting denominators disappeared, the previous inequality is held if $e^{\frac{Q_j - Q_i}{T}} < e^{\frac{-k}{T}}$, $j = 1,2,...,n$, $j \neq i$, which yields $Q_j - Q_i < -k$, thus $Q_i - Q_j > k$. In the preconditions of the theorem it was stated that Q_i is the highest among all Q-values, therefore

$$k = \min_{\substack{j=1,..,n \\ j\neq i}} |Q_i - Q_j| \qquad \text{(A11)}$$

Returning to $1-p_i < e$, the inequality must also be true, if the value p_i is replaced by a stronger constraint, by the left-hand side part of inequality A10. So,

$$1 - \frac{1}{1+\sum_{\substack{j=1 \\ j\neq i}}^{n} e^{\frac{-k}{T}}} < e,$$

$$1 - e < \frac{1}{1+\sum_{\substack{j=1 \\ j\neq i}}^{n} e^{\frac{-k}{T}}} = \frac{1}{1+(n-1)e^{\frac{-k}{T}}},$$

$$(1-e)\left[1+(n-1)e^{\frac{-k}{T}}\right] < 1,$$

$$e^{\frac{-k}{T}} < \frac{\frac{1}{1-e}-1}{n-1} = \frac{e}{(1-e)(n-1)},$$
$$\frac{-k}{T} < \ln\frac{e}{(1-e)(n-1)}.$$

Due to the division by a negative number the relational signal turns over, and the lower temperature bound is expressed as

$$T < \frac{-k}{\ln\frac{e}{(1-e)(n-1)}} = \frac{k}{\ln\left[\left(\frac{1}{e}-1\right)(n-1)\right]}. \blacksquare \qquad \text{(A12)}$$

Appendix B

Routing Table Example from a CISCO 12000 Router

```
#sh ip route
Codes: C-connected, S-static, I-IGRP, R-RIP, M-mobile, B-BGP
       D-EIGRP, EX-EIGRP external, O-OSPF, IA-OSPF inter area
       N1-OSPF NSSA external type 1, N2-OSPF NSSA external type 2
       E1-OSPF external type 1, E2-OSPF external type 2, E-EGP
       i-IS-IS, L1-IS-IS level-1, L2-IS-IS level-2, ia-IS-IS inter ar.
       *-candidate default, U-per-user static route, o-ODR

Gateway of last resort is not set

B    208.221.13.0/24 [20/0] via 62.40.103.73, 1w3d
B    206.51.253.0/24 [20/0] via 62.40.103.73, 1w3d
B    205.204.1.0/24 [20/0] via 62.40.103.73, 1w3d
B    216.103.190.0/24 [20/0] via 62.40.103.73, 1w3d
B    213.239.59.0/24 [20/0] via 62.40.103.73, 1w3d
B    213.152.76.0/24 [20/0] via 62.40.103.73, 00:19:06
B    212.205.24.0/24 [20/0] via 62.40.103.73, 1w0d
     209.16.192.0/25 is subnetted, 1 subnets
B    209.16.192.128 [20/0] via 62.40.103.73, 1w3d
B    207.254.48.0/24 [20/0] via 62.40.103.73, 1w3d
B    205.152.84.0/24 [20/0] via 62.40.103.73, 1w3d
B    203.171.97.0/24 [20/0] via 62.40.103.73, 1w3d
B    203.1.203.0/24 [20/0] via 62.40.103.73, 1w3d
B    198.205.10.0/24 [20/0] via 62.40.103.73, 4d16h
B    192.35.226.0/24 [20/0] via 62.40.103.73, 6d12h
     170.171.0.0/16 is variably subnetted, 4 subnets, 2 masks
B    170.171.0.0/16 [20/0] via 62.40.103.73, 1w3d
B    170.171.251.0/24 [20/0] via 62.40.103.73, 1w3d
B    170.171.253.0/24 [20/0] via 62.40.103.73, 1w3d
B    170.171.252.0/24 [20/0] via 62.40.103.73, 1w3d
S    193.224.167.0/24 [1/0] via 193.6.21.142
O    157.181.141.0/29
           [110/2] via 195.111.97.170, 1d06h, GigabitEthernet2/1.912
O    193.225.57.232 [110/2] via 195.111.97.68, 1d06h, POS1/2
C    193.6.27.0/25 is directly connected, GigabitEthernet3/2
S    193.6.27.62/32 [1/0] via 193.188.137.31
S    193.6.27.62/32 [1/0] via 193.188.137.31
S    193.6.27.63/32 [1/0] via 193.188.137.46
B    192.23.11.0/24 [20/0] via 62.40.103.73, 1w3d
B    192.6.26.0/24 [20/0] via 62.40.103.73, 05:27:22
O IA 195.111.97.225/32 [110/3] via 195.111.97.67, 1d06h, POS1/1
O IA 195.111.97.224/32 [110/3] via 195.111.97.68, 1d06h, POS1/2
O IA 195.111.97.227/32 [110/3] via 195.111.97.66, 1d06h, POS1/0
```

