Solving Weapon Target Assignment Problem with Dynamic Programming

Topi Sikanen

55670A TFY N

November 17, 2008
Contents

1 Introduction 3

2 Dynamic optimization 4
   2.1 The basic dynamic optimization problem 4
   2.2 Dynamic programming 4
   2.3 Limited lookahead policies 6

3 Weapon target assignment 7
   3.1 Definition of the weapon target assignment problem 9
   3.2 Dynamic programming formulation 11
   3.3 Limited lookahead policies 12

4 Implementation of the dynamic programming algorithm 14

5 Examples 15
   5.1 Simple example 15
   5.2 Larger example 17
   5.3 Value of cooperation 18
   5.4 Evaluation of approximate dynamic programming 19

6 Discussion 20

References 21

A Usage of the solver 23

B Source code 23
   B.1 J.m 23
   B.2 Jbar.m 26
   B.3 Solver.m 26
   B.4 getIntersections.m 29
   B.5 getProb.m 32
1 Introduction

Weapon target assignment [9, 4, 7] is a special case of general resource allocation problem arising in defense related operations research problems. The task is to assign $N_w$ weapons to $N_t$ targets over $N$ stages (time steps) so that the total expected effect the attack (e.g., the destruction of as many targets as possible) is maximized. Another variant of the problem seeks to minimize the threat posed to defense assets by offense weapons.

Literature on the weapon target assignment (WTA) problem goes back at least to the 50s (see, e.g., [1, 4, 9] for a review of relevant literature). The WTA problem can be formulated in two ways [1, 9]. First way is to consider the static problem, where numbers and locations of all targets and weapons are known a priori. The second formulation is the dynamic problem formulation where the weapon assignments are made in discrete time steps. The dynamic version of the problem is considered in this research project.

Each assignment carries a certain risk of failure. Depending on the type of the weapon, this might mean that there is a risk of missing the target or a risk of being shot down by enemy defenses. Missiles and unmanned aerial vehicles (UAV) are increasingly used in modern warfare and in this assignment all weapons are considered to be unmanned aerial vehicles.

A successful target assignment can mean reduced risk for assignments made later. Thus, the cooperation of the weapons becomes important in planning the target assignment: A subset of the weapons could be used to open a pathway through enemy defenses, increasing the chances of successfully destroying the target they are defending. This also means that the order of attacks is important.

Various approaches have been taken in attacking the dynamic WTA problem. These include integer programming solutions [8], heuristic methods [6], genetic programming methods [5] and dynamic programming [1].

In this assignment a simple matlab demonstration of dynamic weapon target assignment in risky environment is presented. A solution method for task assignment for unmanned aerial vehicles (UAVs) developed by Alighanbari and How [2] is used to solve the dynamic WTA problem. The solution method is evaluated with respect to computational efficiency and quality of the solutions. Further a comparisons of approximate solution methods is conducted using Monte Carlo simulation. Flexibility of the methods for extending the problem, e.g., adding new constraints is also considered.
2 Dynamic optimization

In this section brief overview of the theory of dynamic optimization (esp.,
dynamic programming) is given. The treatment is kept short and only the
parts of theory needed for this assignment are discussed. The reader is re-
ferred to the wealth of literature on dynamic programing for a more thorough
treatment.

2.1 The basic dynamic optimization problem

The basic dynamic optimization problem consists of a discrete time dynamic
system and an additive cost function over time. The dynamic system is
modeled by equation of state

\[ x_{k+1} = f_k(x_k, u_k), \quad k \in \{1, \ldots, N - 1\}. \] (1)

Here \( k \) is the discrete time variable, \( x_k \) is the state variable of the system,
\( u_k \) is the control (or decision) variable, \( f_k \) is the (possibly time dependent as
indicated by the subscript) state function that controls the evolution of the
system variable and finally \( N \) is the horizon of the problem. For each state
variable \( x_k \) there is a cost \( g_k \) associated with choosing any control \( u_k \). This
cost function is assumed to be additive over time, such that the total cost
over \( N \) stages is

\[ g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k), \] (2)

where \( g_N(x_N) \) is the terminal cost for reaching the horizon in state \( x_N \). The
goal in dynamic optimization is to find a sequence of controls

\[ \pi = \{u_0(x_0), \ldots, u_N(x_N)\} \] (3)

that minimizes (or maximizes) the total cost over time.

2.2 Dynamic programming

Dynamic programming (DP) is an optimization method that considers deci-
sions made in stages [3]. A characteristic feature of dynamic programming
problems is that these decisions cannot be considered in isolation from previous decisions. A low cost of the present decision might mean a high cost for the decisions made down the line. The dynamic programming algorithm deals with this problem by utilizing the principle of optimality due to Bellman [3]. Roughly the principle of optimality states that if the decision to go from point A to C through B is optimal, then the decision to go from B to C directly must also be optimal.

First, we define a cost-to-go function $J_k$ that describes the optimal cost to go from state $x_k$ to stage $x_N$. The optimal cost-to-go from $x_k$ to $x_N$ assuming that all subsequent decisions are optimal is

\[
J_k^* (x_k) = \max_{u_k \in U_k} \left[ g_k(x_k, u_k) + \lambda J_{k+1}^*(f(x_k, u_k)) \right], \tag{4}
\]

where $0 < \lambda \leq 1$ is a time discount factor and $U_k$ is the set of feasible controls at state $x_k$. This recursion formula is also called the Bellmans equation of the problem. Solution of Equation (4) leads to closed-loop solutions where the controls $u_k(x_k)$ depend on the state of the system $x_k$. The opposite of the closed-loop control is open-loop control where the solution is fully determined from the initial state $x_0$. For deterministic problems the open-loop and closed-loop control lead to the same solution.

