Distributed Reinforcement Learning for Multiple Objective Optimization Problems

Carlos E. Mariano
Instituto Mexicano de Tecnología del Agua
Paseo Cuauhnáhuac 8532
Jiutepec, Morelos, México 62550
cmariano@tlaloc.imta.mx

Eduardo F. Morales
ITESM Campus Morelos
Paseo de la Reforma 182-A
Temixco, Morelos, México 62589
emorales@campus.mor.itesm.mx

Abstract

This paper describes the application and performance evaluation of a new algorithm for multiple objective optimization problems (MOOP) based on reinforcement learning. The new algorithm, called MDQL, considers a family of agents for each objective function involved in a MOOP. Each agent proposes a solution for its corresponding objective function. Agents leave traces while they construct solutions considering traces made by other agents. The solutions proposed by the agents are evaluated using a non-domination criterion and solutions in the final Pareto set for each iteration are rewarded. A mechanism for the application of MDQL in continuous spaces which considers a fixed set of possible actions for the states (the number of actions depends on the dimensionality of the MOOP), is also proposed. Each action represents a path direction and its magnitude is changed dynamically depending on the evaluation of the state that the agent reached. Constraints handling, based on reinforcement comparison, considers reference values for constraints, penalizing agents violating any of them proportionally to the violation committed. MDQL performance was measured with "error ratio" and "spacing" metrics on four test bed problems suggested in the literature, showing competitive results with state-of-the-art algorithms.

1 Introduction

Real world optimization problems involve more than one, often incommensurable and conflicting, objective function. In MOOP, unlike single objective optimization problems where there is a unique optimal solution, there exist a set of them responding to the dominance optimality criterion known as *non dominated* or *Pareto optimal* solutions.

In recent years there has been a growing interest by researchers in developing, testing, comparing and adapting methods for the solution of multiple objective optimization problems. In particular, there is a special interest by the Evolutionary Algorithms community, where more that 200 publications related to evolutionary methods for the solution of this kind of problems have been produced since 1984 [12]. In this paper, an alternative approach based on reinforcement learning, and in particularly on Q-learning, is presented.

In Q-Learning an autonomous agent learns an optimal policy that outputs the appropriate action given the current state from the environment [10]. Learning the optimal policy depends on the estimations for states values in the environment, which are incrementally updated until optimal states values are found. This contrasts with evolutionary algorithms and other search strategies that search in the space of policies without ever appealing to value functions. If the space of policies is sufficiently small, or can be structured so that good policies are common or easy to find, then evolutionary methods can be effective. Q-Learning has been widely used in control and as it will be described in this paper, can be stated for optimization problems and for the solution of combinatorial optimization problems.

A new distributed approach based on Q-Learning and its extension to solve Multiple Objective Optimization problems (MDQL) was recently proposed in [6]. In this paper, the performance of MDQL using "error ratio" and "spacing" metrics over four test-bed problems proposed in the literature is presented.

An introduction to Q-Learning and a description of the distributed approach proposed for Q-Learning are given in sections 2 and 3. Section 4 describes how to applied reinforcement learning to optimization problems. In section 5 a description of MDQL is given. Section 6 describes the experiments and results on which MDQL's performance was measured. Finally, conclusions and future research directions are discussed in section 7.

2 Q-Learning

A general Reinforcement Learning model is described in Fig. 1. The agent and environment interact at each sequence of discrete time steps, $t = 0, 1, 2, 3, \ldots$. At each time step t, the agent receives some representation of the environment's *state*, $s_t \in S$, where S is the set of all possible states, and on that basis selects an *action*, $a_t \in A(s_t)$, where $A(s_t)$ is the set of actions available on state s_t. One step later, in part as a consequence of its actions, the agent receives a numerical *reward*, $r_{t+1} \in \Re$, and finds itself in a new state, s_{t+1}.

