Application of Markov decision processes to search problems

Eindhoven University of Technology

Department of Mathematics and Computing Science

Application of Markov Decision Processes to Search Problems

by

L.B. Hartman and K.M. van Hee

94/03

Computing Science Note 94/03
Eindhoven, January 1994

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. M. Philips
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

Application of Markov Decision Processes to Search Problems

Leo B. Hartman and Kees M. van Hee[1]
Department of Computer Science
University of Waterloo, Ontario

Abstract

Many *decision problems* contain, in some form, a NP-hard combinatorial problem. Therefore *decision support systems* have to solve such combinatorial problems in a reasonable time. Many combinatorial problems can be solved by a *search method*. The search methods used in decision support systems have to be *robust* in the sense that they can handle a large variety of (user defined) constraints and that they allow *user interaction*, i.e. they allow a decision maker to control the search process manually.

In this paper we show how *Markov decision processes* can be used to *guide* a *random* search process. We first formulate search problems as a special class of Markov decision processes such that the search space of a search problem is the state space of the Markov decision process. In general it is not possible to compute an optimal control procedure for these Markov decision processes in a reasonable time. We therefore, define several *simplifications* of the original problem that have much smaller state spaces. For these simplifications, *decompositions* and *abstractions*, we find optimal strategies and use the *exact* solutions of these simplified problems to *guide* a *randomized* search process.

The search process selects states for further search at random with probabilities based on the optimal strategies of the simplified problems. This *randomization* is a substitute for explicit backtracking and avoids problems with local extrema. These randomized search procedures are repeated as long as we have time to solve the problem. The best solution of those generated during that time is accepted. We illustrate the approach with two examples: the N-puzzle and a job shop scheduling problem.

1 Introduction

Many *decision problems* contain a NP-hard combinatorial problem. A *decision support system* (dss) that assists a decision maker, needs a *solver* for the underlying combinatorial problem, that computes an *approximation* of a solution, because in most cases there is not enough time to compute an exact solution. There are two good reasons to solve a combinatorial problem by a *search method*: search methods are *robust*, i.e. they can be adapted easily if the problem is changing a bit, and they allow *user interaction*, i.e. they enable a decision maker to do search steps manually. Search problems are studied both in the field of combinatorial optimization and in artificial intelligence and have many different formulations. Normally the solution of a search problem is an element in a finite set, called the *search space*, that satisfies some *criterion*. The elements of the search space are called *search states*. Often the search space is specified in an *implicit form*, i.e., is specified by a *method* to compute the *neighbourhood* of a search state, i.e., the set of search states adjacent in the search space. A neighbourhood is usually small compared to the total search space. The neighbours of a search state are determined in two steps: first an *action* is chosen and secondly a *transition function* computes the neighbour, based on the action and the current search state. Sometimes there is a one-one relation between actions and neighbours, in that case the selection of an action is identified with the selection of a neighbour.

Searching is the process of starting at some element, *selecting* an action, making a *transition* to the corresponding neighbour and repeating these steps until a *solution*, i.e. a search state that satisfies some criterion,

[1] Visiting professor from Eindhoven University of Technology, The Netherlands

is found. The criterion is often membership in a particular set and is tested by an algorithm implementing the characteristic function for the set. In other cases the criterion is expressed in terms of a (real-valued) function, called the *criterion function* and a solution is a search state such that the value of the criterion function has a minimal or maximal value. This formulation is usually called an *optimization problem*. In general a method for controlling the search process is called a *search method*. Strategies for optimization problems that may stop at a local optimum of the criterion function are called *local search methods*.

Most interesting search problems are NP-hard so it is unlikely that there is an efficient algorithm to solve them. We are, therefore, interested in good approximations that involve a reasonable amount of computation.

There are specific algorithms for specific classes of search problems, e.g., the traveling salesman problem and the graph colouring problem. We use the term *problem type* for these classes. (See for instance [Papadimitriou and Steiglitz, 1982] for a survey of optimization problem types.) While many researchers try to exploit all the knowledge they have about the structure of problem type to obtain an efficient algorithm, other researchers focus on what we call *robust methods* that work on a large variety of problem types. For example, branch-and-bound methods and *heuristic search* methods like the A* algorithm are robust in this sense. (See, for instance, [Pearl, 1984].) An interesting class of robust methods is based on analogies with physical or biological processes, for instance, *simulated annealing* [Aarts and Korst, 1991] and *genetic algorithms* [Goldberg, 1989]. In both cases the search method *randomizes* its choices to simulate a natural process. In [Aarts, Eiben and Van Hee, 1991] a general search method is presented that subsumes, for instance, genetic algorithms and simulating annealing. It is shown that the search processes of this method behave like Markov chains and this property is used to prove convergence of the search method. In these *randomized search methods* the neighbourhood is searched at random according to some distribution over its elements. In simulated annealing, for example, the neighbours are selected with probabilities depending on the difference of the criterion value of the neighbour and the starting search state. A nice feature of random search methods is that no explicit backtracking is necessary since there is always some chance of returning to search states already visited.

A feature that is usually not considered in search problems is the *cost of computation* in relation with the quality of the solution. (See [Hartman, 1990], [Mayer and Hansson], [Russell and Wefald, 1991], [Minton, 1988].) In many practical cases, however, it is not worthwhile to carry out a long search for the best solution; instead we would accept a suboptimal solution found at a reasonable cost. In the case of a criterion function, we always get a value if we stop searching. In case we are looking for a search state in a given set there is, in fact, also a criterion function, namely the characteristic function of the given set. If however, we stop without obtaining a search state of this set, we fail to solve the problem even though we might be very "close" to a solution. We therefore, concentrate on problem types where there is a criterion function that expresses the *quality* of each of the search states in the search space. For instance, in scheduling problems the search space is the set of partial schedules and there might not be a schedule that meets all our constraints. In practice, however, it is usually possible to give a criterion value to partial schedules as well.

We concentrate on robust search problems where we apply random search and where we consider the search cost relative to the solution criterion value.

A second approach to decision making is based on *Markov decision processes*. A Markov decision process is characterized by a *state space*, an *action set* for each state and a *transition probability*. In each state some *utility* (cost or reward) is obtained and the goal is to control the process in such a way that the expected total utility is maximized. Markov decision processes deal explicitly with actions whose outcome is probabilistic and so have wider application than do search problems. In practice, however, Markov decision processes have a serious drawback because the known algorithms to determine an optimal control procedure algorithms, *value iteration* and *policy iteration*, iterate over the whole state space and are limited to "small" problems (cf. [Denardo, 1982], [Ross,1983]). An advantage of Markov decision processes is that they offer a useful *framework* for the *specification* of decision problems.

We discuss how the theory of Markov decision processes applies to search problems. The similarity with a search problem is evident: the state space is the search space, the transition probability describes the

random probes over the neighbourhood of a search state, given an action. Further, the utility is the cost of choosing a neighbour if we continue the search and is the criterion value when we stop searching. In fact the Markov decision process that is equivalent to a search problem is a *controlled stopping problem*(cf. [Hordijk, 1986]).

