Dynamic priority broadcasting channels: a multi-objective planning problem

Chiel Kooijman - 5743028
University of Amsterdam

Abstract

This article explores the broadcasting channel problem, focusing on dynamic priority planning. We will show that using the Pareto-optimal set of solutions significantly reduces the set size and computation time, but that it is still infeasible. We make suggestions for a method for reducing computation time of generating the Pareto-optimal set and show that pruning is necessary.

Keywords: Dynamic Programming, MDP, Markov Decision Process, Multi-objective, Planning, Broadcasting Channel, Dynamic Priority, Pareto-optimal set

1 Introduction

In this research we extend the broadcasting channel problem as defined by Ooi and Wornell (1996), in order to account for situations in which multiple objectives are important, such as workload distribution in a data environment, distribution of traffic, or prioritisation of more important messages, in addition to throughput. In section 2 we redefine the problem to account for these considerations. Section 3 covers the methods used to meet these requirements, and some implementation specifics. The results are described in section 4, and section 5 proposes an algorithm that exploits the properties of the data to reduce the time complexity of calculating the Pareto-optimal set.

2 Problem definition

Multiple agents broadcast messages over a single channel, but only one message can be sent at any time, otherwise conflicts will arise and no message will come through. After a message is sent, the agents will know whether a message was sent, a conflict has arisen, or no message was sent. Agents have a single buffer that may contain a message. A visualisation of a two-agent setup is shown in figure 1.

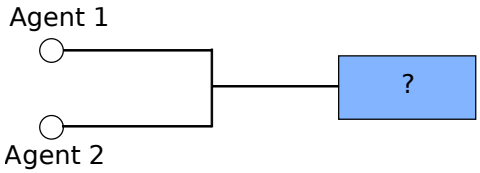


Figure 1: A setup with two agents

At any concrete time-step an agent i with an empty buffer may get a new message with probability $p_i(s'|s, \vec{a})$. The problem of optimising throughput has been solved for POMDPs (Ooi and Wornell, 1996; Hansen et al., 2004), but in this case we would also like to take into account other goals such as avoiding dominance of a single agent over the channel or guaranteeing a certain throughput for an important agent.

Hansen et al. (2004) have provided a solution with dynamic programming. This method will be extended so that for any weight vector $\vec{w}$ it will return the optimal Pareto-optimal set of solutions $\vec{w} \cdot \vec{V}^{\pi}$ (Vamplew et al., 2011), in a reward representation that can dynamically adapt priorities by changing

the weights (Barrett and Narayanan, 2008; Natarajan and Tadepalli, 2005).

2.1 Relevance

Real-world situations that may benefit from this research may include operating system schedulers, collaborative multi-agent communication, and home networks.

For instance the Completely Fair Scheduler (CFS) that is used in the Linux kernel (as of the 2.6.23 release), prioritises tasks that use less time. This may not be the ideal solution, as tasks that only need to be performed on a slow interval are often not of a high priority. Tasks such as user interaction may require a fair amount of computation time, but this should not reduce its precedence with regard to background tasks, as the user will notice lags former more than in the latter.

An example of a collaborative system are rescue robots that work together on mapping an environment in order to effectively find and rescue victims from sites such as collapsed buildings or mines. Sharing all information as it is acquired could be too heavy a burden on the network, and dynamic prioritisation could be used to make the most efficient use of the available bandwidth. Priority of the agent would be based on measures such as expected information gain.

A home network could include telephones, televisions, and personal computers. Although we may want to prioritise the telephones and television to guarantee continuity, we would also like to reduce lag in tasks that are requested by the user such as loading a website. Telephone and television prefer a specific continuous amount of throughput, but after some value do not benefit much more, whereas other tasks may always benefit from more throughput, such as downloading large files. When an alarm call is made, we would like to guarantee the quality of the line before all else, but not drop other tasks if it can be avoided. Many requirements may change over time as devices phone calls are ended or personal computers are connected to the network.

3 Approach

3.1 Reward Representation

There are several ways to represent throughput and dominance in a vector. One approach is to have a two-dimensional vector $\vec{r}$ that contains the total throughput and some measure of dominance, for instance entropy or variance. An advantage of this approach is that the size of the reward vector is independent of the number of agents.