At each time step, the agent implements a mapping from states to probabilities of selecting each possible action. This mapping is called the agent's *policy* and is denoted as π_t, where $\pi_t(s, a)$ is the probability that $a_t = a$ if $s_t = s$. Reinforcement learning methods specify how the agent changes its

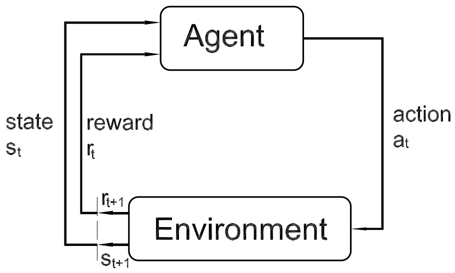

Figure 1: The agent-environment interaction in reinforcement learning.

Q-Learning
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode)
Initialize s
Repeat (for each step of episode)
Choose a from s using policy derived from Q (e.g., $\varepsilon-$greedy)
Take action a, observe r, s'
$Q(s,a) \leftarrow Q(s,a)+$
$\alpha\,[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
$s \leftarrow s'$
Until s is terminal

Table 1: Q-Learning: An off-policy Temporal Difference algorithm

policy as a result of its experience. The agent's goal, roughly speaking, is to maximize the total amount of reward it receives in the long run.

One of the most important reinforcement learning method is Q-Learning [15]. In this case, the learned action-value function, Q, directly approximates Q^*, the optimal action-value function, independently of the policy being followed. In this setting, Q-values are updated according to:

$$Q(s,a) \quad \leftarrow \quad Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right] \tag{1}$$

where, $Q(s,a)$ is the action value function of taking action a in state s, $Q(s',a')$ is the action value function in the resulting state (s') after taking action a, α is the step-size parameter $[0,1]$, r is the reward, and γ is a discount rate. It is clear, from Equation 1, that the agent needs only to consider each available action a in its current state s and choose the action that maximizes $Q(s,a)$.

It may at first seem surprising that one can choose globally optimal action sequences by reacting repeatedly to the local values of $Q(s,a)$ for the current state. This means the agent can choose the optimal action without ever conducting a lookahead search to explicitly consider what state results from the action. Part of the beauty of Q-Learning is that the evaluation function is defined to have precisely this property, the value of $Q(s,a)$ for the current state and action summarizes in a single number all the information needed to determine the discounted cumulative reward that will be gained in the future if action a is selected in state s.

Table 1 describes the Q-Learning algorithm. The agent's estimation of $Q(s,a)$ converges, in the limit, to the actual Q^*, provided the system can be modeled as a deterministic Markov decision process, the reward function r is bounded, and actions are chosen so that every state-action pair is visited infinitely often.

3 Distributed Q-Learning

In order to reduce convergence times a set of *m* independent agents is considered. Agents try to establish an optimal policy over states and actions (identify the best action for every state that let the agent reach a previously defined goal), however, during the learning process, decisions made by the agents affect the rest of the agents behavior. To induce this cooperative behavior to the agents, every time an agent visits a state and selects and action, it leaves a trace, updating a value function over the current state-action pair. The rest of the agents consider these traces in their decision process of which action to take.

At the beginning of an episode, agents are assigned initial states and all the actions value functions $Q(s,a)$ are copied[1]). Every agent make a state transition following the Q-Learning criterion (selects an action for its current state) using the copies of the action value functions $Q_c(s,a)$. Once a transition is made (the agent goes from its current state to the resulting state performing the selected action), the agent modifies the copied action value function using Equation 2.

$$Q_c(s,a) \quad \leftarrow \quad Q_c(s,a) + \alpha\left[\gamma \max_{a'} Q_c(s',a') - Q_c(s,a)\right] \tag{2}$$

Since all the $Q_c(s,a)$ are common to all agents, the more frequently visited state-actions pairs will be more attractive for the agents.

An episode finishes when an agent completes a number of steps or when it reaches the goal. When all the agents have finished an episode, the proposed solutions are compared and evaluated finding the best of them. The best solution elements (state-action pairs visited) are rewarded, and the original action value functions are updated using Equation 1 (see Table 2).

4 From RL to optimization

One of the main subjects in this work is to establish the criteria under which Q-Learning can be applied to the solution of

[1]There is one $Q(s,a)$ for each s, a pair.