Our approach is to solve the search problem by *guided random search*, which means that we simulate one or more random search processes and we use the best of them as our solution. A solution is a *search path*. If it contains cycles we may of course cut these cycles out to obtain a better search path. The actions in each step of the random search process are selected from distributions that are determined by a Markov decision process. We say that the search is *guided* by the Markov decision process. We call the Markov decision process that is equivalent with the search process the *equivalent process*. There are three ways of guidance.

- In each step we compute an optimal action for the equivalent process to the search problem; the computation of this action does not require iteration over the whole state space, but only computations over a part that can be reached from the current state in a limited number of steps. We call this the *exact method*.
- We define one or more *abstractions* of the equivalent process to the search problem. An abstraction has a much smaller state space than the equivalent process and actions for the abstracted process must be translated to the equivalent process. We use the actions of the abstractions for the search process. We call this the *abstraction method*.
- We *decompose* the state space of the equivalent process into several subsets of a "manageable" size and we define for each of these subsets a Markov decision process that has the same structure, except for the fact that we stop as soon as we leave the (sub) state space. We call these smaller systems *decompositions*. For these decompositions we compute the optimal strategy as soon as we reach one of their states and we use this strategy until we leave the state space of the decomposition. We call this the *decomposition method*.

Note that in the exact method we solve (a part of a) Markov decision process in every step of the search process, in the abstraction method we only solve some Markov decision processes before the search process starts and in the decomposition method we solve Markov decision processes only when we enter the state space of a decomposition.

The size of the state spaces of abstractions or decompositions should be such that the necessary computations can be carried out in internal memory. Since we have to maintain some functions over the states, a reasonable size is 10^5 states. The number of steps in a random search process should be large enough to be able to reach goal states (if they are defined) and the number of runs, i.e. the number of simulations of the search process, should be determined by the amount of computing time we may spend to solve the problem. We illustrate the results of some methods with numerical examples. The examples we have chosen are simple and well-known: the N-puzzle and a jobshop scheduling problem. The method is however intended for more complex problems for which no efficient algorithms are known. Note that in the guided random search process all kinds of constraints may be added while the guides are computed with models that might not be able to deal with these constraints.

The idea of approximating a Markov decision process with a very large state space by one with a much smaller state space is not new at all (see for instance [Norman, 1972]). However the use of these exact solutions for controlling a random search process for the original problem seems to be new. So we still solve the original problem and not another problem that "looks" similar.

2 Search Problems

In this section we formalize the notion of a *search problem*. A search problem is characterized by a 5-tuple

$$(S,\ A,\ T,\ c,\ r)$$

in which:

- S is a finite set, called the *search space*
- A is a set-valued function, such that $\mathrm{dom}(A) = S$ and for all $s \in S$ the set $A(s)$ is finite and is called the set of *allowable actions* in state s. There is one special action, called *stop* such that $\forall s \in S : stop \in A(s)$
- T is a function, called the *transition function*

  $$\mathrm{dom}(T) = \{(s,a) \mid s \in S \wedge a \in A(s) \backslash \{stop\}\}$$

  with

  $$\forall s \in S, a \in A(s) \backslash \{stop\} : T(s,a) \in S$$

  T is such that if we are in state s and we choose action $a \in A(s) (a \neq stop)$ then we move to state $T(s,a)$. If we take the action *stop*, the search stops.
- c is a function with $c \in S \rightarrow \mathbb{R}^+$, called the *cost* function. If we are in state s and we choose action $a \in A(s)$ and $a \neq stop$ we incur a cost $c(s)$. It is assumed that

  $$\exists \epsilon \in \mathbb{R}^+ : \forall s \in S \backslash \{stop\} : c(s) \geq \epsilon$$

- r is a function with $r \in S \rightarrow R^+$, called the *terminal reward* function. If we choose action *stop* in some state s, we receive a final reward $r(s)$.

A *search path* is a (finite) sequence of the following form:

$$\langle (s_0, a_0), \ldots, (s_{n-1}, a_{n-1}), (s_n, stop) \rangle$$

where $a_i \neq stop$ for $0 \leq i \leq n-1$ and

$$\forall i \in \{0, \ldots, n-1\} : T(s_i, a_i) = s_{i+1}$$

The *objective* is to find a search path with a maximal *total return*:

$$r(s_n) - \sum_{j=0}^{n-1} c(s_j),$$

if the search is started in some given *initial state* $s_0 \in S$. We call such a path a ***solution*** of the search problem. A *search method* chooses the next action in a state given a partial search path. This concept will be formalized in the next section. Note that our formulation differs from the more standard formulation of a search problem in the sense that we have not defined *goal states* that have to be reached. In fact our formulation is a *generalization* of the the standard formulation. To verify this let S' be the subset of S that contains the *goal states*. To enforce that we are looking for search paths that stop if and only if a goal state is reached, we define the terminal reward function as:

$$\begin{aligned} r(s) &= \text{'large' if } s \in S' \\ &= 0 \quad \text{otherwise.} \end{aligned}$$

Here the value 'large' denotes a value that is larger than (an upper bound for) the total cost of the set S'.

We have chosen this generalization because in many search problems it is impossible to find a search path to the goal set in a reasonable time or there might be no search path to the goal set at all. In these cases the decision maker is satisfied with a *partial* solution, i.e. a search state with some good quality measure. This quality measure is expressed by the terminal reward function. So we forget the concept of a goal and we just look for a path from the initial state to a final state such that the reward of the final state minus the cost of the visits to other states is maximal.

We next consider another generalization of the standard search problems: we introduce ***random search actions***. A random search action is a probability distribution over the set of actions in a state. We exclude the stop action from this distribution. Formally: for all $s \in S$ we define a finite set $Q(s)$ of probability distributions over $A(s)\backslash\{stop\}$. Hence $q \in Q(s)$ is a function:

$$q : A(s)\backslash\{stop\} \rightarrow [0, 1]$$

such that

$$\Sigma_{a \in A(s)\backslash\{stop\}} q(a) = 1$$

If we choose $Q(s)$ such that all of its members are *degenerate* distributions, i.e. all distributions that give one action probability one, then the set of all randomized search procedures enclosed the set of all ***deterministic*** search procedures. We conclude this section with the definition of a ***randomized search problem*** which is characterized by a 6-tuple:

$$(S,\ A,\ Q,\ T,\ c,\ r)$$

where $(S,\ A,\ T,\ c,\ r)$ is a search problem as defined above and, for $s \in S$, $Q(S)$ is a finite set of randomized actions over $A(s)$.

3 Markov Decision Processes

We now define one version of a Markov decision process and we summarize some old and well-known properties of these processes. Our discussion will be restricted to the class of Markov decision processes called *negative dynamic programs*. These processes have been extensively studied in [Strauch, 1966] (see also [Denardo, 1982] and [Ross, 1983]).

