Stochastic dynamic programming for resource allocation

Ronnie Johansson, Christian Mårtenson, Robert Suzić, Pontus Svenson

FOI-R-- 1666 --SE
ISSN 1650-1942

Command and Control Systems
Methodology report

September 2005

Stochastic dynamic programming for resource allocation

FOI-R--1666--SE
ISSN 1650-1942

Command and Control Systems
Methodology report

September 2005

Issuing organization FOI – Swedish Defence Research Agency Command and Control Systems P.O. Box 1165 SE-581 11 Linköping	**Report number, ISRN** FOI-R--1666--SE	**Report type** Methodology report
	Research area code 4. C4ISTAR	
	Month year September 2005	**Project no.** E7097
	Sub area code 41 C4I	
	Sub area code 2	
Author/s (editor/s) Ronnie Johansson Christian Mårtenson Robert Suzić Pontus Svenson	**Project manager** Pontus Hörling	
	Approved by Thomas Kaijser	
	Sponsoring agency FM	
	Scientifically and technically responsible Pontus Svenson	

Report title

Stochastic dynamic programming for resource allocation

Abstract (not more than 200 words)

Resource allocation and management is an important part of the future network-based defence. In order to provide an adequate situation picture for commanders in the field, sensor platforms must be guided correctly and the needs of different users must be prioritized correctly.

Other important problems which require resource allocation include determining where soldiers should be posted in order to maintain peace in an area and determining where to place ambulances.

In order for sensor allocation to take opponent's future activities into account, detailed analysis of risks and consequences is needed. An important help for doing this is stochastic optimization.
One important method for such optimization is stochastic dynamic programming, which can be used to compute the best possible policy when we have a probabilistic model of the opponent's behaviour. The method can also be used when we have stochastic models for, e.g., reliability of our own communication networks.
In this report, we describe the basic methods and algorithms of stochastic dynamic programming and their relation to reinforcement learning. We also give a short overview of some relevant applications from the literature and suggest future work in this area.

Keywords

Stochastic dynamic programming, sensor management, resource allocation, simulation

Further bibliographic information	**Language** English
ISSN 1650-1942	**Pages** 18 p.
	Price acc. to pricelist

Utgivare FOI - Totalförsvarets forskningsinstitut Ledningssystem Box 1165 581 11 Linköping	**Rapportnummer, ISRN** FOI-R--1666--SE	**Klassificering** Metodrapport
	Forskningsområde 4. Ledning, informationsteknik och sensorer	
	Månad, år September 2005	**Projektnummer** E7097
	Delområde 41 Ledning med samband och telekom och IT-system	
	Delområde 2	
Författare/redaktör Ronnie Johansson Christian Mårtenson Robert Suzić Pontus Svenson	**Projektledare** Pontus Hörling	
	Godkänd av Thomas Kaijser	
	Uppdragsgivare/kundbeteckning FM	
	Tekniskt och/eller vetenskapligt ansvarig Pontus Svenson	

Rapportens titel (i översättning)

Stokastisk dynamisk programmering för resursallokering

Sammanfattning (högst 200 ord)

Resursallokering är en viktig del i det framtida nätverksbaserade försvaret. För att uppnå tillräckligt bra lägesbilder krävs det att sensorplattformar skickas till rätt ställen och att rätt prioriteringar görs mellan olika användares behov. Andra viktiga problem som kräver resursallokering är t ex att bestämma var soldater eller ambulanser ska placeras.

För att ta hänsyn till motståndarens framtida aktivitet när sensorstyrning görs krävs noggrann konsekvens- och riskanalys. Ett viktigt hjälpmedel vid sådan analys är stokastisk optimering.
En viktig metod för sådan optimering är stokastisk dynamisk programmering, som kan användas för att beräkna bästa möjliga handlingsalternativ när vi har probabilistisk kännedom om motståndarens uppträdande. Även i de fall där man har stokastiska modeller för t ex tillförlitlighet i våra kommunikationsnät kan stokastisk dynamisk programmering användas.

I den här rapporten beskriver vi grunderna för stokastisk dynamisk programmering samt dess relation till reinforcement learning. Vi beskriver också kort några relevanta tillämpningar från litteraturen samt ger förslag på framtida arbete inom området.

Nyckelord

Stokastisk dynamisk programmering, sensorstyrning, resursallokering, simulering

Övriga bibliografiska uppgifter	**Språk** Engelska
ISSN 1650-1942	**Antal sidor:** 18 s.
Distribution enligt missiv	**Pris: Enligt prislista**

Chapter 1

Introduction

Sensor allocation and management [1] is one of the most important problems in a future network based defence. In order to obtain adequate common operational pictures, sensor actions must be coordinated and planned so that fusion of their observations provides the maximal possible amount of information regarding the enemy.

An efficient system for such sensor management will consist of three important parts:

Commanders and analysts must have high-level planning tools available that help them decide which areas of the battlefield that they need information on. Such methods could be fully or semi-automatic, perhaps simulating futures and fusion systems in order to determine areas of interest [2]. They could also rely on mixed-initiative reasoning (*e.g.*, [3]), where the computer acts as a "discussion partner" for the commander, helping them to describe their plans and intentions and determining what sensor coverage to request.

Sensor platforms must also be able to optimize their paths and sensor parameters so that they can perform the tasks assigned to them. This requires sophisticated low-level platform optimization methods that determine, *e.g.*, flight paths so that the risk of loss is minimized and fuel consumption tolerable.

Finally, there must be a system that links the requests of the commanders to the services provided by the platforms and determines which requests are possible to fullfill. This system will act as a bridge between the commanders and the platforms [4, 5].

The support systems needed for these different stages differ in both the response-time needed and the degree to which microscopic details of, *e.g.*, sensor and platform characteristics need to be taken into account as well as the need for user interaction. In the systems used for high-level planning, there is an urgent need for fast response in order to facilitate experimenting and mixed-initiative reasoning. Such systems should aid humans in the same way as they are now aided by paper and pencil. The planning systems could be combined with impact analysis modules to allow for simulation of both enemy and own behaviour.

On the other hand, the algorithm that actually determines how the platform should move often does not need the same "faster than real-time" characteristics. Hence, they can use more detailed models and more sophisticated optimization algorithms. (Note that it is of course necessary in some cases to have "faster than real-time" behaviour also in the algorithms that control the platforms. This could be the case, for example, when a platform is facing threats to its existence.)