DQL
Initialize $Q(s,a)$ arbitrarily
Repeat (for n episodes)
Repeat (for m agents)
Initialize s, copy $Q(s,a)$ to $Q_C(s,a)$
Repeat (for each step of episode)
Take action a, observe r, s'
$Q_C(s,a) \leftarrow Q_C(s,a)+$
$\alpha\,[\gamma \max_{a'} Q_C(s',a') - Q_C(s,a)]$
$s \leftarrow s'$;
Until s is terminal
Evaluate the m proposed solutions
Assign rewards to the best solution found
$Q(s,a) \leftarrow Q(s,a)+$
$\alpha\,[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$

Table 2: General algorithm for DQL, in this case a greedy evaluation policy is considered, but other policies are possible.

optimization problems. RL has been generally applied to discrete spaces where its goal is to converge to an optimal policy that reaches a known goal state starting from an initial state. Solutions to optimization problems involve the localization of (near) optimal points in a (generally) continuos space. To apply RL techniques in the solution of optimization problems a discrete solution space with a finite number of states, each state with a set of available actions is considered. It can be said that RL agents search two things in optimization problems, states which evaluation in the objective space define them as optimal, and the establishment of the optimal policy to reach them starting from an initial state. So, for the solution of optimization problems the agent must identify states in the environment with high evaluation function, that can be considered as goals, and define an optimal way to reach them. In other words, for the solution of optimization problems with RL, an initial state is defined for a particular optimization problem and an optimal policy is learned to reach (near) optimal states from it.

In order to discretize the solution space, two previous works were considered. George Bilchev, in [1], proposed an adaptation of the Ant Colony model for continuous problems. His proposal considers for each state a set of possible directions for the trajectory of agents. The magnitude of this trajectory is bounded by an upper limit. The selection of the trajectories and magnitudes is performed randomly at the beginning and then the most frequently selected directions are assigned higher probabilities of selection.

In an earlier work, Deniz Yuret [16] proposed an algorithm, named Dynamic Hill Climbing (DHC), which uses a similar mechanism. DHC is based on three heuristics: storage of local minimum and restarts from distant points from them, adjustment of the magnitudes of the trajectories used during the search, and use of successful trajectories for faster climbings (descendings).

In this paper, we take ideas from both proposals to deal with continuous functions. Each state has a set of possible actions which depends on the problem to be solved. The actions are associated with the directions that an agent can take, which depends on the dimensionality of the problem to be solved. For one dimension there are two possible directions, for 2 there are 8 (considering both directions for each dimension and all the directions of the combinations of them), for 3 there are 26, and so on.

The magnitudes of the trajectories are adjusted according to their evaluation (as it will be explained below). If a state-action pair increases the evaluation function, its magnitude is doubled, otherwise it is divided by 2. The initial magnitude is given arbitrarily. The magnitude of the trajectories represent the rewards that the agent receives when selecting a state-action pair (r in Equation 1).

4.1 Constraint handling

In the proposed algorithm, constraint violation determines the amount of penalization to be assigned to the state-action pairs involved in a solution. The implementation uses a mechanism similar to the reinforcement comparison method proposed by Sutton [9]. The reward for a state-action pair involved in a solution violating any of the constraints in the optimization problem, is affected by a punishment based on a measure of the violation. Rewards are modified dynamically in proportion to the current reward. For feasible solutions:

$$\overline{r}_{t+1} = \begin{cases} \overline{r}_t + \beta\,[r_t - \overline{r}_t] & \text{if violation of constraints} \\ \overline{r}_t \times N & \text{otherwise} \end{cases}$$

where $\overline{r}_{t+1}$ is the reward for the state-action pair of the next interaction, $t+1$, $\overline{r}_t$ is the reward for the state action pair in the current interaction, t, β, $0 \leq \beta \leq 1$, is a step-size parameter, and r_t, is the punishment. r_t considers the difference between the current reward and the addition of all the committed violations:

$$r_t = \overline{r}_t - \sum_{j=1}^{q} f_j(\overline{X}), \tag{3}$$

where, $f_j(\overline{X})$, measures the violation of the j-th. constraint and q is the number of constraints.