A *Markov decision process* is defined by a 4-tuple

$$(X,\ D,\ P,\ u)$$

in which:

- X is a finite (or countable) set called the *state space*
- D is a set-valued function, with $\mathrm{dom}(D) = X$ and for $s \in X$ the set $D(s)$ is finite and it denotes the set of *allowable actions* in state s
- P is a *transition probability*, i.e. P is a function with $\mathrm{dom}(P) = \{(s, a) \mid s \in X \wedge a \in D(s)\}$ and

$$\forall s \in X, a \in D(s) : \forall s' \in X : P(s' \mid s, a) \in [0, 1] \wedge \Sigma_{s' \in X} P(s' \mid s, a) = 1$$

- u is a real-valued function, such that $\mathrm{dom}(u) = \mathrm{dom}(P)$, called the *utility* function

The next concept we define is a *strategy*. Let a Markov decision process be given. A strategy is an infinite sequence $\pi_0, \pi_1, \pi_2, \ldots$ such that

$$\forall n \in Nat : \pi_n \in (X \times \overline{D})^n \times X \rightarrow \overline{D}$$

where $\overline{D} = \bigcup_{s \in X} D(s)$ and Nat is the set of natural numbers including 0. The meaning of a strategy is that it determines for each *path* of the form $\langle (s_0, a_0), \ldots, (s_{n-1}, a_{n-1}), s_n \rangle$ what the next action has to be, namely $\pi_n(s_0, a_0, \ldots, s_{n-1}, a_{n-1}, s_n)$. The set of all strategies is denoted by Π.

It can be proven that given a strategy π and a starting state s, a *stochastic process* is determined. We denote the probability distribution over the paths of this process by $\mathbb{P}_s^\pi$ and the expectation operator by $\mathbb{E}_s^\pi$. Let X_n denote the state of the system after the n-th transition and A_n the action chosen in that state for a (stochastic) process *starting in* s *with strategy* π. Then X_n and A_n are random variables with joint distribution $\mathbb{P}_s^\pi$ and $\langle X_0, A_0, X_1, A_1, \ldots \rangle$ is a stochastic process. The *expected total return*, denoted by $v(s, \pi)$ is defined by:

$$v(s, \pi) = \mathbb{E}_s^\pi [\sum_{n=0}^{\infty} u(X_n, A_n)]$$

We are interested in a strategy π^* that satisfies:

$$v(s, \pi^*) = \sup_{\pi \in \Pi} v(s, \pi)$$

Such a strategy is called *optimal*. A strategy π with the property that for all n π_n depends only on the last visited state is called a *stationary* strategy and if π_n depends only on n and the last visited state it is called *memoryless*. (Note that a stationary strategy is also memoryless.)

We define $v(s) = \sup_{\pi \in \Pi} v(s, \pi)$ and we call it the *value function*. Further we introduce similar functions for finite processes:

$$v_k(s, \pi) = \mathbb{E}_s^\pi [\sum_{n=0}^{k} u(X_n, A_n)]$$

and

$$v_k(s) = \sup_{\pi \in \Pi} v_k(s, \pi).$$

The following condition makes the Markov decision process a *negative dynamic program*:

$$\forall s \in X, a \in D(s) : u(s, a) \leq 0.$$

For negative dynamic programs with finite sets of allowable actions the following properties hold:

1. the value function v satisfies, for all $s \in X$:

$$v(s) = \max_{a \in D(s)} \{u(s, a) + \sum_{s' \in X} P(s' \mid s, a).v(s')\}$$

2. there exists a stationary strategy that is optimal for all initial states, and the strategy that always takes an action that maximizes the right hand side in the equation above is such a strategy.

3. the sequence of functions w_n on X, defined, for $s \in X$, by: $w_0(s) = 0$ and

$$w_{n+1}(s) = \max_{a \in D(s)} \{u(s, a) + \sum_{s' \in X} P(s' \mid s, a).w_n(s')\}$$

   satisfies:

   - $\forall n \in Nat, s \in X : w_n(s) \geq w_{n+1}(s)$
   - $\forall s \in X : v(s) = \lim_{n \to \infty} w_n(s)$
   - $\forall s \in X, k \in Nat : v_k(s) = w_{k+1}(s)$

4. for every initial state $s \in X$ and every $k \in Nat$ there is a memoryless strategy π such that

$$v_k(s, \pi) = v_k(s)$$

and this strategy selects in state s' at stage $n = k,\ k-1, \ldots, 1$ an action that maximizes

$$\max_{a \in D(s)} \{u(s,a) + \sum_{s' \in X} P(s' \mid s,a).w_{n-1}(s')\}$$

Note that the "stage" means the number of steps to go. It is an immediate consequence of these properties that

$$\forall s \in X, n \in Nat : w_n(s) \geq v(s)$$

so with w_n we compute an upperbound for the value function.

Approximating v by the sequence $\{w_n, n \in Nat\}$ is called *value iteration.*

4 Search Problems as Markov Decision Processes

We are now ready to verify that a search problem is, in fact, a Markov decision process. We will consider only randomized search problems because the deterministic search problems are a special case of this class.

Let a randomized search problem $(S,\ A,\ Q,\ T,\ c,\ r)$ be given. It defines the Markov decision process $(X,\ D,\ P,\ u)$ in the following way:

- $X = S \cup \{end\}$, where *end* is a new state that denotes the situation after the search process has been stopped
- $\forall s \in S : D(s) = Q(s) \cup \{stop\}$
- $D(end) = \{stop\}$
- $\forall s,\ s' \in S, q \in Q : P(s' \mid s, q) = \sum_{\{a \in A(s) \mid T(s,a) = s'\}} q(a)$
- $\forall s \in X : P(end \mid s, stop) = 1$
- $\forall s \in S, d \in D(s) : (d \neq stop \Rightarrow u(s,d) = -c(s)) \wedge u(s, stop) = r(s)$
- $u(end, stop) = 0$

There is only one action, *stop*, that is possible in state *end* and it keeps the system in state *end*. The reward r is obtained if and only if *stop* is chosen in a state different from *end*.

Note that we cannot apply the properties of negative dynamic programs because the utility u is not non-positive. However we will see later that it is straightforward to modify the reward function a bit, such that we obtain a negative dynamic program. From now on we only consider Markov decision processes that represent randomized search problems.

First we introduce the concept of a *stopping time*. Each strategy determines a stopping time τ which is a random variable like the X_n and A_n, and that satisfies:

$$(\tau = 0 \Rightarrow A_0 = stop) \wedge \forall k \in Nat \backslash \{0\} : \tau = k \Leftrightarrow (A_k = stop \wedge A_{k-1} \neq stop)$$

In case all $A_n \neq stop$ then $\tau = \infty$. Note that if $A_{n-1} \neq stop$ then all former actions differ from *stop* also. For $\tau \neq \infty$ we observe that X_τ is the state where action *stop* is chosen and so $r(X_\tau)$ is the terminal reward if the

search is stopped. For $\tau = \infty$ we define $r(X_\tau) = 0$. Now we are able to give a more convenient expression for the expected total return of a search problem, let $s \in S$ and $\pi \in \Pi$:

$$v(s,\pi) = I\!E_s^\pi[r(X_\tau) - \sum_{n=0}^{\tau-1} c(X_n)]$$

First we show how to transform this Markov decision process into a negative dynamic program.