Another motivation for having different methods for high and low level planning is the new kinds of surroundings and enemies that face us in international operations. Operations in urban or semi-urban terrain entail new demands on the amount of sensors that are needed to adequately observe an area. The use of a large number of autonomous UxV's[1] will probably be necessary. These will require sophisticated control algorithms to control their search patterns. However, their use also makes it even

[1]Unmanned Air, Ground, or Under-water Vehicles

more important to have the kind of fast and simple-to-use high-level planning systems described above, since the large number of platforms makes it difficult for humans to keep track of all possible resource allocations. The planning systems need to take into account both how the output from the sensors should be used in the fusion systems and how this output will help the commander attain their goal/effect.

In this report, we describe a class of optimization methods that we believe may be useful both for the high-level planning tools and for the low-level platform optimization systems. The class of methods is called stochastic dynamic programming (SDP). SDP differs from normal dynamic programming in that it can be used when we do not have deterministic models for the environment and utility of performing various actions.

The main purpose of this report is to be a brief introduction and survey of SDP. We describe the background of SDP, including its relation to dynamic programming, and give the most important algorithms for solving SDP problems in chapter 2. As we were learning about SDP, we discovered the more general field of reinforcement learning. A brief survey of some of the most basic concepts in this area is contained in section 2.5

In chapter 3 we list and briefly describe some interesting applications of SDP for problems that we believe could be important for military applications. This list is far from complete and is meant to serve as a sample only.

We conclude in chapter 4 by briefly listing some reasons why SDP and related concepts might be especially useful in the new kinds of operations that the Swedish armed forces are facing.

Our interest in SDP was originally raised by the suggestion that it could be used to directly improve a sensor plan evaluation method that we previously implemented in the IFD03 Information Fusion demonstrator [6] and subsequently improved [2]. Since our knowledge of SDP was very limited, we decided that we needed to learn more about it. In order to do this as fast and efficiently as possible, we divided the subject between us and undertook to summarize the parts for each other. This report is the result of this process. We hope that it will prove useful for others who are interested in resource management.

Chapter 2

Problem formulation and algorithms

In this chapter we formulate a simple version of the stochastic dynamic programming problem and describe some of the algorithms used to solve it. We also briefly describe the standard dynamic programming problem and reinforcement learning. More detailed descriptions and reviews can be found in [7] and [8].

2.1 Problem formulation

Consider the situation of a decision maker (agent) acting in an environment. Every decision being made influences the environment and brings the decision maker into a new situation (state). This situation can be more or less valuable, which can be modeled mathematically by associating it with a value or a cost. The objective of the decision maker is to find an optimal policy of how to act in each given situation in order to maximize the accumulated value over time. The described problem is called a Markov Decision Process (MDP) and solutions are studied in the fields of Reinforcement Learning[1] and Optimal Control.

More formally, consider an agent acting in an environment during a sequence of time-steps ($t = 0, 1, 2...$). In a finite MDP the environment is always presented to the agent as being in one of a finite number of states. For each such state, $x_t \in \mathcal{X}$, the agent can make a decision by choosing from a finite set of actions/controls, $U(x_t)$. The consequences of performing action $u \in U(x_t)$ when in state $x_t = x$ is described by a transition probability,

$$p(x, u, x') = Pr(x_{t+1} = x' | x_t = x, u_t = u), \tag{2.1}$$

which for each state $x' \in \mathcal{X}$ gives the probability of ending up in that state. As can be seen, this model for determining the next state only depends on the immediate preceding state, which makes the process Markovian.

After performing an action the agent gets feedback in the form of a reward. The reward is a number, $R(x, u)$, that can depend on the old state and the action taken to get there. The agent's goal is to find an optimal policy $\pi = \{\pi_0, \pi_1, \pi_2, ...\}$ that maximizes the accumulated reward over time. A policy is a time-dependent function that gives the preferred action u for each state x.

If we have complete knowledge of the interactions with the environment, *i.e.*, we know all $p(x, u, x')$ and $R(x, u)$, the MDP can in theory be solved exactly using dynamic programming, which is described in the next section. To have such a complete model of the environment is seldom the case, and even if we did, the number of states and actions are often so large that making an exhaustive computation is not feasible. Fortunately, a number of well established approximation methods are available.

[1]Reinforcement Learning is also known as Neuro-Dynamic Programming, referring to its characteristics of combining methods inspired by dynamic programming and neural networks.

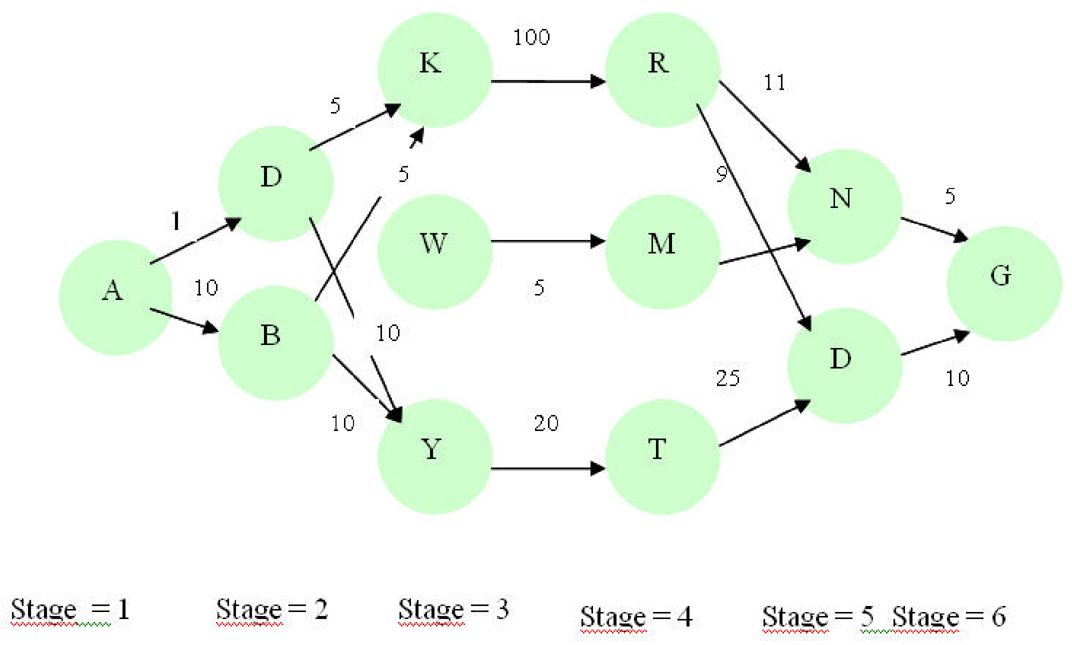

Figure 2.1: Example problem to be solved using dynamic programming. Links are labelled with their costs.