In the case where there is no violation of constraints, rewards are scaled by N which can take values of 2 or 1/2, depending on whether there is or is not an improvement on the evaluation function. With this adjustment in the magnitude of the trayectories, agents reach in a smaller number of steps promising space solutions and can use the remaining steps to improve on previous solutions.

MDQL
Initialize $Q(s,a)$ arbitrarily
Repeat (for n episodes)
Initialize s, copy $Q(s,a)$ to $Q_C(s,a)$
Repeat (for m agents)
Repeat (for f families)
Repeat (for each step of episode)
Take action a, observe r, s'
$Q_C(s,a) \leftarrow Q_C(s,a) +$
$\alpha\left[\gamma \max_{a'} Q_C(s',a') - Q_C(s,a)\right]$
$s \leftarrow s'$;
Until s is terminal
compromise_solution=**negotiation**(m solutions)
if (*compromise_solution* are *non-dominated*)
Pareto=Pareto + *compromise_solution*
For all Pareto solutions do
$Q(s,a) \leftarrow Q(s,a) +$
$\alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$

Table 3: General algorithm for MDQL, the same number of agents is considered for all families.

5 Distributed Q-Learning for Multi-objective Optimization Problems

DQL can be extended for multiple objective optimization problems. The idea is to assign a family of agents to each objective. The solutions obtained by the agents in one family are compared with the solutions obtained by the agents from the rest of the families. A negotiation mechanism is used to find compromise solutions satisfying all the objectives. This mechanism was originally proposed and tested in [4, 5] using Ant-Q [3]. The limitations and extra parameters associated with Ant-Q based algorithms lead us to propose a more theoretical sustained algorithm based exclusively on Q-learning. This algorithm, called MDQL, is described in Table 3.

Each agent of each family proposes a solution for the objective that it is optimizing. The proposed solution of the i-th. agent of all the families (i.e., a solution for all the objectives) is evaluated and compared against the rest of the $m-1$ solutions proposed by the rest of the agents. A negotiation mechanism, described in Table 4, is responsible for the creation of Pareto sets. The associated rewards of the states involved in the solutions are updated according to their evaluation function (this is described in the following section). By delaying the adjustment in the rewards MDQL achieves a better distribution of solutions (Pareto set) while keeping its exploration capabilities.

In the negotiation mechanism all solutions are evaluated for all the objectives. If a solution satisfies all the objectives it is considered as a candidate compromise solution. If there is more than one possible solution, the best one is returned. Ties are broken arbitrarily. *Compromise_solution* follows a non-dominant criterion to construct a Pareto front. Non-dominated solutions are kept in the Pareto front and rewards

Negotiation(m solutions)
Let *compromise_solution*=NULL
Repeat (for each solution i)
Repeat (for each objective j)
if (objective$_j$(solution$_i$) achieves
all objective$_j$(solution$_j$))
compromise_solution=
best_of(solution$_i$, *compromise_solution*)
if(*compromise_solution*=NULL)
compromise_solution=**random**(m solutions)
return *compromise_solution*

Table 4: Negotiation mechanism that compares the solution proposed by an agent with the solutions proposed by agents from other families.

are assigned to their components. In case where there in no solution which dominates the rest, a new solution is chosen randomly from the rest of the solutions at that episode. The number of episodes and steps are given arbitrarily, as long as a sufficiently large number is chosen to ensure convergence.

6 Experiments

In order to test MDQL performance, two metrics proposed in the literature were used over the results obtained on four test-bed problems. A description of the metrics is given below, where PF_{MDQL}, represents the results reported by the algorithm and PF_{true} represent the real solutions.

Error Ratio: Solutions in PF_{MDQL} which are not members of PF_{true} are considered as errors. This metric is mathematically represented by [13]:

$$E \triangleq \frac{\sum_{i=1}^{n} e_i}{n}$$

where n is the number of vectors in PF_{MDQL} and

$$e_i = \begin{cases} 0 \text{ if vector } i \in PF_{true} \\ 1 \text{ otherwise} \end{cases}$$

For example, $E = 0$ indicates that every vector reported by MDQL in PF_{MDQL} is actually in PF_{true}; $E = 1$ indicates that none are.