Lemma 1 *Let $M = \max_{s\in S} r(s)$ and let $\tilde{r}$ be defined by*

$$\forall s \in S : \tilde{r}(s) = r(s) - M$$

Then we have $\forall s \in S, \pi \in \Pi$:

$$I\!E_s^\pi[r(X_\tau) - \sum_{n=0}^{\tau-1} c(X_n)] = I\!E_s^\pi[\tilde{r}(X_\tau) - \sum_{n=0}^{\tau-1} c(X_n)] + M$$

This lemma allows us to use $\tilde{r}$ instead of r, and with $\tilde{r}$ we have a negative dynamic program. It is easy to transform the results of the case with $\tilde{r}$ to the case with r by adding M. From now on we assume r is not positive. We call a Markov decision process having these properties a *search process* and a strategy is now called a *search method.*

So we may apply the results for negative dynamic programs. Properties 1 and 3 of the former section can be reformulated.

Lemma 2 *The functional equations for a search process have the following form:*

- *the value function of a search process satisfies for all $s \in S$:*

$$v(s) = \max\{r(s), \max_{q\in Q(s)}\{-c(s) + \sum_{a\in A(s)} q(a).v(T(s,a))\}\}$$

- *the sequence of functions w_n on S satisfy, for $s \in S$: $w_0(s) = 0$ and*

$$w_{n+1}(s) = \max\{r(s), \max_{q\in Q(s)}\{-c(s) + \sum_{a\in A(s)} q(a).w_n(T(s,a))\}\}$$

Remember that $\forall s \in X, n \in Nat : w_n(s) \geq v(s)$, so we have an upperbound on v. The next lemmas will be used to derive a *lowerbound.* First we introduce subsets of Π, the set of all strategies. With Π_n we denote the set of all strategies that stop before the n+1-th transition and

$$\tilde{\Pi} = \bigcup_{n=0}^{\infty} \Pi_n$$

the set of all strategies that stop. For these strategies we define a sequence of functions on X by:

$$\forall s \in X, n \in Nat : z_n(s) = \sup_{\pi\in\Pi_n} v_n(s,\pi)$$

The next lemma gives some intuitive clear properties for the functions z_n.

Lemma 3 *The functions z_n satisfy the following properties:*

- $z_0(s) = r(s)$
- *for* $n \in Nat$: $z_{n+1}(s) = \max\{r(s), \max_{q \in Q(s)}\{-c(s) + \sum_{a \in A(s)} q(a).z_n(T(s,a))\}\}$
- *for a given initial state and a given search time limit* k *there is a memoryless strategy* π *that chooses in search state* s *at stage* $n = k, k-1, \ldots, 1$ *a (randomized) action that maximizes:*

$$\max\{r(s), \max_{q \in Q(s)}\{-c(s) + \sum_{a \in A(s)} q(a).z_{n-1}(T(s,a))\}\}$$

which satisfies $v_k(s, \pi) = z_k(s)$.

The proof of this lemma is not trivial, but uses standard arguments of the field of Markov decision processes. The next lemma gives the desired bounding properties.

Lemma 4 *The following properties hold for all* $s \in S$:

- $\lim_{n \to \infty} z_n(s) = v(s)$
- $\forall m, n \in Nat : z_m(s) \leq v(s) \leq v_n(s) = w_{n+1}(s)$

We are now able to approximate v as precisely as we want. Suppose we have computed lower and upper bounds z_m and v_n. The next lemma tells us how to determine an optimal strategy; however for every state we have to perform this computation.

Lemma 5 *Let* $s \in S$ *be given, as well as* z_m *and* v_n. *If*

$$r(s) \geq \max_{q \in Q(s)}\{-c(s) + \sum_{a \in A(s)} q(a).v_n(T(s,a))\}$$

then choose stop and if;

$$r(s) \leq \max_{q \in Q(s)}\{-c(s) + \sum_{a \in A(s)} q(a).z_m(T(s,a))\}$$

then continue the search. In the second case choose random search action $\tilde{q}$ *if:*

$$\sum_{a \in A(s)} q(a).z_m(T(s,a) \geq max_{q \in Q(s)} \sum_{a \in A(s)} q(a).v_n(T(s,a)$$

If there is a case that is not covered by one of the conditions above we have to improve our the bounds by further iteration.

Note that it is not necessary to compute z_m and v_n for all $s \in S$ in order to determine the optimal randomized action in some state s because to compute for instance $v_n(s)$ we need for $k \in \{1, \ldots, n-1\}$ the values $v_k(s')$ for those s' that can be reached by T in $n - k$ steps. So this gives us an *exact method* to determine an optimal action for each search state.

Since, for all search methods, $\pi \in \grave{\Pi}$ and all $s \in S$ we have:

$$-M \leq v(s, \pi) \leq -I\!E_s^{\pi}[\tau].\epsilon$$

we may conclude

$$I\!E_s^{\pi}[\tau] \leq M/\epsilon$$

which gives an estimate for the required number of steps to find that v_n and z_n are very close.

5 Simplification and Guided Random Search

Although the *exact method* is elegant it can be very (computing) time consuming because the functions v_n and z_m have to be computed (as far as we need them) in every step of the random search process. The standard method for Markov decision processes, *value iteration*, is not applicable because it requires iteration over the whole state space.

We, therefore consider *simplified* Markov decision processes, translate their solutions to the equivalent process and use them in a *guided random search*, which is an approximation of the equivalent process.

Our main concern is the size of the search space; the simplified problems should have smaller search spaces. These smaller search spaces may be obtained by either *decomposition* or *abstraction*.

Decomposition splits the search space into subsets and for each of these subsets a search problem, a *decomposition*, is solved. Decompositions have almost the same structure as in the equivalent process, except that it stops when the *boundary* (defined below) of the state space is reached. Let $S'_1, \ldots, S'_n$ be a partioning of the search space S in non-empty sets. We then define the search space S_i of decomposition i by:

$$S_i = \{s \in S \mid s \in S'_i \vee \exists x \in S'_i, a \in A(x) : s = T(x, a)\}$$

Here $S_i \backslash S'_i$ is called the *boundary* of the state space of decomposition i and S'_i the *internal* state space of i. Further $A_i(s) = A(s)$, if $s \in S'_i$ and $A_i(s) = \{stop\}$, if s is in the boundary. The cost and reward functions are the same as in the equivalent process.

The guided search process uses the optimal actions of a decomposition as long as the system is in the internal state space of the decomposition. If the process enters the boundary an optimal search method for the entered decomposition is computed and that one is used until again a boundary is hit. Note that if the search process leaves an internal state space it enters an internal state space of exactly one other decomposition. Further note that the search process is only stopped in internal states.

Abstraction is a more general technique. Often we consider several abstractions simultaneously.