2.2 Dynamic programming

Here we briefly describe the concept of dynamic programming, which can be seen as the basis for all stochastic dynamic programming methods.

Let us start with an example where dynamic programming is used for solving a shortest path problem. The example is depicted in figure 2.1.

Our goal is to find the shortest path from node A to node G by following the arcs. The cost of moving between two nodes x and y is denoted $c(x, y)$ and shown in the figure; it is only possible to move between those nodes that have an edge between them. This requires a policy or *sequence of decisions* divided into stages where at each stage a decision is made. A decision in this case is the choice of moving from a node at one stage to another node at the following stage. Each decision, in this case movement, is associated with a certain cost (negative reward).

Mathematically we introduce a cost function that is constructed stepwise, starting from the end. The cost function $f_j(x)$ represents the minimal cost of moving from a generic node x to destination node by taking an optimal decision at each stage j. We must also store the actual node to visit in each stage, *i.e.*, the one that minimizes the cost.

The problem can be written as

$$f_j(x) = \min_{\text{nodes y in stage j+1}} \{c(x, y) + f_{j+1}(y)\}, \tag{2.2}$$

which represents a set of equations that is solved backwards.

In our example we start first with the base case $f_6(G) = 0$ meaning that there is no cost associated with moving from the goal state to the goal state. We proceed by calculating the cost function recursively:

At stage $j = 5$ we use our local cost functions. However, no decision choices have to be made because in both cases, the decision ”go to end state” is the only alternative. From equation 2.2, we get $f_{j=5}(N) = 5$ and $f_{j=5}(D) = 10$

Next come the nodes at stage 4, R, M , and T. Nodes M and T only have one possible successor, while for node R there are two alternatives, giving

$$f_4(R) = \min_{y \in \{N,D\}} \{c(R,y) + f_5(y)\} . \tag{2.3}$$

The result is that $f_4(R)$ is 16 and the policy at node R should be ”go to node N”.

We now perform the same procedure recursively for stage three and so on backwards. By calculating pairs of values and decisions at each level, we can determine which decision to make at each stage.

Dynamic programming can be used to solve a large number of different problems. Even if those problems can differ significantly from application to application there are some general (common) characteristics for all DP problems:

1. The problem can be divided into *stages* where decisions can be made at each stage
2. At each stage there are a number of states (in our previous example a node could be interpreted as certain state)
3. The decision at one stage transforms one state into a new state in the next stage.
4. Given the current state, the optimal decision for each of the remaining states does not depend on the previous states or decisions. (In our example it was not necessary to know how you got to a node, only that you did).
5. There exists a recursive relationship that identifies the optimal decision for stage j, given that stage $j + 1$ has already been solved.
6. The final stage must be solvable by itself. In deterministic dynamic programming both the immediate payoff (cost) and the next state given a certain decision are known. If there is *uncertainty* in state, pay-off or in the decisions executed then we have a *stochastic dynamic programming* problem. The basic ideas of determining stages, states, decisions, and recursive formulae still hold but can be calculated in different manner using different algorithms.

In the following section, we will see how the dynamic programming algorithm briefly described above can be augmented to handle also such stochastic problems.

2.3 Algorithms

In this section we will describe two of the most important algorithms used for solving stochastic dynamic programming, value iteration and policy iteration. We begin by formulating the problem in terms of policies.

Our aim is to find an optimal policy, $\pi^*(x)$. The policy, which is a function that maps states to actions, should be optimal in the sense that it maximizes the *cumulative reward* $V^\pi(x_0)$ of the acting agent given its starting state x_0,

$$\pi^* \equiv \arg\max_\pi V^\pi(x_0). \tag{2.4}$$

The cumulative reward function $V^\pi(x)$ for a policy $\pi(x)$ is sometimes expressed as a γ-*discounted* reward,

$$V^\pi(x_t) = R(x_t, \pi(x_t)) + \gamma R(x_{t+1}, \pi(x_{t+1})) + \gamma^2 R(x_{t+2}, \pi(x_{t+2})) + \ldots = \sum_{i=0}^{\infty} \gamma^i r_{t+i}, \tag{2.5}$$

which simply means that we sum the instantaneous rewards $r_t = R(x_t, \pi(x_t))$ for each t but weigh each term by a factor that decreases for future rewards.

Thereby, the reward function (and, hence, the agent) prefers rewards closer in time. Other alternatives are the *finite horizon* reward and *average* reward. The finite horizon reward corresponds to equation 2.5 with $\gamma = 1$ and an upper limit to the summation, *i.e.*, only the rewards from the next h steps are taken into account. The following discussion will merely concern the discounted, infinite-horizon reward.

Note that the expression in equation 2.5 can be rewritten as a recursive function

$$V^\pi(x_t) = R(x_t, \pi(x_t)) + \gamma V^\pi(x_{t+1}). \tag{2.6}$$

In equation 2.5, the reward function $R(x, u)$ and transition function $p(x_t, u, x_{t+1})$ (implicit in equation 2.5) are assumed to be deterministic and known. These assumptions can be relaxed. For example, the transition function may be known but non-deterministic, *i.e.*, $P : \mathcal{X} \times U \rightarrow \Pi(\mathcal{X})$, where a member of $\Pi(\mathcal{X})$ is a probability distribution over states. If so, we need to form an average over all possible states, and the value function becomes

$$V^\pi(x) = R(x, \pi(x)) + \gamma \sum_{x' \in \mathcal{X}} p(x, \pi(x), x') V^\pi(x'), \tag{2.7}$$

where $p(x, \pi(x), x')$ is the probability of ending up in state x' from state x after taking action $\pi(x)$.

In equation 2.4, we determined the optimal policy by calculating the cumulative reward function $V^\pi(x_0)$ for all possible policies and choosing the one that gave the maximum cumulative reward. Interestingly, it is often more convenient to work in terms of the *optimal value function* V^* instead of the optimal policy.