Another way is to represent the vector as an n-dimensional vector:

$$\vec{r} = \begin{bmatrix} t_1 \\ \vdots \\ t_n \end{bmatrix}$$

where n is the number of agents, and each value represents the throughput of the corresponding agent. An advantage is that the vector contains more information than in the aforementioned representation, which allows us to use different measures of optimality, but there are many more states when the number of agents increases. It does however allow for prioritising messages, and when we choose not to prioritise, the number of states can be greatly reduced, because every vector is equal to all of its permutations (provided we reorder the state vector and the reward vector in the same way). For example in a two-agent system, a state with a reward vector $\vec{r}$ and function parameter $\vec{\theta}$ vectors

$$f_{\vec{\theta}}(\vec{r}) \text{ with } \vec{r} = \begin{bmatrix} 7 \\ 3 \end{bmatrix}, \ \vec{\theta} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

the value is equal to that of

$$f_{\vec{\theta}}(\vec{r}) \text{ with } \vec{r} = \begin{bmatrix} 3 \\ 7 \end{bmatrix}, \ \vec{\theta} = \begin{bmatrix} w_2 \\ w_1 \end{bmatrix}$$

because all agents are equal and connected to the network in the same way. For these reasons the second representation was chosen.

This method can also be extended to a multi-channel set-up: Where we would reward only one agent, and only when it was the only agent to send a message in the single-channel set-up, we can reward any number of agents that sent a message, if

there were no more messages sent than the number of channels.

3.2 State Representation

The Bellman equation 1 describes the value V of a state s given policy π. It is defined by the return R that is obtained in state s, plus the expected return in the following states s'given the policy. Discount value γ is a value in $[0, 1)$, and defines the weight of future rewards. Low values can be seen as making the agent impatient, because a states value is mostly defined by expected return in the near future.

$$V(s)^{\pi} = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) V^{\pi}(s') \quad (1)$$

In a system with an event horizon we can represent a state as a vector of booleans, that represents for each agent whether it has a message to send or not. Furthermore, discounting is unnecessary when calculating state values, whereas in a system without a horizon discounting the reward is necessary to make the values converge. With Dynamic Programming (DP) we can calculate the optimal policy. Values for the states can be redefined as shown in equation 2.

$$V(s)^{*} = R(s) + \max_{a} \sum_{s'} P(s'|s, a) V^{*}(s') \quad (2)$$

In our case it may be more useful to represent values of state-action pairs, as the immediate return R is dependent on the action given the state. Equation 3 shows how this value Q can be calculated.

$$Q(s, a)^{*} = \sum_{s'} P(s'|s, a) \left(R(s, a) + \max_{a'} Q^{*}(s', a') \right) \quad (3)$$

As the max-operator is not defined for vectors, the output consists of a set of vectors instead of a scalar. To reduce the size of this set, we only use the subset of Pareto-optimal values, which contains all optimal values for any definition of optimality we would choose in this task. A vector belongs in a Pareto-optimal set if there are no other vectors that are higher or equal in all dimensions.

Equation 4 shows the definition for the Pareto-optimal set.

$$\left\{ V \,\middle|\, \forall V' [V \neq V' \wedge \exists n [V_n > V'_n]] \right\} \quad (4)$$

An example of a Pareto-optimal set for a two-dimensional system is shown in figure 2.

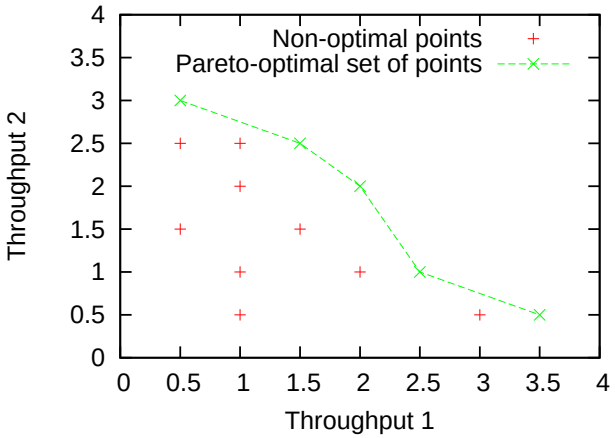


Figure 2: Pareto-optimal set in a two-dimensional system

The number of different possible states is 2^n where n is the number of agents. In the fully observable system there are up to $n + 1$ possible actions: any one agent can send a message or none at all. In a system where agents are unable to observe the states of other agents there are up to 2^n possible actions as any combination of agents can send a message.

3.3 Implementation

The planning algorithm was implemented in C++, as it allows for low-level optimisations as well as object-oriented programming and generic programming, and a wide array of libraries such as The Standard Template Library (STL), are available. The goal of the algorithm is to compute the values for every action in every state of every time, given that it will follow an optimal policy.

States and actions are represented as integers, which can be seen as arrays of booleans. For example, binary 0100 represents the *state* where only the third agent has a message, or the *action* where only the third agent sends a message, dependent on context. This representation is used to improve performance.