Let (S, A, T, Q, c, r) be a randomized search problem, then the randomized search problems $(S_i, A_i, Q_i, T_i, c_i, r_i)$ $(i \in \{1, \ldots, n\})$ are called *abstractions* if and only if there exist functions α_i and β_i and γ such that (for $(i \in \{1, \ldots, n\})$:

- $\alpha_i \in S \rightarrow S_i$ and $\mathrm{dom}(\alpha_i)$ is much larger than $\mathrm{rng}(\alpha_i)$.
- $\beta_i \in \overline{A} \rightarrow \overline{A}_i$ and $\forall s \in S, a \in A(s)$:

  $$\beta_i(a) \in A_i(\alpha_i(s)) \wedge T_i(\alpha_i(s), \beta_i(a)) = \alpha_i(T(s, a))$$

  where $\overline{A} = \bigcup_{s \in S} A(s)$ and $\overline{A}_i = \bigcup_{s \in S_i} A_i(s)$
- $\gamma \in \overline{A}_1 \times \ldots \times \overline{A}_n \rightarrow \overline{D}$ such that $\forall s \in S$:

  $$\bigwedge_{i \in \{1, \ldots, n\}} a_i \in A_i(\alpha_i(s)) \Rightarrow \gamma(a_1, \ldots, a_n) \in D(s)$$

So α_i and β_i form a *morphism* from the original problem to the abstraction i. Note that α_i and β_i map the states and actions of the equivalent process to the state and actions of the decompositions, and that γ combines the actions of the decompositions into an action for the equivalent process. Note that γ produces either the stop action or a distribution over $A(s)$, which may be degenerate. The functions β_i do not play a role in the search method, they just guarantee consistency between the equivalent process and the abstractions. The function γ can be constructed in many ways. Sometimes the actions of abstractions can be executed simultaneously (as in the job shop example) and in other cases they are actions for the equivalent

process themselves and so only one of them can be executed (as in the N-puzzle example). A reasonable requirement for γ is: $\gamma(stop, \ldots, stop) = stop$.

We did not specify any requirement for the terminal reward and the cost function of the abstractions. A natural requirement is that for all $s \in S$: $c_i(\alpha_i(s))$ is "close" to $c(s)$ and $r_i(\alpha_i(s))$ is "close" to $r(s)$. An obvious choice for c_i is:

$$c_i(s_i) = \frac{\sum_{\{s \in S \mid \alpha_i(s) = s_i\}} c(s)}{\#\{s \in S \mid \alpha_i(s) = s_i\}}$$

where $\#A$ is the cardinality of set A. A similar choice is possible for r_i.

Before we can start the guided random search we have to compute stationary search methods for all abstractions. Let us call these search methods π_i, such that π_i is a function that assigns to each state $s_i \in S_i$ either the value *stop* or a distribution over $A_i(s_i)$.

The guided random search procedure is the following algorithm:

$s \leftarrow s_0$;
$a \leftarrow a_0$; $\{a_0 \in A(s_0) \backslash \{stop\}\}$
list $\leftarrow \langle\rangle$;
while $a \neq stop$ do
    $s \leftarrow T(s, a)$;
    list $\leftarrow$ *list* $\circ (s, a, c(s))$;
    forall $i \in \{i, \ldots, n\}$ do
        if $\pi_i(\alpha_i(s)) = stop$ then $a_i \leftarrow stop$
        else a_i is a random drawing from $\pi_i(\alpha_i(s))$ fi;
    if $\gamma(a_1, \ldots, a_n) = stop$ then $a \leftarrow stop$
    else a is a random drawing from $\gamma(a_1, \ldots, a_n)$ fi
list $\leftarrow$ *list* $\circ$ (s, *stop*, $r(s)$).

(Here $\circ$ denotes concatenation of an element to a list.) The variable *list* contains a solution of the problem. To determine the total return we simply have to add the third components of the elements of the list. The *search path* is obtained from this list by deleting the last component from each element of the list.

It is possible that the search is *recurrent* with the search returning to a previously visited state. In that case cycles can be eliminated from a search path to obtain a better solution.

We may apply this random search procedure repeatedly and return the best path found when the available computing time is exhausted.

6 Example: the N-puzzle

The N-puzzle is a generalization of the 15-puzzle, a child's toy in which tiles numbered 1 to 15 are arranged in a 4 by 4 grid. An empty grid position allows tile in adjacent grid positions to be moved into that space, thus allowing the configuration of the puzzle to change, i.e., allows the puzzle to occupy different states. Solving the puzzle involves returning the puzzle to its goal state:

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

The experiments discussed in this section actually involve the 8-puzzle rather than the 15-puzzle version previously described. The value of $N = 8$ was chosen because of the convenient size of the search space. There are $9! = 362880$ distinct configurations of the 8-puzzle only half of which are reachable from the goal state, thus resulting in a state space with 181440 states. A search space of this size is large enough to be interesting but small enough to analyze and manipulate in explicit form. For a similar reason the 8-puzzle was chosen for other empirical studies [Hansson and Mayer, 1989, Russell and Wefald, 1991].

For $N = k^2 - 1$ for some k the state space S_N is such that

$$S_N = \{s | s \in \{1, ..., k\}^2 \rightarrowtail \{0, 1, ..., N\}$$

where $A \rightarrowtail B$ is the set of one-one mappings from set A into set B and 0 is the position without a tile. That is, members of S_N are permutations of {0,...,N}. For $N = 8$, states have the form

$$\{((1,1),1),((1,2),2),((1,3),3),$$
$$((2,1),4),((2,2),5),((2,3),0),$$
$$((3,1),7),((3,2),8),((3,3),6)\}$$

or more graphically

1	2	3
4	5	
7	8	6

The actions involve moving one of the tiles adjacent to the blank position into that position thus leaving the original position of the tile blank. We refer to these moves by the direction in which the blank moves and the set of possible moves is therefore {up, right, down, left}.

If we use row(s) and col(s) to refer to the first and second indices of the blank element in state s then

$$\text{left} \in A(s) \text{ if } \text{col}(s) > 1$$
$$\text{up} \in A(s) \text{ if } \text{row}(s) > 1$$
$$\text{right} \in A(s) \text{ if } \text{col}(s) < k$$
$$\text{down} \in A(s) \text{ if } \text{row}(s) < k$$

We distinguish a goal state such that row(s) = 1 and col(s) = 1 and $s(i,j) = (i-1)*k+j-1$ and refer to it by the term g_N. That is, g_N is the state

	1	...	$k-1$
k	$k+1$	...	$2k-1$
	...		
$N-k+1$	$N-k$	...	N

The transition function T is defined as follows:

$$T(s, \text{left}) = s'$$

such that

$$\forall (i,j) \in \{1, ..., k\}^2 : (((i,j) \neq (\text{row}(s), \text{col}(s))) \wedge (i,j) \neq (\text{row}(s), \text{col}(s)-1) \Rightarrow s(i,j) = s'(i,j)) \wedge$$
$$s'(\text{row}(s), \text{col}(s)) = s(\text{row}(s), \text{col}(s)-1) \wedge$$
$$s'(\text{row}(s), \text{col}(s)-1) = s(\text{row}(s), \text{col}(s)).$$

In short, the blank and the tile to its left exchange position. $T(s, m)$ for $m \in \{\text{up, right, down}\}$ is defined in a similar way. The state space S_N then is the set $\{g\}$ closed under application of the actions defined by A.