The optimal value function is defined as

$$V^*(x_0) \equiv \max_\pi V^\pi(x_0). \tag{2.8}$$

Note the subtle similarity to equation 2.4. In analogy to equation 2.6, we can also determine the optimal value function by solving the set of equations

$$V^*(x) = \max_u \{R(x, u) + \gamma \sum_{x'} p(x, u, x') V^*(x')\} \tag{2.9}$$

for all states x. (Note the similarity to the toy problem described in the previous section, equation 2.2.) Given the optimal value, it is of course possible to determine the optimal policy.

The approach using the optimal value function is used in the value iteration algorithm, while the formulation using the cumulative reward is more useful in understanding policy iteration.

2.3.1 Value iteration

Value iteration [9] is an algorithm that can be used to find the optimal policy for a given problem. The idea is to first establish the optimal value function $V^*(x) = V^{\pi^*}(x)$ and then to define the optimal policy $\pi^*(x)$ as

$$\pi^*(x) = \arg\max_{u \in \mathcal{U}} Q(x, u). \tag{2.10}$$

Here, $Q(x, u)$ is an estimate of the maximum discounted cumulative award that can be achieved starting from state x and using the first action u. The algorithm proceeds by first calculating $Q(x, u)$ from the current approximation to $V^*(x)$ and then changing the optimal value function for each state x by the maximal possible $Q(x, u)$. The algorithm stops when a sufficiently good policy has been found; it can be shown that it converges to the optimal value function.

The algorithm [8] is

```
initialize $V(\mathbf{x})$ arbitrarily
while policy not good enough do
  for all $x \in \mathcal{X}$ do
    for all $u \in U$ do
      $Q(x,u) \leftarrow R(x,u) + \gamma \sum_{x' \in \mathcal{X}} p(x,u,x')V(x')$
    end for
    $V(x) \leftarrow \max_{u \in U} Q(x,u)$
  end for
end while
```

It is not trivial to determine when to stop, *i.e.*, when a good enough policy has been found. Some bounds on how much additional iterations change the utility have been obtained, but in practice trial runs must be used in order to determine when to stop. It has been observed that the convergence rate slows considerably when the discounting factor γ approaches 1.

The value iteration algorithm lends itself very well to parallelization, since the updates of V can be done asynchronously. For more details on this as well as proofs of convergence, we refer the reader to [10, 11, 12].

It would be interesting to try to combine the value iteration algorithm with the concept of satisfying rationality (as described in [13]) and possibly user interaction. Very briefly, satisfying rationality compares the benefits and costs of various decision alternatives and provides a way of resolving some of the apparent paradoxes that appear in how humans actually make decisions. A decision support system could run the value iteration algorithm in the background while displaying the currently best found policies along with their benefits and costs to the user.

2.3.2 Policy iteration

Instead of determining the correct value function and optimizing it to determine the policy to use, the policy iteration algorithm constructs the optimal policy incrementally. The algorithm consists of two steps. First, an approximation of the value function for the current policy is computed. Then, this value function is used to improve the policy.

The algorithm is

```
choose an arbitrary initial policy $\pi'$
repeat
  $\pi \leftarrow \pi'$
  compute approximate value function $V^{\pi}(x)$ by solving equations 2.11
  $\pi'(x) \leftarrow \arg\max_{u \in U}\{R(x,u) + \gamma \sum_{x'} p(x,u,x')V^{\pi}(x')\}$
until $\pi = \pi'$
```

The equations to solve in the first step are (compare equations 2.7)

$$V^{\pi}(x) = R(x,\pi(x)) + \gamma \sum_{x' \in \mathcal{X}} p(x,\pi(x),x')V^{\pi}(x'), \tag{2.11}$$

Exact expressions for the algorithms worst-case behavior are missing.

Some work has been done on combining the two approaches, see [8]

2.4 A note on problem formulation

Our formulation of MDP as well as the algorithms we have briefly described assumes that several parameters are given, in particular that we know the transition probabilities $p(x,u,x')$ and reward function

$R(x, u)$. When trying to apply the methods described in this report to a military problem, it is important to remember that the most important source of errors is not approximations made in the algorithm, but rather inaccuracies in these parameters.

In order to be useful, any application of SDP and related techniques in the military domain must include also methods for determining and validating these parameters. Unfortunately, such methods are lacking in the currently available literature. Depending on the application at hand, this might be more or less difficult. For some problems, there might be some sort of *universality*, so that the solution to the problem does not depend sensitively on the problem parameters. In other cases, however, small changes of parameters might lead to significantly different results.

One possible approach to dealing with these problems is to use robust Bayesian methods (*e.g.*, [14, 15, 16]) to quantify the uncertainty in the parameters. This method, however, might prove difficult to implement in practice. Another possible approach is to apply learning methods also to the parameters. This could be done by running several implementations using different parameters simultaneously and then comparing their output [17, 18], or perhaps by adapting the indirect learning methods described below.

In practice, it might be simpler to accept that we may not be able to determine the optimal solution to the problem, and instead try to construct an algorithm that finds a good enough solution.

2.5 Reinforcement learning

In this section, we will briefly describe the more general method of reinforcement learning. Stochastic dynamic programming can be seen as a special case of reinforcement learning; it is included here for completeness. For more details on reinforcement learning, we refer the interested reader to [7].

When calculating the optimal policy using dynamic programming there is not much of real learning involved. The model of the environment is fully known from the beginning and the process can be performed entirely off-line, and is hence often referred to as *planning*. The large complexity of the planning process can often make the method infeasible. A way of getting around this is to use Monte Carlo techniques to calculate the value function from samples.

However, an agent acting in an new and maybe hostile environment very rarely has complete knowledge. Hence, other reinforcement learning methods are needed, based on direct agent-environment interactions and experiences built on trial-and-error. These kinds of methods can be divided into two categories, *direct* and *indirect learning*.