In some cases a analytical solution to Equation (4) can be derived. Usually however this is not the case. Fortunately Equation (4) is at least in theory easy to solve with the aid of a computer. The solution proceeds backwards from the final state $x_N$, and the optimal cost-to-go for each possible state is solved. A simple algorithm might be written as follows:

1. set $k = N - 1$
2. For each $x_k$ calculate

\[
J_k^* (x_k) = \max_{u_k \in U_k} \left[ g_k(x_k, u_k) + \lambda J_{k+1}^*(f(x_k, u_k)) \right].
\]

and

\[
u_k^* (x_k) = \arg \max_{u_k \in U_k} \left[ g_k(x_k, u_k) + \lambda J_{k+1}^*(f(x_k, u_k)) \right].
\]
3. if $k \neq 0$ let $k = k - 1$ and go to 2.

After the algorithm has finished we have the optimal controls $u_0^* (x_0), \ldots, u_{N-1}^* (x_{N-1})$ from any initial state $x_0$. 

The Dynamic Programming algorithm has a few shortcomings. The major difficulty is the curse of dimensionality [3]: Since we need to solve and store the optimal solution for each possible state at each possible state and time, the computational and memory requirements grow exponentially. Consider a system with two possible states represented by a state variable $x_k$ of ones and zeros: $x_k \in \{0, 1\}$, $k \in \{1, \ldots, N_t\}$. The dynamic programming solution for this problem is going to require $2^{N_t}$ memory locations to store the optimal costs $J^*_k$ to all the subproblems.

Furthermore, depending on the set of feasible controls, over which the minimization in 4 is carried out, might grow as a function of possible states. For these reasons the full DP-problem is seldom used. Instead approximations to the real cost-to-go function are employed. This is called suboptimal control of which there are roughly three types: limited lookahead methods, heuristic methods and roll-out algorithms [3].

### 2.3 Limited lookahead policies

An effective way to reduce the computational requirements of dynamic programming is to truncate the time horizon of the problem and use at each state a control derived by considering only limited number of stages. The simplest way is to use one-step lookahead [3], where only the next stage is considered. For one-step lookahead the optimal control $u_k$ at state $x_k$ and stage $k$ maximizes the equation

$$
\max_{u_k \in U_k} \left[ g_k (x_k, u_k) + \lambda \tilde{J}_{k+1} (f_k (x_k, u_k)) \right].
$$

Here $\tilde{J}_{k+1}$ is some approximation to the real cost-to-go function $J^*_{k+1}$. Selecting the approximation for the cost-to-go function $\tilde{J}_{k+1}$ is a key issue in implementing limited lookahead. Such approximations can derived from solutions to a simplified version of the problem [3]. Other approaches are the so called roll-out approach, where the cost-to-go is approximated by the cost of some suboptimal policy, and heuristic approach, where the cost-to-go function is approximated with a function whose parameters are set using some heuristic scheme [3].

Obvious extension to the one-step lookahead is two-step lookahead, where at stage $k$ and state $x_k$ the optimal control $u_k$ maximizes the equation
\[
\max_{u_k \in U_k} \left[ g_k (x_k, u_k) + \lambda \tilde{J}_{k+1} (f_k (x_k, u_k)) \right], \quad k \in \{1, \ldots, N - 1\}. \quad (6)
\]

Here \( \tilde{J}_{k+1} \) is obtained from one-step lookahead
\[
\tilde{J}_{k+1} (f_k (x_k, u_k)) = \max_{u_{k+1} \in U_{k+1}} \left[ g_k (x_{k+1}, u_{k+1}) + \lambda \tilde{J}_{k+2} (f_{k+1} (x_{k+1}, u_{k+1})) \right]. \quad (7)
\]

Limited lookahead could be extended to \( N_{\text{steps}} \)-step lookahead by simply truncating the problem at stage \( k + N_{\text{steps}} \). For \( N_{\text{steps}} \)-step lookahead the optimal control \( u_k \) at stage \( k \) and state \( x_k \) maximizes
\[
\max_{u_k \in U_k} \left[ g_k (x_k, u_k) + \lambda \tilde{J}_{k+1} (f_k (x_k, u_k)) \right], \quad k \in \{1, \ldots, N - 1\}. \quad (8)
\]

Here the approximate cost-to-go \( \hat{J}_k \) is obtained from
\[
\hat{J}_k = \max_{u_k \in U_k} \left[ g_k (x_k, u_k) + \lambda \hat{J}_{k+1} (f_k (x_k, u_k)) \right], \quad k \in \{1, \ldots, N - 1\},
\]
\[
\hat{J}_{k+N_{\text{steps}}} (x_{k+N_{\text{steps}}}, u_{k+N_{\text{steps}}}) = \tilde{J}_{k+N_{\text{steps}}} (x_{k+N_{\text{steps}}}, u_{k+N_{\text{steps}}}) \quad (9)
\]

The one-step lookahead approximation \( \hat{J} \) is again used to truncate the problem at stage \( k + N_{\text{steps}} \). Obviously using longer lookahead can quickly become computationally infeasible.

### 3 Weapon target assignment

The research on weapon target assignment goes back to 1950s and 60s [7] and the problem as received a fair bit of attention over the years, see [1] for a review of the literature. To this date exact solutions are only known for special cases of the problem.

The dynamic weapon target assignment problem considers assigning a set of \( N_w \) weapons to a set of \( N_t \) targets over several time steps (time step and stage are used interchangeably). The dynamic WTA problem can be formulated as a offensive or defensive version [9]. In the offensive version the goal is to assign the weapons to targets so that the expected damage to the targets is maximized. A target is any entity or collection of entities that has some military value (i.e., building, vehicle, fleet etc.). Each target has a value (or score) for successfully destroying the target. Each target also
Figure 1: Illustration of the weapon target assignment problem.

has a time discount factor which means that the value of the target when
destroyed now is greater than if it is destroyed later. In the defensive version
of the problem, the targets themselves are considered to be weapons that
threaten some defense assets. The goal is to minimize the threat posed by
these weapons. Only the offensive version of the WTA problem is considered
here.