The *cost* of visiting a state is the same for all states:

$$\forall s \in S_N : c(s) = 1$$

The *reward* function is zero except for the distinguished goal state:

$$\begin{aligned} r(s) &= \rho \quad \text{if } s = g \\ &= 0 \quad \text{otherwise} \end{aligned}$$

where $\rho > 0$.

We define $q_i(a)$ for $i \in \{\text{up, right, down, left}\}$. For $\xi \in [0, 1]$

$$\begin{aligned} q_i(a) &= \xi & \text{if } a = i \text{ and } a \in A(s) \\ &= (1 - \xi)/(\#A(s) - 1) & \text{if } a \neq i \text{ and } a \in A(s) \end{aligned}$$

where $\#A(s)$ is the cardinality of $A(s)$. Then the set of randomized moves from state s is

$$Q(s) \subset \{q_{\text{up}}, q_{\text{right}}, q_{\text{down}}, q_{\text{left}}\}$$

with $q_i \in Q(s)$ iff $i \in A(s)$. For example, if s_1 is

1	2	3
4	5	
7	8	6

then for $q_{up} \in Q(s_1)$:

$$\begin{aligned} q_{\text{up}}(\text{up}) &= \xi \\ q_{\text{up}}(\text{right}) &= 0 \\ q_{\text{up}}(\text{down}) &= (1 - \xi)/2 \\ q_{\text{up}}(\text{left}) &= (1 - \xi)/2 \end{aligned}$$

We see that $G_N = (S_N, A, Q, T, c, r)$ is a randomized search problem. To obtain an ***abstraction*** we use proper subsets of the N tiles to specify equivalence classes of states in S_N. If we have $\delta \subset \{1, ..., N\}$ then the equivalence class of s with respect to δ is

$$[s]_\delta = \{s' \in S_N | s(i, j) \notin \delta \Rightarrow s'(i, j) = s(i, j)\}$$

The tiles in δ are permuted among themselves. Equivalent states are only required to agree on the position of tiles not in δ. δ names positions in s that are '*don't cares* or *wildcards* and that match any member of δ. Let S_δ be the set of equivalence classes induced by δ of states in S. This is the search space of the abstract problem. The goal in such an abstract search space is just $[g]_\delta$ where g is the goal state in S. The function α for this abstraction as

$$\alpha(s) = [s]_\delta$$

Since there are the same set of allowable moves for each abstract state as for the corresponding state in the original search problem G_N, we have simply for $a \in A(s) : \beta(a) = a$. In the experiments we only consider combinations of at most two abstractions. For the case of only one abstraction δ we have

$$\gamma_\delta(a) = a$$

and we call the randomized search problem $G_{N,\xi,\delta}$. For the case of two abstractions we define

$$\gamma \in \overline{A} \times \overline{A} \rightarrow [0,1]$$

as

$$\gamma_{\delta,\epsilon}(a_\delta, a_\epsilon) = \{(a_\delta, 1/2), (a_\epsilon, 1/2)\}$$

where a_δ and a_ϵ are the actions obtained from the abstractions induced by δ and ϵ. Thus we obtain a randomized search problem $G_{N,\xi,\delta\epsilon}$ that combines two abstractions defined by δ and ϵ.

6.1 Experiments

In order to measure the quantitative effect of *abstraction* we carried out a number of experiments with the 8-puzzle, varying the degree of abstraction, i.e., the number of wildcards, and the parameter ξ of the probability distributions that determine the outcome of a move. Each of the experiments consisted of applying an abstract strategy to 1000 games. Given an initial state, the strategy generated actions until either the goal state was reached or the limit of 4000 steps was exceeded. For each game an initial state was selected by following a sequence of 21 randomly chosen steps. Doubling back was prohibited so a move could not be undone by the immediately following move. It was possible, however, for a loop to occur and for steps to be retraced thereafter. (Note that, by empirical verification, there are no states in the concrete space more than 31 steps from the goal.) The random steps need not all be on paths away from the goal node and, therefore, it is possible for an optimal solution for an initial state generated in this way to be less than 21 steps long.

We carried out experiments with the following strategies:

- G_{8,ξ,δ_i} for $i = 3, ...7$ and for $\xi = \{.6, .7, .8, .9\}$
- $G_{8,\xi,\delta_i\epsilon_i}$ for $i = 3, ...7$ and for $\xi = \{.6, .7, .8, .9\}$

for $\delta_i = \{8-i, 8-i+1, ..., 8\}$ and $\epsilon_i = \{1, 2, ..., i\}$. For clarity the subscripts for δ and ϵ are omitted below. No confusion should arise. We refer to $G_{N,i,\delta}$ and $G_{N,\xi,\delta\epsilon}$ as *simple* and *combined* strategies, respectively. Thus, the experiments involved from 3 to 7 wildcards and a range of values for ξ for both the simple and combined abstract strategies.

Table 1 shows the relative size of the concrete and abstract state spaces as a function of the number of wildcards. An abstract state defines an equivalence class of concrete states. The size of this equivalence class represents the factor by which the concrete search space is compressed by the abstraction mapping. As the size of equivalence classes grow, the optimal action of the abstract state matches fewer of the optimal actions for concrete states in its equivalence class. This property, of course, reduces performance of guided random search.

Number of Wildcards	Size of Equivalence Classes
3	6
4	12
5	60
6	360
7	2520

Table 1: The size of equivalence classes of abstract states as a function of the number of wildcards.

The tables below summarize experiments for both simple and combined strategies. The data suggest that, for the conditions of the 8-puzzle, the guided randomized search performs significantly better than pure

random search and the combined strategy, $G_{8,\xi,\delta\epsilon}$, performs better than the corresponding simple strategy, $G_{8,\xi,\delta}$.

Table 2 shows the rate at which the guided random searches succeed in find the goal state. As a point of comparison the pure random search under the same conditions succeeds only 1.5% of the time. Even if the random search is allowed to continue for 100,000 steps, the success rate remains less than 17%. Thus from table 2 both the simple and combined guided searches perform much better.

		ξ .6	.7	.8	.9
i	3	87 / 100	87 / 100	88 / 100	88 / 100
	4	77 / 100	76 / 100	80 / 100	81 / 100
	5	81 / 100	75 / 100	62 / 100	50 / 100
	6	51 / 71	50 / 65	40 / 59	35 / 60
	7	11 / 20	11 / 20	11 / 20	11 / 20

Table 2: Percentage of successes as a function of the number of wild cards i and the randomization parameter ξ for the simple and combined strategies.

The percentage of successes is the percentage of the total number of games for which a solution was found within 4000 steps. The value ξ is the probability of taking the optimal move at each step and i is the number of wild cards. Each entry gives the percentage for the simple strategy $G_{8,\xi,\delta}$ and the combined strategy $G_{8,\xi,\delta\epsilon}$.

It is also clear that the combined strategy performs better than the simple strategy. The ϵ abstraction provides additional information about the direction of the goal even though moves provided the δ and ϵ abstractions are selected with equal probability.

The following tables summarize how quickly the combined strategy discovers solutions relative to the simple strategy.