Algorithm 1 shows the main loop of the algorithm. Starting from the last time-step (at $t = 0$) we calculate the reward of all possible actions. While the value at $t = 0$ is only dependent on direct reward, for the other time-steps it is the sum of the direct reward for taking the action and the weighted average of the values of the possible transition states (multiplied with the probability of reaching that state given the action). On line 12 the action is converted to an index (i.e. 0100 to 3), and 1 (the direct reward) is added to the vectors in that dimension.

Algorithm 1 Main Loop

```
 1: procedure MAIN(n_agents, time, p_msg[])
 2:     n_states ← POW(2, n_agents)
 3:     Q[max_t][n_states]                                   ▷ Initialise Q
 4:     for t ← 0, time do
 5:         for s ← 0, n_states do
 6:             actions ← GET_ACTIONS(s, n_agents)
 7:             for all a ∈ actions do
 8:                 rewards[] ← TRANS_REWARDS(a, s, n_states, p_msg)   ▷ States multiplied with probabilities
 9:                 new_rewards = ADD_SETS(rewards)
10:                 if a then                                ▷ Action is not 'do nothing'
11:                     for all r ∈ rewards do
12:                         INCREMENT r[ACTION2INDEX(a)]     ▷ Add direct reward
13:                         Q[t][s].TRY_ADD(a, r)            ▷ Try to add to set
14:                     end for
15:                 end if
16:             end for
17:         end for
18:     end for
19: end procedure
```

The TRY_ADD method described in algorithm 2 adds the vector to the set $Q(s_t, a)$ if it is Pareto-optimal. And if it is optimal, it also tries to add it to $V(t_s)$. TRY_ADD_V works in the same way as TRY_ADD. The loop exits early when it is apparent that the new vector does not belong in the $Q(s_t, a)$ set. When it is sub-optimal it also does not belong in the $V(s_t)$ set, as it contains the optimal vectors from all actions, and is thus always as least as strict as $Q(s_t, a)$.

Algorithm 2 Adding vectors to the set

```
1: procedure Q.TRY_ADD(a, r)
2:     for all r_old ∈ actions[a] do
3:         if SAME(r_old, r) then
4:             return                                  ▷ The same vector already in set
5:         else if BETTER(r_old, r) then
6:             return                                  ▷ Better vector already in set
7:         else if BETTER(r, r_old) then
8:             actions[a].ERASE(r_old)                 ▷ Remove from set and try to remove more
9:         end if
10:    end for
11:    actions[a].INSERT(r)                            ▷ Done removing, add to set
12:    TRY_ADD_V(r)                                    ▷ Try to add to V
13: end procedure
```

The function described in algorithm 3 adds the sets of sets of vectors. It is analogous to the summation symbol in the Bellman equation. The resulting set consists of each possible sum of vectors from each set. The algorithm works by setting the first set aside. It is then replaced by all sums with the vectors of the next set. This is repeated for all sets. The number of sets is dependent on the number of actions that can be taken in the current state.

Algorithm 3 Adding sets of vectors

```
1: function ADD_SETS(sets)
2:     temp = {}
3:     result = sets.POP()
4:     for all s ∈ sets do
5:         temp = COPY(result)
6:         result = {}
7:         for all v ∈ s do
8:             for all w ∈ temp do
9:                 result.INSERT(v + w)
10:            end for
11:        end for
12:    end for
13:    return result
14: end function
```

The TRANS_REWARDS function generates all possible states and compares them to the state minus the action. For example if the state is 1111 and the action is 0100, the result is 1011. Then all possible transition states are compared to this state. In this case only 1111 and 1011 can follow. In the actual implementation a static mapping between the *least_state* and the matching return value, so that it can be looked up and is only calculated if it was not in the set yet. The transition probabilities are then calculated by multiplying each 0 in the base with the probability $p\_msg[i]$ or $1 - p\_msg[i]$, dependent on whether it becomes a 1 (gets a new message) or stays a 0 (does not get a new message) respectively. Since the transitions and transition probabilities are the same for every time-step, it is possible to calculate the return value of TRANS_REWARDS only once for each possible state.