Spacing: To measure the spread (distribution) of vectors throughout PF_{MDQL} a metric proposed in [7] was implemented. This metric measures the range (distance) variance of neighboring vectors in PF_{MDQL}:

$$S \triangleq \sqrt{\frac{1}{n-1}\sum_{i+1}^{n}\left(\overline{d} - d_i\right)^2},$$

where $d_i = \min_j \Big(\mid f_1^i(\overrightarrow{x}) - f_1^j(\overrightarrow{x}) \mid + \mid f_2^i(\overrightarrow{x}) - f_2^j(\overrightarrow{x}) \mid \Big)$, $i,j = 1,\ldots,n$, $\overrightarrow{d}$ is the mean of all

d_i, and n is the number of vectors in PF_{MDQL}. $S = 0$ indicates all members of PF_{MDQL} are equidistantly spaced.

MDQL has been successfully tested on several MOOPs suggested in the literature [6]. In this paper, the results on MDQL on four instances of MOOP considered to be difficult are presented.

The first problem is Viannet's third three-objective function [14]. Its Pareto front (PF_{true}) is a single, convoluted three-dimensional Pareto curve and its Pareto set (P_{true}) consists of disconnected areas in the solution space.

$$
\begin{aligned}
\textit{minimize} f_1(x,y) &= 0.5(x^2+y^2)+ \\
&\quad \sin(x^2+y^2), \\
\textit{minimize} f_2(x,y) &= \frac{(3x-2y+4)^2}{8}+ \\
&\quad \frac{(x-y+1)^2}{27}+15, \\
\textit{minimize} f_3(x,y) &= \frac{1}{(x^2+y^2+1)}- \\
&\quad 1.1e^{(-x^2-y^2)} \\
\textit{subject to:} &\quad -30 \leq x,y \leq 30
\end{aligned}
$$

The problem proposed in [2] was selected as the second test problem. Their P_{true} and PF_{true} (four Pareto curves) are disconnected.

$$
\begin{aligned}
\textit{minimize} f_1(x,y) &= x, \\
\textit{minimize} f_2(x,y) &= (1+10y)[1-\left(\frac{x}{1+10y}\right)^{\alpha}- \\
&\quad \frac{x}{1+10y}\sin(2\pi q x)] \\
\text{subject to:} &\quad 0 \leq x,y \leq 1, q=4, \alpha=2
\end{aligned}
$$

Two side-constrained problems were also considered. The first problem is a two objective problem proposed by Srinivas [8] whose P_{true} is disconnected and PF_{true} is connected.

$$
\begin{aligned}
\textit{minimize} \quad f_1(x,y) &= (x-2)^2+(y-1)^2+2. \\
\textit{minimize} \quad f_2(x,y) &= 9x-(y-1)^2 \\
\text{subject to} &: -20 \leq x,y \leq 20, \\
0 &\geq x^2+y^2-255, \\
0 &\geq x-3y+10.
\end{aligned}
$$

The second constrained problem, proposed by Tanaka [11], is a P_{true} connected and PF_{true} disconnected and convoluted two objective MOOP.

$$
\begin{aligned}
\textit{minimize} \quad f_1(x,y) &= x, \\
\textit{minimize} \quad f_2(x,y) &= y, \\
\text{subject to} &: 0 \leq x,y \leq \pi, \\
0 &\geq -(x^2)-(y^2)+1+ \\
&\quad 0.1\cos(16\arctan\frac{x}{y}), \\
0 &\geq (x-\frac{1}{2})^2+(y-\frac{1}{2})^2.
\end{aligned}
$$

Table 5: Parameters used by MDQL for the four test problems.

	Episodes	Steps	States	Agents	Time[2] (sec.)
Viannet	500	500	500	100	200
Deb	500	500	2000	150	250
Srinivas	500	500	2000	200	68
Tanaka	500	500	2000	250	300

Table 6: Comparison between MDQL and MOMGA.