		ξ .6	.7	.8	.9
i	3	21 / 20	17 / 17	15 / 15	13 / 13
	4	35 / 27	35 / 21	27 / 17	37 / 15
	5	677 / 43	669 / 37	1111 / 27	3015 / 23
	6	3697 / 393	- / 97	- / 85	- / 55
	7	- / -	- / -	- / -	- / -

Table 3: Number of steps within which 50% of the games are solved by the simple and combined strategies. and the combined strategy $G_{8,\xi,\delta\epsilon}$

Each entry shows the number of steps for the simple and combined strategy respectively. A "-" indicates that for the data collected not enough games were solved. The value ξ is the probability of taking the optimal move at each step and i is the number of wild cards. Each entry gives the percentage for the simple strategy $G_{8,\xi,\delta}$. We also did some experiments with decomposition. It turned out that the greater the state spaces of the decompostions are the better the guide works.

7 Example: job shop scheduling

The problem is defined by a 4-tuple (M, K, D, J):

1. M is a finite set of machine types
2. K is a function such that $K \in M \rightarrow N$ and $K(m)$ is the number of machines of type m.
3. D is a finite set of job identities
4. J is a set of jobs, in fact it is a function: $J \in D \rightarrow M^*$, so a job is a pair consisting of a job identity and a sequence of machine type, required for the subsequent operations for the job.

We introduce some notation. For sequences, *head* gives the head, *tail* gives the tail and *size* the size. Further we need:

$$rest(0, p) = p$$
$$rest(k, p) = tail(rest(k-1, p))$$

From these objects we derive a search problem (S, A, T, c, g):

1. $S \subset D \rightarrow M^*$ such that $s \in S$ if and only if $\forall n \in D : \exists k \in N : s(n) = rest(k, J(n))$
   The search state is, therefore, the amount of work still to be done.
2. $A(s) \subset D \not\rightarrow M$ such that

   $$\forall a \in A(s) : \forall n \in \mathrm{dom}(a) : a(n) = head(s(n)) \wedge \forall m \in M : \#\{n \in D \mid a(n) = m\} \leq k(n)$$

   An action is, therefore, an assignment of tasks to machine types such that the machine is appropriate for the task (i.e., $a(n) = head(s(n))$ and we do not exceed the number of available machines.
3. $\forall s \in S, a \in A(s)$:
   $\forall n \in D : (n \notin \mathrm{dom}(a) \Rightarrow T(s,a)(n) = s(n)) \wedge (n \in \mathrm{dom}(a) \Rightarrow T(s,a)(n) = tail(s(n)))$
4. $c(s) = 1$ for all $s \in S$
5. $r(s) = -max_{n \in D} size(s(n))$
   The "reward" is, in fact, a penalty for unfinished work. In the initial state it is $r(s) = -max_{n \in D} size(J(n))$ and in the 'goal' state it is just 0. Note that, since the reward function is already negative, it *does not* require a transformation.

An optimal search method corresponds to a *schedule* with a minimal *make span*. Finding such a schedule for this problem type is an NP-hard problem (see [Garey and Johnson, 1979] or [Lenstra and Rinooy Kan, 1978]) and it is reasonable to adopt an approximation to the problem type.

7.1 Experiments

We give the results for some very small problems in order to be enable the reader to calculate the optimal policies by hand.

We consider three strategies:

1. Decomposition by restricted look-ahead
2. Abstraction with "wildcard" machines
3. Abstraction by splitting in job sets

For the first strategy we consider the following jobshop problem.

A	B	C	B	A	C
A	C	A	B	A	B
B	C	B	A	C	B
B	A	C	A	B	C
C	B	A	B	C	A

In this table A, B and C represent machine types and each row represents a job. For each machine type the number of available machines is one. It is easy to verify that the optimal schedule takes 9 time units.

In the following table we compare purely random search with two decomposition strategies with three and five steps look-ahead. The entries in the table are the percentages of runs for different completion times.

schedule time	random	3 ahead	5 ahead
9	18.8	71.2	77.4
10	56.4	93.2	96.6
11	88.4	99.2	100

For the second strategy we consider only the first four jobs of the table above. Further the problem is the same. Each abstraction has only two machine types with one machine and "enough" of the third machine type. The three abstractions are combined. In the following table we display the percentages of runs for different completion times.

schedule time	random	abstraction
9	19	44.0
10	55.4	90.0
11	87.4	98.8
12	99.8	100

For the last strategy we added one job:

C	A	C	B	A	B

Further the problem is the same. For this problem the minimal makespan is 13. The abstractions are obtained by considering two groups of three jobs each: one group with the first three and the second group with the last three (including the extra job). The results are displayed in the following table.

schedule time	random	abstraction
13	35.0	41.4
14	77.4	85.5
15	93.8	98.6
16	99.4	99.8

8 Conclusions

Applying techniques of Markov decision processes to implicitly defined search problems is a challenge. We have shown that the application of such techniques to large implicitly defined search spaces is feasible. The process of abstracting or decomposing the search and "translating" the actions of the resulting Markov decision back to the original search problem provides guidance for a randomized search. In addition experiments show that randomization accomplishes the same effect as explicit backtracking. Randomization sufficiently perturbs the search process so that the process does not get stranded at local optima. The same effect can

be obtained by repeating the randomized search in the case of search problems involving actions that are irreversible.

We considered two techniques, decomposition and abstraction. The first one requires the solution of many Markov decision problems during the search process while the second one only requires the solution of some Markov decision problems before the search process starts. The choice of good abstractions is far from trivial, while the decompostions can be obtained easily.

While the experiments presented here allow us to claim interesting properties of the approach regarding two specific problems, the 8-puzzle and simple jobshop scheduling problem, additional experiments are required before more general statements can be made. Future directions include both experimenting with more realistic decision problems and comparing our approach with more traditional heuristic methods such as A^*. Such directions will extend our understanding of the empirical properties of abstraction and decomposition. It would also be useful to measure the frequency with which the optimal actions for the concrete states of an equivalence class coincide with optimal action of the corresponding abstract state. The 8-puzzle is small enough that such measurements are feasible. We would also like to obtain methods for automatically generating abstractions and decompositions. For decompositions this seems to be quite simple while we have no idea to do this for abstractions. Finally we are interested in analytical results, such as bounds on the expected value of guided search.

The main point of this paper is that the guided randomized search performs much better than pure random search. That is, an exact solution for an abstracted problem captures useful information about the original concrete problem.

Acknowledgement
The authors wish to thank Arno van Haastert for performing some of the experiments.

9 Appendix

In this section the proofs of the lemma's of section 4 are presented.