Algorithm 4 Getting transitions and probabilities

```
1:  function TRANS_REWARDS(a, s, n_states, p_msg)
2:      states = {}                ▷ In the real implementation this variable is static and used as a cache
3:      least_state ← s ⩒ a                                  ▷ Remove message that has been sent
4:      for mask ← 0, n_states do
5:          new_state ← least_state ∨ mask                   ▷ Transition states only add messages
6:          p ← TRANS_P(new_state, least_state, p_msg)
7:          states.INSERT(p · new_state)                     ▷ states is a set, so duplicates are not added
8:      end for
9:      return states
10: end function
1:  function TRANS_P(new_state, least_state, p_msg)
2:      p_total ← 1.0
3:      mask ← 1
4:      for all p ∈ mathitp_msg do
5:          if ¬(mask ∧ state) then                          ▷ State can change (agent has no message)
6:              if mask ∧ new_state then
7:                  p_total ← p_total · (1 − p)              ▷ Agents gets no message
8:              else
9:                  p_total ← p_total · p                    ▷ Agents gets message
10:             end if
11:         end if
12:         mask ← 2^mask                                    ▷ Bit shift
13:     end for
14:     return p_total
15: end function
```

	1 agent	2 agents	3 agents	4 agents	5 agents
$p = 0.1$	100157	190211	271604	344064	410397
$p = 0.2$	200015	360030	488788	590683	672028
$p = 0.5$	500050	749407	875208	937568	968696
$p = 0.7$	699996	909630	972937	991940	997638

Table 1: Total throughput for turn-based policy after 1000000 turns

4 Results

To explore where a planning algorithm would benefit most in terms of total throughput, a small experiment was conducted where a number of agents with the same transition probabilities followed a turn-based policy. Table 1 shows that the turn-based approach is more efficient as the number of agents and the probability of obtaining a new message increases. This is because after the all agents have had their first turn, the probability of having a message to send is $P(m)^n$, as there are n times that a new message can fill the buffer with probability $P(m)$. The throughput suffers most when the probability of only one agent having a message, but not being able to send it because it must wait for its turn is most prevalent.

In figure 5 we see that the size of the Pareto front is not uniform for all probability distributions. The sizes are smallest where probabilities are 0.5 and on the diagonals of the quadrants. The shape of the graph is similar with and without optimisation, and figure 5d shows that most reduction in size is achieved in the larger sets. The smaller set sizes can be explained by the fact that these probability distributions produce more duplicate vectors, which are only included once in the sets.

Note that some of the values shown are redundant, because with two agents with different transition probabilities, the size of the set is independent of which agent has which probability.

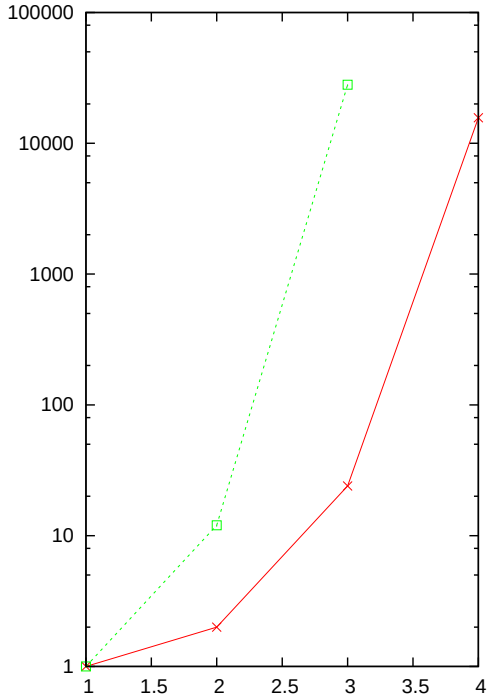
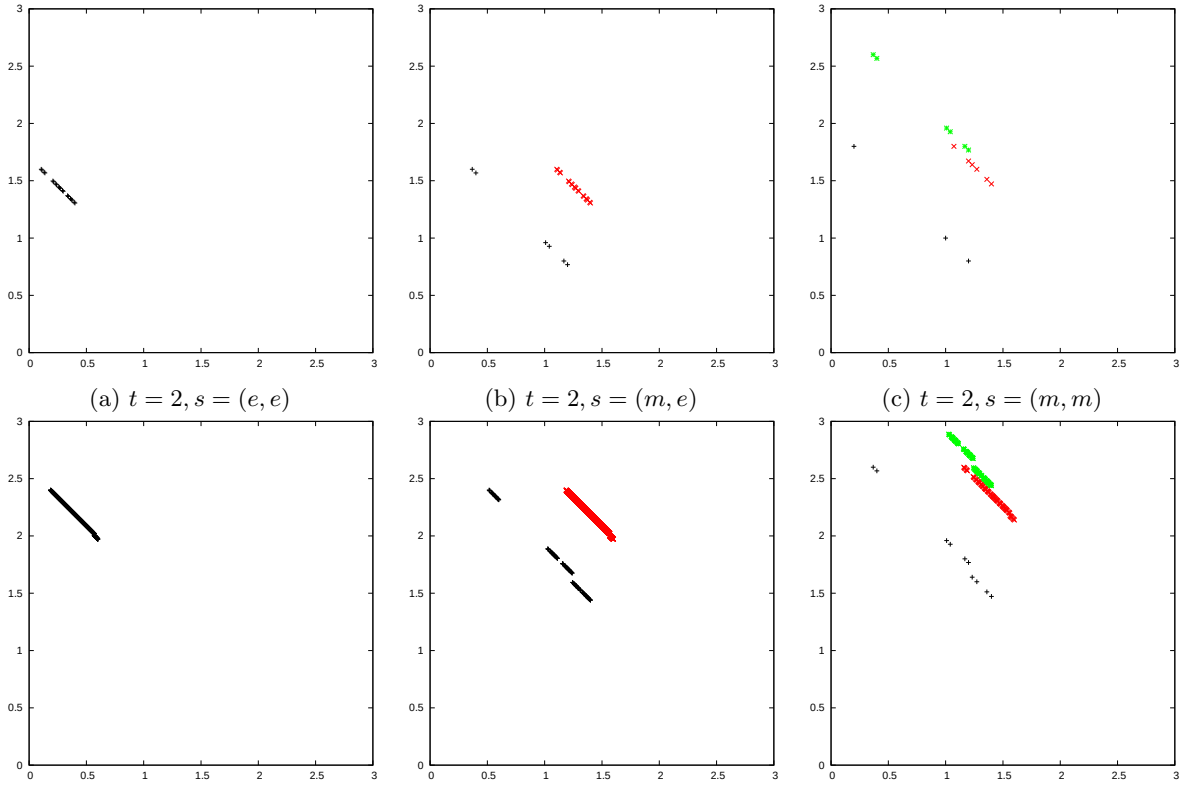