The problem setting considered here is illustrated in Figure 1. Only two
types of targets are considered: high-value targets (marked with star in
Figure 1) and Surface to Air Missile (SAM) sites that are marked with a
cross in Figure 1. The weapons are considered to be UAVs. SAM-sites are
equipped with defensive capacity which makes the area around them risky
(risky areas are the shaded areas in Figure 1). When a weapon flies over the
risky area it may be shot down. Following [2], several assumptions are made
in order to make the problem tractable.

• All the weapons are considered to be identical and are launched from
  the same location.

• The time discount factor is assumed to be equal for all targets.
• Targets are assumed to be always destroyed when a weapon is assigned to them.
• The weapons are assumed to fly in straight lines.
• Targets don’t move and their locations are known.

Flight path of a weapon is shown in Figure 1 with a dashed line for safe part of path and a thick line for the risky part of path. Clearly it would be beneficial to assign some weapons to the SAM-sites guarding the high-value target before assigning the weapon to the high value target. This opens the door for cooperation. By cooperation, we mean that the probability of success for future assignments can be dramatically altered by current assignments, e.g., by destroying a SAM site [2].

3.1 Definition of the weapon target assignment problem

A mathematical formulation of the weapon target assignment problem is now given. Consider a set of $N_w$ weapons and a set of $N_t$ targets. We want to assign weapons to targets over $N$ stages. Each assignment carries a certain risk of failure that depends on the number of remaining SAM-sites on the weapons flight path. Each target is also assigned a value $s_i$, which might be different for each target depending on, e.g., whether they are SAM sites or high-value targets.

We denote the decision to assign weapon to target $i$ at stage $k$ by the variable $x_{ik} \in \{0, 1\}$. As was previously mentioned, this action carries with itself a risk of failure. The probability of success for this assignment is $p_i(k)$. The expected value for target $i$ is then $p_i(k) \lambda^k s_i x_{ik}$, where $s_i$ is the value of target $i$ and $\lambda^k$ is the time discount factor for target $i$. The WTA problem can then be formulated as an optimization problem [2]:

$$\max_{x_{i1}, \ldots, x_{iN}} \sum_{k=1}^{N} \sum_{i=1}^{N_t} p_i(k) \lambda^k s_i x_{ik}$$

s.t.

$$\begin{align*}
\sum_{k=1}^{N} x_{ik} & \leq 1 \quad \forall i \in \{i \ldots N_t\} \\
\sum_{k=1}^{N} \sum_{i=1}^{N_t} x_{ik} & \leq N_w
\end{align*}$$

$$x_{ik} \in \{0, 1\}, \quad i \in \{1 \ldots N_t\}, \quad k \in \{1 \ldots N\}.$$
The first inequality states that at most one weapon is assigned to a target through all stages. The second inequality states that no more weapons are assigned to the targets than we actually have in the beginning. The time discount factor \( \lambda \) controls the number of stages over which the optimal assignments take place. The probability of success \( p_i(k) \) depends on the flight path of the weapon and on the SAM-sites. Removing SAM-sites first to open up a pathway to a high-value target may bring better results and therefore this formulation is called the cooperative formulation.

The probability \( p_i(k) \) of successfully hitting target \( i \) can be formulated as

\[
p_i(k) = \prod_{j \in A_k} \tilde{p}_j.
\]

Here the set \( A_k \) denotes the set of active (here “active” means simply that no weapon has been assigned to that site yet) SAM sites the weapon must fly over in order to hit the target and \( \tilde{p}_i \) denotes the probability of not being shot down by SAM site \( i \). The survival probability for flying over a SAM site \( i \) for \( d_i \) units of length is defined as

\[
\tilde{p}_i = \hat{p}^{d_i}.
\]

Here \( 0 < \hat{p}_i \leq 1 \) is the survival probability for flying a unit distance within the SAM sites range. In the future if all the targets are to have the same probabilities \( \hat{p}_i \), we refer to this common probability as \( p_s \).

The non-cooperative formulation can be derived by dropping the time dependence of \( p_i(k) \). This means that the effect of the other weapons on the environment is ignored. The first line in Problem (10) then becomes

\[
\max_{x_{ik}} \sum_{k=1}^{N} \sum_{i=1}^{N_t} p_i \lambda^k s_i x_{ik}.
\]

This is called the non cooperative formulation because destroying targets doesn’t affect the survival probability of following weapons. The time-discount factor \( \lambda \) then forces all the weapons to be assigned in the next stage. Solving (11) thus reduces to a sorting problem, where we choose \( m_k \) targets with the largest expected values \( p_i \lambda s_i \) and assign weapons to those targets.
3.2 Dynamic programming formulation

In practice, solving Problem (10) can be difficult. The time-dependence of risk leads to a solution in stages and so the Dynamic Programming algorithm is a natural solution method. Here a brief summary of the Dynamic Programming algorithm for the Problem (10) developed by Alighambari and How [2] is presented. Indeed if the time dependence of risk were dropped from the problem definition the time discount of objective values would force all the weapons to be allocated in the first stage.

To proceed with the solution several additional definitions are needed. The state of the system at given stage $k$ consists of the set of remaining targets $r_k$ and the number of remaining weapons $m_k$. Since targets are assumed to be destroyed once a weapon has been assigned to them the set of remaining targets is simply

$$ r_k = \{i| \text{site } i \text{ is active}\} . \quad (12) $$

In this formulation of the WTA problem, a site is considered “active” if a weapon has not been assigned to it during previous stages. At the first stage, the set of remaining targets is simply $r_0 = \{1 \ldots N\}$. The control variable $u_k$ is defined to be the set of targets to hit at the stage $k$:

$$ u_k = \{i| \text{a weapon is assigned to target } i \text{ at stage } k\} . \quad (13) $$