	MDQL		MOMGA	
	E	S	E	S
Viannet	60-75	0.00-0.80	NA	0.00-1.07
Deb	10-23	0.00-0.05	20-60	0.00-0.03
Srinavas	20-35	0.00-0.15	NA	NA
Tanaka	95-100	0.15-0.30	NA	NA

7 Results

The four test functions were run ten times in order to apply the performance metrics. Figures 2 to 5 show the best results obtained with MDQL for the four problems. For the first two problems the following parameter values were considered: $\alpha = 0.1$, $\gamma = 1.0$. For the third problem $\alpha = 0.5$ and $\gamma = 0.7$ were finally the parameter values with which best performance were obtained, while fourth problem's parameter values were 0.5 and 0.95 for α and γ respectively.

Table 5 shows the parameters used in the problems solution. Results from the two performance metrics applied to the four problems are given in Table 6. In this case, a comparison on the first two problems is shown against MOMGA, the best evaluated algorithm in [12]. In particular, Table 6 shows the best and worst error-ratio (E) and spacing (S) metrics of MDQL for the 10 runs (the same comparative conditions as MOMGA). MOMGA results were taken from [12].

As can be seen from the results, MDQL is able, in general, to successfully solve a wide range of multiple objective optimization problems. It also behaves favorably when compared against MOMGA. From a qualitative comparison against the results reported in [8] and [11] it appears that MDQL behaves favorably in the first case but worst in the second, although a more thorough comparison is needed.

[2]MDQL is written on Borland C++ V4.5 and runs on a Pentium at 75 MHz.

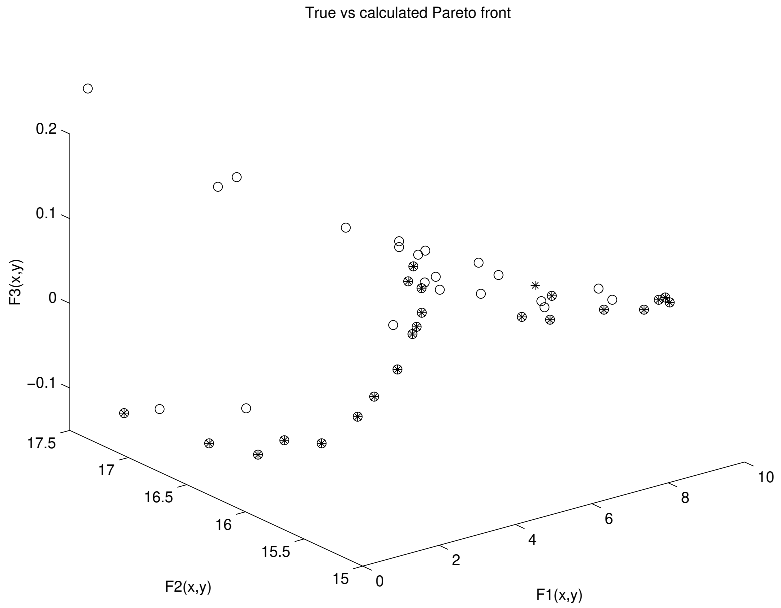

Figure 2: True vs calculated Pareto front for Viannet's three-objective MOP, * represent PF_{true}, ∘ represent PF_{MDQL}.

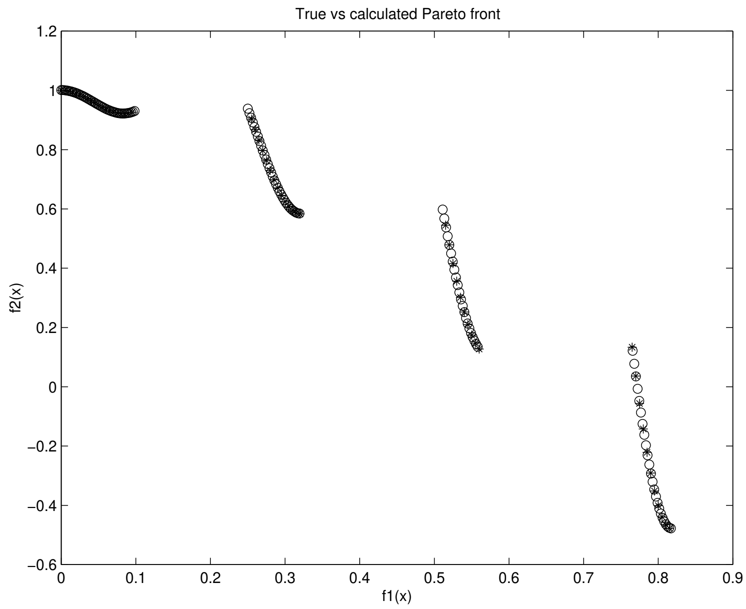

Figure 3: True vs calculated Pareto front for Deb's two-objective MOP, * represent PF_{true}, ∘ represent PF_{MDQL}.