Figure 3: Size of the Pareto-optimal set with (red) and without (green) Pareto-optimisation on a logarithmic scale for different time-steps

Figure 3 shows the difference in the set size with and without Pareto-optimisation for time-steps up to 4. Although we can see that the optimisation reduces the size more than an order of magnitude, it also shows that that the size of the set grows faster than exponential in both cases (note that the y-axis is on la logarithmic scale).

$$Q(s,a)^* = \sum_{s'} P(s'|s,a)\left(R(s,a) + \max_{a'} Q^*(s',a')\right)$$

The reason for this complexity can be found



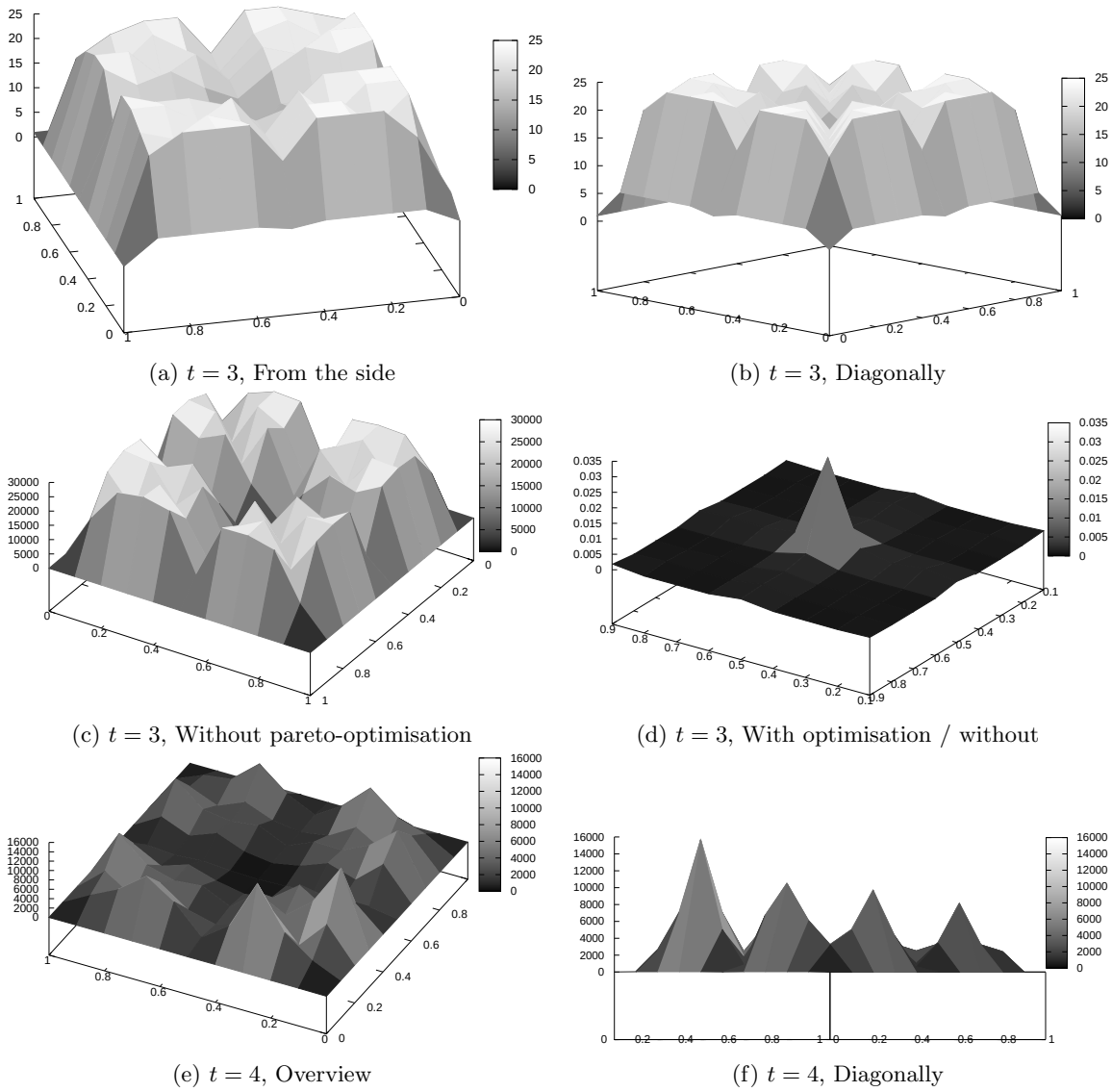
(a) $t = 2, s = (e, e)$ (b) $t = 2, s = (m, e)$ (c) $t = 2, s = (m, m)$