At each stage, at most $m_k$ weapons can be assigned to the remaining targets and weapons can be assigned only to active sites. The set of feasible controls $U_k$ can be defined with the power set (the set of all subsets of a set) of $r_k$, $\mathcal{P}(r_k)$ as

$$ U_k = \{u \in \mathcal{P}(r_k)| |u| \leq m_k\} . \quad (14) $$

The state equation of the system is

$$ r_{k+1} = r_k \setminus u_k, \quad (15) $$

$$ m_{k+1} = m_k - |u_k| . \quad (16) $$

Here $|u_k|$ denotes the size (number of elements) of the set $u_k$. There is also an implicit assumption made that all the assigned targets are destroyed or at least we don’t attempt to hit the same target twice. This follows from the formulation where the list of targets at stage $k+1$ is simply the set of remaining targets minus the targets we decided to hit: $r_k \setminus u_k = \{i \in r_k|i \notin u_k\}$ and the fact that the time discount factor forces at least one
weapon to be assigned at each stage. This also means that the horizon of
the problem $N$ is finite and equal to the minimum of $N_t$ and $N_w$

$$N = \min (N_w, N_t).$$

(17)

The value associated with choosing the control $u_k$ is the expected value of
the assigned targets

$$g_k (r_k, u_k) = \sum_{i \in u_k} p_i (r_k) s_i.$$  

(18)

Here the probability of success $p_i (r_k)$ is written as function of the state $r_k$ instead of $k$ to highlight its dependence on the number of active sites.

We can now write the bellmans equations of Problem (10).

$$J_k^* (r_k, m_k) = \max_{u_k \in U_k} \left[ g_k (r_k, u_k) + \lambda J_{k+1}^* (r_k \setminus u_k, m_k - |u_k|) \right], \quad k \in \{1, \ldots, N-1\},$$

(19)

$$J_N^* (r_N, m_N) = 0, \quad g_k (r_k, u_k) = \sum_{i \in u_k} p_i (r_k) s_i, \quad k \in \{1 \ldots N\}.$$  

To get the optimal sequence of weapon assignments we must solve (19) for $r_o$ and $m_o$, which are simply the set of all targets and the number of available weapons $N_w$ respectively. Solution to (19) gives the optimal sequence of weapon assignments $u^*_0, \ldots, u^*_{N-1}$. The optimal cost for the optimal solution beginning at state $(r_0, m_0)$ is $J_0^* (r_0, m_0)$.

### 3.3 Limited lookahead policies

When the number of weapons and targets grows, the full dynamic programming solution can quickly become infeasible. For this reason approximate solutions can be desireable. Next we consider limited lookahead policies. For one-step lookahead, at stage $k$ and state $(r_k, m_k)$ the optimal control $u_k$ maximizes the equation

$$\max_{u_k \in U_k} \left[ g_k (r_k, u_k) + \lambda \bar{J}_{k+1} (r_k \setminus u_k, m_k - |u_k|) \right], \quad k \in \{1, \ldots, N-1\},$$

$$\bar{J}_{N} (r_N, m_N) = 0,$$

(20)

where $\bar{J}_{k+1}$ is some approximation of the true cost-to-go function. Here we follow the method proposed by Alighambari and How [2]: The approximate
cost-to-go function $\bar{J}_k (r_k, m_k)$ is the solution to the non-cooperative (i.e. non time dependent) formulation of Problem (10)

$$\max_{x_{ik}, \ldots, x_{iN}} \sum_{t=k}^{N} \sum_{i=1}^{N_t} p_i \lambda_i^t s_i x_{it}$$

s.t.

$$\sum_{t=k}^{N} x_{it} \leq 1 \quad \forall i \in r_k$$

$$x_{it} = 0 \quad i \notin r_k$$

$$\sum_{t=k}^{N} \sum_{i=1}^{N_t} x_{it} \leq m_k$$

$$x_{it} \in \{0, 1\}, \quad i \in r_k, \quad t \in \{k, \ldots, N\}.$$  \hspace{1cm} (21)

Here time variable $t$ is used to emphasize that the problem is solved from stage $k$ onwards. The first inequality states that only one weapon can be assigned to each remaining target. The second inequality ensures that weapons are assigned only to active targets. The third inequality states that we can assign at most $m_k$ weapons to targets. The control variable $u_k$ is related to the state variable in (21) by

$$u_k = \{i | x_{ik} = 1\}.$$ \hspace{1cm} (22)

Similarly a two-step lookahead can be defined. For two-step lookahead, at stage $k$ and state $(r_k, m_k)$ the optimal control $u_k$ maximizes the equation

$$\max_{u_k \in U_k} \left[ g_k (r_k, u_k) + \lambda \hat{J}_{k+1} (r_k \setminus u_k, m_k - |u_k|) \right], \quad k \in \{1, \ldots, N - 1\},$$ \hspace{1cm} (23)

where the approximate cost-to-go function $\hat{J}_{k+1}$ itself is obtained by one-step lookahead.

$$\hat{J}_{k+1} (r_{k+1}, m_{k+1}) =$$

$$\max_{u_{k+1} \in U_{k+1}} \left[ g_{k+1} (r_{k+1}, u_{k+1}) + \lambda \hat{J}_{k+2} (r_{k+1} \setminus u_{k+1}, m_{k+1} - |u_{k+1}|) \right],$$

$$\hat{J}_N (r_N, m_N) = 0.$$ \hspace{1cm} (24)

This approach can be generalized to $N_{\text{steps}}$ lookahead where at stage $k$ and state $(r_k, m_k)$ the optimal control $u_k$ maximizes

$$\max_{u_k \in U_k} \left[ g_k (r_k, u_k) + \lambda \hat{J}_{k+1} (r_k \setminus u_k, m_k - |u_k|) \right], \quad k \in \{1, \ldots, N - 1\},$$ \hspace{1cm} (25)
where

$$\hat{J}_{k+1}(r_{k+1}, m_{k+1}) =$$

$$\max_{u_{k+1} \in U_{k+1}} \left[ g_{k+1}(r_{k+1}, u_{k+1}) + \lambda \hat{J}_{k+2}(r_{k+1} \setminus u_{k+1}, m_{k+1} - |u_{k+1}|) \right],$$

$$\hat{J}_{k+N_{\text{steps}}}(r_{k+N_{\text{steps}}}, m_{k+N_{\text{steps}}}) = \hat{J}_{k+N_{\text{steps}}}(r_{k+N_{\text{steps}}}, m_{k+N_{\text{steps}}}),$$

$$\hat{J}_N(r_N, m_N) = 0. \quad (26)$$

Selecting larger value for $N_{\text{steps}}$ leads to solutions that approximate the optimal solution more accurately. However the computational cost also grows very fast. It will be seen later that lookahead longer than two steps is probably not worth the computational cost.