Figure B1

Routing table excerpt from a CISCO 12000 router

Appendix C

IP Packet Format Extensions of the Boltzmann-exploration Q-routing Algorithm

In this appendix the protocol specification of the Boltzmann-annealing based routing algorithm can be found. Though the simulator is written over UDP protocol, the concept can be implemented using slight modifications in the IP header. Figure C1 illustrates the original IP header specified in [R35], Figure C2 illustrates IP header holding the proposed extensions for data packet types.

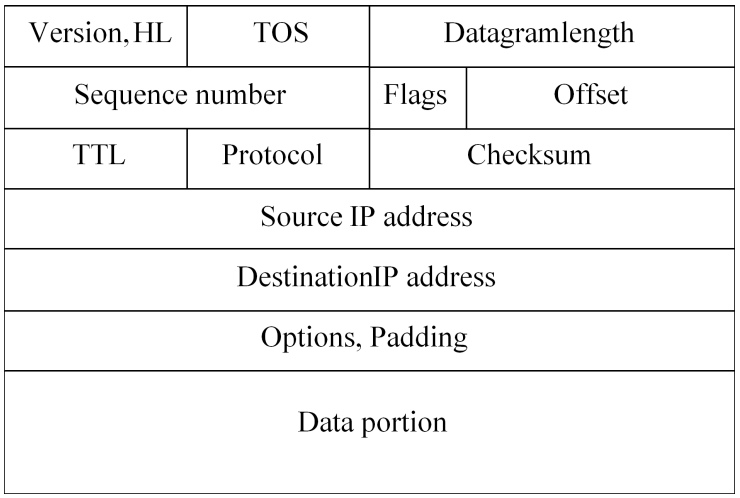


Figure C1
The specification of the IP packet header. Each row consists of four bytes

The meaning of the individual fields is as follows:

- Version: In case of IP version 4 this field is set to 4.
- Header Length (HL): the length of the header in 32-bit units. The maximal allowable length is 60 bytes.
- Type of Service (TOS): This field can be used by routers which implement priorities and quality of service in transmitting IP-packets.
- Datagram length: The length of the whole datagram including the header.
- Sequence number: Unique identifier of a packet on particular source.
- Flags: Used for indicating fragmentation.
- Offset: Used for indicating fragments' order.

- Time to Live (TTL): The maximum allowable life-time of the packet.
- Protocol: The field is used for indicating higher level protocols such as TCP or UDP.
- Checksum: CRC checksum computed for the whole packet.
- Source IP address: Identifier of the source node.
- Destination IP address: Identifier of the destination node.
- Options: Special options for specific usage, e.g. source routing.
- Padding: Empty placeholder to extend the header to end up on 32-bit boundary.

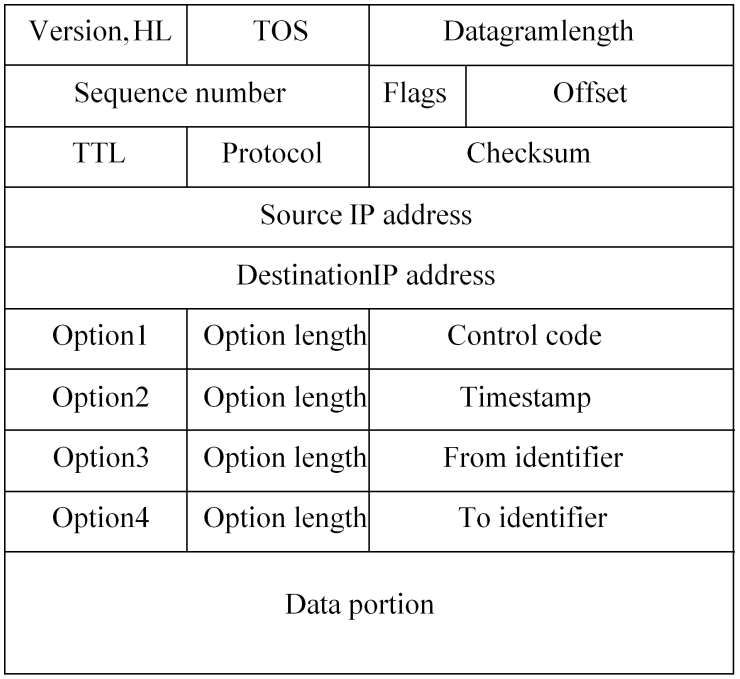


Figure C2
Proposed extensions in the IP header. Data packet example

The different fields are as follows:

- Control word: Holds control bits which are used as packet type identifiers (1-hello, 2-control, 4-data, 8-acknowledgement, 16-looped bit, 32-raise temperature, 64-send me Q (SMQ), 128-SMQ acknowledgement). The control word determines the remaining options as well. The following three options are valid for ordinary data packets only. Figure C3, Figure C4, Figure C5, Figure C6 and Figure C7 indicates the option field of acknowledgement, hello, raise temperature, send me Q and send me Q acknowledgement packets respectively.
- Timestamp: When the packet header is stored on routers, this filed is applied to hold the storage time.
- From identifier: the identifier of the link where the packet has come from.
- To identifier: the identifier of the link where the packet has sent to.

Option1	Option length	Control code
Option2	Option length	Q-value
Q-value		Padding

Figure C3
Option part of acknowledgement packets

The meaning of extra options is as follows:

- Control code: The same as in the case of data packets.
- Q-value: The floating point representation of the estimated Q-value[21].

Option1	Option length	Control code

Figure C4
Option part of hello packets

Note that in the case of hello packet there is no need for extra information introduced in the extended data packet format, only the control code is required. The same applies for raise temperature packets and SMQ packets as well.

Option1	Option length	Control code

Figure C5
Option part of raise temperature packets

Option1	Option length	Control code

Figure C6
Option part of send me Q (SMQ) packets

[21] Estimated by the neighbor.

Option1	Option length	Control code
Option2	Option length	Q-value
Q-value		Padding

Figure C7
Option part of SMQ acknowledgement packets.

Appendix D

The Flow-shop Schedule-evaluation Function

Let processing machines be denoted by A and B, jobs by $j_1, j_2, ..., j_n$. Let the job sequence be the same on both machines, i.e. pipeline processing. Processing times of the individual jobs on machines A and B are denoted by $A_1, A_2, ..., A_n$, and $B_1, B_2, ..., B_n$ respectively. Let $X_1, X_2, ..., X_n$ denote off-machining. The reason why off-machining times exist is that job j_{k+1} cannot start on machine A, since job j_k has not finished (jobs may not be interrupted), and similarly job j_k cannot start on machine B since it has not finished on machine A. Gantt-chart illustration is shown in Figure D1.

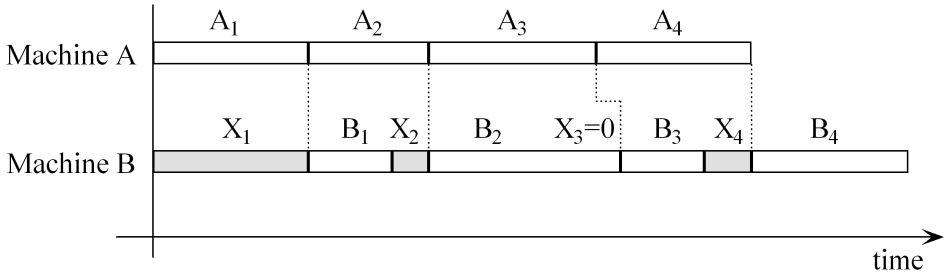


Figure D1
Gantt-chart of two-machine in flow-shop scheduling. Job sequence j_1, j_2, j_3, j_4 is assumed

Off machining times can be computed by the following equation:

$$X_k = \max(0, \sum_{j=1}^{k} A_j - \sum_{j=1}^{k-1} B_j - \sum_{j=1}^{k-1} X_j). \qquad \text{(D1)}$$

The total off-machining time is, therefore, given by

$$X = \sum_{k=1}^{n} X_k = \sum_{k=1}^{n} \max(0, \sum_{j=1}^{k} A_j - \sum_{j=1}^{k-1} B_j - \sum_{j=1}^{k-1} X_j). \qquad \text{(D2)}$$

and computed by the algorithm shown in Figure D2.