2.5.1 Direct learning

Direct learning is also known as model-free learning. The learning is completely based on trial-and-error and no learning of the environment model is obtained while performing a task. An important class of direct learning algorithms is Temporal-Difference Learning. The most basic representative of this type of method is called $TD(0)$. In $TD(0)$, after choosing an action according to the policy, $\pi(x_t)$, the next state x_{t+1} is observed and used directly to update the value function,

$$V(x_t) = V(x_t) + \alpha(R(x_t, \pi(x_t)) + \gamma V(x_{t+1}) - V(x_t)). \tag{2.12}$$

The parameter α controls the learning rate and should be slowly decreasing to guarantee convergence. By iterating this procedure, we get a better and better approximation of the value function. The drawback of this method is that in order to converge to the correct value function, we need to perform the update in equation 2.12 many times.

As its name suggests, $TD(0)$ is a simpler version of the more sophisticated $TD(\lambda)$ method. In this algorithm, we do not restrict ourselves to only updating the value function for the old state x_t. Instead, we calculate an *eligibility* function $e(x)$ for all states and update all states using

$$V(x) = V(x) + \alpha[R(x_t, \pi(x_t)) + \gamma V(x_{t+1}) - V(x_t)]e(x). \tag{2.13}$$

The eligibility is defined as

$$e(x) = \sum_{k=1}^{t} (\lambda\gamma)^{t-k} \delta_x^{x_k}, \tag{2.14}$$

where δ is the Kronecker delta. In other words, we use the reward gained by performing action $\pi(x_t)$ in state x_t to change the value function for all states x that have been visited in the past. The degree to which the value function of a state is updated depends on the discounting rate γ and λ. Note that for $\lambda = 0$ we get the $TD(0)$ method.

Another important Temporal Difference method is Q-learning. This method takes its name from the fact that it learns the expected reward function Q introduced in the discussion of value iteration. Recall that $Q(x, u)$ is the estimated reward of taking action u in state x and that the optimal policy is obtained by choosing actions so that this function is maximized. In contrast to value iteration described above we here need to learn the Q function, since our lack of knowledge of the environment and the relations between states and actions prevents us from calculating it. The method uses *experience* in the form of 4-tuples (x, u, r, x') describing the reward r gained by taking action u in state x as well as the resulting new state x' of the environment. The Q function is updated using a learning rate α as

$$Q(x, u) = Q(x, u) + \alpha(r + \gamma \max_{u'} Q(x', u') - Q(x, u)). \tag{2.15}$$

2.5.2 Indirect learning

Indirect learning is sometimes called model-based learning. The direct methods described in the previous section gave us an optimal policy and its value function directly, without providing us with approximations of the reward and transition functions. Sometimes, it might be necessary to obtain also these, in which case indirect learning is appropriate.

If we learn also the reward and transitions functions, we do not need as much experience to determine optimal policies. This means that indirect learning could be very important in military applications, where we typically want our agents/soldiers to perform as few actions as possible. After learning the transition and reward functions during the first steps of the agents, we can later use simulations instead of real-world experience to determine the optimal policy. In doing this, we must of course be careful not to extrapolate too much from the real experience.

The simplest indirect learning method is simply to store histograms of rewards and state-transitions seen by the agent. This method, however, does not make optimal use of the experience gained: it takes too long to obtain accurate histograms.

One of the most important indirect learning methods is the Dyna method [19]. Dyna combines updates of its knowledge of the environment with value iteration. Each iteration of the algorithm has four steps

- Update the model of the environment (reward function and transition function) using the experience tuple (x, u, r, x')
- Update $Q(x, u)$ as in Q-learning
- Select k state action pairs (x_i, u_i) at random and update $Q(x_i, u_i)$
- Use the Q function to choose which action to perform in the current state x'

Compared to normal Q-learning, Dyna requires considerably less experience to determine the optimal policy to use, but it uses much more computational resources.

An important improvement of the Dyna method is to select the k additional state-action pairs to update more intelligently. This can be accomplished by associating each state with a priority, and choosing the states with highest priority in the update step [8].

It would be interesting to see if methods for indirect learning could be combined with knowledge stored in an organizational memory system. Ideally, such a system would use the experience stored to determine the first few actions, and then gradually switch over to using the knowledge gained from the current operation.

Chapter 3

Applications

In this chapter we briefly review some of the most interesting applications of stochastic dynamic programming that we have found. The chapter is not meant to be an exhaustive listing of interesting material, but instead aims to provide a brief survey of some of the different types of problems relevant for military applications that SDP can be used to solve. A common characteristic of all the papers reviewed here is that they only apply their methods to simulated experiments; none of the methods have been implemented in a real system.

In [20], the problem of assigning *indistinguishable*, *unreliable*, and *non-renewable* resources to tasks is discussed. Multiple resources may be assigned to the same task. The resources are indistinguishable in the sense that they are unlabeled (*i.e.*, all resources have the same capabilities), unreliable in the sense that they may fail to complete the task that they have been assigned, and non-renewable in the sense that once employed they cannot be reused. It is, furthermore, assumed that success or failure of employed sensors, in the first stage of assignments, can be observed which should affect the second (and final) stage of assignments. Possible applications discussed in the paper include employment of sonobuoys for submarine detection. Note that the paper does not apply the method to a real problem: all test data is generated randomly.

The formal components of the problem are the N tasks to be handled by M resources. Each task i is assigned a reward value V_i that is related to the importance of the task and is obtained only if the task is completed. To each resource a cost C is assigned.

Assuming that the probabilities $p_i(1)$ that a resource completes task i in the first stage are independent, the probability that task i is not completed in the first stage is $P_S(i,1) = (1 - p_i(1))^{x_i(1)}$, where $x_i(1)$ denotes the number of resources assigned to task i in the first stage. After the first stage has been executed, we observe which tasks have completed. This information is stored in a vector $\omega \in \Omega = \{0,1\}^N$, where the ith component ω_i is 0 if the task was completed in stage one and 1 otherwise.

The solution sought is a *resource allocation* vector $\mathbf{x}(1)$, and a set of so called *recourse strategies* $\mathbf{x}(2,\omega)$. The recourse vector tells us which resources to allocate for which tasks in the second stage, given that we have observed a specific outcome ω of the first state.