4 Implementation of the dynamic programming algorithm

The solution algorithm was implemented as a matlab (version 2008a) program. The actual program is broken into a user interface part and solver part. Only an outline for the solution methodology is presented here, the actual source code for the solver part is presented in the appendices.

The heart of the solution algorithm is the cost-to-go function $J_N$. For each stage we calculate the optimal cost-to-go $\hat{J}_k^*$ and the optimal control $u_k^*$ by dynamic programming (see Table 1). For each problem maximum of $N_t \times N_w$ memory locations are needed for storing the optimal value and control for each of the possible states. This is an worst-case scenario approximation. In practice most of these states are not feasible, e.g., we cannot have states for which $|r_0| - |r_k| > N_w$ (i.e., more targets destroyed than we have weapons). Further storage space is needed for the set $U$ which is explicitly created every time it is needed.

Limited lookahead solutions are found by algorithm laid out in Table 2. It is almost identical to the algorithm in Table 1, except that instead of recursing the full depth of the problem for each stage $k$ we solve a truncated problem where we employ the approximate cost-to-go $\hat{J}_k$ when we reach recursion depth $k + N_{\text{steps}}$. Here $N_{\text{steps}}$ is the number of stages to lookahead and is 1 for one-step lookahead and 2 for two-step lookahead.
Let $t = 0$, $m = m_0$, $r = r_0$ and $J^* (t, m, r) = -\infty$
Call $J (t, m, r)$

FUNCTION $J (t, m, r)$:

if $t >= N$ or $r = \emptyset$ or $m = 0$ then
    return 0
end if

if $J^* (t, m, r) > 0$ then
    return $J^* (t, m, r)$
end if

for all $u \in \{ u \subseteq r | |u| \leq m \}$ do
    $V(u) = S(u) + J (t + 1, m - |u|, r \setminus u)$
end for

$J^* (t, m, r) = \max_u V(u)$

$u^* (t, m, r) = \arg \max_u V(u)$

return $J^* (t, m, r)$

Table 1: Dynamic programming solution with recursion

5 Examples

Here two example problems are described and the solutions analyzed. Both problems are solved with the full DP algorithm.

5.1 Simple example

We first consider a simple example with $N_w = 3$ and $N_t = 6$. Here one of the targets is set to be a high value target with no shooting capability. The placement of the problem along with a full DP solution is presented in Figure 2.

SAM sites are plotted with triangles and their range are plotted as circles. High value targets are marked with a cross. The index number and value of the target are plotted below and above the symbol marking the target. For each stage, lines are drawn from the weapons to the assigned targets. Dotted line marks safe path and thick line marks risky part of path. Circles plotted with dotted lines depict inactive sites.

In the first stage, the two foremost SAM sites guarding the high-value target are selected as targets. In the second stage the last remaining weapon is targeted at the high value target 6 (value 100) that is directly behind the
Let $t = 0$, $k = 0$, $m = m_0$, $r = r_0$ and $\tilde{J}^*(t, m, r) = -\infty$.

**while** $t < N$ **do**

Call $\tilde{J}(k, m, r)$

$m = m - |\tilde{u}^*(t, m, r)|$

$r = r \setminus \tilde{u}^*(t, m, r)$

$k = \max [0, t + N_{steps} - N]$

$t = t + 1$

**end while**

FUNCTION $\tilde{J}(t, m, r)$:

**if** $t >= N$ or $r = \emptyset$ or $m = 0$ **then**

return 0

**end if**

**if** $t = N_{steps}$ **then**

return $\bar{J}(t, m, r)$

**end if**

**if** $\tilde{J}^*(t, m, r) > 0$ **then**

return $\tilde{J}^*(t, m, r)$

**end if**

for all $u \in \{ u \subseteq r | |u| \leq m \}$ **do**

$V(u) = S(u) + \tilde{J}(t + 1, m - |u|, r \setminus u)$

**end for**

$\tilde{J}^*(t, m, r) = \max_u V(u)$

$\tilde{u}^*(t, m, r) = \arg \max_u V(u)$

return $\tilde{J}^*(t, m, r)$

**Table 2:** Dynamic programming solution with recursion, limited lookahead
Figure 2: Example problem with \( p_s = 0.9, \lambda = 0.9, N_w = 3 \) and \( N_t = 5 \) SAM sites (triangles) plus one high value target (cross). Figures show the full DP solution for stages 1 and 2. Dotted lines denote targeted weapons, and dotted circles mark destroyed targets. The risky portion of each line is plotted with thick line. In the first stage two SAM sites are targeted. In the second stage the remaining weapon is targeted at the high value site.

SAM site 4. Expected value for the non-cooperative solution is about 114 where as for the cooperative solution it is about 126. This shows quite clearly what is meant when we talk about cooperation. First assigning weapons to the two guarding SAM sites has increased the expected value for the problem.

5.2 Larger example

To evaluate functionality of the program a slightly larger problem is considered. A larger problem and its full DP solution is depicted in Figure 3. This is a problem similar to the one presented in [2], thus it also works as a verification case for the implemented solver.

In the first stage almost all non-high value targets are targeted. The sites blocking the path to high value target 6 are selected in the first stage since the expected value of taking them out consecutively would mean that the expected value derived from the high value target would diminish due to time discounting. The targets 2, 3 and 5 are blocking the path to high value target 10 and are selected for removal for the same reasons as the target 6. The target 4 is blocking the path to (slightly higher value) the target 1 and
Figure 3: Optimal solution of the problem with 10 targets and 10 weapons, with $\lambda = 0.9$ and $p_s = 0.9$.

is thus selected for removal. In the second stage all remaining targets are selected.