Note that equation D2 does not involve any off-machining times on machine A. An improved version of the model is when there are also two processing units, for the future consistency, B and C, and there are off-machining times on both of them denoted by $X_1, X_2, ..., X_n$ and $Y_1, Y_2, ..., Y_n$, on machines B and C respectively. Equations D1 and D2 are then modified

$$Y_k = \max(0, \sum_{j=1}^{k} B_j + \sum_{j=1}^{k} X_j - \sum_{j=1}^{k-1} C_j - \sum_{j=1}^{k-1} Y_j), \tag{D3}$$

and

$$Y = \sum_{k=1}^{n} Y_k = \sum_{k=1}^{n} \max(0, \sum_{j=1}^{k} B_j + \sum_{j=1}^{k} X_j - \sum_{j=1}^{k-1} C_j - \sum_{j=1}^{k-1} Y_j). \tag{D4}$$

Gantt-chart corresponding to the new layout is shown in Figure D3. An algorithm that computes the sum of off-machining times is shown in Figure D4.

```
input: Ai, Bi, i=1..n
output: Xsum
function two_machine_task([A1..An],[B1..Bn])
begin
  A:=A1; B:=B1; X:=A1; Xsum:=X;
  for i=2 to n do
    A:=A+Ai;
    X:=max(0,A-B-Xsum);
    Xsum:=Xsum+X;
    B:=B+Bi;
  end
end
```

Figure D2
Computation of off-machining times for two-machine flow-shop scheduling

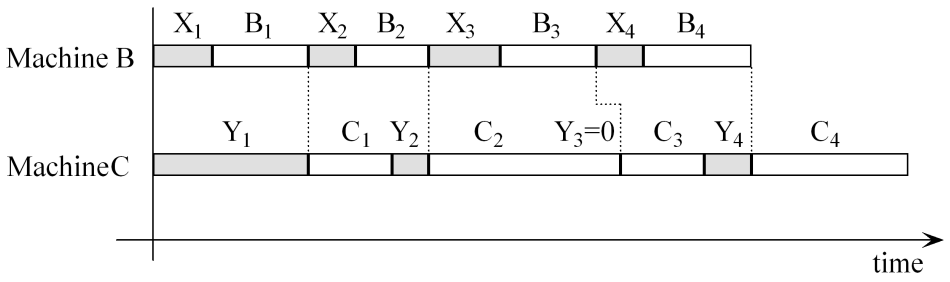


Figure D3
Gantt-chart of two-machine flow-shop scheduling with initial off-machining times

```
input: Bi, Ci, Xi, i=1..n
output: Ysum
function two_machine_extended_task([A1..An],[B1..Bn],[X1..Xn])
begin
  B:=B1; C:=C1; X:=X1; Y:=B1+X1; Ysum:=Y;
  for i=2 to n do
    B:=B+Bi;
    X:=X+Xi;
    Y:=max(0,B+X-C-Ysum);
    Ysum:=Ysum+Y;
    C:=C+Ci;
  end
end
```

Figure D4
Computation of off-machining times on two-machine flow-shop scheduling with initial off-machining times on the first equipment

When the two tasks are joined together, i.e. scheduling in Figure D1 and Figure D3, the resulting task is a three-machine scheduling with off-machining times on the second and the third machines only. Note that off-machining times on the third machine directly depend only on off-machining times as well as machining times on the second equipment, so when values of X_i have been computed Y_i can be done as well. Figure D5 illustrates the three-machine layout; Figure D6 gives formal algorithmic description to compute off-machining times for the 3-machine task.

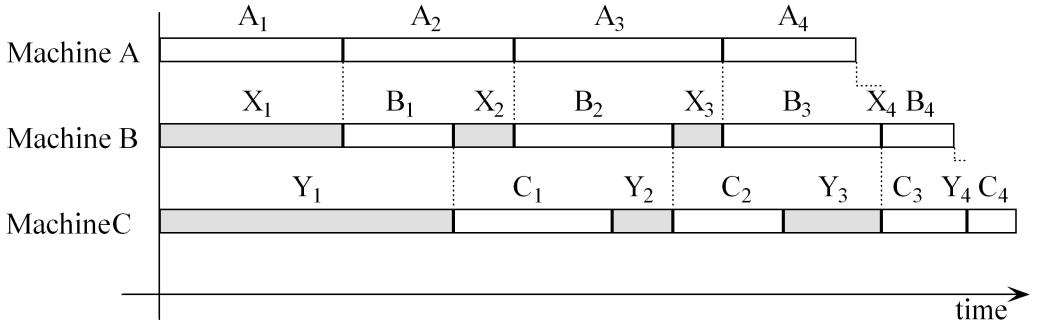


Figure D5
Three-machine flow-shop scheduling task with off-machining times occurring on machines B and C

Since there are repetitions in the core of the cycle, the algorithm can be extended to arbitrary number of jobs and machines. By induction and rearrangement of operations it is shown that algorithm in Figure D7 computes off-machining times for n jobs and m machines.