Lemma 1:

Proof:
Remember that S is finite, so M is defined properly. Using some well-known property of conditional probabilities, for the case

$$\mathbb{P}_s^\pi[\tau = \infty] \neq 0 \wedge \mathbb{P}_s^\pi[\tau < \infty] \neq 0$$

we obtain:
$\mathbb{E}_s^\pi[\hat{r}(X_\tau) - \sum_{n=0}^{\tau-1} c(X_n)] =$

$$\mathbb{E}_s^\pi[\hat{r}(X_\tau) - \sum_{n=0}^{\tau-1} c(X_n) \mid \tau < \infty]\mathbb{P}_s^\pi[\tau < \infty] + \mathbb{E}_s^\pi[\hat{r}(X_\tau) - \sum_{n=0}^{\tau-1} c(X_n) \mid \tau = \infty]\mathbb{P}_s^\pi[\tau = \infty]$$

Since $\forall s \in S: c(s) > \epsilon > 0$ we have that

$$\mathbb{E}_s^\pi[\sum_{n=0}^{\infty} c(X_n) \mid \tau = \infty] = \infty$$

So, if $\mathbb{P}_s^\pi[\tau = \infty] > 0$ then

$$\mathbb{E}_s^\pi[\hat{r}(X_\tau) - \sum_{n=0}^{\tau-1} c(X_n)] = -\infty$$

which is also the case if $\tilde{r}$ is replaced by r. On the other hand, if $I\!P_s^\pi[\tau = \infty] = 0$ we have:
$I\!E_s^\pi[\tilde{r}(X_\tau) - \sum_{n=0}^{\tau-1} c(X_n)] =$

$$I\!E_s^\pi[\tilde{r}(X_\tau) - \sum_{n=0}^{\tau-1} c(X_n) \mid \tau < \infty] I\!P_s^\pi[\tau < \infty] = I\!E_s^\pi[r(X_\tau) - M - \sum_{n=o}^{\tau-1} c(X_n)]$$

which gives the desired result.
□

Lemma 2:

Proof:
We only proof the first functional equation, the second one proceeds along the same lines. First note that $v(end) = 0$. So we may rewrite the functional equation for v as:

$$v(s) = \max\{r(s), \max_{q \in Q(s)} \{-c(s) + \sum_{s' \in X} P(s' \mid s, a).v(s')\}\}$$

If we substitute the definition of P, we only have to check that

$$\sum_{s' \in X} (\sum_{\{a \in A(s) | T(s,a) = s'\}} q(a)).v(s') = \sum_{a \in A(s)} q(a).v(T(s,a))$$

To verify this we rewrite the left hand side into:

$$\sum_{s' \in X} \sum_{\{a \in A(s) | T(s,a) = s'\}} q(a).v(T(s,a))$$

Note that every $a \in A(s)$ appears only once in this double summation so we may rewrite this formula into

$$\sum_{a \in A(s)} q(a).v(T(s,a))$$

which gives the desired result.
□

Lemma 3:

Proof:
The proof proceeds along the same lines as the proof of the similar properties for the functions w_n and v. See [Strauch, 1966] or [Ross, 1983].
□

Lemma 4:

Proof:
Fix some $s \in S$. First note that for $\pi \in \Pi_n$: $v_n(s,\pi) = v(s,\pi)$ because after stopping the utility is 0 forever. Since, for all $n \in Nat$: $\Pi_n \subset \Pi_{n+1}$ we have $z_n(s) \leq z_{n+1} \leq 0$. Therefore $\lim_{n \to \infty} z_n(s)$ exists. Let us call this limit $z(s)$. Note that:

$$\lim_{n \to \infty} \sup_{\pi \in \Pi_n} v(s,\pi) = \sup_{\pi \in \tilde{\Pi}} v(s,\pi)$$

Hence $z(s) = \sup_{\pi \in \tilde{\Pi}} v(s,\pi)$. To prove the first property we have only to show that the limit over $\tilde{\Pi}$ equals the limit over Π. To verify this, note that all strategies π for which $I\!P_s^\pi[\tau = \infty] > 0$, have $v(s,\pi) = -\infty$,

as we have seen in the lemma (1). Since there is at least one strategy that does better, namely the strategy that stops immediately, we may delete the strategies in $\Pi \backslash \tilde{\Pi}$ if we have compute the supremum. Hence $z(s) = v(s)$.

To prove the second property note that

$$z_n(s) = \sup_{\pi \in \Pi_n} v(s, \pi, n) \leq \sup_{\pi \in \Pi} v_n(s, \pi) = v_n(s)$$

□

Lemma 5:

Proof:
To prove this replace z_m and v_n by v in the formulas. Then we obtain exactly property (2) of negative dynamic programs as given in the section 3.
□

References

(Aarts and Korst, 1989) E.J.L. Aarts and J. Korst. *Simulated Annealing and Boltzman Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing.* Wiley, 1989.

(Aarts *et al.*, 1991) E.J.L. Aarts, A.E. Eiben, and K.M. van Hee. Global convergence of genetic algorithms: a markov chain analysis. In H.P. Schwefel and R. Maenner (eds.), editors, *Parallel problem solving from nature, Lecture Notes in Computer Science.* Springer Verlag, 1991.

(Denardo, 1982) E.V. Denardo. *Dynamic programming: Models and Applications.* Prentice Hall, 1982.

(Goldberg, 1989) D.E. Goldberg. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley, 1989.

(Hansson and Mayer, 1989) O. Hansson and A. Mayer. Probabilistic heuristic estimates. In *Proceedings of the Fifth Workshop on Uncertainty in Artificial Intelligence*, 1989.

(Hartman, 1989) L.B. Hartman. Decision theory and the cost of planning. Technical Report 355, Department of Computer Science, University of Rochester, Rochester, NY, March 1989.

(Hordijk, 1986) A. Hordijk. Markov decision processes and potential theory. *MC Tracts*, 1986.

(J.R. Garey, 1979) D.S. Johnson J.R. Garey. *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, 1979.

(Korf, 1985) Richard E. Korf. Macro-operators: A weak method for learning. *AI*, 26:35–77, 1985.

(Lenstra and Kan, 1978) J.K. Lenstra and Rinooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26:22-35, 1978.

(Minton, 1988) Steven Minton. Qualitative results concerning the utility of explanation-based learning. In *AAAI-88*, pages 564–569, 1988.

(Norman, 1972) J.M. Norman. *Heuristic procedures in dynamic programming.* Manchester University Press, 1972.

(Papadimitriou and Steiglitz, 1982) C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, 1982.

(Pearl, 1984) J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, 1984.

(Ross, 1983) S.M. Ross. *Introduction to stochastic dynamic programming.* Academic Press, 1983.

(Russell and Wefald, 1991) S. Russell and E. Wefald. *Do the Right Thing.* 1991.

(Strauch, 1966) R.E. Strauch. Negative dynamic programming. *Annals of Mathematical Statistics*, pages 871–890, 1966.

In this series appeared:

91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.
91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.

91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.
91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.

92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.
92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.
92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.

92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for Fω, p. 15.
92/26	T.H.W.Beelen W.J.J.Stut P.A.C.Verkoulen	A modelling method using MOVIE and SimCon/ExSpect, p. 15.
92/27	B. Watson G. Zwaan	A taxonomy of keyword pattern matching algorithms, p. 50.
93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.
93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.

93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness?, p. 24.
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions, p. 38.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.
93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.
93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.

93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxomomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Pathwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Kornatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.
94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen	The object-oriented paradigm, p. 28.