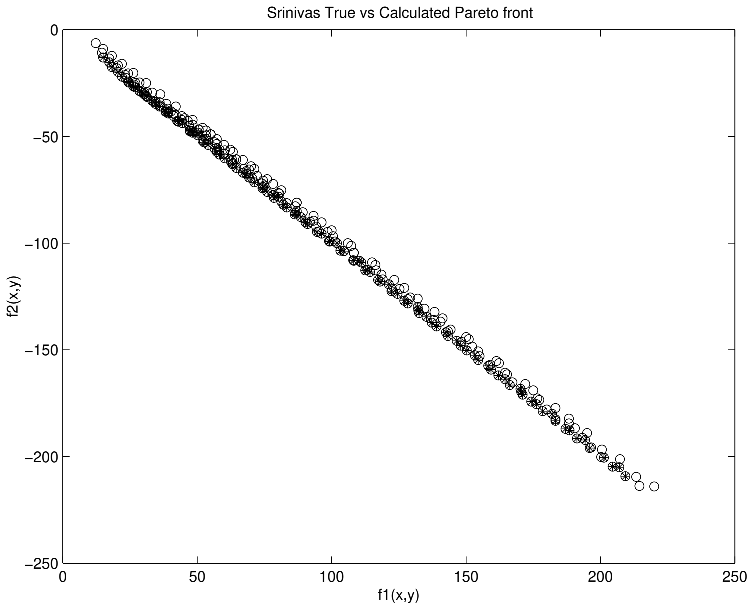

Figure 4: True vs calculated Pareto front for Srinivas' two-objective MOP, * represent PF_{true}, ∘ represent PF_{MDQL}.

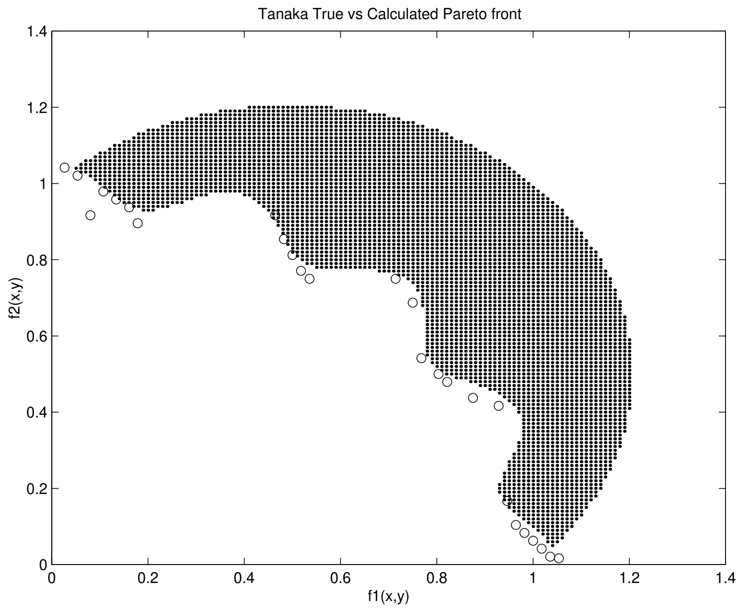

Figure 5: True vs calculated Pareto front for Tanaka's two-objective MOP, + represent PF_{true}, ∘ represent PF_{MDQL}.

8 Conclusions

This paper has shown the performance of a new algorithm, based on reinforcement learning, for multiple objective optimization problems. The algorithm, called MDQL, is able to approximate relatively well the Pareto front for a wide range of multiple objective optimization problems reported in the literature. It was also shown that MDQL has a competitive behavior when compare against recent genetic algorithms.