(d) $t = 3, s = (e, e)$ (e) $t = 3, s = (m, e)$ (f) $t = 3, s = (m, m)$

Figure 4: Pareto fronts in time-steps 2 and 3 for different states. Colours denote different actions m and e represent a full and an empty buffer respectively

by analysing the Bellman equation (3). The max-operator returns a set of solutions for each action, and the sets are summed. As an example to illustrate the problem, consider four actions each containing a set of four rewards. The result contains each possible sum of any vector from each set. This results in a set of $4^4 = 256$ elements. For the next time-step, this would result in a set of $4^{256} \approx 1.34 \cdot 10^{154}$ vectors (assuming that there were no duplicates and all states had the same set size).

In figure 4 we can see the distribution of the Pareto-fronts per action for several different probabilities with two agents. Black represents sending no message, red and green are for the different agents, and in the last image the overlapping vectors are shown in yellow. Striking is that the points are all located on up to two lines (or hyperplanes) per action, where the lines are always perpendicular to the line that makes half a right angle from the origin. In figure 6 we can see that the points are also divided in two lines per action for time-step 4, which suggests that the number of hyperplanes is independent of the time-step, and presumably dependent on the number of agents. Figure 7 suggests that this may also hold true for more dimensions, although this may also be because the transition probabilities are the same for all agents.