The data structures take a more compact form:
Matrix **M** stores machining times on different machines for different jobs, where machines are indexed by $i = 1,2,...,m$ and jobs by $j = 1,2,...,n$.

Matrix **D** has the same dimensional properties as **M** has and is used for storing off machining times during computation. It indicates how much the off-machining time is on machine *i* before starting job *j*.

```
input: Ai, Bi, Ci, i=1..n
output: Xsum, Ysum
function three_machine([A1..An],[B1..Bn],[C1..Cn])
begin
  A:=A1; B:=B1; C:=C1; X:=A1; Y:=A1+B1;
  Xsum:=X; Ysum:=Y;
  for i=2 to n do
    A:=A+Ai;
    X:=max(0,A-B-Xsum);
    Xsum:=Xsum+X;
    B:=B+Bi;
    Y:=max(0,B+Xsum-C-Ysum);
    Ysum:=Ysum+Y;
    C:=C+Ci;
  end
end
```

Figure D6
Computation of off-machining times having three machines and off-machining times on the second and the third one

Property D1: *For all machines the sum of off-machining times can be computed by:*

$$d(i) = \sum_{j=1}^{n} D(i, j) . \qquad \text{(D5)}$$

Property D2: *It is reasonable to accept that the first machine does not produce any off-machining, so*

$$D(1, j) = 0 , \text{ for } j = 1,2,...,n . \qquad \text{(D6)}$$

Machining times measured on individual processors are summed up in vector **s**. Since any algorithm that uses off-machining time evaluation may set up arbitrary job sequence, it is reasonable to address matrices not directly, but through a permutation vector that stores a certain job sequence. The job permutation vector is denoted by **p** and it stores a possible permutation of $1,2,...,n$.

In some cases there are already-running jobs on the other machines when the first job starts to execute on the first machine. The expected finish times of these initial operations may influence the rest of the schedule, thus, they need to be built into the

model. The expected initial processing time on a single machine appears as a boundary condition, and is denoted by **b** for the whole machine set. If it is smaller than the accumulated initial waiting time on a particular machine, it is simply subtracted from the corresponding initial off-machining time value. If it is larger, it delays the schedule on that machine; the if-then rule indicates this special case.

```
input: M(m,n), p(n), b(m);
output: v;
storage: d(n), D(m,n), g(m);
function n_machine (M,p,b)
begin
  v:=0;
  for i:=1 to m do
    g(i):=b(i)-v;
    s(i):=M(i,p(1));
    d(i):=max(0, -g(i));
    D(i,1):=d(i);
    if g(i)<0 then g(i):=0;
    v:=v+M(i,p(1));
  end
  for j:=1 to n do D(1,j):=0;
  for j:=2 to n do
    for i:=1 to m do
      s(i):=s(i)+M(i,p(j));
      D(i+1,j):=max(0,s(i)+d(i)-s(i+1)-d(i+1));
      d(i+1):=d(i+1)+D(i+1,j);
    end
    s(m):=s(m) +M(i,p(j));
  end
  v:=0;
  for i:=1 to m do v:=v+d(i);
end
```

Figure D7

Algorithm to compute off-machining times to arbitrary number of machines and jobs

Property D3: *The computation cost of the algorithm is as follows:*

$$O(m+n+(n-1)(m-1)+m) = O(nm+m+1) \approx O(nm)\,. \qquad \text{(D7)}$$

■

Appendix E

Structure of the CD-ROM

The structure of the CD-ROM appendix can be found in Figure E1.

```
DOC
 |-Dissertation.pdf
 |-Dissertation.doc
 |-Theses.pdf
 \-Theses.doc
CHAPTER3
 |-Boltzmann_annealing
 |-Variable_temperature_bounds
 \-Figures
CHAPTER4
 |-Routing_simulator
 |-Routing_analyzator
 \-Figures
CHAPTER5
 |-Static_comparison
 \-Dynamic_architecture
README.txt
```

Figure E1
Structure of the CD-ROM appendix

The README.txt file contains further information about the software as well as the documents located on the disk.