We now need an expression for the probability that task i is not completed after the second stage. Using $p_i(2)$ to denote the probability that a resource completes task i in this stage, and again assuming independence, we get

$$P_S(i,2) = \sum_{\omega \in \Omega} P(\omega|\mathbf{x}(1))\delta^1_{\omega_i}[1 - p_i(2)]^{x_i(2,\omega)}, \tag{3.1}$$

where δ is the Kronecker-delta and we have introduced $P(\omega|\mathbf{x}(1))$ as the probability that we observe a specific ω given that we chose a certain resource allocation vector $\mathbf{x}(1)$ in the first stage. An approximation for $P(\omega|\mathbf{x}(1))$ can be readily obtained by again assuming independence of the events that a resource completes a task,

$$P(\omega|\mathbf{x}(1)) = \prod_{\{i:\omega_i=0\}} [1-(1-p_i(1))^{x_i(1)}] \times \prod_{\{j:\omega_j=1\}} (1-p_j(1))^{x_j(1)}. \tag{3.2}$$

The resource allocation vector and recourse strategy should be chosen so that the expected incomplete task value plus the expected cost of using the resources is minimized, *i.e.*, we want to find

$$\min \mathbb{E}\left\{\sum_{i=1}^{N} V_i P_S(i,2) + C\sum_{i=1}^{N}(x_i(1)+x_i(2,\omega))\right\}, \tag{3.3}$$

subject to constraints

$$\sum_{i=1}^{N}(x_i(1)+x_i(2,\omega)) \leq M, \forall \omega \in \Omega.$$

In order to formulate the problem as a stochastic dynamic programming problem, the authors first obtain a fast algorithm for determining the optimal recourse strategy $\mathbf{x}(2,\omega)$ and its associated optimal cost $J_2^*(\omega, M_2)$ assuming that M_2 resources remain after stage one has been completed. The algorithm uses an *incremental optimization* property of the problem, wherein a solution for a given number of resources M_2 is used to construct the solutions for any number $M > M_2$ of resources; please see the paper [20] for details. The solution for the first state can then be obtained by solving the recursive stochastic dynamic programming problem for the optimal value

$$J^* = \min_{\mathbf{x}(1)\in\{0,\ldots,M\}^N} \sum_{\omega} P(\omega|\mathbf{x}(1)) J_2^*(\omega, M - \sum_{i=1}^{N} x_i(1)) + C\sum_{i=1}^{N} x_i(1),$$

with the constraint $\sum_{i=1}^{N} x_i(1) \leq M$.

The problem can be solved approximately by incremental optimizing in a similar manner as the solution to the second stage. Note, however, that for the first stage the incremental optimizing property is only approximately valid, hence we only get an approximate solution (again, we refer to the paper for additional details on the specific algorithm). A severe problem with this solution, however, is that the time-complexity of the algorithm is exponential in the number of tasks.

To achieve a more efficient solution the authors replace the constraints of the original problem in equation 3.3 with a single average utilization constraint,

$$\sum_{i=1}^{N}\sum_{\omega\in\Omega} P(\omega|\mathbf{x}(1))(x_i(1)+x_i(2,\omega)) \leq M. \tag{3.4}$$

The restated problem is one of monotropic programming. The new solution is established where the solution is expressed in so called mixed *local strategies*. A mixed strategy is one that selects one of the pure strategies with some probability, and hence an infinite number of mixed strategies are available. A local strategy has the property that $x_i(2,\omega) = x_i(2,\omega_i)$. The authors motivate such a solution with the claim that given any pure strategy, there is a mixed strategy using only local strategies that achieves the same expected performance and the same expected resource use. Finally, the time complexity of the solution is $O\left((M+N)\log N\right)$.

The conclusion is that although the SDP solution finds optimal (or nearly optimal solutions) its time complexity is daunting, and the relaxed problem with a monotropic programming solution is considerably faster with little efficiency loss.

A similar application is studied in [21]. Here, the combinatorial explosion in the number of states that must be explored is handled by Monte Carlo simulations.

In [22], the problem of mission assignments in Joint Air Operations is approached using an approximate SDP-solution (ASDP). A number of striker and weasel aircrafts are to be assigned to different air packages. An air package is a composition of m strikers and n weasels sharing target and launch time. The targets differ in value and some are hidden during random time intervals. In addition, SAM-sites threaten the aircrafts on the route to and from the targets. The objective is to hit as many targets as possible while preserving own aircraft.

The actors in the scenario (the aircrafts, the SAM-sites and the targets) are modeled as finite-state, stochastic automatons. For instance, the states of an aircraft are $\{$BASE, INGRESS, EGRESS, DEAD$\}$ and the event set to move between the states is $\{$LAUNCH, THREAT ENGAGE, TARGET ENGAGE, LAND$\}$. An aircraft in state INGRESS that engages a threat can end up either DEAD or stay in INGRESS with some given probability. The complete model of the scenario is given by the composition of each actor's automaton.

The authors solve the mission assignment problem by first formulating it as an SDP. Then, to cope with the inherent complexity an approximation known as the Rollout Algorithm is applied. A Rollout policy is a one-step lookahead policy, meaning that all possible controls are evaluated only for the first step, $t = 0$. For $t = 1, ..., N$ the same pre-determined *base-line* policy is always used. Still, evaluating this policy can be quite complex and a Monte Carlo strategy is used to further reduce computations.

The ASDP approach is compared and shown superior to a heuristic using only the base-line policy mentioned above for a number of small example scenarios. The scenarios are of the "toy problem" variety, and deal with a limited number of aircraft. The paper should be seen as a proof of concept paper only; it remains to be seen if the method can also be applied to larger, more realistic scenarios.

The technical report [23] describes simulation experiments where reinforcement learning is used as a method of controlling cooperating UAVs performing multi-target tracking. The core problem is to find a good balance between the indiviual goal of each UAV to track high value targets and the cooperative team-goal of covering as many targets as possible. To manage this the state-space of each UAV must incorporate information of the distances to all targets and their respective values and the distances to the other UAVs. Handling these as separate parameters will yield a state-space too large to learn efficiently. Approximations are needed, and the authors describe a way of summarizing the information from the state-space in terms of two potentials. The first potential summarizes the target information,

$$\sum_{n=1}^{N} \frac{V_n}{1 + d_n^2} \tag{3.5}$$

where V_n is the value of target n and d_n is the distance to it. The second potential describes the spatial relations between the cooperating UAVs,

$$\sum_{m=1}^{M} \frac{1}{1 + d_m^2} \tag{3.6}$$

where d_m is the distance to UAV n.