(a) $t = 3$, From the side

(b) $t = 3$, Diagonally

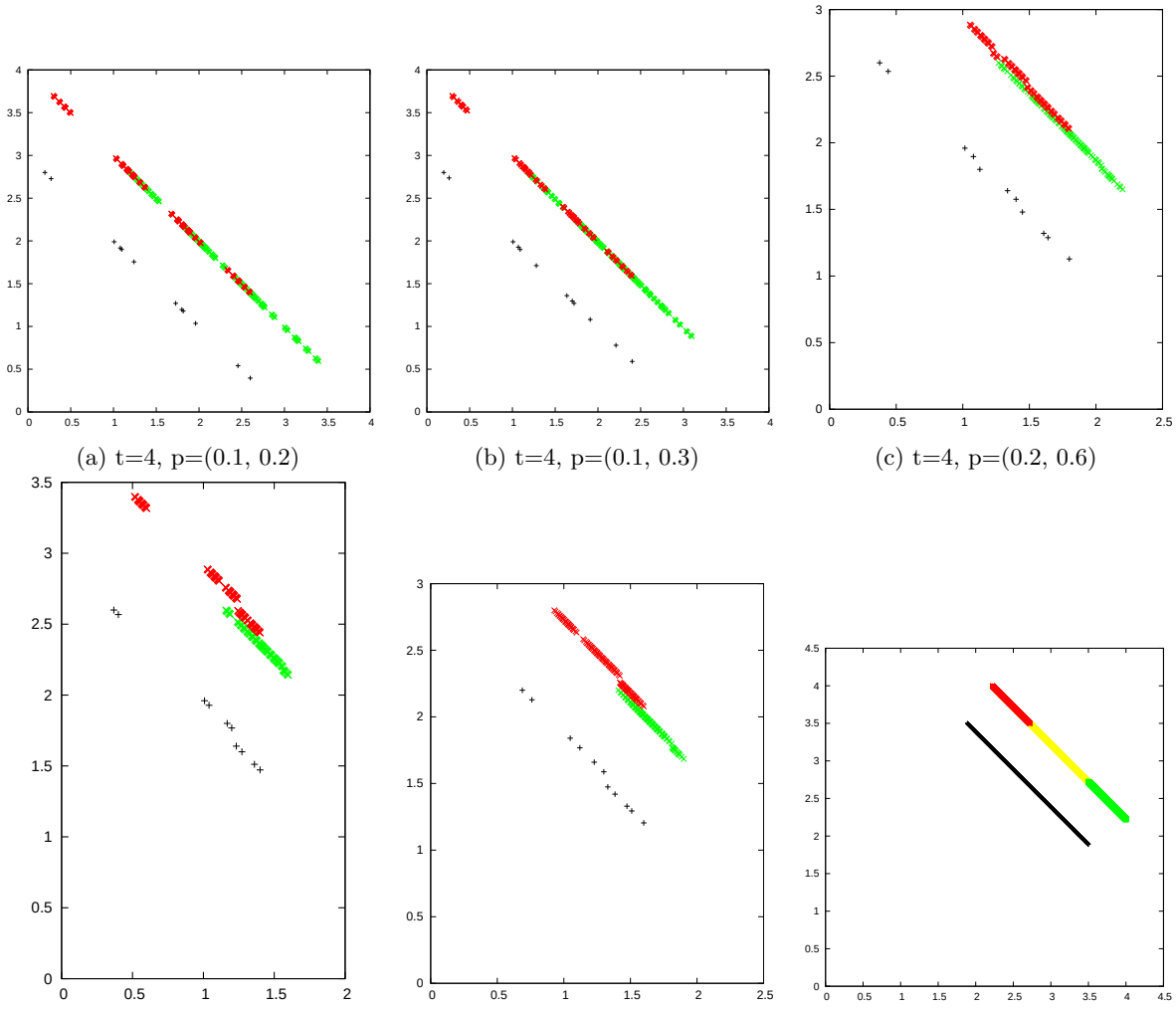
(c) $t = 3$, Without pareto-optimisation

(d) $t = 3$, With optimisation / without

(e) $t = 4$, Overview

(f) $t = 4$, Diagonally

Figure 5: Size of the pareto-front over different probability distributions



(a) t=4, p=(0.1, 0.2) (b) t=4, p=(0.1, 0.3) (c) t=4, p=(0.2, 0.6)

(d) t=4, p=(0.2, 0.8) (e) t=4, p=(0.4, 0.7) (f) t=7, p=(0.5, 0.5)

Figure 6: Shape of the Pareto-front for different time steps and probabilities with two agents. Black indicates no action, green and red are for the different actions. Yellow in figure 6f is used to show the overlap between values of the two actions. The axes represent the throughput of the respective agents

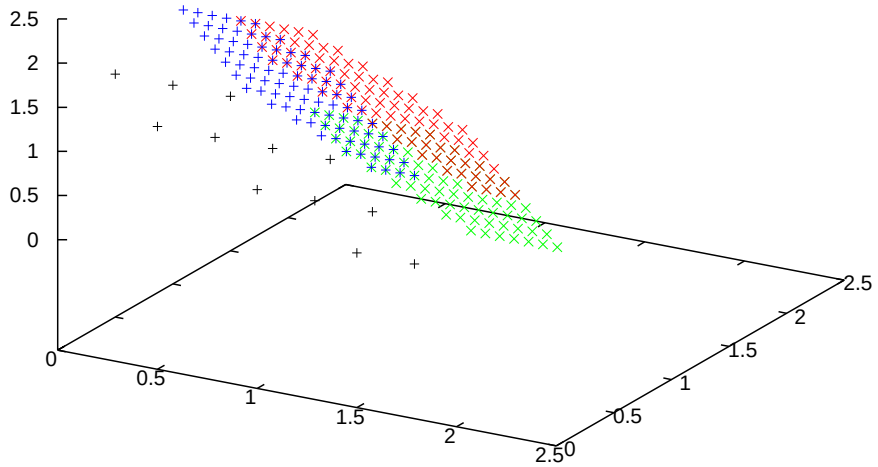


Figure 7: t=3, p=(0.5, 0.5, 0.5)

5 Discussion

Due to computational complexity it was infeasible to calculate the fronts for more than two agents, except in the case where all probabilities are 0.5. In future research more aggressive pruning (5.2) could help to analyse the shape of the Pareto-set and may confirm or deny our suspicions about the distribution of vectors described in section 4.