5.3 Value of cooperation

Value of cooperation is studied by varying the parameters $\lambda$ and $p_s$ for the problem depicted in Figure 3, while $N_w$ and $N_t$ are kept constant. Effect of increasing the time discount factor $\lambda$ is to favor solutions that play out over larger number of stages, whereas decreasing $\lambda$ has the opposite effect. Increasing $p_s$, the probability of successfully flying over unit distance of risky path, should favor cooperative solutions. Thus cooperation should bring the largest reward when the environment is relatively risky and the time-discount factor is relatively large.

Following [2] we consider the degree of suboptimality of the solution

$$ D = 100 \times \frac{J_{optimal} - J_{approximation}}{J_{optimal}} \%.$$  \hspace{1cm} (27)

In Figure 4 the degree of suboptimality for one- and two-step lookahead policies is plotted as a function of $\lambda$ and $p_s$. It can be seen that increasing $\lambda$ makes limited lookahead perform worse. This follows from increased time discount factor favoring solutions where fewer weapons are targeted at targets where there is a SAM site obstructing the path. Rather it is more
beneficial to target these obstructing sites before targeting anything behind them. This leads to longer solution sequences to be optimal and thus the myopic approximate solvers perform worse. The success probability $p_s$ doesn’t seem to have much effect on the performance of the solvers.

5.4 Evaluation of approximate dynamic programming

The effectiveness of limited lookahead was studied by simulating a set of random problems similar to the larger example problem depicted in Figure 3. The test problems were created so that $N_w = 10$, $N_t = 10$, $p_s = 0.9$ and $\lambda = 0.9$ were held constant. The positions of the targets are chosen randomly from a uniform distribution over $(20 \ldots 100) \times (20 \ldots 100)$. The ranges for each site were also drawn from a uniform distribution between 10 and 30. Further three target sites were randomly chosen to be high value targets with range 0 and value of 100. Finally, the position of weapons was set to be $(0, 0)$.

Performance of one- and two-step lookahead is studied by calculating the degree of suboptimality for each simulation. Results from a 100 random simulations are depicted in Figure 5. It can be seen that the two-step lookahead is indistinguishable from the full DP solution in the majority of the cases (Degree of suboptimality close to 0 %). In running times there is a
clear difference in favor of one- and two-step lookahead. Average computation times were, 195, 106 and 2 seconds for full DP, two- and one-step lookahead respectively. The difference in favor of the one-step lookahead is quite dramatic.

6 Discussion

The dynamic weapon target assignment problem was presented and a dynamic programming solution to the problem was implemented. A presentation of the implemented solver was then given.

The code presented here is not perhaps very efficient. Using recursion makes the programming easy but the added function calls may add significant overhead. Some of the slowness is also due to using Matlab as the coding platform. Matlab is best suited for vectorized calculations and vectorizing the dynamic programming solver isn’t very simple if even possible. Rewriting the solver to solve without recursion and vectorizing the code should bring some speed improvements.

The example problems presented clearly show the benefit of cooperation in risky environment. One and two-step lookahead solutions were also measured against the full DP solution and it was found that the two-step lookahead
performed almost as well as the full DP solver for most situations and found the solution in about half the time. One-step lookahead was dramatically faster than either of the other solvers and produced only slightly worse solutions. Two-step lookahead performed very nearly equally as well as the full DP- solution but was significantly faster.

In the formulation the problem, the assumption was made that all weapons are identical and located in single location. A more realistic formulation would be to consider several types of weapons located at several locations. Some weapons might also be cheaper (thus yielding greater value, if costs are considered) and others more effective and thus have a higher probability of success. The same logic applies to the SAM sites: some of them might be more effective than the others. A more realistic formulation would be to consider several types of weapons located at several locations. These improvements would be easy to implement.

Another assumption was that locations of the targets (SAM-sites and high-value targets) are known and stay the same from stage to stage. In reality, the locations of the targets are very likely never known exactly and they probably aren’t constant. Furthermore a target is not necessarily destroyed when a weapon is assigned to it.Accounting for these things would lead to a shoot-look-shoot type of stochastic dynamic optimization problem (see, e.g., [9]) and cannot be handled by the method presented here without some heavy modification.

References


A Usage of the solver

Problems can be entered by the aid of text files containing comma separated values. Problems defined in `.csv` files can be loaded with the function `WTAloader`. The format of the file should be

\[ x, y, r, s, p, \text{type} \]

are the \( xy \)-coordinates, range, value and probability \( p \) and type of the target respectively. Type should be 1 for a SAM site and 0 for a high-value target with no shooting capability. The last line in the \( \text{csv} \) file should be \( x_0, y_0, N_w, \lambda, \text{NSTEPS}, \text{FIGS} \). The entries on the last line are \( xy \)-coordinates of the weapons, number of weapons, time discount factor, Number of lookahead steps and a number indicating if figures should be shown (=1) or not (=0). For full DP-solution set \( \text{N\_STEPS} \) to a number that is larger than the number of targets. For shorter lookahead set the number lower. Setting \( \text{N\_STEPS} \) to 0 causes the problem to be solved non-cooperatively. For example the input files for the smaller and larger example introduced in this paper would be

for the smaller example and

B Source code

Here the matlab code for the programs demonstrated here, except for the GUI are presented. The GUI code is left due to it’s considerable size and irrelevance to the actual solutions. For better description of the problem structure the reader may refer to the section 4