The authors show that these potentials are sufficient for constructing the reward function for each state. This means that it is enough for each UAV to only consider these two potentials when determining its next move. Briefly, the individual UAV's should move towards locations with many targets, but move away from locations with many other UAV's. The paper combines fuzzy rules with the TD(λ) learning method and describes an experiment with 3 UAV's and 6 targets. The report describes the findings of the first phase of an ongoing project, and the authors conclusion is that their results indicate not only that the solution method is feasible but also adequate.

Next, we turn to two applications of SDP techniques to the optimal stopping problem. The optimal stopping problem refers to the problem of determining when an obtained solution is good enough that it is not worthwhile to continue searching for a better. An example of this problem type is that of a female individual choosing a mate. For each potential mate the individual meets, she has two options: to choose the current mate, or to consider another.

Military problems that can be mapped into this problem include determining who in a crowd to search or arrest, or, more generally, when a decision support system should stop planning or computing a situation-picture and present its conclusions to the user. One way of implementing this could be to run a stopping problem algorithm as a controller on top of the planning/analysis system. Methods for solving the stopping problem could also be used in mixed-initiative reasoning systems, where the computer needs to determine when to interrupt the reasoning of the user. This can be modeled as a stopping problem: the user should be interrupted when the probability that it will improve its current analysis or plan is sufficiently low. The main difficulty in applying stopping problem algorithms to these two problems is to find sufficiently accurate models to use for the reasoning processes (compare section 2.4); this makes the approach described in [24] extra interesting.

The first example of SDP techniques for the stopping problem is [24]. Typical approaches to the problem assume that an apriori distribution of potential mates are known. In [24], however, an SDP approach is used that allows the female (or more generally, the decision-maker) to learn the male distribution while inspecting the males. More specifically, the authors assume that the family of the distribution of males is known. In their example, $W_i, i = 1, \ldots$ are random variables that represent the quality (in some sense) of the ith male inspected by the female. All W_i are assumed to be governed by the same normal distribution $N(\mu, \sigma)$, where the parameters μ and σ are the mean and standard deviation, respectively. Each observed W_i incurs a cost c, which may correspond to the time consumed by observing or loss of energy. At each observation the female can update its knowledge of the distribution $N(\mu, \sigma)$. She updates the mean as

$$\mu_{n+1} = \mu_n + (\omega_{n+1} - \mu_n)/(n+1), \tag{3.7}$$

where ω_{n+1} is the outcome of W_{n+1}, *i.e.*, observed quality of the $n+1$th male. The standard deviation is subsequently updated in the following way:

$$\sigma_{n+1} = \sqrt{\frac{1}{n+1}\left[n(\sigma_n^2 + \mu_n^2) + \omega_{n+1}\right] - \left[\mu_n + \frac{\omega_{n+1} - \mu_n}{n+1}\right]^2}. \tag{3.8}$$

Now, let the Bellman function

$$V(\omega_n, \mu_n, \sigma_n, n) = \max\left\{\omega_n, \int_R V(\omega, \mu_{n+1}, \sigma_{n+1}, n+1) dF_{\mu_n, \sigma_n}\right\}, n = 2, 3, \ldots \tag{3.9}$$

be the *expected maximal gain* of the female. Here F_{μ_n,σ_n} is the distribution function for each W_i and R its domain. Moreover, μ_{n+1} is shown in equation 3.7 and σ_{n+1} in equation 3.8.

In the final solution, an intermediary function is utilized in a rewritten form of equation 3.9

$$V(\omega, \mu, \sigma, n) = \max\left\{\omega, -c + \mu + \delta_n(\sigma)\right\}, \tag{3.10}$$

and some of the values of $\delta_n(\sigma)$ for different σ and costs c are precalculated.

The resulting strategy is then for the female (or decision-maker) to select the nth mate, if the previous $n-1$ have been rejected and the quality w_n of the currently observed candidate mate is greater than $-c + \mu_n + \delta_n(\sigma_n)$. Otherwise, she decides to observe also the $(n+1)$th candidate.

The other stopping problem application [25] deals with the problem of finding a policy for selecting among a set of sensing techniques. The application in mind is one of *robotic assembly*, where the successful combination of parts by a robot manipulator can be described by a sequence of *discrete contact states*, denoted $\gamma_1 \rightarrow \gamma_2 \rightarrow \ldots \rightarrow \gamma_n$. The aim is to collect high quality sensory information while keeping the sensory processing time low.

In order to detect a state transition, an *event* e, a number of alternative sensing techniques, called *process monitors* (PMs), are at one's disposal. The output of a PM is a pair $\{e, C\}$. $C \in [0, 1]$ is a measure of the confidence of the recognised event $e \in E_\gamma$, where E_γ is a set of all possible events (i.e., state transitions) for state γ.

A PM encapsulates the combination of one or more sensor measurements and algorithms for analysing the information from sensors. The assembly task, in the paper, has three process monitors at its disposal. Two of them are fast, and the third one is slow, but accurate. The time required by the ith PM is called its cost c_i.

The problem of finding a policy for selecting a process monitor during robotic assembly is formalized as an optimal stopping problem and solved using SDP. The solution describes a confidence state space $\mathcal{S} = \{s_i\}$, an action space $\mathcal{A} = \{a_k\}$, a reward function $R(s_i, a_k)$ and state transition probabilities $P_{s_i,s_j}(a_k)$. The actions are a_1 ***terminate**, **i.e.**,* activate no more process monitors, and a_2 ***continue**, **i.e.**,* that the next "best" PM should be used.

The confidence state space contains different levels of confidence
$\mathcal{S} = \{0.0, 0.5, 0.75, 0.875, 0.9375, 0.96875, 0.984375, 0.9921875, s_\infty\}$. The reward function is defined by: $R(s_i, a_1) = \mathcal{R}(i)$ and $R(s_i, a_2) = \max_l c_l - c_n$, where $\mathcal{R}(i)$ denotes the i'th component of the vector $\mathcal{R} = [0, 0, 1, 2, 3, 4, 5, 6, 0]$.

The SDP problem then, for each ordering of PMs, becomes

$$V_n(\gamma, s_i) = \max \left[R(s_i, a_1), R(s_i, a_2) + \beta \sum_{j=1}^{N_s} P_{s_i,s_j}(\gamma, a_2) V_{n-1}(\gamma, s_j) \right], \tag{3.11}$$

where β is the DP discount factor.