There are basically three advantages that can be easy detected in MDQL; a) MDQL does not need a search space encoding, operating in the phenotypical level, b) the evolution of the Pareto front is made without the use of additional mechanisms such as Niches or fitness sharing to keep diversity, so no additional parameters are needed to setup the algorithm, and c) the algorithm uses an efficient mechanism based on reinforcement comparison to handle objective functions and constraints.

There are several research directions that need to be explored. In particular, we would like to experiment with different policies and negotiation mechanisms and evaluate their influence on the performance of the algorithm. We would like to perform a sensitivity analysis on the parameters involved in the algorithm. Finally, a more thorough comparison with alternative algorithms must be performed.

Bibliography

[1] Bilchev, G. and Parmee, I. (1995) The Ant Colony Metaphor for Searching Continuous Design Spaces, in T. Fogarty, ed., *Lecture Notes in Computer Science 993*, Springer-Verlang, 1995, pp. 25-39.

[2] Deb, K. (1998) *Multiobjective Genetic Algorithms: Problem Difficulties and Construction of Test Problems*. Technical Report TR CI-49/98, University of Dortmund, Germany: Department of Computer Science/XI, 1998.

[3] Gambardella, L.M. and Dorigo, M. (1995) Ant-Q: A Reinforcement Learning approach to the traveling salesman problem. In *Proceedings of the 12th. International Conference on Machine Learning* (ML-95), pp. 252-260, Morgan Kaufmann.

[4] Mariano, C. and Morales E. (1999a) MOAQ an Ant-Q Algorithm for Multiple Objective Optimization Problems. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., & Smith, R. E. (eds.). *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference* pp. 894-901, San Francisco, CA: Morgan Kaufmann.

[5] Mariano, C. and Morales E. (1999b) MOAQ a Distributed Reinforcement Learning Algorithm for the Solution of Multiple Objectives Optimization Problems. In *Memorias del Segundo Encuentro Nacional de Computación ENC99*, paper No. 111, Pachuca Hidalgo, Mexico, Sociedad Mexicana de Ciencias de la Computación.

[6] Mariano, C. and Morales, E. (2000) A New Distributed Reinforcement Learning Algorithm for the solution of Multiple Objective Optimization Problems. In *Lecture Notes in Artificial Intelligence 1793, MICAI-2000: Advances in Artificial Intelligence*, O. Cairo, L.E. Sucar, F.J. Cantú (eds.), Springer Verlag, pp. 212-223,

[7] Schott, J. (1995) *Fault Tolerant Design Using Simple and Multicriteria Genetic Algorithms*. MSc. thesis, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1995.

[8] Srinivas, N. and Deb, K. (1994) Multiobjective Optimization Using Nondominated Sorting in Genetic Algorithms, *Evolutionary Computation*, 2(3), pp. 221-248.

[9] Sutton, R.S. (1984) *Temporal Credit Assignment in Reinforcement Learning*. PhD. thesis, University of Massachusetts, Amherst.

[10] Sutton, R.S. and Barto, A.G. (1998) *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

[11] Tanaka, M., et al., (1995) GA-Based Decision Support System for Multicriteria Optimization. *Proceedings of the International Conference on Systems, Man, and Cybernetics* 2, pp. 1556-1561.

[12] Van Veldhuizen, D. (1999) *Evolutionary Algorithms: Classification, Analysis, and New Innovations*. PhD Thesis Dissertation, Faculty of the Graduate School of Engineering, Air Force Institute of Technology, Air University.

[13] Van Veldhuizen, D. and Lamont, G. (1999) Multiobjective Evolutionary algorithms Test Suites, *Proceedings of the 1999 ACM Symposium on Applied Computing*, Janice Carrol et al., editor pp. 351-357.

[14] Viannet, R., et al. (1996) Multicriteria Optimization Using a Genetic Algorithm for Determining a Pareto Set, *International Journal of Systems Science*, 27(2):255-260.

[15] Watkins, C.J. (1989) *Learning from Delayed Rewards*. Ph.D. thesis, Cambridge University.

[16] Deniz Y. (1994) From Genetic Algorithms to Efficient Optimization, *MSc. Thesis in Electrical Engineering and Computer Science*, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.