5.1 Sorted Sets

As the C++ implementation of the `set` class is sorted, it is possible to select subsets based on separators. In the case of adding a new vector to the set there are three distinct possible situations:

1. Elements must be removed and the new vector added
2. No elements must be removed but the new vector must be added
3. No elements must be removed and the new vector must not be added

When we consider the whole set per dimension, we can separate the vectors in the set into three different groups of interest:

1. Elements that may be strictly better
2. Elements that may be strictly worse
3. Elements that are not better or worse

Sets 1 and 2 may intersect when the values of the vector that have been compared are the same as the vector that is to be added. The fist and second categories contain values that are equal or larger or equal or smaller in each evaluated dimension respectively. Vectors that have a higher value in at least one dimension and a lower value in another, fall in the third category. All vectors that have been identified as belonging in the third category can be excluded from evaluation, as they do not affect the actions that must be performed. As we explore more dimensions the size of the third group will increase and the first two will decrease.

When we have reached the last dimension, the sets tell us what action to take:

- If set 1 is not empty: do not add (situation 3)
- Else if set 2 is not empty: remove all vectors from this set and add the new vector (situation 1),
- Else (sets 1 and 2 are empty): add the new vector (situation 2).

Since in the case of two agents, the sets are in a large part distributed on a few lines, as can be seen in figures 4, and 6, we could represent them as several sets, grouped by the lines they are on. As all points are on a single line, the vectors in the set are sorted in both the first and the second dimension. The set in which a vector belongs can be characterised by the distance between the origin and the projection of the vector on the line that makes a half right angle from the origin (equation 5).

$$|v \cdot \frac{\vec{1}}{|\vec{1}|}| \tag{5}$$

Table 2 shows an example of the Pareto-set for three agents. Vectors are shown horizontally in the order that it is contained in the set. When we consider all vectors that share a value in the first dimension, we see that the last two vectors are both sorted in similarly to the case with two agents. As the container type allows us to divide such sets based on separators (with UPPER_BOUND and LOWER_BOUND from the STL), this allows us to analyse the last two dimensions simultaneously in $O(\log(n))$.

0	0.1	1.9
0	1	1
0	1.7	0.3
0.1	0	1.9
0.1	0.9	1
0.3	1	0.7
0.3	1.7	0
0.7	1	0.3
0.9	0.1	1
1	0	1
1	0.2	0.8
1	0.8	0.2
1	1	0
1.2	0	0.8
1.2	0.8	0

Table 2: Example Front: $s = (e, e)$, $p = (.1.3.8)$, $t = 2$

5.2 Aggressive Pruning

Figure 3 shows that the complexity of the algorithm is worse than exponential, and therefore it is not usable beyond around four time-steps in a set-up with two agents, and fewer time-steps for more agents. This makes it necessary to prune the set, for example by using only the ε-approximate Pareto-set. Clustering algorithms could be used to keep the set on a constant maximum size for any time-step.

References

L. Barrett and S. Narayanan. Learning all optimal policies with multiple criteria. In *Proceedings of the 25th international conference on Machine learning*, pages 41–47. ACM, 2008.

E.A. Hansen, D.S. Bernstein, and S. Zilberstein. Dynamic programming for partially observable stochastic games. In *Proceedings of the National Conference on Artificial Intelligence*, pages 709–715. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2004.

S. Natarajan and P. Tadepalli. Dynamic preferences in multi-criteria reinforcement learning. In *Proceedings of the 22nd international conference on Machine learning*, pages 601–608. ACM, 2005.

J.M. Ooi and G.W. Wornell. Decentralized control of a multiple access broadcast channel: Performance bounds. In *Decision and Control, 1996., Proceedings of the 35th IEEE*, volume 1, pages 293–298. IEEE, 1996.

P. Vamplew, R. Dazeley, A. Berry, R. Issabekov, and E. Dekker. Empirical evaluation methods for multiobjective reinforcement learning algorithms. *Machine learning*, 84(1):51–80, 2011.