Finally, the best policy $a_n(\gamma, s_i)$ at stage n is

$$a_n(\gamma, s_i) = \begin{cases} a_1, & \text{if } V_n(\gamma, s_i) = R(s_i, a_1) \\ a_2, & \text{otherwise} \end{cases} \tag{3.12}$$

The SDP algorithm is run off-line and calculates V_n and a_n for every possible ordering of the process monitors, *e.g.*, monitor orderings $1-2-3$, $1-3-2$, etc. The algorithm terminates when the optimal action is a_1 or when there are no PMs left. The method is well suited for real-time operation as only the lookup tables are used on-line.

The authors emphasize, however, that for applications where the parameters of the monitors are time-varying, the off-line generation of a lookup table is undesirable. For such systems, adaptation on the monitor parameters and an on-line algorithm are needed. This, however, reduces the real-time performance of the sensor selection method. The method is also most suitable for highly structured environments, such as industrial, where states and state transitions can easily be defined and anticipated.

Chapter 4

Discussion

As shown in many of the applications described in chapter 3 , SDP can be fruitfully utilized for sensor allocation. While most of the applications of reinforcement learning so far seem to be to autonomous systems, we hope that it and SDP could play an equally important role in higher-level optimization and planning systems.

In the new kinds of operations that face us in military operations other than war (MOOTW), we are less likely than before to meet opponents that follow a rigorous doctrine that we have detailed knowledge of. Instead, we may face adversaries that base their decisions on cultural or religious facts which we do not understand. This means that our models for the other actors in an operation must include stochastic simulation and learning. By implementing learning in the high-level planning systems, we might be able to help human analysts understand adversary actions that, at first glance, seem irrational. This could be done by implementing an organizational memory that describes situations encountered previously, and using this in an SDP algorithm to predict what the enemy is trying to do.

An interesting extension of many of the planning systems currently in use is to consider the problem of allocating surveillance and other resources when there are several antagonistic sides. This is a situation that will arise in, for example, peace-enforcing or nation-building operations. The planning systems will also need to provide support for coordinating actions and platform routes with several different coalition partners. Here, too, there is a need for stochastic simulation, since we will need to model uncertainties in the actions and capabilities of our partners.

Another area where planning tools based on stochastic dynamic programming might be important is logistics. Here it is important to take into account the randomness of both demand and supply of resources at various sites as well as of, *e.g.*, the possibility that transports are intercepted and destroyed.

In conclusion, we believe that stochastic dynamic programming, reinforcement learning and related algorithms are important parts of the toolbox needed to build the high-level planning, bridging and low-level optimization systems of future network-based defence systems.

Bibliography

[1] Ning Xiong and Per Svensson. Sensor management for information fusion - issues and approaches. *Information Fusion*, 3:163–186, 2002.

[2] Christian Mårtenson and Pontus Svenson. Evaluating sensor allocations using equivalence classes of multi-target paths. In *Proc 8th International Conference on Information Fusion*, 2005.

[3] J. Allen. Mixed initiative interaction. *Proc. IEEE Intelligent Systems*, 14:14, 1999.

[4] Ronnie Johansson and Robert Suzić. Bridging the gap between information need and information acquisition. In *Proc 7th Int Conf Information Fusion*, volume 2, pages 1202–1209, 2004.

[5] Robert Suzić and Ronnie Johansson. Realization of a bridge between high-level information need and sensor management using a common dbn. In *2004 IEEE International Conference on Information Reuse and Integration (IEEE IRI-2004)*, November 2004.

[6] J. Schubert, C. Mårtenson, H. Sidenbladh, P. Svenson, and J. Walter. Methods and system design of the IFD03 information fusion demonstrator. In *CD Proceedings of the Ninth International Command and Control Research and Technology Symposium, Copenhagen, Denmark*, pages 1–29, Washington, DC, USA, Track 7.2, Paper 061, 2004. US Dept. of Defense CCRP.

[7] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[8] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[9] R Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[10] D P Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, 1987.

[11] D P Bertsekas and J N Tsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, 1989.

[12] D P Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 1995.

[13] Henrik Kriisa. Ett adekvat beslutsstöd utan optimala ambitioner. C-uppsats, Uppsala Universitet, Filosofiska Institutionen, 2004.

[14] Stefan Arnborg. Robust Bayesianism: Imprecise and paradoxical reasoning. In *Proc 7th Int Conf Information Fusion*, pages 407–414, 2004.

[15] L. Ronnie M. Johansson and Robert Suzić. Particle filter-based information acquisition for robust plan recognition. In *Proc 8th Int Conf Information Fusion*, 2005.

[16] Johan Schubert and Per Svensson. Methodology for guaranteed and robust high level fusion performance: a literature study. FOI-D–0216–SE, 2005.

[17] H. Sidenbladh, P. Svenson, and J. Schubert. Comparing multi-target trackers on different force unit levels. In Proc. SPIE Vol. 5429 *Signal Processing, Sensor Fusion, and Target Recognition XIII*, pages 306–314, 2004.

[18] H. Sidenbladh, P. Svenson, and J. Schubert. Comparing future situation pictures. In *Proc 8th Int Conf Information Fusion*, 2005.

[19] R S Sutton. Planning by incremental dynamic programming. In *Proceedings of the 7th International Conference on Machine Learning*, page 353, 1991.

[20] David A. Castañón and Jerry M. Wohletz. Model predictive control for dynamic unreliable resource allocation. In *Proceedings of the IEEE Conference on Decision and Control*. IEEE, 2002.

[21] Milos Hauskrecht and Tomas Singliar. Monte-Carlo optimizations for resource allocation problems in stochastic network problems. In *Proceeding UAI*, page 305, 2003.

[22] D A. Castañon J. M. Wohletz and M. L. Curry. Closed-loop control for joint air operations. In *Proceeding of the American Control Conference*, volume 6, pages 4699–04, 2001.

[23] D. Vengerov H. R. Berenji and J. Ametha. Perception-based co-evolutionary reinforcement learning for uav sensor allocation. Technical report, Intelligent Inference Systems Corp, 2003.

[24] Vladimir Mazalov and Eduard Kochetov. Mate choice and optimal stopping problem. In *Proceedings of the 7th conference on probability theory and statistics*, pages 317—326, 1995.

[25] Geir E. Hovland and Brenan J. McCarragher. Dynamic sensor selection for robotic systems. In *Proceedings of the 1997 IEEE International Conference on Robotics and Automation (ICRA)*, pages 272–277. IEEE, 1997.