B.1 J.m

This file implements the objective function \( J_N \) used in the dynamic programming solution. Global variables `Jast` and `uast` are used to track the optimal solution \( J^* \) and \( u^* \) respectively. The solutions \( J^*_k (r, m) \) are stored
in an array of size $N \times 2^{N_t} \times N_w$. If we describe the state of target sites by a vector $r$, where the elements of $r$ are 1 if the site is active and 0 if the site has already been targeted, we can map each state of the active sites to a natural number by setting

$$R = \sum_{i=1}^{N_t} r_i \cdot 2^i$$

This has the benefit of very fast lookup (no comparisons or dynamic memory management needed) and storing of the optimal control and cost-to-go values, with the penalty of wasting memory. This procedure also handles the $N$-step lookahead versions of the problem. The $N$-step lookahead requires setting the global variable `N_STEPS` to a value that is less than the horizon of the problem $N$.

```matlab
1 % cost-to-go for weapon target allocation
2 % arguments
3 % t simulation time
4 % r vector of length Nt with 0 for inactive and 1 for active site
5
6 function [Jopt uopt]=J(t,r,m)
7 global N N_STEPS Nw Nt Jast uast lambda st visit_counter
8
9 activeSites=r;
10 activeSitesInd=find(r>0);
11 r_index=sum(bitset(0,activeSitesInd));
12 if(t==N_STEPS)
13    [Jopt uopt]=Jbar(t,r,m);
14    return
15 end
16 if(t==N || ~any(r) || m==0)
17    Jopt=0;
18    uopt=zeros(1,Nt);
19    return
20 end
21 if(t>0)
22    if(Jast(r_index,m)>=0)
23        Jopt=Jast(r_index,m);
24        uopt=bitget(uast(r_index,m),1:Nt);
25        return
26    end
27 end
28```

24
end
end
visit_counter=visit_counter+1;
Jopt=-1;
uopt=zeros(1,Nt);
absr=sum(activeSites);
%create the set of feasible controls:
% we can target the maximum of m and abs(r)
max_targets=min(absr,m);
u=zeros(1,Nt);
u=logical(u);
for i=1:max_targets
    %set of feasible controls when targeting i targets
    U=nchoosek(activeSitesInd,i);
    for j=1:size(U,1)
        u(:)=0;
        u(U(j,:))=1;
        Jtmp=S(r,u)+lambda*J(t+1,xor(r,u),m-i);
        %Jtmp=Jtmp+S(u);
        if(Jtmp>Jopt)
            Jopt=Jtmp;
            uopt=u;
        end
    end
end
if(t>0)
    Jast(r_index,m)=Jopt;
    uast(r_index,m)=sum(bitset(0,find(uopt>0)));
end
function EV=S(r,u)
    EV=sum(getProb(r,u).*st(u));
end
B.2 Jbar.m

This file implements the approximate objective function $\tilde{J}_N$. It returns the sum of the largest $m$ expected values.

```matlab
1 1%Cheap cost-to-go approximation..
2 2% sort remaining sites by their expected value.
3 3% target as many as possible
4 function [Jopt uopt]=Jbar(t,r,m)
5 global N Nt Nw Jast uast st Nc
6 activeSites=r;
7 activeSitesInd=find(activeSites==1);
8 r_index=sum(bitset(0,activeSitesInd));
9 EV=zeros(1,Nt);
10 if(t==N || isempty(activeSitesInd)|| m==0)
11     Jopt=0;
12     uopt=zeros(1,Nt);
13 else
14     uopt=zeros(1,Nt);
15     EV=getProb(r,r).*st(find(r>0));
16     [EV idx]=sort(EV,1,'descend');
17     num_targets=min(m,length(idx));
18     uopt(idx(1:num_targets))=1;
19     Jopt=sum(EV(1:num_targets));
20     if(t>0)
21         Jast(t,r_index,m)=Jopt;
22     end
23 end
24 end
25 if(any(uopt>0) & t>0)
26     uast(t,r_index,m)=sum(bitset(0,find(uopt)));
27 end
28 end
```

B.3 Solver.m

This is the main solver file.

```matlab
1 1% DP algorithm solver for Weapon Target Assignment
```
function [Jopt uopt]=Solver(problem,solverOpts)
global Nt Nc Nw xt yt rt st Jast uast mask N p lambda N_STEPS x0 y0 Jappr uappr visit_counter
Nt=problem.Nt; %Number of targets
Nw=problem.Nw; %Number of weapons
N=min(problem.Nw,problem.Nt); %Maximum time-steps
x0=problem.x0; % xy coords of weapon launch position
y0=problem.y0;
N_STEPS=solverOpts.N_STEPS; % N_STEPS lookahead
Jopt=[];
uopt=[];
xt=problem.data(:,1); % xy-coordinates of target sites
yt=problem.data(:,2);
rt=problem.data(:,3); % range of SAM sites 0 for non-sam site
st=problem.data(:,4); % values of target sites
probs=problem.data(:,5);
type=problem.data(:,6);
lambda=problem.lambda;
% set r=0 for all non-SAM sites
for k=1:length(type),
    if(type(k)==0)
        rt(k)=0;
    end;
end;
p=ones(Nt); % precalculate probabilities
for i=1:Nt,
    [I d ix iy]=getIntersections(ones(1,Nt),i);
    if(isempty(I))
        p(i,:)=1;
    else
        p(i,I)=probs(I).'.^d;
    end
end
Nc=uint32(2^Nt); % Number of possible target states
r0=logical(ones(1,Nt));
m0=Nw;
% template for indexing;
if(N_STEPS<N)
    Jast=-1*ones(sum(bitset(0,1:Nt)),Nw); % optimal value for each state
    uast=zeros(sum(bitset(0,1:Nt)),Nw); % optimal control for each state as integer
    else

Jast=-1*ones(sum(bitset(0,1:Nt)),Nw); % optimal value for each state
uast=zeros(sum(bitset(0,1:Nt)),Nw); % optimal control for each state as integer
end
m0=Nw;
t=0;
uopt=zeros(N+1,Nt);
Jopt=zeros(N+1,1);
visit_counter=0;
t1=cputime;
if(N_STEPS<N)
    [Jtmp uopt(1,:)]=J(0,r0,m0);
    Jopt(1)=sum(getProb(r0,uopt(1,:)>0).*st(uopt(1,:)>0));
    r=xor(r0,uopt(1,:));
    m=m0-sum(uopt(1,:));
    % solve the simplified problem for each stage
    % simplification handled by function J
    for t=1:N-1,
        if(t>N-N_STEPS)
            N_STEPS=max(0,N_STEPS-1);
        end
        T=0;  % Never look beyond the horizon
        [Jtmp uopt(t+1,:)]=J(T,r,m);
        % Calculate the real cost-to-gos
        Jopt(t+1)=sum(getProb(r,uopt(t+1,:)>0).*st(uopt(t+1,:)>0));
        Jopt(1:t)=Jopt(1:t)+lambda.^(t:-1:1).'*Jopt(1+t);
        r=xor(r,uopt(t+1,:));
        m=m-sum(uopt(t+1,:));
        if(m<=0 || sum(r)==0), break;end;
        Jast(:)=-1;
        uast(:)=0;
    end
else
    %get solution for the first stage
    [Jtmp,utmp]=J(t,r0,m0);
    %lookup the rest of the solution
    Jopt(1)=Jtmp;
    uopt(1,utmp)=1;
    r=r0;
    m=m0;
for i=1:N-1,

m=m-sum(uopt(i,:));
r=xor(r,uopt(i,:));
if(m<=0 || sum(r)==0), break;end;
Jopt(i+1)=Jast(sum(bitset(0,find(r))),m);
uopt(i+1,:)=bitget(uast(sum(bitset(0,find(r))),m),1:Nt);
% optimal value for each state
% optimal control for each state as integer

end

t2=cputime;
if(solverOpts.FIGS>0)
    fprintf('J visited %d times. Time: %d',visit_counter,t2-t1);
drawSolution(Jopt,uopt,r0,m0,problem,solverOpts);
end

B.4 getIntersections.m

This function takes the index number of the target site as an argument and returns the list of all intersection points of a line drawn from \((x_0, y_0)\) to \((x_c, y_c)\) with circles drawn at SAM sites. It returns the x and y coordinates of the intersection points in arrays ix and iy, the numbers of the sites intersected J and the lengths of each intersection d. If the target is a SAM site, the last intersection ends at the center of the site. The details of the function might require some explaining. The algorithm for finding the intersection points is

1. Let \(\mathbf{p}_0\) be the position vector of starting point of the weapon launch site and \(\mathbf{p}_0\) the position vector for the assigned target of the weapon. Let \(d = (\mathbf{p}_1 - \mathbf{p}_0) / \|\mathbf{p}_1 - \mathbf{p}_0\|\) be the direction vector from \(p_0\) to \(p_1\).
2. For all targets do:

(a) Let \( \mathbf{c} \) be the \( i \)th targets position vector and \( r \) the range.
(b) Let \( \mathbf{V} = \mathbf{c} - \mathbf{p}_0 \) be the direction vector to SAM site \( i \)
(c) Let \( t = \langle d, V \rangle \). Now \( \mathbf{D} = \mathbf{p}_0 + t\mathbf{d} \) is the point on line from \( \mathbf{p}_0 \) to \( \mathbf{p}_1 \) closest to \( \mathbf{c} \)
(d) If \( \|\mathbf{D}\| \geq r \) we have at most an tangent line, if there are targets left go to a) other wise stop
(e) The two intersection points reside symmetrically around point \( \mathbf{p}_0 + t\mathbf{d} \):

\[
\mathbf{ic} = \mathbf{p}_0 + t\mathbf{d} \pm \lambda \mathbf{d} \\
\lambda = \sqrt{r^2 - \|\mathbf{D}\|^2}
\]

(f) If SAM sites left goto a)

3. Do some final cleaning up.

```matlab
function [J d ix iy]=getIntersections(activeSites,i)
    global xt yt rt x0 y0
    J=[];
    d=[];
    ix=[];
    iy=[];
    p1=[x0 y0];
    p2=[xt(i) yt(i)];
    dir=p2-p1;
    dir=dir/norm(dir);
    for k=1:length(xt)
        if(activeSites(k)<1), continue; end;
        r=rt(k);
        c=[xt(k) yt(k)];
        if(xt(k)-r>max(x0,xt(i)) || xt(k)+r<min(x0,xt(k))), continue; end;
        if(yt(k)-r>max(y0,yt(i)) || yt(k)+r<min(y0,yt(i))), continue; end;
        diff=c-p1;
        t=dot(diff,dir);
        \%t=max(t,0);
        \%t=min(t,1);
    end
end
```
closest=p1+t*dir;
dist=c-closest;
if(norm(dist)<1E-10)
  if(i==k)
    ic2=c;
    ic1=c-rt(k)*dir;
  else
    ic2=c+rt(k)*dir;
    ic1=c-rt(k)*dir;
  end
  ix=[ix; ic1(1) ic2(1)];
  iy=[iy; ic1(2) ic2(2)];
  J=[J k];
  d=[d norm((ic2-ic1))];
else
  t=sqrt(r^2-norm(dist)^2);
  ic1=closest-t*dir;
  ic2=closest+t*dir;
  ix=[ix; ic1(1) ic2(1)];
  iy=[iy; ic1(2) ic2(2)];
  J=[J k];
  d=[d norm((ic2-ic1))];
end
end
%
% clean up overshoots
if(~isempty(J))
  ind=ix(:)>max(xt(i),x0);
  ix(ind)=max(xt(i),x0);
  ind=iy(:)>max(yt(i),y0);
  iy(ind)=max(yt(i),y0);
  ind=ix(:)<min(xt(i),x0);
  ix(ind)=min(xt(i),x0);
  ind=iy(:)<min(yt(i),y0);
  iy(ind)=min(yt(i),y0);
  ind=(ix(:,1)==ix(:,2) & iy(:,1)==iy(:,2));
  ix=ix(~ind,:);
  iy=iy(~ind,:);
  J=J(~ind);
B.5 getProb.m

This function calculates the success probability of shooting at target i. It uses function getintersections to calculate the intersection distances and then returns the probability of success as \( p = \prod_{j \in A} p_j d_j \). Where \( A \) is the set of active sites intersected.

```matlab
function prob=getProb(activeSites,targets)
    global p
    prob=prod(p(targets,activeSites),2);
end
```